Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Laurea Magistrale in Informatica

Solving the European Air Traffic Flow Management Problem with Kernel Search Matheuristics and Machine Learning

Master degree thesis

Thesis Supervisor

Prof. Luigi De Giovanni

Graduand

Gabriele Garbin

mat. 2020578

Academic year 2021-2022

Abstract

The Air Traffic Flow Management (ATFM) problem has the goal of planning flights within a set of constraints representing both capacity limits of the air space and airline company needs, consisting in a delay and a preference assigned to each trajectory; several mathematical linear programming models exist to solve this problem, and the main issue is their size, since they may contain up to millions of variables for real instances. As a consequence, the computational effort required to solve the model to optimality is huge, and not suitable for practical use.
This thesis presents a heuristic method based on Kernel Search. The goal of Kernel Search is to solve the model using only an initial subset of variables, called kernel, and dividing the remaining variables into small groups called buckets, ordered by *"promising impact on the solution"*, that is computed from variable information obtained through the resolution of the linear relaxation of the problem. Each iteration of the Kernel Search method consists in solving a small subproblem given by variables from the kernel and from a single bucket, whose size allows to solve it to optimality in a small amount of time; furthermore, in this thesis, Machine Learning techniques have been used in the process of defining the *"quality"* of each variable, in order to see if such modification in the bucket defining procedure can lead to more efficient or effective methods. The developed algorithms have been implemented and tested on real instances obtained from European data repositories, showing their ability to find optimal or very close to optimal solutions.

Thanksgiving (English)

First of all, I thank my thesis supervisor, prof. Luigi De Giovanni, for giving me the opportunity to work in this project, for giving me suggestions and support during the development of this interesting and engaging work. Furthermore, I'd like to thank other students that worked on this project before me, because they realized the basis for my part, and also I thank them for being open to help me understand such parts.

A particular thank goes to my family for pushing me to continue my studies after the bachelor degree, for being always by my side and for encouraging me whenever I had doubts and uncertainty about my future.

I greatly thank all my friends, to the ones I've known for a lot of time, to whom I lost and then I found again, and to the ones I met during my journey; we shared many good times, that gave me beautiful memories, there were also bad moment, but we overcame them together, enforcing our bond.

A final thank to everyone that has been a positive presence in my life, to who supported me in my decisions, helping me to realize my own dreams and ambitions.

Padova, December 2022 Gabriele Garbin

Ringraziamenti (Italiano)

Innanzitutto, vorrei ringraziare il mio relatore, il prof. Luigi De Giovanni, per avermi dato la possibilità di svolgere questa tesi, per avermi consigliato a supportato durante lo svolgimento di questo progetto interessante e coinvolgente. Inoltre vorrei ringraziare gli altri studenti che hanno lavorato a questo progetto prima di me, realizzando ciò che è stato il mio punto di partenza, e per essere stati disponibili ad aiutarmi a comprendere le parti svolte da loro.

Desidero ringraziare la mia famiglia per avermi spinto a continuare il mio percorso di studi dopo la laurea triennale, per essere sempre stati al mio fianco e per avermi stimolato quando avevo dei dubbi e perplessità sul mio futuro.

Un ringraziamento particolare va a tutti i miei amici, a quelli che conosco da una vita, a quelli che ho perso e poi ritrovato, e a chi ho conosciuto durante il mio percorso; abbiamo condiviso tanti bei momenti, di cui ho ricordi stupendi, ci sono stati anche dei brutti periodi, ma li abbiamo superati insieme rafforzando il nostro legame.

A tutte le persone che hanno costituito una presenza positiva nella mia vita, che mi hanno supportato nelle mie decisioni, aiutandomi a realizzare i miei sogni e ambizioni.

Padova, Dicembre 2022 Gabriele Garbin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the world, hundreds of thousands flights are operated every day, transporting both people and goods from a point of the Earth to another, thus it is imperative to regulate the traffic inside the airspace, taking into account physical and legislative limits, such as the flight level range, traffic capacity assigned to a specific air space as determined by institutions, etc. An airline company must evaluate all these information to propose a flight plan, taking into account that different trajectories lead to different flight times and to different costs, which are determined by the variable fuel cost (depending, among others, on time of flight) and by the overflight taxes that might be applied by some countries.
Depending on the adopted business model, each company has its specific metrics to determine a preferred trajectory: some evaluate more the time of arrival (preference to minimize the delay), others might prefer to avoid paying too many overflight taxes, even if that means scheduling a delay on the arrival time.
The needs of scheduling and assigning flight plans that are both coherent to the constraints previously defined, and acceptable by airline companies, have found an answer in the mathematical approach to this problem, called *ATFM (Air Traffic Flow Management)* problem, which aims to assign a flight plan to each flight, respecting all constraints, while maximizing the preferences of airline companies.

The goal of this thesis is to implement a heuristic algorithm to solve the mathematical model associated to the ATFM problem, called Kernel Search, which consists in iteratively solving a sequence of subproblems obtained by ranking the variables according to the resolution of the relaxation of the problem. In particular, such information is used to sort variables in decreasing *"promising impact on the solution"*, so that they are divided in small groups called buckets. The first group is picked and fixed as a starting point, and the model is solved using only variables that belongs to the fixed bucket and one extra bucket at a time, obtaining small problems that are manageable in terms of computational resources required.
The mathematical model for ATFM considered in this thesis is taken form literature and consists of a path formulation that uses binary decision variables associated to assignment of a realistic trajectory to each flight.
The first part of the Kernel Search based algorithm, which solves the relaxation of the full model, is faced through a dynamic column insertion algorithm, taken from literature.
Different versions of the Kernel Search approach have been developed in this thesis.

As for the standard procedure, the relevance of each variable is established from its value and reduced cost, without considering the features of the flight and the trajectory described by the variable itself. Another approach proposed in this thesis is to integrate Machine Learning techniques in the standard algorithm schema, to provide a different, and possibly more accurate and effective, determination of *"promising variables"*. In particular, clustering is used to classify variables based on trajectory, in order to identify similarities and provide a different sorting, leading to different subgroups of variables that limit redundancy in the buckets. Moreover, decision tree classification is also used, on each single bucket, as a filter that further discriminates between variables that are worth to be included or not in the submodel, based on the features of both variables and current solution.
The algorithms proposed in this thesis have been implemented and compared to the results obtained through dynamic column insertion and a rounding algorithm to get a solution of the integer problem, in order to determine if the Kernel Search algorithm finds better solutions than such rounding procedure. In this case, it is also necessary to determine if the required higher computational cost, specifically time of execution, can be effectively sustained in the resolution of the ATFM problem in practice.

1.1 Content and contributions

The second chapter describes tools and technologies used during the thesis project.

The third chapter describes in detail the ATFM problem, giving also some related literature and information on the mathematical model that will be considered by this thesis, and on existing methods for its resolution.

The fourth chapter describes the design of the Kernel Search procedure for the ATFM problem. In particular, it discusses parameter tuning and describes the implementations realized to fit the procedure for the ATFM problem, which represent a first contribution of this thesis.

The fifth chapter proposes Machine Learning integrations into the standard Kernel Search procedure, which lead to original implementations that feature clustering and decision tree filters.

The sixth chapter describes results obtained through tests of the proposed versions of the Kernel Search procedure on real-world instances, with a focus on differences between the standard and the Machine Learning implementations of the Kernel Search procedure. Results represent a further computational contribution of the thesis.

The seventh chapter summarizes up all the work done, and provides conclusive remarks.

1.1.1 Text highlighting

For this document, the following norms have been chosen:

- class, methods, package names and programming language keywords are highlighted as `keyword`;
- mathematical elements, such as sets and variables, are highlighted in *cursive*;

- foreign language terms (non-English words), acronyms, or terms belonging to technical language are highlighted in *cursive*, only when their meaning is explained, following occurrences will not be highlighted;
- bibliography references are represented by a blue number closed by square parenthesis, for example [1].

Chapter 2

Tools and Technologies

In this chapter, the main tools and technologies used in this thesis are described, giving details about the role each one played in the development of this project.

2.1 Linear Programming

A mathematical programming model is used to solve optimization problems. It involves a measure to optimize, a set of constraints representing the feasible region of the problem, and a set of variables that model decisions of the problem; a mathematical programming model can be written as:

$$\min c^T x \tag{2.1}$$
$$s.t.\ Ax \geq b \tag{2.2}$$
$$x \geq 0 \tag{2.3}$$

where $x \in \mathbb{R}^n$ (or $\mathbb{Z}^n$), $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ (having $n =$ number of decision variables, and $m =$ number of constraints).
Expression 2.1 is called objective function, 2.2 is the set of constraints and 2.3 indicates the domain of variables.
Linear Programming models are a special class of mathematical programming models, and involve, as the name suggests, linear equalities/inequalities for both the objective function and the set of constraints. A mathematical programming model is formed by the following elements (examples are given with reference to the ATFM model elements):

- sets: they group the elements of the system, for example F is the set of flights, P is the set of trajectories, and D is the set of delays;
- objective function: an expression involving decision variables, the goal is to maximize or minimize this quantity, e.g. to minimize the total delay or to maximize the sum of preferences;
- decision variables: elements unknown to the system, e.g., in the case of ATFM, y variables (indexed by flight, trajectory and delay) determine a real flight plan for a specific flight, while z variables (indexed only by flight) indicate if such flight is unplanned. Each variable denotes a column of the model, the one collecting the coefficients associated to the variable itself in the constraint matrix A of

expression 2.2. The determination of value for all decision variables that satisfies all constraints constitutes a feasible solution of the problem;

- constraints: a set of equalities/inequalities that describe restrictions regarding decision variables, each constraint denotes a row of the model, i.e., a row of the coefficient matrix A; for example, the constraint *"Flight 4 must have exactly 1 flight plan or be unplanned"* can be expressed as:

$$\sum_{p \in P, d \in D} y_{4pd} + z_4 = 1$$

- parameters: the known elements of the model, such as the capacity of each sector, the preference for each flight and the delay coefficient for each variable.

A linear programming model can be of one of the following types:

- *LP (Linear Programming)* models involve real-valued decision variables;
- *ILP (Integer Linear Programming)* models involve decision variables of integer type;
- *BILP (Binary Integer Linear Programming)* models involve decision variables of boolean type. It's a particular case of ILP where decision variable values are restricted to only 0 and 1;
- *MILP (Mixed Integer Linear Programming)* models involve decision variables of both real and integer type.

2.2 Methods for linear and integer linear programming

In this section, methods to solve mathematical programming models are described. Only methods involved in this thesis are illustrated, together with the off-the-shelf software library that allows to use such methods.

2.2.1 Simplex method

The simplex method is used to solve continuous models, i.e. LP models. It requires the problem LP to be in standard form, which is $min\{c^T x : Ax = b, x \geq 0\}$, and a feasible basis B, the procedure is the following:

1. write the LP canonical form with respect to B, the *canonical form* explicitly express basic variables and the objective function using only non-basic variables:
   $z = \overline{z}_B + c_{N_1} + \cdots + \overline{c}_{N_{(n-m)}}$
   $x_{B_i} = \overline{b}_i - \overline{a}_{i_{N_1}} - \cdots - \overline{a}_{i_{N_{n-m}}} \quad (i = 1 \ldots m)$
   where:

   $\overline{z}_B$ is the objective function value given by the basis B;
   $\overline{b}_i$ is the value of basic variable i;
   B_i is the index of the i-th basic variable $(i = 1 \ldots m)$;
   N_j is the index of the j-th non-basic variable $(j = 1 \ldots n - m)$;

$\bar{c}_{N_j}$ is the *reduced cost*, which is coefficient of the j-th non-basic variable in the objective function, with respect to the basis B;

$-\bar{a}_{i_{N_j}}$ is the coefficient of the j-th non-basic variable in the constraints that makes explicit the i-th basic variables;

2. if $\bar{c}_j \geq 0 \; \forall j$ then B is an optimal basis, `terminate`;

3. if $\exists h : \bar{c}_h < 0 \; and \; \bar{a}_{ih} \leq 0 \; \forall i$ then the problem is unlimited, `terminate`;

4. insert in B any variable with negative reduced cost $(x_h : \bar{x}_h < 0)$, and remove from B the variable x_{B_t}, where $t = arg \min\limits_{i=1 \ldots m} \{\frac{\bar{b}_i}{\bar{a}_{ih}} : \bar{a}_{ih} > 0\}$;

5. repeat all steps until the optimum solution is found, or the problem is known to be unlimited.

2.2.2 Relaxation of a model

Each model can be solved in its original formulation, or it can be solved in a different version called *relaxation*. The linear relaxation of a model keeps all the elements of the initial formulation, but all variable types are changed to real; note that, by definition, no change is made for LP models.
The relaxation is used because a complete real-valued model is easier to solve than its integer counterpart. Its solution is used by many algorithms, since its value represents a lower bound for the original problem, as a starting point to obtain a solution of the original problem that is more likely to be close to the optimal value.

2.2.3 Dynamic Column Insertion

Dynamic column insertion is an algorithm used to solve linear programming models, it is used mainly when the model to solve is too big, in terms of number of variables, to be solved in its totality. This algorithm can give the optimal solution of LP models and, hence, of the linear relaxation (described in §2.2.2) of any model. Note that this does not mean a solution of the original problem with integrality constraints will be found, but the optimal solution of the relaxation can be used as a starting point for other algorithms to obtain a feasible integer solution.
The dynamic column insertion schema is as follows:

1. find a feasible solution to the original problem, this can be obtained by a greedy algorithm for example;

2. initialize the model taking into consideration only variables (columns) from the initial solution, obtaining the initial restricted LP problem;

3. until `(stop condition)` do:

   (a) solve the relaxation just defined;

   (b) compute all reduced costs for both columns in the model and columns that are not included in the model;

   (c) select one or more columns whose reduced cost, is negative, that means they can bring an improvement in the objective function value;

   (d) add such columns to the model;

(e) `stop condition`: there is not any column with a negative reduced cost, the last solution is optimal.

There are different variants of this algorithm, which involve different features to manage the size of the intermediate restricted models such as a limit on the number of columns inserted at each iteration, and the application of an *aging mechanism*: when a column of the model is not selected in the solution (its value is 0) for `k` (parameter) consecutive iterations, then this column is removed from the restricted model.
A dynamic column insertion algorithm for the ATFM problem has already been implemented in [18], together with a rounding algorithm to determine a good feasible integer solution.

2.2.4 Branch & Cut

Branch & Cut is an exact method to solve an ILP (or MILP/BILP) model starting from a solution of the relaxation of the model. It does so by exploring all possible solutions in a tree style. The initial problem constitutes the root, and the initial bound is set equal to the optimal solution value of the linear relaxation. Subset of solutions are generated with a *branching* operation, adding a constraint that splits the current node (called parent) in different nodes (called children); for example, in BILP models, a constraint may be based on the variable x_1, having a child contain all solutions where $x_1 = 0$ and a second child with all solutions where $x_1 = 1$.
Before actually exploring a node, a check with the current best solution is performed: if the node potentially contain a better solution (based on the bound given by the solution of the relaxation of the original problem with the additional constraints given by the node) then it is explored. If the relaxation provides a feasible integer solution better than the incumbent one, the bound is updated: all nodes whose bound is worse than the current best solution are discarded and will never be explored. This procedure is guaranteed to find the optimal solution (within the set of variables involved). The procedure also adds *cutting planes*, which are constraints that affect only fractional solutions, leaving integer solutions unchanged, thus allowing to produce bounds that are strictly closer to the best integer of solution of the current node.

2.2.5 IBM ILOG CPLEX Optimization Studio

IBM ILOG CPLEX Optimization Studio is a software, property of IBM, centered on decision optimization technology, which consists mainly in the use of optimization engines to solve mathematical programming problems. One of the most important solvers is CPLEX, that implements, among others, the simplex method (described in §2.2.1).
In particular, for this thesis, the `CPLEX Callable C API` is used (whose documentation is available at [8]), that allows to create, manipulate and solve a linear programming model inside a C++ program; specifically, the `C API` has been chosen over the `C++ API` because it does not require the instantiation of a dedicated class to work with the library, and macros to simplify the use are available in [9].

2.3 Kernel Search

Kernel Search is a matheuristics method, described in [19], to obtain heuristic solutions for MILP problems. Kernel Search is first introduced to solve the multidimensional

knapsack problem, but later contributions ([19], [16] and [15]) show that this approach is applicable also to problems that do not necessary model item choices.
The principle is to solve a relaxation of the problem, with any method, and, based on this result, to determine an initial subset of good variables, while sorting the remaining such to have the best variables in the first positions of this sort. The set obtained is then partitioned into small subsets called buckets; to obtain the solution of the problem, the solver is called multiple times on smaller problems, determined by the union of the initial subset and one bucket at the time.
The Kernel Search method, as it is described in [19], is divided into two main phases, *initialization* and *expansion*:

- initialization goal is to solve the relaxed problem, determine the initial set of good variables, called kernel, sort remaining variables and divide them into buckets;
- expansion is about creating and solving subsequent subproblems, each one formed by kernel variables and variables that belongs to the bucket i, where i is the number of the current subproblem; the first subproblem gives a solution value that is used as a feasible bound for next iterations; more specifically, each subproblem (except the first one that has no additions) has the same constraints as the original problem, with two additional constraints:

  1. at least one bucket variable must be in the basis, that means the bucket must contribute to the new solution;

  2. the objective function value must be better than (or equal to) the best objective function value found so far; in conjunction with the previous constraint, this consider solutions with the same objective function value, but with different variables, feasible, as long as the bucket participates in the swap of variables.

  Each time a subproblem has a solution, that means it is an improving solution, the bound is updated, and bucket variables involved in the solution (with a non-zero value) are added to the kernel; subproblems are solved under some kind of restrictions, such as time or CPU usage limits.

Parameters and choices to be made when implementing a Kernel Search matheuristics are the following:

- type of relaxation: can be a linear relaxation, a dynamic column insertion relaxation, a Lagrange relaxation, etc.;
- kernel: which variables form the initial kernel;
- sorting algorithm: how to determine the quality of a variable, and, based on that, perform the sorting of all variables;
- bucket size: strictly related to the subproblems size, a high value can put in the same bucket many good variables, but it will take more time to solve the related subproblem; on the opposite side, a too small value may not include enough variables to obtain a good solution, because a combination of different variables to perform a swap in the solution may be required;
- subproblems restrictions: determine under which conditions the subproblems have to work in order to find a solution;

- kernel update: which variables are added to (or removed from) the kernel at each iteration;
- stop condition: when to terminate the algorithm, for example, stop after a fixed number of iterations, after a fixed amount of time or after a better solution has not been found after a given number of consecutive iterations.

The basic Kernel Search uses as sorting method the reduced cost of variables: the lower it is, the higher is the probability it will improve the current solution; this version has many analogies with dynamic column insertion: both methods are based on an initial set of variables (the kernel and the starting feasible solution), and provide the solution to the model by solving subproblems, whose variables are determined by the reduced cost values. The main difference is given by the fact that in the Kernel Search, buckets are created in the initialization phase, and are not changed, so they are built statically. On the other hand, dynamic column insertion re-computes all reduced costs at each iteration, thus having a dynamic determination of variables to enter in the model.

2.4 Machine Learning techniques

In this section, the Machine Learning techniques used in this thesis for integration into Kernel Search are described. Basic concepts and standard functioning of such techniques are illustrated here.

2.4.1 KMeans

KMeans is a specific *clustering* algorithm, which is a Machine Learning technique, that consists in grouping a set of objects in different subsets, called clusters, each of them containing similar objects, as described in [2]. It is the most common form of *unsupervised learning*, that features the lack of pre-classified examples (differently from *supervised learning*, where pre-classified examples are used to train the algorithm). Clustering algorithm are different from each other for the type of similarity used, and for the algorithm used to build, add, and expand clusters.
KMeans is a flat clustering algorithm, meaning that the number of clusters is assigned from the beginning and never change, and it assumes examples to be classified are real-valued vectors. The algorithm starts from a random partition of objects into clusters, and, for each cluster, the algorithm aims to minimize the distance between examples and the centroid of the cluster (the center point); labels are assigned based on the distance between the example and the centroids of the current clusters. The following step consists in recalibrating centroids values using the means of examples that belongs to the same cluster as the new position; this process is repeated until centroids stabilize.

2.4.2 Decision Tree

Decision Tree is a Machine Learning technique that allows to learn functions that are representable as trees (described in [2]). It can be viewed as a series of `if-then` rules, which reduce to `if-then-else` rules in the case of a binary tree, this feature makes decision trees very understandable by a human interlocutor.
Elements composing a decision tree are the following:

- an inner node represents an attribute of any type, such as the delay of a flight of the day of the week;
- a branch indicates a possible value (or range of values) the attribute of the parent node can assume, for example `delay` $>$ `5` or `day of the week = Saturday`;
- a leaf node assigns a classification, e.g. of type `Yes/No`.

Decision Trees are included in the category of supervised learning: training is performed on pre-classified examples and, based on example labels, branch rules are created, building the actual tree.

2.5 Technological and development tools

In this section, tools used to manage the project and interact with it are described. Basic definitions are provided, together with the description of the role that each tool plays in the thesis.

JSON

JSON (JavaScript Object Notation) (logo in Figure 2.1) is an open standard file format, structured in the following way:

- the file content in included inside graph parenthesis, { };
- the content is saved as a pair `key: value`. The key is generally a string or a number, while value can be almost everything, from a single value to list of different values and types, and could also be another `key-value` pair, hence giving the option to have nested keys;
- each pair is separated by the following one with the comma (`,`) and a line break (the comma is omitted in the last value of the group).

In this thesis, this format is used to store instances for the ATFM problem, to save the result of the clustering procedure, and to track the delay value constraint for the maximization of preference version of the problem.

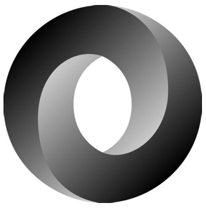

Figure 2.1: JSON logo.

C++

C++ (logo in Figure 2.2) is a programming language, documented in [6], created as an extension of the C language. It involves Object Oriented Programming features, but it is multi-purpose, because it has many different characteristics, that belongs also to the worlds of generic and functional programming. The main strengths of this language are performance and efficiency. Instead of having a garbage collector like many languages, it leaves memory management to the programmer, who has the possibility (and the goal) to optimize memory usage by allocating (and deallocating) memory at the best moment.
It is the main programming language used to develop this project, because there is a CPLEX library available for C++ (as mentioned in §2.2.5), together with many other libraries to manage different aspects of this work, such as file manipulation (input/output), JSON parsing and optimized data structures.

Figure 2.2: C++ logo.

CMake

CMake (logo in Figure 2.3) is a software available at [7] for build automation, testing, packaging and installation of software using a compiler-independent method, with few dependency. One of them consists in simply having a C++ compiler on the system; it is used mainly because it is the chosen way to build and execute the previous parts of this project, where each part has its specific CMake build directory, allowing to run single parts easily.

Figure 2.3: CMake logo.

Python

Python (logo in Figure 2.4) is an interpreted programming language, documented in [22], that features dynamic typing, resulting in a very versatile language, particularly used for scripting and for the quick development of applications, where the programmer does not need to manage every single detail like in C++, allowing to focus more on the program logic. The use of this language is greatly increased in the last years, mainly because of the huge variety of libraries available, combined with the expressiveness of its syntax; such libraries cover many different fields, and the one regarding this thesis is Machine Learning, which is covered mainly by the `scikit-learn` library (documented in [23]), together with the `numpy` library (documented in [20]) that provides improved data structures for mathematical computations.

Figure 2.4: Python logo.

Visual Studio Code

Visual Studio Code (logo in Figure 2.5) is an *IDE (Integrated Development Environment)*, which provides many tools to support software development, such as syntactic code analysis, compilation, execution, testing and debugging, that are acquired through different extensions that can be downloaded. One important extension used in this thesis is *Remote Explorer*, which allows to connect remotely via *SSH (Secure SHell)* to the machines of *Servizio di Calcolo, Torre Archimede, Dipartimento di Matematica (Università degli Studi di Padova)*, that are the computers used to execute the project. It is worth mentioning that Visual Studio Code will not work on Windows OS for this project, because for Microsoft environments the `CPLEX Callable C API` is recognized by the system only through Microsoft Visual Studio, that must be used to compile and execute the C++ parts of the project (Python parts are not using the CPLEX library so there are no restrictions).

Figure 2.5: Visual Studio Code logo.

GitHub

GitHub is an internet hosting service for software development and version control using *Git*, which is a distributed open source *VCS (Version Control System)*, used to manage software projects: GitHub is available at [14]. The environment in which each project is controlled by this system is called *repository*; it is the platform where the repository of this thesis is stored. Such repository contains also all previous work related to the ATFM problem, developed by both researchers and other students.

Chapter 3

The ATFM problem

In this chapter, the ATFM problem is described in detail, discussing its importance in the current scenario and the role of a linear programming model to solve it. We will also provide details about existing solution approaches proposed by the scientific literature.

3.1 Managing the Air Traffic Flow

The ATFM problem involves many actors, everyone providing different type of preferences and constraint when scheduling flight plans:

- institutions provide the subdivision of the air space into sectors, set their capacities, and approve flight plans;
- airlines show preferences for certain routes and time of departure and arrival, and manifest the intention of performing a flight, giving to the institutions related information;
- economical and political status of different countries can influence the choice for a flight plan, for example a country can deny a specific air company the crossing of its air space, or set an overflight tax;
- people indirectly influence flight plans, because flights are performed to satisfy their need to move, or to receive goods coming from other places of the world. An excessive delay, for example, may discourage travellers to take again the same flight, and the related air company may review its preference metrics, choosing a different flight plan.

The main actor is formed by the institutions, because they set the majority of constraints, which are hardly negotiable, forcing air companies to eventually give up the most preferred flight plan in favor of one with less preference or more delay. We consider the *pre-tactical* phase of flight plan scheduling, that happens a couple of days before the actual day of operation, considering a prediction of actual sector capacities. Other planning phases, not considered in this thesis, are the strategic one, which precedes the pre-tactical one and takes traffic demand and a rough approximation of capacities into account, and the tactical one (up to a few hours before operations), which takes into account every event that may disrupt the plan formulated in the pre-tactical phase, for example weather condition.

In this thesis, the European air space is analyzed, and the planning is viewed from the perspective of Eurocontrol, responsible of the *ECAC (European Civil Aviation Conference)* area; the database available for this project is based on Eurocontrol repositories and contains all information on flights occurred in Europe during summer 2016.

3.2 Literature review

During time, various models for the ATFM problem have been proposed: the easier ones allow to determine the optimal plan for one single airport, called *SAGHP (Single-Airport Ground-Hondilng Problem)*. Later, they have been extended such as to consider a group of airports instead, this models refer to *MAGHP (Multi-Airport Ground-Holding Problem)*, as described in [1]. Models just defined do not take into consideration the air space, and the only possible action is to delay flight departure, keeping aircraft inside the airport for more time (*ground holding*).
In 1987, Odoni [21] proposed a model that uses binary variables to track sector occupation by a specific flight at a specific time.
Five years later, in 1992, Helme formulated in [17] a model that formulates the problem as a *MMCFP (Multicommodity Minimum Cost Flow Problem)* considering, in addition to ground holding, the *airborne holding* technique: it allows to delay flight arrival by keeping the aircraft in flight near the destination airport.
In 1998, a model is proposed in [4] by Bertsimas and Patterson, which allows to change the flight speed, thus avoiding to keep the aircraft in place (ground holding and airborne holding). This is done by considering trajectories as the sequence of crossed sectors over time; it also consider sequences of two "continuous flights": so it takes into account aircraft that perform stopovers.
In 2000, the same authors presented in [5] a new model that allows to redirect flights over different trajectories when the current trajectory would cross congested regions, within the goal to minimize the total delay.
In 2011, Bertsimas, Lulli and Odoni proposed in [3] a model that combines all previous features (ground holding, airborne holding, speed control and flight rerouting) giving, within the context of flight rerouting, more alternative sectors for each flight. This combination defines a model that is flexible and expressive, allowing to obtain reasonable running times also for large-size instances.
Recently, in 2017, Fomeni, Lulli and Zografos formulated in [11] a model reguarding the optimization of *TBO (Trajectory Based Operation)*. The main feature of this approach is to consider compromises between airspace users needs, that aim to optimize their operations, and the performance optimization of the whole system. In this model, each trajectory is represented by sequences of points in four dimensions (latitude, longitude, flight level and time); trajectories are obtained by minimizing the deviation from the optimal preferred route. As said before, this model considers airspace user needs, so it takes into account preferences of involved airline companies.
A new model considering trajectories and preferences assigned to them by each flight is proposed in [10] by De Giovanni, Lancia and Lulli. This thesis makes references to such model, so its features are described later, with more details, in §3.3.2. Moreover, in §3.4, solution methods based on the model presented in [10] are illustrated accurately, as they constitute the base of this thesis.

3.3 An assignment-based linear programming formulation

We assume the following information is available:

- division of air space into sectors;
- capacity limits for each sectors;
- flights to be planned;
- scheduled time of departure;
- scheduled time of arrival;
- possible trajectories for each flight;
- possible delay;
- preferences for each trajectory (defined in §3.3.1).

All this data suggest constraints within the problem, as well as coefficients to compute the total amount of delay and/or preference of all scheduled flights. Having the pairs `(flight - flight plan)` as decision variables, we can formulate the ATFM problem using a mathematical model, specifically a linear programming model, because all constraints can be expressed as linear equalities/inequalities, with binary decision variables, hence a BILP model, since every flight plan can or cannot be assigned to a flight, having all variables in the model of boolean type, representable just by 0 or 1 values. The mathematical model that uses these definitions (presented in [10]) is described later in §3.3.2, in two different versions: the minimization of delay (that does not consider air company preferences) and the maximization of preferences.

3.3.1 Determination of preferences

The issue is given by the fact that this mathematical approach requires the preference values from air companies for each possible trajectory in the model, and most of the time the metrics used to determine a favorite flight plan are confidential. However, some information on preferences can be extracted thanks to the availability of historcal data on flown trajectories. In particular, a Dbscan clustering (presented in [10]) has been performed to all known trajectories, to group them by similarity; outliers may be caused by a special event, so they are not to be considered, and thus they are removed. This clustering gives classes that will be used by a decision tree in a further step, that takes as parameters:

- day of the week;
- day of the week with respect to the year;
- time slot;
- airline;
- airline type (low-cost or legacy);
- airplane type.

The training phase has been performed on trajectories known from historical data, then the tree classifier is applied to the clusters obtained previously, in the following way: for each leaf of the classifier the number of trajectories classified for each cluster n_i^c is computed, then these values are normalized such that each leaf contains the probabilities for a trajectory to belong to a specific cluster, $\overline{n}_i^c = \frac{n_i^c}{\sum\limits_c n_i^c}$.
Finally, the preference value for a flight f with plan p is defined as:

$$G_p^f = \overline{n}_{i(f)}^{c(p)}$$

where *c(p)* is the cluster of plan *p*, and *i(f)* is the tree classification of flight *f*.

3.3.2 Mathematical model

A linear programming model requires different elements in order to be built, the elements for the ATFM problem, used by the model proposed in [10] are the following:

Sets

- F set of flights to schedule;
- T set of natural numbers corresponding to time slots, a time slot has a fixed duration (for example, with a time slot equal to 5 minutes, the number $t = 2 \in T$ corresponds to the time slot starting 10 minutes after time 0);
- S set of air space sectors;
- *B(f)* set of possible departure delays for the flight *f*, expressed in number of time slots;
- *P(f)* set of flight plans available for flight *f*.

Parameters

- G_p^f is the preference of flight f for the plan p, it is a number in the range $[0, 1]$;
- STD^f is the time slot corresponding to the departure time of flight *f*, without considering any delay;
- STA^f is the time slot corresponding to the arrival time of flight *f*, without considering any delay;
- D_p is the flight duration of plan p, expressed in number of number slots;
- R is the maximum total delay accepted for the maximization of preferences problem;
- $C_s(t)$ is the capacity of sector s at time $t \in T$, expressed as number of flights per unit of time (commonly one hour) that enter such sector (this is the ECAC definition for capacity); if the time slot value is different from this unit, then the capacity at time t is associated with all flights within k time slots, where k $*$ *(time slot duration)* = *unit of time*, starting from time slot t (that means all time slots corresponding to the interval $[t, t + 1]$);
- $A_{ps}(t)$ is a boolean-valued parameter whose value is equal to 1 if a flight plan p $\in$ *P(f)*, $f \in F$ affects the capacity of sector $s \in S$ after $t \in T$ time slots from departure, otherwise its value is 0; practically, the value is equal to 1 if the flight plan p schedules to enter the sector s in the time slot $u \in \{t, t+1, t+2, \dots, t+k-1\}$, where k is the number of time slots contained in the capacity time interval.

Decision variables

y represent a chosen flight plan p for flight f with delay d; it is of boolean type because a flight plan is or is not assigned.

$$y^f_{pd} = \begin{cases} 1 & \text{if flight } f \in F \text{ uses the plan } p \in P(f) \text{ with a delay of } d \in B(f) \\ & \text{time slots} \\ 0 & \text{otherwise} \end{cases}$$

z represent flights with no flight plans assigned, that means such flight will be unplanned. This has a remarkable penalty on the solution, but the introduction of unplanned flights allows the use of many heuristics, that in some steps may require to have some flight unplanned in order to improve the current solution. Having all flights unplanned constitutes a starting solution, the worst possible, but it is a solution obtained with no computational cost, so the model will always have a solution.

$$z^f = \begin{cases} 1 & \text{if flight } f \text{ is unplanned} \\ 0 & \text{otherwise} \end{cases}$$

The model used to solve the minimization of delay, as presented in [10], is the following:

$$\min \sum_{f \in F} (Mz^f + max(0, STD^f + \sum_{p \in P(f)} \sum_{d \in B(f)} (d + D_p) y^f_{dp} - STA^f)) \tag{3.1}$$

$$s.t.\ z^f + \sum_{p \in P(f)} \sum_{d \in B(f)} y^f_{pd} = 1 \tag{3.2}$$

$$\sum_{f \in F} \sum_{p \in P(f)} \sum_{d \in B(f)} A_{ps}(t - STD^f - d) y^f_{pd} \leq C_s(t) \qquad \forall s \in S, t \in T \tag{3.3}$$

$$y^f_{pd} \in \{0, 1\} \qquad \forall f \in F, p \in P(f), d \in B(f) \tag{3.4}$$

$$z^f \in \{0, 1\} \qquad \forall f \in F \tag{3.5}$$

Below there are the descriptions of the model constraints:

3.2 each flight must have exactly one flight plan assigned, or it must be unplanned;

3.3 each assigned flight plan must not go above the fixed capacity of any sector;

3.4 the domain of y variables is boolean (values $\{0, 1\}$);

3.5 the domain of z variables is boolean (values $\{0, 1\}$).

For the maximization of preferences version, the objective function 3.1 is moved to the set of constraints, and the objective function is changed to maximize the sum of all preferences for chosen flight plans, with a penalty for unplanned flights, all other constraints remain unchanged, the model is the following:

$$\max \sum_{f\in F} (-Mz^f + \sum_{p\in P(f)} \sum_{d\in B(f)} G_p^f y_{dp}^f) \tag{3.6}$$

$$s.t. \sum_{f\in F} (Mz^f + max(0, STD^f + \sum_{p\in P(f)} \sum_{d\in B(f)} (d + D_p) y_{dp}^f - STA^f)) \leq R \tag{3.7}$$

$$z^f + \sum_{p\in P(f)} \sum_{d\in B(f)} y_{pd}^f = 1 \tag{3.8}$$

$$\sum_{f\in F} \sum_{p\in P(f)} \sum_{d\in B(f)} A_{ps}(t - STD^f - d) y_{pd}^f \leq C_s(t) \qquad \forall s \in S, t \in T \tag{3.9}$$

$$y_{pd}^f \in \{0, 1\} \qquad \forall f \in F, p \in P(f), d \in B(f) \tag{3.10}$$

$$z^f \in \{0, 1\} \qquad \forall f \in F \tag{3.11}$$

The constraint 3.7 fixes a limit to the total delay, otherwise the model could select the preferred route for each flight just by choosing the most preferred trajectory for each one with a different assigned delay to respect other constraints; the amount of maximum delay R can be set as a parameter, but it is typically based on the minimization of delay value.
Both models assign a penalty for unplanned flights, this penalty is equal to a constant M, whose value is:

- equal to the number of flights to schedule for the minimization of delay;
- equal to the number of flights to schedule multiplied by the maximum total delay allowed for the maximization of preferences.

The value of M is large enough to determine, just looking at the solution value, if there is any unplanned flight, because:

- for the minimization of delay, each delay number correspond to a fixed time slot, with a limit such that the delay does not go to the next day; under these conditions, a solution planning all flights has a value which is lower than the number of flights, so just one unplanned flight will make the solution having a value higher than the amount of flights to schedule;
- for the maximization of preferences, each preference is a number in the range $\{0, 1\}$, so the theoretical maximum preference is equal to the number of flights, a single unplanned flight has a higher penalty with opposite sign, so solutions with unplanned flights will be constituted by a negative number.

Practically, in our project, instead of solving the maximization of preferences, we create the model to solve the minimization of the opposite of preferences, changing sign to the solution at the end to obtain the correct result; in this way the solver always deals with a minimization problem, thus requiring one less parameter to determine the objective function verse based on the type of problem.

3.4 Methods to solve the model

Many methods have been implemented to solve the ATFM problem, formulated by the model presented in §3.3.2, in this section, two methods are presented: dynamic

column insertion and Local Search.
Dynamic column insertion (proposed in [18]) is presented because parts of its implementation, that is detailed in [18], are used for the Kernel Search, and its final results are used to compare the solution provided by the methods implemented in this thesis. In [18], this approach is called column generation, but it is a dynamic column insertion, so any further mention to this algorithm will use the proper name (dynamic column insertion).
Local Search (described in [12]) is presented because it features Machine Learning techniques that will be used in Machine Learning Kernel Search implementations: more specifically, the decision trees implemented in [12] are taken without any modifications, and adapted for use in the Kernel Search procedure.

3.4.1 Dynamic column insertion and integer-rounding heuristic

The dynamic column insertion procedure has been applied in conjunction with a rounding algorithm to solve the ATFM problem, this approach is proposed in [10] and described in detail in [18]. A general description and important parts for the purpose of this thesis are now described.
This method is used to solve to optimality the relaxation of the problem, and it is based on the general dynamic column insertion procedure (described in §2.2.3), with some modifications:

- the initial solution is given by a greedy algorithm, or it is provided by a file. This last one is generally the case for the maximization of preferences problem, where the solution of the minimization of delay is used;
- each iteration adds a maximum of 1500 column to the model, to avoid dealing with a large-sized problem that will require a lot of time to be solved, while inserting a good amount of new variables;
- after each iteration, all columns that are not in the solution for two consecutive iterations are discarded from the model, i.e. there is a mechanism of aging, with a parameter k equal to 2;
- a limit ε has been added, with a negative value close to 0, if the solution value between two consecutive iteration is lower than this limit, then the algorithm stops; this will handle possible numeral errors and it guarantees termination: in fact, after a final solution is found, there could still be columns with negative reduced costs.

After obtaining the optimal solution of the relaxed problem, a solution to the original problem can be found using the final restricted problem given by the dynamic column insertion, working on a reduced set of variables: the chosen algorithm is Branch & Cut, described in §2.2.4.
The drawback of this algorithm is that it is not guaranteed to find the optimal solution of the original integer problem on a restricted set of variables, in particular it is possible to do not even obtain a feasible integer solution. The Branch & Cut method is executed using as stop condition the limit of the gap between the current solution and the best bound available set to of $0,01\%$, in order to reduce the amount of time spent exploring nodes for a minimal improvement in the solution value.
If no solution is found with the Branch & Cut, then another heuristic algorithm is proposed in [18] to obtain a feasible solution: fractional variables are grouped by

corresponding flight, and such groups are sorted based on increasing cardinality, in order to schedule first flights with less alternatives; groups are sorted internally by value in the relaxation, if more variables have the same value, then the precedence is given to the one with the best objective value coefficient. Given the available capacity, obtained from the congestion rate of plans related to integer variables, variables in each group are tested for insertion, if it fails, then the best fractional variable is selected, with a possible violation of some capacity constraints.

3.4.2 Local Search

A second approach to solve the ATFM problem is the local search developed in [12]. *Local Search* indicates a class of algorithm aiming at improving the current solution by exploring similar solutions, called neighbours, thus avoiding to check for every possible solution; a crucial aspect of these algorithm is the neighbourhood definition, given the current solution.
In the algorithm proposed in [12], the solution space is made of models restricted to a small subset of variables, evaluated by solving the models themselves. As a consequence, the effective neighbours are obtained from the current restricted model by adding a further subset of variables that are considered "good", that means they are likely to bring an improvement in the solution. The size of such subsets is experimentally set to 10.000, value that is used in a similar heuristic applied for the ATFM problem, described in [13].
The local search scheme is integrated with Machine Learning techniques, specifically with two different decision trees, one for the minimization of delay, the other for the maximization of preferences. The goal of these decision trees is to classify all variables with respect to the solution of the current restricted model, returning the probability of each of them to be a good addition in the model. Trees have been trained on instances of reduced size, using as initial solution the greedy algorithm used also for the dynamic column insertion procedure.
The decision tree for the minimization of delay is shown in Figure 3.1. Features to determine the quality of each variable, used to produce such tree, are the following:

- delay: the delay of current variable, indicated by its third index;
- similarity: how similar is the variable to the one currently present in the solution for the same flight, similarity is computed by list of traversed sectors, in conjunction with the entry and exit times for each sector;
- congestion rate: the congestion rate of the air space, given by the insertion of the variable in exam and the removal of the one currently in the solution.

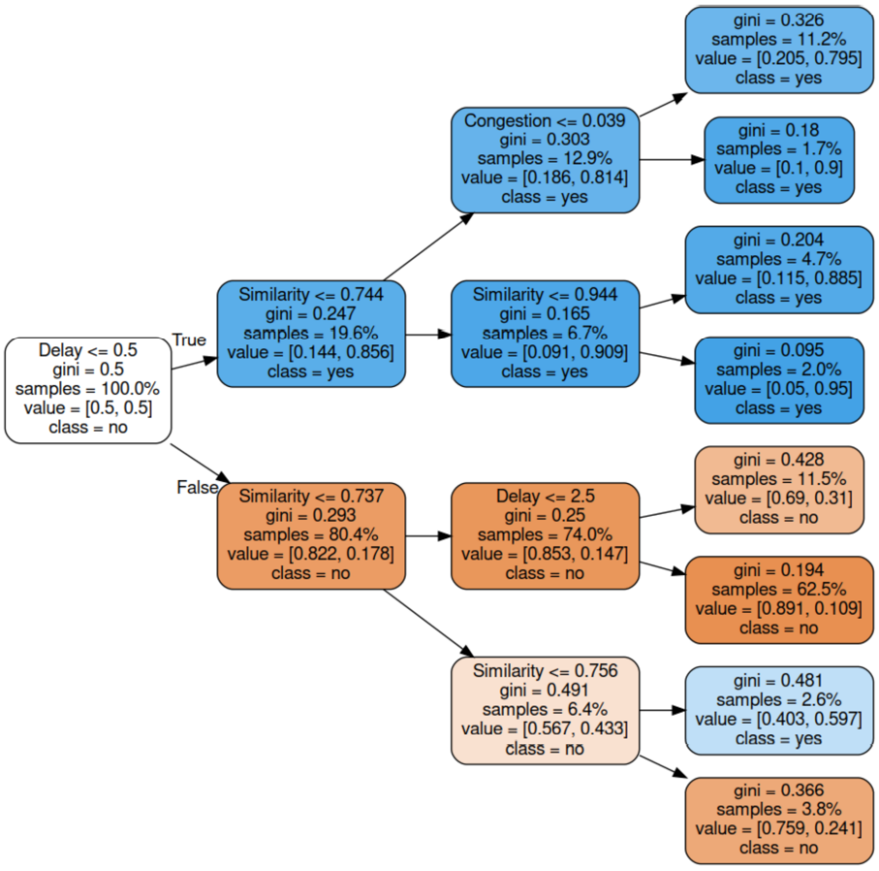

Figure 3.1: Decision Tree for the minimization of delay [12].

For the maximization of preferences, the result of minimization of delay has been used as a starting solution of dynamic column insertion, setting the maximum delay equal to the minimum delay obtained, increased by 10%.
The decision tree for the maximization of preferences is shown in Figure 3.2. Features to determine the quality of each variable, used to produce such tree, are the following:

- delay: the delay of current variable, indicated by its third index;
- similarity: how similar is the variable to the one currently present in the solution for the same flight, similarity is computed by list of traversed sectors, in conjunction with the entry and exit times for each sector;
- preference: the objective function coefficient of the variable.

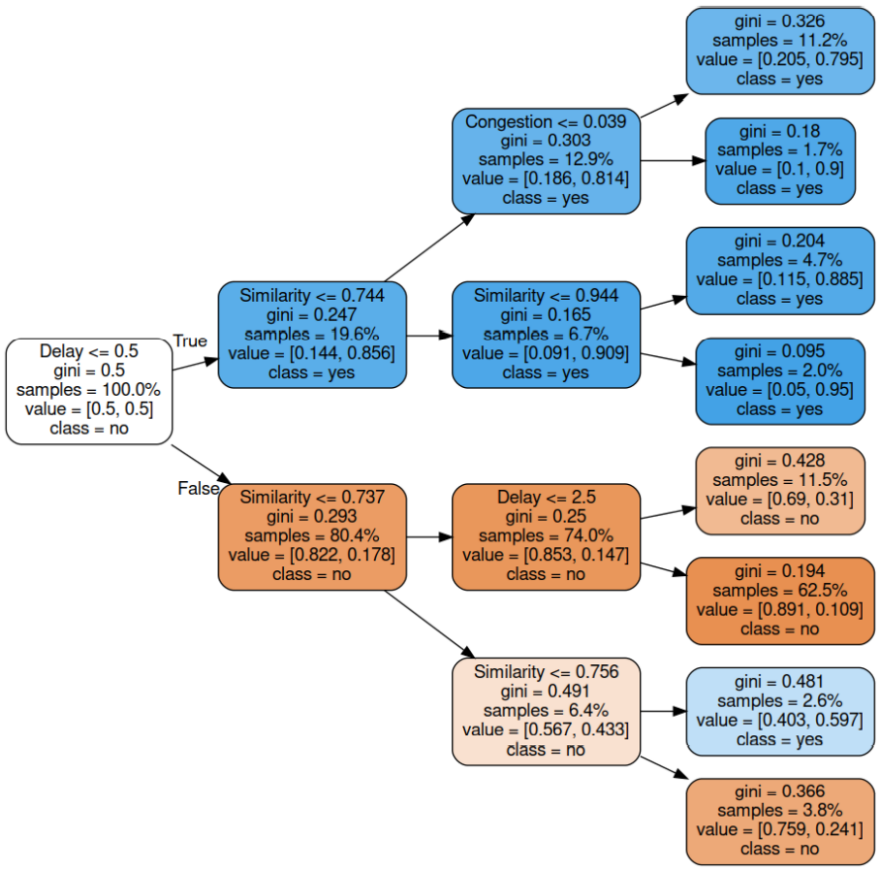

Figure 3.2: Decision Tree for the maximization of preferences [12].

More specifically, to generate a new neighbour a set of flights from which picking variables is chosen. To this end, a random flight is selected, together with interacting flights; such interaction is given by the *potential overlapping*: two flights interact with each other if the interval between planned departure and arrival times is partially shared, and if at least one pair of trajectories (one for the first flight, one for the second) shares a sector. Once the set of flights had been identified, each variable has a probability of being picked equal to its probability of being good, according to the parameter of the leaf reached after the decision tree classification.
After the set of variables has been identified, the model including current variables and such subset is solved, variables selected in the new solution (with value equal to 1) are kept, updating the current solution status, and the previous steps are repeated until a fixed number of iterations (given as parameter) is reached; the procedure proposed in [12] also features a *Simulated Annealing* scheme: a new neighbour is accepted with a probability called temperature, defined by the so-called *cooling schedule* that defines the initial temperature and how it change through iterations; also the difference of values between current and new solution is combined with the temperature to compute the actual probability to accept the incoming neighbour.
Furthermore, the last iterations feature an *intensification phase*: instead of choosing flights by potential overlapping, flights are chosen by *effective overlapping*, which

considers only trajectories present in the current solution (trajectories assigned to a flight plan); in this way the number of flights to consider is reduced, so there are more variables for each flight. This is used in the later stages of the algorithm where a good solution has already been obtained, allowing to refine it by producing some small adjustments on a limited set of interacting flights.

Chapter 4

Standard Kernel Search implementations

In this chapter, the basic Kernel Search approach proposed for the ATFM problem is discussed. In particular, we will describe the design and the implementation of the basic components, related to the solution of the problem relaxation and the basic definition of kernel and buckets. These components will be shared by further extensions of kernel search, including the implementation based on trajectory grouping, described in this chapter, and the ones using Machine Learning approaches, that will be described in the next chapter.

4.1 Applicability to the ATFM problem

The Kernel Search algorithm has been applied to the *single source capacitated facility location problem* in [15]. This problem consists in assigning each customer to a facility. A facility can serve more customers, as long as its capacity can satisfy their needs. Choosing to use a facility results in a cost to open it and the objective function minimizes the cost to open facilities while satisfying all customers. There are many analogies with the ATFM problem:

- each flight represents a customer to be served by a trajectory (plan);
- each trajectory can be used by more flights (each facility can serve more customers), as long as the capacity constraints are satisfied;
- each assignment has a cost in the objective function;
- each unplanned flight (unserved customer) has a penalty in the objective function, whose goal is to plan all flights (serve all customers) while minimizing costs (total delay), or maximizing flight preferences.

The main difference is that the capacity of a trajectory (facility) is not directly related to the trajectory itself, but to further entities, sectors, which are shared by more trajectories. Nevertheless, given the features of the ATFM problem, in comparison with the single source capacitated facility location problem, the Kernel Search approach seems to be reasonable.

4.2 Relaxation of the problem

The first step of the Kernel Search procedure is to solve a relaxation to obtain information about variables. The relaxation through dynamic column insertion (described in §3.4.1) has been chosen, for the following reasons:

1. it is a fast enough procedure that finds the optimal relaxed solution. In our implementation, it takes less than 25 minutes, with the exception of the two critical instances represented by flight planning on 27^{th} and 28^{th} August;

2. using the same relaxation method allows to perform a direct comparison between Kernel Search and the rounding algorithm used in [18] (and mentioned in §3.4.1), because the starting point is the same;

3. there is no need to implement a new relaxation method, as adapting dynamic column insertion to the Kernel Search will require minor adjustments;

4. the linear relaxation, which is the standard choice for the Kernel Search, cannot be directly solved with off-the-shelf solvers, because these solvers would use the model with all variables, changing only their domain from boolean to real; there are different millions of variables in the complete instances, and solving a complete relaxation to optimality would require too much time.

The existing dynamic column insertion approach proposed and implemented in [10] and [18] has been adapted easily to be used by the Kernel Search, the following modifications has been adopted:

- variables that compose the initial solution are saved in a map, using their name as keys; the map exists only to check if a variable belong to the initial solution, and the access to a map using a known key takes constant time;

- the procedure has two further maps to track variables in the model (described in §4.4.2), each time the model is modified by inserting or removing variables, the maps are updated;

- all names of variables that constitute the initial kernel are saves in a list;

- all remaining variables are inserted in list as instances of the class `Variable` (described in §4.4.2), in order to allow sorting;

- the rounding procedure described in §3.4.1 has been cut off because of no use for the Kernel Search.

4.3 Kernel Search basic elements

This section provides the description of all elements required to define the Kernel Search algorithm, as it is described in §2.3. Parameters are discussed, as well as any choice that has been made, making reference also to provisional implementations and parameter values (obtained after calibration through test on some problem instances).

4.3.1 Kernel definition

Following the Kernel Search basic schema described in [19], the kernel is initialized with all selected variables in the relaxation. The number of such variables is on average equal to the number of flights to plan plus 800, which corresponds to the number of fractional variables. In the first implementation this was the kernel definition, but this subset of variables does not guarantee to produce an integer feasible solution, resulting in the first subproblem to have no solution. The first adopted adjustment is to add z variables, which model unplanned flights, that are related to flights having variables with fractional values; this set guarantees a feasible solution, because the solution formed by integer variables in the relaxation and z variables for fractional variables always exists.
A better kernel initialization would use variables leading to a model that does not need unplanned flights to be feasible, so to always start with a good integer solution. The idea to find such subset is already implemented in the dynamic column insertion algorithm of [10] and [18], because it uses as starting point an integer solution which is provided as input or created using a greedy algorithm. Since there is no guarantee it contains a good subset of variables, that are more likely to be selected by solutions close to the optimal one, this group of variables is added as an extension of the original kernel formed only by positive variables of the relaxation: note that in this definition there is not any z variable, unless it is present in the starting or in the final solution of the dynamic column insertion.
Using this approach, the size of the kernel is up to 60.000-70.000 variables, and this size is considered too big, even if very small compared to the original problem that contains millions of variables. In fact, preliminary computational experience set the desired size of kernel and bucket below 50.000, in order to solve subproblems able to find the optimal solution in a small amount of time. Indeed, the set of positive variables and the set of variables that belong to the initial solution share many elements. In fact, with relation to tested instances, on average, the number of variables that belongs only to the initial solution is:

- 7.400 for minimization of delay and for maximization of preferences only when the starting solution is given by input, that corresponds to the integer solution of minimization of delay obtained using the rounding algorithm;
- 8.300 for maximization of preferences when a starting solution is not provided, but it is generated using the greedy algorithm; specifically, on most instances this number is pretty much equal to the previous case, but for others, like the 27th and 28th August instances, it goes up to 10.000.

In the worst case scenario, the kernel size is around 45.000, on average it is 40.000, which is below the maximum size established to have a limited subproblem size, so this kernel definition has been implemented.
Regardless of the Kernel Search implementation, the kernel has been built in the same way, in particular, with the same set of variables using the same sorting criterion: variables are sorted based on their features in the solution of the relaxation, first by decreasing value, and then by increasing reduced cost for variables with the same value. This choice has been made to uniform the first subproblem execution, because a different order will result in a different execution time, and may lead to different results in case the respective problem goes on timeout; the current kernel internal sorting is not guaranteed to be the best sorting in any way, it is just to obtain coherent results between different implementations.

4.3.2 Bucket size

Following the basic Kernel Search procedure, the suggested bucket size is the kernel size, so it corresponds to the number of positive variables in the relaxation, for a total subproblem size of approximately twice the number of flights to plan for the relative instance. Each instance contains 28.000-32.000 flights, so the subproblem size would be 56.000-64.000, that goes over the desired size, without even considering the kernel extension that includes also variables from the dynamic column insertion starting solution.
Before excluding any *a priori* size, the Kernel Search has been implemented to take the bucket size as parameter, having a special value that indicates the size to be equal to the kernel size, in order to easily perform executions on different sizes. The following sizes have been tested on all implementations of the Kernel Search algorithm (described in next sections):

- 10.000;
- 20.000;
- number of positive variables in the relaxation.

Results show, with few exceptions for the maximization of preferences, that the reduction of bucket size does not affect negatively the objective function value; as for the exceptions, the difference in value is in the order of 10^{-3}, so we decided to consider a bucket size of 10.000.

4.3.3 Subproblems creation and execution

After building the kernel and the buckets, the initialization phase of Kernel Search procedure is finished, and the expansion phase starts. Before initializing any subproblem, the empty model is created with all constraints, so there is no need to create a new problem for each subproblem, but it is sufficient to modify the set of variables included in such model.
The first subproblem is formed only by kernel variables, and does not contain any additional constraint. The goal of this subproblem is to give an initial solution, whose value will be the starting upper bound for next iterations. For this reason, the kernel subproblem is the most important among all subproblems to solve, and so it is reasonable to execute it with less restrictions compared to other subproblems. The chosen restriction is only about execution time and different timeouts have been tested:

- 10 minutes;
- 20 minutes;
- 30 minutes;
- 1 hour (this has been tried only for 27^{th} and 28^{th} August instances).

Timeouts below 30 minutes are enough for most instances to obtain the optimal solution in the minimization of delay. For the maximization of preferences version of the problem, instead, many instances do not return a feasible solution for lower timeouts: in 30 minutes these instances does not find the optimal solution, but they still provide a good value, so this is the chosen timeout, because it represents a good compromise between result and execution time.

After the kernel subproblem is solved, the kernel is updated (details are in §4.3.4), by eventually removing variables from both the model and the kernel. Constraints of the problem and remaining variables are not modified in this process, allowing bucket subproblems to be created easily. The bucket subproblems are created and run following the bucket order:

1. variables that belong to bucket `i` are inserted in the model;
2. an additional constraint is added to the model: the objective function value must be greater than (or equal to) the best solution value obtained so far (initially it is the kernel solution value);
3. an additional constraint is added to the model: at least one bucket variable must appear in the solution (with a value equal to 1).

With the two additional constraints, the feasible solution of each subproblem can be of two types:

1. a solution with a better objective function value, which is an effective improvement;
2. a solution with the same objective function value, but involving different variables.

Infeasibility is modelled with a special solution value:

- -1 for the minimization of delay, because such problem does not have any negative objective function coefficient, so negative solutions do not exist;
- $+1$ for the minimization of the opposite of preferences (the transformation of the maximization of preferences), because any solution that plans all flights has a negative objective function value, while a solution that features unplanned flights has a positive value in the order of millions of units, so small positive solutions do not exist.

Subproblems are run under a time constraint and different timeouts have been tested:

- 5 minutes;
- 10 minutes;
- 15 minutes;
- 20 minutes;
- 30 minutes.

Results show that the first couple of subproblems takes more time to find the optimal solution, while later iterations take only few seconds: in order to find a balance between the total execution time and the quality of the result, the timeout for subproblem has been fixed to 10 minutes, because additional time is giving minor improvement. More specifically, the improvement is in the order of 10^{-3} for the maximization of preferences, while it do not affect the minimization of delay executions, with the exception of 27^{th} and 28^{th} August instances.

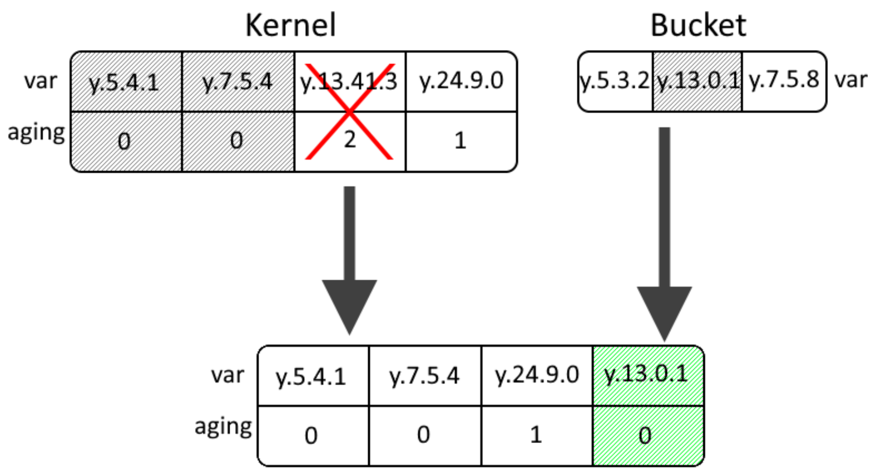

Figure 4.1: Kernel update after iteration with a bucket.

4.3.4 Kernel update

After each iteration, the kernel is updated in order to include new variables and prepare all structures for the next iteration; furthermore, a mechanism of aging is introduced to remove unused variables and keep the kernel within a reasonable size. The complete procedure is the following:

1. reset the aging counter for all kernel variables that appear in the solution;
2. add to the kernel all bucket variables that appear in the solution;
3. increase the aging counter by 1 for all other kernel variables;
4. remove from the model all bucket variables with a value equal to 0 in the solution;
5. remove kernel variables with the aging value higher then the fixed limit from both the model and the kernel;
6. remove the two specific subproblem constraints.

In all implementations, the considered value of aging is equal to 2 for y variables and it is equal to 1 for z variables: the value 2 has been indicated by [16] as a good value to keep the kernel size limited while not immediately discarding variables that may reappear in future solutions; z variables, on the other hand, are removed instantly when they are not used, because they represent unplanned flights, with a huge penalty in the objective function, so, once a flight is planned, it is very unlikely to return in an unplanned state in order to obtain a better solution.
Each step that modifies variables in the model also updates the related structures, that keeps track of variable names and indexes in the model (described in §4.4.2).

Figure 4.1 illustrates the kernel update procedure after solving a subproblem formed by the kernel and a bucket: variables with cells in grey appear in the solution with a

positive value, i.e. such variables contribute to the last solution, which is improved with respect to the previous one (or it has the same value). The aging counter for variables in the current solution is set to 0, according to the definition of this procedure. Bucket variables that appear in the solution (in Figure 4.1 it is the case of variable $y.13.0.1$) are inserted into the kernel with aging counter initialized at 0: inserted variables are highlighted as a green cell in Figure 4.1.
Kernel variables that are not selected in the current solution, represented in Figure 4.1 with a white cell, have the respective aging counter increased by 1. Variables with the counter equal to 2 are removed from the kernel: in Figure 4.1 removed variables are marked with a red cross.
If the subproblem does not find a feasible solution, then the kernel remains unchanged, and bucket variables are simply removed from the model, together with the specific subproblem constraints.

4.3.5 Stop conditions

There can be three different stop conditions for the Kernel Search procedure:

1. a total timeout of the procedure is reached;

2. a fixed number of consecutive non-improving iteration is observed;

3. a fixed number of iteration is performed.

In the implementations for the ATFM problem, the following stop conditions have been set:

1. total timeout of 2 hours for the expansion phase, so the time starts when the kernel subproblem is created;

2. after an unfeasible solution is returned from a subproblem. According to the principle of Kernel Search, variables are sorted such as to have the best ones in the first buckets, so, once a bucket does not improve a solution, it is very unlikely that a bucket with "worse" variables can bring a better result;

3. no limit is set to the number of iterations, so, if the procedure keeps improving and does not reach the timeout, it can examine all buckets, but such limit can be provided as a parameter.

The current stop conditions have been set in order to limit the duration of execution, because without any stop condition in some instances the procedure runs for around 20 hours, mostly on non-improving or unfeasible subproblems.

4.4 Standard Kernel Search implementation

The standard Kernel Search implementation is realized completely with the C++ programming language, according to the basic elements described above, using existing code where possible. All structures and functions are briefly described, giving information about their behaviour, while also pointing which elements are reused and how they are adapted to the Kernel Search procedure.

4.4.1 Interaction with CPLEX

The `CPLEX Callable API` (documented in [8]) are used through a set of macros defined by other people working at this project in the past years, available in [9], the most used macros are:

- `DECL_ENV(env)` initializes a CPLEX environment and assigns it to the variable `env`;
- `DECL_PROB(env, prob)` initializes an empty CPLEX problem in the environment `env` and assigns it to the variable `prob`;
- `CHECKED_CPX_CALL(functionName, env, funcArgs)`: runs the CPLEX function `functionName` with parameters `funcArgs`, in the environment `env`. This is generally to be used for functions that returns the effective result by side effect, modifying a reference of a given parameter; the actual value returned by these functions is a status code that indicates eventual errors, this macro automatically checks the status code for errors (it does not handle them).

For all functions that do not return a status code, but instead return the proper result, the macro `CHECKED_CPX_CALL` has not been used, but the function is called directly, and its result is assigned to a variable.

4.4.2 Classes and auxiliary elements

In order to better manage the information required by the Kernel Search procedure, the class `Variable` has been created. It represents, as the name indicates, a variable of the model, it stores its name, value in the relaxation, and reduced cost in the relaxation. The standard *OOP (Object Oriented Programming)* conventions have been adopted, so:

- attributes are declared as private;
- `get` methods allow to read an attribute;
- `set` methods allow to modify an attribute;
- all read-only methods are marked with `const`.

The comparison operators have been defined, in order to easily implement the sorting algorithm:

- two variables are equal if all their attributes are equal;
- given two variables *var1* and *var2*, *var1* $\geq$ *var2* if the value of *var1* is greater than the value of *var2*; in case such value is equal, then *var1* $\geq$ *var2* if the reduced cost of *var1* is lower than (or equal to) the reduced cost of *var2*;
- all other comparison operators ($<$, $>$, $\leq$) have been redefined according to the definition above.

This class has a static method, `sortVariables(vector<Variable>& vars)`, that implements the merge sort algorithm for a list of variables, using as comparison operator $\geq$, so the resulting sorted list contains all variables, in the following order:

1. variables with value equal to 1 in the relaxation;

2. variables with fractional value in the relaxation, sorted by decreasing value;
3. variables with value equal to 0 in the relaxation, sorted by increasing reduced cost.

The kernel is implemented as an unordered map that maps each kernel variable to its aging value. Since for kernel variables only the name is required, only this information is saved, so kernel elements are pairs `(varName, aging)`; if a variable name does not exist in this map, then it is not a kernel variable (or it was a kernel variable but at the current state it was removed).
In order to manage the kernel update schema described in §4.3.4, an auxiliary element `varToIndex` has been added: it is an unordered map that connects each variable name to the corresponding index in the current CPLEX problem, the index is -1, or the map entry is missing, for variables that do not belong to the current model; this map is required to connect each kernel variable to the corresponding position in the CPLEX model (recall that the kernel is always part of the subproblem, so its variables are always in the model), this index is used to remove the correct kernel variable from the model according to the aging parameter.
Also the reverse map is implemented, `indexToVar`: this is an ordered map that connects each index of the current model to the corresponding variable name. If an index does not exist, or it maps a variable named `"NULL"`, then this is not a valid index of the model. This map is required to connect results of CPLEX functions to the corresponding variables, for example `CPXgetx` returns all variable values as a list, ordered by index in the model, and `CPXdelsetcols` removes a set of variables from the model, returning a list containing the new index of each variable, sorted by old index value.
Variables that are not in the initial kernel are implemented as a single list of `Variable` objects, once they are sorted the corresponding information about value and reduced cost is not used anymore, so the type is changed such to have a list of variable names, in order to reduce the amount of memory occupied by data.

4.4.3 Functions

In order to create subproblems, the corresponding variables must be added to the current model. This is done by modifying the existing functions that add all y variables and all z variables:

- `addSingleYVar` adds a variable modelling a flight plan with the provided name to the model. It requires the variable name, the CPLEX problem and environment, the two tracking maps, and all information about the problem (problem type, maximum delay, objective function and constraint coefficients, constraints involved);
- `addSingleZVar` adds a variable modelling an unplanned flight with the provided name to the model. It requires the variable name, the CPLEX problem and environment, the two tracking maps, and all information about the problem (problem type, maximum delay, objective function and constraint coefficients, constraints involved);
- the two functions above are used by the function `addColumn` that, performing a case distinction on the first letter of the variable name (y or z), calls the right function.

The following functions constitute the implementation of the standard Kernel Search:

- `initFromRelax` checks if the relaxation of current instance has already been performed. In this case, this function reads the corresponding files to initialize the kernel and the variable list, and performs the sorting of variables;
- `columnGen` performs the dynamic column insertion relaxation. It takes, in addition to all parameters required by the proper procedure, the two tracking maps, a reference to a list in which saving kernel variables, and two references to save the execution time and the objective function value. It returns the list of all variables not added in the kernel.
- `saveVariableInfo` saves the list of non-kernel variables into a file, in order to avoid repeating the relaxation. This file is used on repeated runs of the same instance, and for other Kernel Search versions that implements a different sorting algorithm;
- `createBuckets` initializes the kernel with all corresponding variables and assigns to each one the aging value 0, and creates all buckets by dividing the sorted variable list into lists of size 10.000 (the chosen bucket size);
- `createKernelProblem` uses the existing functions to create all constraints of the model, and adds to it all kernel variables;
- `solveSubProblem` solves a subproblem within the given timeout (30 minutes for the kernel problem, 10 minutes for all other subproblems), returning the objective function value and the execution time;
- `createSubProblem` adds all variables that belong to the upcoming bucket to the model, and create the two specific subproblem constraints;
- `updateKernel` extracts information about the obtained solution (this function is called only if a feasible solution is found) and, based on that, resets the aging counters for variables appearing in the solution, increases the counter by 1 for variables with value equal to 0, adds to the kernel bucket variables that appear in the solution, removes from the kernel and from the model variables that reached the aging limit, and removes from the model all bucket variables that do not contribute to the solution;
- `removeAllBucketVars` removes all bucket variables from the model, this function is called instead of `updateKernel` in case of non-feasible solutions, eventually allowing to continue with the procedure (it is not the case of the current stop condition);
- `removeBucketConstraint` removes the two subproblem specific constraints 1 and 2. They are always removed, and they will be replaced in the next iteration with new constraints;
- `saveToFile` saves all results into different files, allowing to store data in organized *csv (Comma-Separated Values)* tables.

4.4.4 Algorithm

The standard Kernel Search algorithm implemented is here provided, divided in its two phases (expressed as pseudo-code that can be matched easily with functions defined in §4.4.3).

Parameters

- `instanceName`: the path to the specific instance file;
- `maxDelay`: indicates the maximum delay allowed, a negative number indicates a minimization of delay problem, otherwise this value will be used for the specific maximization of preferences constraint;
- `problemParams`: includes different parameters used for the creation of the problem, such as the unique max delay and the unique delay step (it is grouped here to avoid having a list of parameters too long for functions);
- `CplexVars`: includes the CPLEX variables regarding the environment and the problem;
- `maps`: includes the two maps `indexToVar` and `varToIndex`, initialized as empty;
- `bucketSize`: the size of each bucket. As discussed in §4.3.2, it has a value equal to 10.000;
- `aging`: determines how many iterations unused variables remain in the kernel, it has a value equal to 2;
- `subProbTimeout`: the timeout assigned to each subproblem (excluding the first kernel subproblem), its value is 600 seconds, that corresponds to 10 minutes.

The initialization phase is defined as following:

```
begin
  if a relaxation has already been executed then
    variables, kernelVarNames = initFromRelax(instanceName,
        maxDelay)
  else
    variables, kernelVarNames = columnGen(instanceName, maxDelay,
         problemParams, maps)
    save kernel variables into file
    sortVariables(variables)
    saveVariableInfo(instanceName, maxDelay, variables)
end
```

The expansion phase is defined as following:

```
begin
  kernel, buckets = createBuckets(variables, kernelVarNames,
      buckSize)
  createKernelProblem(CplexVars, kernel, maxDelay, problemParams,
       maps)
  bestSol = solveSubProblem(CplexVars, 1800) // 30 minutes,
      timeout for kernel problem
  if bestSol == "unfeasible solution" then
    bestSol = +∞
  counter = 0
  repeat
    currentBucket = buckets[counter]
    createSubProblem(CplexVars, bestSol, currentBucket, maps)
    newSol = solveSubProblem(CplexVars, subProbTimeout)
    if newSol <= bestSol then
      bestSol = newSol
      updateKernel(CplexVars, kernel, currentBucket, maps, aging)
    else
      removeAllBucketVars(CplexVars, currentBucket)
    removeBucketConstraint(CplexVars)
    counter = counter + 1
  until (stop condition)
  saveToFile(instanceName, maxDelay, bestSol)
end
```

4.5 Kernel Search with trajectory groups

The trajectory groups Kernel Search is a different version of the procedure, that uses all functions of the standard algorithm defined previously, with the exception of the sorting criterion, so it defines buckets differently from the canonical sorting of variables. Variable ordering is defined in this way:

1. all trajectories are divided into groups, and each group contains trajectories that refers to flights with the same pair *"origin-destination"*;

2. inside every group, each trajectory is replaced by the list of variables that refers to such trajectory: for example the trajectory 24 will be replaced by [y.2_24_0, y.5_24_5, y.6_24_1, . . .];

3. variables in each group are sorted internally by increasing reduced cost (the same order used by the standard implementation, but applied to single groups separately). Value is not considered because all variables involved in this stage have a value equal to 0;

4. groups are sorted based on the average reduced cost of their first 6 elements (6 is a number arbitrary fixed based on the average number of variables contained in each group);

5. buckets are built one at the time using a round robin schema on groups: the first variable is the best variable of the first group (that will be removed from the group), the second one is the best variable of the second group, and so on. Once the last group is reached, the procedure starts again from the first group, until the bucket has been built;

6. buckets are built until there are variables available to be picked.

This order makes good flight plans for each *"origin-destination"* pair available in the first buckets, differently from the standard implementation that could have good flight plans regarding only few *"origin-destination"* pairs, because the number of such plans can be similar to a complete bucket size. Instead, the proposed method guaantees that more fligths, related to several differentiated *"origin-destination"* pairs, can be potentially improved, even if the expected improvement per flight may be smaller.
Trajectory groups sorting has been implemented using the Python programming language, because it shares many steps with the version of Kernel Search that involves clustering (see §5.1), and the ordering inside each group of variables is implemented

using the Python built-in function for list, `list.sort()`, applied to the list of variables, in order to do so the class `Variable` (described in §4.4.2) has been implemented also in Python, with the same exact attributes and methods. Communication with the C++ program is made through files: the C++ part of the initialization phase provides the variables file, the Python program reads it, performs the sorting, and saves the new order on another file, that will be read by the C++ part of the expansion phase.

4.5.1 Functions

Since the entire implementation of the trajectory groups sorting is made with Python, the only C++ function added is `initFromGroups`, it initializes the list of variables using the Python output file (exactly like `initFromRelax`, but it opens a different file). The following functions has been realized in Python for the trajectory groups sorting implementation:

- `getVariables` reads the variables file that corresponds to the current instance and type of problem (minimization of delay or maximization of preferences), saving all variables into a list of `Variable`;
- `commonTrajectories` divides all trajectories into groups based on common flights: if two trajectories are used by the same flights, then they share the same *"origin-destination"* pair;
- `connectTrajToVars` creates a dictionary that maps each trajectory to the list of variables that use it (that have such trajectory as their second index);
- `convertToVars` transforms a list of trajectories into a list of variables, by substituting each trajectory with the list of variables that use it;
- `sortGroups` performs variable sorting within each group;
- `prioritizeGroups` sorts all groups, putting in first position groups with the best (lowest) average of reduced cost, computed on the first 6 variables of each group;
- `createBuckets` performs the round robin procedure to create a list of list of variable names, each list is a bucket;
- `saveVariables` saves the list of buckets in a file.

4.5.2 Algorithm

The trajectory groups Kernel Search algorithm implemented is here provided, divided in its two phases (expressed as pseudo-code that can be matched easily with functions defined in §4.5.1).

Parameters

- `instanceName`: the path to the specific instance file;
- `maxDelay`: indicates the maximum delay allowed, a negative number indicates a minimization of delay problem, otherwise this value will be used for the specific maximization of preferences constraint;

- `problemParams`: includes different parameters used for the creation of the problem, such as the unique max delay and the unique delay step (it is grouped here to avoid having a list of parameters too long for functions);
- `CplexVars`: includes the CPLEX variables regarding the environment and the problem;
- `maps`: includes the two maps `indexToVar` and `varToIndex`, initialized as empty;
- `bucketSize`: the size of each bucket, it has a value equal to 10.000;
- `aging`: determines how many iterations unused variables remain in the kernel, it has a value equal to 2;
- `subProbTimeout`: the timeout assigned to each subproblem (excluding the first kernel subproblem), its value is 600 seconds, that corresponds to 10 minutes.

The initialization phase is identical to the standard implementation (see §4.4.4), so its pseudo-code is omitted.
After this phase, the execution flow passes control to a Python program, which performs variable ordering (the outer repeat-until loop is a detail about the implementation of `createBuckets`):

```
begin
  trajGroups = commonTrajectories(instanceName)
  variables = getVariables(instanceName, maxDelay)
  trajToVar = connectTrajToVars(variables, instanceName)
  varGroups = convertToVars(trajGroups, trajToVar)
  varGroups = sortGroups(varGroups)
  varGroups = prioritizeGroups(varGroups)
// Start of createBuckets pseudo-code
  allBuckets = ∅
  repeat
    currentGroup = 0, insertedVars = 0
    currentBucket = ∅
    repeat
      if size(varGroups[currentGroup]) != 0 then
        // pop() removes the first element of the list returning
           it
        currentBucket = currentBucket ∪
           varGroups[currentGroup].pop()
        insertedVars = insertedVars + 1
        currentGroup = next group index (the first one if this is
            the last)
      else // empty groups are removed
        remove varGroups[currentGroup]
    until (insertedVars == bucketSize or varGroups == ∅)
    allBuckets = allBuckets ∪ currentBucket
  until (varGroups == ∅)
// End of createBuckets pseudo-code
  saveVariables(allBuckets, instanceName, maxDelay)
end
```

The expansion phase is defined as follows (differences with the standard implementation are highlighted in red):

```
begin
  kernelVarNames = names contained in the kernel file
  variables = initFromGroups(instanceName, maxDelay)
  kernel, buckets = createBuckets(variables, kernelVarNames,
      buckSize)
  createKernelProblem(CplexVars, kernel, maxDelay, problemParams,
       maps)
  bestSol = solveSubProblem(CplexVars, 1800) // 30 minutes,
      timeout for kernel problem
  if bestSol == "unfeasible solution" then
    bestSol = +∞
  counter = 0
  repeat
    currentBucket = buckets[counter]
    createSubProblem(CplexVars, bestSol, currentBucket, maps)
    newSol = solveSubProblem(CplexVars, subProbTimeout)
    if newSol <= bestSol then
      bestSol = newSol
      updateKernel(CplexVars, kernel, currentBucket, maps, aging)
    else
      removeAllBucketVars(CplexVars, currentBucket)
      removeBucketConstraint(CplexVars)
      counter = counter + 1
  until (stop condition)
  saveToFile(instanceName, maxDelay, bestSol)
end
```

Chapter 5

Kernel Search using Machine Learning

In this chapter, the proposed integration of Machine Learning techniques to boost the Kernel Search procedure is described in detail, with focus on both theoretical idea and practical implementation.

5.1 Integrating clustering into Kernel Search

Clustering is applied to objects with the goal of dividing them into different groups, where each group contains similar objects. Different types of clustering are already implemented and executed on the procedure that generates instance files for the ATFM problem proposed and described in [10], [12] and [18]. KMeans (see section §2.4.1) is chosen because it is simple, and it perfectly fits the type of data involved in the procedure.

5.1.1 Idea and application

The ideal data type for the KMeans algorithm is formed by multi-dimensional vectors, because clustering labels are assigned based on the distance between centroid vectors and each object vector, recomputing centroids after each iteration, until they stabilize. The idea is to use clustering to group similar trajectories together, and to create buckets using trajectories from different groups, but in the current state trajectories are simply an integer number, representing their corresponding identifiers. The procedure is implemented using the Python programming language, using the library `scikit-learn`, available at [23], that provides all procedures and structures to operate with Machine Learning techniques.
A trajectory is described with a list of triplets `(sectorID, entryTime, exitTime)` that indicates crossed sectors, with the number of time slots giving information about both geographical and temporal position of such trajectory. Trajectories are transformed into matrices of `numSectors` rows and `totalTimeSlots` columns (`numSectors` is the number of overall sectors used by the instance, `totalTimeSlots` is the total number of time slots considered for the current instance), where:

$$cell(s,t) = \begin{cases} 1 & \text{if the trajectory is inside sector } \texttt{s} \text{ at time slot } \texttt{t} \\ 0 & \text{otherwise} \end{cases}$$

This is a sparse matrix, where the number of non-zero values is equal to the number of time slots used by the current trajectory, that is sensibly lower than the total number of time slots available. It is important mentioning that all trajectory descriptions are normalized on time, such that the starting time slot is 0 for every trajectory, in order not to have duplicates just because the starting times are different.
The existing clustering has not been executed globally for all trajectories, but multiple clusterings are run on each *"origin-destination"* group. The trajectory groups implementation (§4.5) gives exactly the same subdivision, but the number of trajectories involved is much lower, because the previous clustering had the goal of grouping all similar trajectories into a single one, represented by the centroid of each cluster, that are the trajectories used by instances; this fact forces to choose a different number of clusters to obtain a good separation: if the number is greater than the amount of trajectories involved, each trajectory will have its own cluster; on the other hand, a too small number will group different trajectories into the same label. The number of clusters established is $nClusters = \lfloor\sqrt{\frac{number\ of\ trajectories}{2}}\rfloor$, which is the value suggested in [18] and [10], but note that the actual number of clusters will be different because the number of clusters is a parameter that depends on the number of trajectories. Moreover, the KMeans parameters used are the default parameters for KMeans in the `scikit-learn` library.
Clustering is executed once per instance, and the results are saved in a JSON file, in order to avoid repeating it for repeated executions of the same instance, or for execution of a different solution method, even because the clustering is independent from the type of the problem (minimization of delay or maximization of preferences). After the clustering has been performed for each *"origin-destination"* group, trajectories will be identified by:

- *identifier*: the number of the trajectory as in the instance file;
- *group*: the identifier of the *"origin-destination"* group in which the trajectory is included;
- *label*: the specific cluster in which the trajectory is included.

Labels have value from 0 to *nClusters[group]*, so the group information is necessary to distinguish between two trajectories that belongs to different groups, but with the same label.
Once the clustering procedure terminates, trajectories are replaced by corresponding variables in the model, using the same function of the trajectory groups implementation (`connectTrajToVars`), but keeping label information that will be copied on each variable; then groups are sorted internally by increasing reduced cost, and externally by the average reduced cost on the first 6 elements. Finally, the round robin procedure is started, with a modification: if a variable with the same *"origin-destination"* and label is already inserted in the current bucket, then this variable is skipped, and will be inserted in the next bucket (with the exception given by the scenario where no other variable can be inserted, then this check is ignored, which specifically occurs for the last buckets).
Notice the similarity with the trajectory groups implementation: in fact, the groups

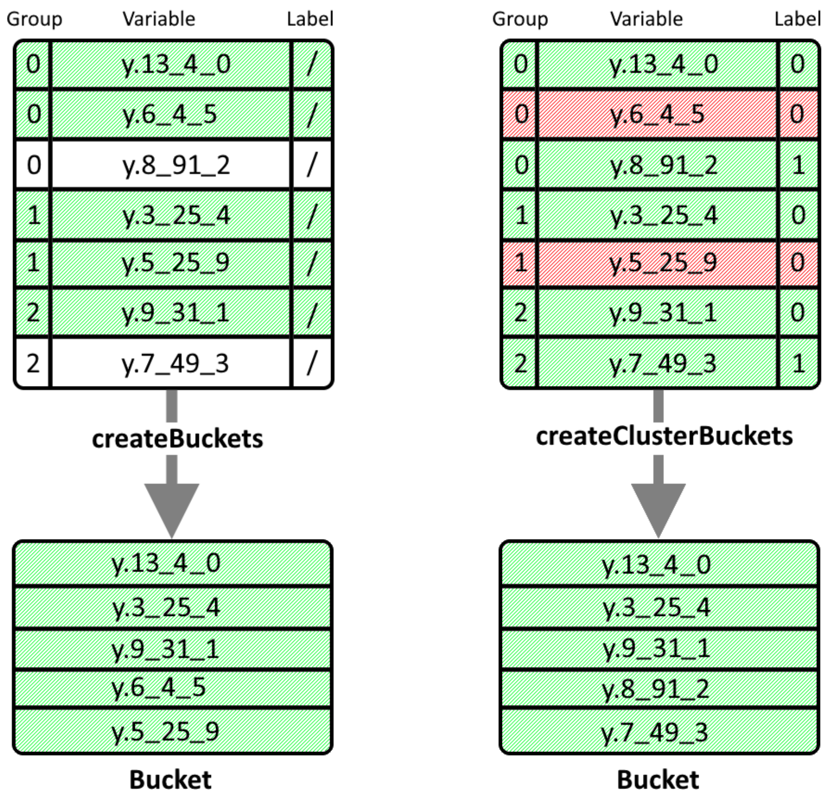

Figure 5.1: Difference between trajectory groups (left) and clustering (right) bucket creation.

identified by the trajectory groups procedure (§4.5) can be viewed as a particular case of the clustering implementation where each trajectory has a different cluster label, so every trajectory is different from all others, and the check does not prevent any variable from being picked. This difference is illustrated in Figure 5.1 that shows the creation of a bucket from the list of variables sorted according to the *"origin-destination"* group and to their respective reduced cost: in green picked variables, in red variables that are discarded because of the label check.

Notice that in both cases variables inserted in the first round are identical, because there are not other variables of the same group already in the bucket, so the check returns true. On the other hand, the check starts filtering variables after the first iteration, and discards variables that share a similar trajectory.

5.1.2 Classes and auxiliary elements

In order to manage the extra information of labels, the class `ClusterVarResult` has been implemented, it has the following attributes:

- `originDest`: the identifier of the corresponding *"origin-destination"* group;
- `variable`: the current variable, of type `Variable` (defined in §4.4.2);
- `label`: integer representing the cluster label of the trajectory used by the variable.

The comparison operators have been implemented in order to consider only the `variable` attribute, so to have compatibility with the sorting algorithms already implemented; this specific implementation completely adapt `ClusterVarResult` to existing functions, that now will be called on list of `ClusterVarResult` instead of list `Variable`, so no other auxiliary element is required.

5.1.3 Functions

The following functions have been defined for the clustering implementation:

- `getTrajectories` returns a list of matrices containing trajectory descriptions, each matrix is defined as in §5.1.1;
- `executeClustering` performs KMeans clustering on the list of trajectory descriptions for each distinct *"origin-destination"* group;
- `saveClustering` saves the lists of clustered trajectories into a file (each list represents a distinct *"origin-destination"* group);
- `loadClustering` reads the file of the existing clustering result and creates the list of trajectories, divided by *"origin-destination"*, with label information;
- `createClusterBuckets` is identical to the definition of `createBuckets` for the trajectory group implementation, with the addition of the filter to check for the same *"origin-destination"* group and label in the current bucket (and the escape rule to avoid non-termination).

Because of the similarity with the trajectory groups method and the clustering implementation (in particular with the consideration of trajectory groups as a special case of clustering), the following functions have simply been renamed:

- the C++ function `initFromGroups` has been renamed to `initFromCluster`;
- the Python function `sortGroups` has been renamed to `sortClusters`.

5.1.4 Algorithm

The clustering Kernel Search algorithm implemented is here provided, divided in its two phases (expressed as pseudo-code that can be matched easily with functions defined in §5.1.3).

Parameters

- `instanceName`: the path to the specific instance file;
- `maxDelay`: indicates the maximum delay allowed, a negative number indicates a minimization of delay problem, otherwise this value will be used for the specific maximization of preferences constraint;
- `problemParams`: includes different parameters used for the creation of the problem, such as the unique max delay and the unique delay step (it isgrouped here to avoid having a list of parameters too long for functions);
- `CplexVars`: includes the CPLEX variables regarding the environment and the problem;

- `maps`: includes the two maps `indexToVar` and `varToIndex`, initialized as empty;
- `bucketSize`: the size of each bucket, it has a value equal to 10.000;
- `aging`: determines how many iterations unused variables remain in the kernel, it has a value equal to 2;
- `subProbTimeout`: the timeout assigned to each subproblem (excluding the first kernel subproblem), its value is 600 seconds, that corresponds to 10 minutes.

The initialization phase is identical to the standard implementation (see §4.4.4), so its pseudo-code is omitted.
After this phase, the execution flow passes control to a Python program, which performs variable ordering (the outer repeat-until loop is a detail about the implementation of `createClusterBuckets`, in red the differences with the trajectory groups implementation are highlighted):

```
begin
  if clustering has been already performed then
    trajClusters = loadClustering(instanceName)
  else
    trajGroups = commonTrajectories(instanceName)
    trajDescriptions = getTrajectories(instanceName)
    trajClusters = executeClustering(trajGroups, trajDescriptions)
    saveClustering(trajClusters)
  variables = getVariables(instanceName, maxDelay)
  trajToVar = connectTrajToVars(variables, instanceName)
  varGroups = convertToVars(trajClusters, trajToVar)
  varGroups = sortClusters(varGroups)
  varGroups = prioritizeGroups(varGroups)
// Start of createClusterBuckets pseudo-code
  allBuckets = ∅
  repeat
    currentGroup = 0, insertedVars = 0
    currentBucket = ∅
    repeat
      if size(varGroups[currentGroup]) != 0 then
        // clusterCheck returns true if and only if a similar
            variable has not been inserted yet
        // escapeRule is set to true if and only if for one
            entire cycle I didn't insert any variable
        if clusterCheck(varGroups[currentGroup]) or escapeRule then
          // pop() removes the first element of the list
              returning it
          currentBucket = currentBucket ∪
              varGroups[currentGroup].pop()
          insertedVars = insertedVars + 1
        currentGroup = next group index (the first one if this is
            the last)
      else // empty groups are removed
        remove varGroups[currentGroup]
    until (insertedVars == bucketSize or varGroups == ∅)
    allBuckets = allBuckets ∪ currentBucket
  until (varGroups == ∅)
// End of createClusterBuckets pseudo-code
```

```
  saveVariables(allBuckets, instanceName, maxDelay)
end
```

The expansion phase is identical to the trajectory groups implementation (see §4.5.2), so its pseudo-code is omitted.

5.2 Integration of classification techniques into the Kernel Search

The classification features implemented by the decision trees described in §3.4.2 have been reused in the Kernel Search procedure, without any modification, with the goal of using them as an additional tool to refine buckets dynamically, adjusting their composition based on the current solution status.

5.2.1 Idea and application

As a byproduct of classification, decision trees provide, for each variable, a measure of the probability of being a good insertion in the current solution. Buckets are composed by 10.000 variables each, where the first buckets contain the best variables, even if only few of such variables will be actually part of a better solution. The decision tree acts as a filter that keeps only "the best of all good variables", by fixing a probability threshold, and removing all variables whose probability of being good is below such limit.
Decision trees for both preference maximization and delay minimization have already been implemented in [12], using the C++ language, as two different functions written as a series of `if-else` statements. This structure constitutes exactly the direct translation of a decision tree, whose main characteristic is to be easily understandable. Since these functions only provides a value used by the filter, they can be applied to any Kernel Search implementation, by reducing the bucket size before the creation of the corresponding subproblem.

5.2.2 Classes and auxiliary elements

The decision tree filter requires information about the current solution, in order to obtain the required features for each variable. Functions that provide such values have already been implemented in the class `Data`, available from [12] (only important parts for the Kernel Search implementation will be mentioned). Class `Data` has the following attributes:

- one attribute for each characteristic of the problem parameters (list of preferences, max delay, instance name, number of flights etc.);
- `capacity` indicates the capacity of each sector over time;
- `all_vars` contains all possible variables of the model;
- `current_solution` indicates variables with positive value in the current solution;
- `prev_solution` indicates variables with positive value in the previous solution;
- `current_capacity` indicates sector capacities, computed taking into account flight plans currently selected by the solution.

Class `Data` has the following methods:

- `var_at` returns the variable information given the variable characteristic (flight, trajectory, delay);
- `initSolEmpty` creates an empty solution by leaving all flights unplanned (it selects all z variables);
- `updateSolFromResult` reads the current solution to update internal attributes coherently;
- `updateStats` updates current capacities of all sectors;
- `congestionRate` returns the congestion rate of sectors affected by the insertion of the flight plan given as parameter;
- `routeComparison` returns the percentage of similarity between two flight plans given as parameters.

Most of `Data` functions work using flight plans, that are implemented in the class `Route`, available from [18], [10] and [12] (only important parts for the Kernel Search implementation will be mentioned). Class `Route` has the following attributes:

- `flight` is the identifier of the flight;
- `plan` is the identifier of the trajectory;
- `delay` is the number of delay time slots assigned;
- `occupations` indicates, as a list of pairs `(sector-time)`, all sectors affected by the flight plan.

This class only has constructor methods. Functions working with `Route` objects have been implemented in the `Data` class already described.
In order to use the just listed methods, the following adjustments are required in the Kernel Search procedure:

- an object of type `Data`, containing information about all possible variables, is built before the creation of the kernel problem, and it is initialized using `initSolEmpty`;
- an additional map, `flightToVar`, has been added, it connects each flight identifiers to the variable currently selected in the model; in order to make this tracking effective, this map is an extra parameter for the function `updateKernel`;
- the method `updateSolFromResult` has been changed, now it requires only the `flightToVar` map to update the current solution status, other attributes of the class `Data`, such as the index of each variable inside the model, are not modified because they are not used in the Kernel Search procedure (indexes are already tracked with the `indexToVar` and `varToIndex` maps).

5.2.3 Functions

The following functions have been defined for the decision tree implementation:

- `getTreeProb` returns the probability of a variable begin a good insertion into the current solution by calling the existing decision tree functions, it requires the list of features and the type of problem (to choose the right tree);
- `buildFeatures` creates the list of features given the variable name and the information about current solution (`Data` object);
- `getAllVarProbs` performs the decision tree classification for all bucket variables given as parameter;
- `pickDtFromBucket` calls previous functions to build features and compute probabilities, and removes from the bucket all variables that are below a fixed limit.

5.2.4 Algorithm

As said above, the decision tree filter can be applied to any implementation before the creation of bucket subproblems. Below, the focus on such part of the expansion phase is given, modifications from previous implementations are highlighted in red.

Parameters

- `instanceName`: the path to the specific instance file;
- `maxDelay`: indicates the maximum delay allowed, a negative number indicates a minimization of delay problem, otherwise this value will be used for the specific maximization of preferences constraint;
- `problemParams`: include different parameters used for the creation of the problem, such as the unique max delay and the unique delay step (it isgrouped here to avoid having a list of parameters too long for functions);
- `CplexVars`: includes the CPLEX variables regarding the environment and the problem;
- `maps`: includes the two maps `indexToVar` and `varToIndex`, initialized as empty. Additionally, it includes the new map `flightToVar` required for the decision tree procedure;
- `bucketSize`: the size of each bucket, it has a value equal to 10.000;
- `aging`: determines how many iterations unused variables remain in the kernel, it has a value equal to 2;
- `subProbTimeout`: the timeout assigned to each subproblem (excluding the first kernel subproblem), its value is 600 seconds, that corresponds to 10 minutes;
- `data`: information about the current solution, it is the object of type `Data`.

```
begin
// Expansion phase
   ...
  repeat
    currentBucket = buckets[counter]
    currentBucket = pickDtFromBucket(currentBucket, maxDelay, data)
    createSubProblem(CplexVars, bestSol, currentBucket, maps)
    newSol = solveSubProblem(CplexVars, subProbTimeout)
    ...
  until (stop condition)
  saveToFile(instanceName, maxDelay, bestSol)
end
```

Chapter 6

Computational results

In this chapter, the results of the execution of Kernel Search algorithm, in all the variants proposed in this thesis, are presented, and compared to the results obtained by the dynamic column insertion relaxation with rounding algorithm presented in literature.

6.1 Experiment setup

All Kernel Search executions have been run on the machines of *Servizio di Calcolo, Torre Archimede, Dipartimento di Matematica (Università degli Studi di Padova)*, which have the following specifications:

- operating system: Ubuntu 18.04.6 LTS;
- kernel: Linux 4.15.0-194-generic;
- CPU: Intel(R) Core(TM) i7-8700 3.20GHz;
- RAM: 31 GB;
- IBM CPLEX Optimization Studio: version 12.8.

The use of CPLEX version 12.8, instead of the newer version adopted in [10], provides higher computational times for the column insertion procedure, leading us to set a timeout of 2 hours for this procedure.
Before presenting results, it is important to define the notation used to present data:

- **Date** refers to the date of the corresponding instance. We consider the same instances used in [18];
- **Round** refers to the rounding algorithm used to obtain an integer solution, described in [18] and mentioned in §3.4.1;
- **KS** means "Kernel Search". It refers to the implementation of the section where the acronym appears (for example, it means "Standard Kernel Search" in the section devoted to the standard implementation);
- **DT x%** means "Decision tree filter with threshold x%", so, for example, "Clustering DT 50%" refers to the clustering implementation with the decision tree probability threshold fixed to 50% (excludes from each bucket all variables having a probability lower than 50%);

Date	Type	Kernel % Gap (30 min)	Kernel % Gap (60 min)
27 Aug	Delay	14,94	1,72
28 Aug	Delay	0,65	0,46
27 Aug	Pref	1,58	0,41
28 Aug	Pref	0,2	0,16

Table 6.1: Comparison with different timeouts on the kernel subproblem for critical instances.

- **% Gap** is the percentage ratio of the difference with respect to the dynamic column insertion relaxation value;
- times are expressed as number of seconds elapsed from the beginning of the algorithm to the end, it is not the time to find the best solution (recall that an equal solution is feasible and does not stop the procedure). A more accurate performance over time analysis is performed in 6.6. More specifically, rounding algorithm times do not consider the time spent on the dynamic column insertion relaxation, and Kernel Search times start with the expansion phase: relaxation and sorting are not considered, but in each specific section the time spent on sorting will be reported.

Furthermore, the following consideration must be taken into account: the two instances related to days 27th and 28th August will be referred to as "critical instances", because there are known issues with values and times of such instances (also with other methods, not only Kernel Search); in many tables in this chapter, these instances will have values much different from values of other instances. The easiest fix to this issue would be to assign a higher timeout to such problems. Changing the algorithm parameters for specific instances is not a good practice, but it was made to show that with additional time the procedures provide good results even in this cases. As displayed by Table 6.1, a timeout of 1 hour brings significant improvement on the solution of the kernel problem: this is mostly noticeable on the minimization of delay of 27th August instance, in fact, the gap difference is higher than 10%. For other instances presented in Table 6.1, the improvement is smaller, but it is still noticeable within the gap approximation at 2 decimals.

6.2 Filtering effect of the decision tree

In this section, we provide generic results about the effectiveness of the decision trees in filtering hopefully good variables. Statistics consolidate different implementations of the kernel search, and we remark that we will not provide, in this section, information about the overall impact of filters on the quality of the solution obtained, nor on the efficiency of the kernel search implementation, with or without such filters, which will be discussed in §6.4.1 and §6.5.1.
Consolidated statistics are here presented with reference to different thresholds to the probability of a variable being good (used to select or not a variable in the bucket, see §5.2):

- all variables with probability higher than 50% are selected;
- all variables with probability higher than 70% are selected;
- all variables with probability higher than 20% are selected.

Date	**0-10%**	**10-20%**	**20-30%**	**30-40%**	**40-50%**
8 July	0	569	2250	0	0
26 Aug	0	521	2232	0	0
27 Aug	0	532	2324	0	0
28 Aug	0	503	2228	0	0
29 Aug	0	575	2281	0	0
30 Aug	0	538	2302	0	0
31 Aug	0	544	2318	0	0
1 Sept	0	542	2381	0	0
2 Sept	0	502	2273	0	0
9 Sept	0	520	2353	0	0

Table 6.2: Decision tree probability distribution for the minimization of delay on the first bucket obtained with clustering sort (from 0 to 50%).

Date	**50-60%**	**60-70%**	**70-80%**	**80-90%**	**90-100%**
8 July	31	0	2053	4577	0
26 Aug	37	0	2205	4496	0
27 Aug	31	0	2227	4478	0
28 Aug	37	0	2273	4472	0
29 Aug	37	0	2126	4471	0
30 Aug	40	0	2150	4487	0
31 Aug	36	0	2208	4426	0
1 Sept	30	0	2175	4402	0
2 Sept	43	0	2154	4519	0
9 Sept	34	0	2142	4476	0

Table 6.3: Decision tree probability distribution for the minimization of delay on the first bucket obtained with clustering sort (from 50 to 100%).

A fixed threshold has not been established, because there was not much time left to test execution on other values, also this filters were tested only on the trajectory groups and clustering implementations (standard implementation was left out); the probability distribution for variables in the first bucket are shown in Table **??**, 6.3, 6.4 and 6.5, as well as Figure 6.1 : ranges of probabilities are displayed, and each interval is read as $(start, end]$, for example, the range 10-20% includes all variables whose probability is strictly greater than 10% and lower than (or equal to) 20%.
The time spent on filtering buckets is negligible (a total of few seconds for a complete execution with more than 400 buckets), so it is not considered. The distribution of probability computed by decision tree on the first bucket obtained is illustrated only for buckets obtained with the clustering sorting (trajectory groups sorting have an average difference of a couple of units, that is negligible when considering a total of 10.000, so they are not reported), using as current solution the result of the kernel problem: Tables 6.2, 6.3, 6.4, 6.5 show the values on each instance, and the diagram in Figure 6.1 displays the average for the minimization of delay. For the minimization of delay (see Tables 6.2 and 6.3), most filters keep more than 60% of the variables. Removing variables classified with a low probability is the goal of this implementation, and it shows that the sorting actually puts many good variables in the first bucket: by removing bad variables, subproblems will require less time to be solved and the

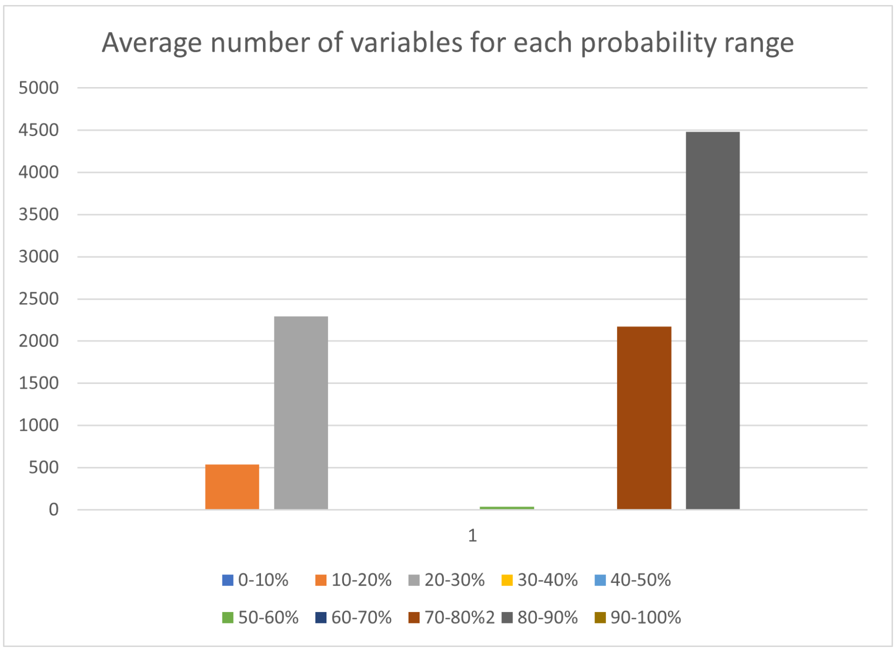

Figure 6.1: Distribution of decision tree probabilities for minimization of delay on the first bucket built with clustering based sorting.

application of filters on this problem is expected to maintain the same results while reducing the execution time, or to even improve results because the new amount of variables avoid reaching the time limit for subproblems. If the result is worse than the one obtained without applying any filter, then that means bad variables discarded by the decision tree were actually good for the current solution, and so the tree didn't discriminate correctly variable probabilities, or the filter limit was set too high. For the maximization of preferences (see Tables 6.4 and 6.5), most filters keep only a couple of hundreds variables; for this reason, the diagram that displays the average is omitted. Less variables means less time to solve the subproblem, but a little amount of variables, even in the best case scenario, can bring only minimal improvement, so many iterations will be needed to notice a considerable difference in the objective function value, assuming the procedure didn't stopped early because it didn't improve; if no improvement is made with respect to the unfiltered version of the method, it means that variables classified with a low probability value actually would have been picked for the subproblem solution. If, instead, the application of filters brings better results, then it means the majority of variables in the first buckets is useless, and the sorting criterion does not put effectively the best variables in the first buckets for the maximization of preferences.
Details about the filtering results are found on trajectory groups (§6.4.1) and clustering (§6.5.1) sections.

6.3 Standard Kernel Search

Below, the results for the standard implementation (described in §4.4) are shown, in comparison with the results obtained with the rounding algorithm (described in

Date	0-10%	10-20%	20-30%	30-40%	40-50%
8 July	9851	15	0	27	7
26 Aug	9869	18	0	17	4
27 Aug	9847	18	0	22	9
28 Aug	9839	24	0	18	4
29 Aug	9841	23	0	25	5
30 Aug	9862	16	0	18	3
31 Aug	9858	21	0	13	5
1 Sept	9850	25	0	11	6
2 Sept	9861	18	0	23	5
9 Sept	9860	20	0	18	6

Table 6.4: Decision tree probability distribution for the maximization of preferences on the first bucket obtained with clustering sort (from 0 to 50%).

Date	50-60%	60-70%	70-80%	80-90%	90-100%
8 July	30	1	27	7	0
26 Aug	28	1	27	6	0
27 Aug	27	1	27	10	0
28 Aug	37	0	24	6	0
29 Aug	30	0	27	10	0
30 Aug	35	0	22	5	0
31 Aug	31	2	22	6	0
1 Sept	40	0	30	1	0
2 Sept	28	0	23	4	0
9 Sept	25	0	20	8	0

Table 6.5: Decision tree probability distribution for the maximization of preferences on the first bucket obtained with clustering sort (from 50 to 100%).

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	7	0,24	86	0,01
26 Aug	11	0,05	405	0,02
27 Aug	350	0,97	3600	9,64
28 Aug	66	0,34	3629	0,64
29 Aug	13	0,25	2019	0,02
30 Aug	25	0,22	3000	0,04
31 Aug	16	0,5	3622	0,05
1 Sept	12	0,07	2258	0,05
2 Sept	4	0,13	53	0,01
9 Sept	12	0,86	74	0,01

Table 6.6: Comparison between rounding algorithm and standard Kernel Search for the minimization of delay.

§3.4.1). We recall that no filtering technique based on decision tree is considered for this implementation. The displayed time does not take into account the cost of sorting all variables, because it starts when the first subproblem is created: such time has a value of a couple of seconds. The standard implementation for the minimization of delay already brings good results with respect to the relaxed objective function value. In fact, Table 6.6 shows that the gap is below $0,1\%$ for all instances, with the exception of critical ones, especially 27^{th} August, but, as illustrated in Table 6.1, just a higher timeout of the kernel problem will translate into an objective function value close to the value found by the rounding algorithm. The maximum time spent by this Kernel Search implementation is one hour, which is below the timeout chosen as stop condition: it means the algorithm did not find an improving solution in a certain iteration, and so it terminated.
As an observation, on some instances the procedure went really fast, taking about one minute, but even in such cases the algorithm is slower than the rounding algorithm, which was expected: in fact, the goal of the Kernel Search implementations is to find better results by spending more time. Where the gap is very small (0,01), the solution is almost optimal, in same cases it is optimal, for example the relaxation value for the 8^{th} July instance is $5771, 14$, while the Kernel Search objective function value is 5772, which is optimal because all objective function coefficients are integer, this is the best value obtainable by using only integer variables. Also for the maximization of preferences (see Table 6.7), the standard implementation provides good results, in particular only the first critical instance presents a worse value; this version of the problem is solved with a larger amount of time (on average), because all kernel subproblems, with the exception of the instance relative to the 2^{nd} of September, go on timeout, and so does the majority of the first bucket subproblems; this is due to the nature of the problem, because there is the maximization of the objective function value, but it has an additional constraint on the total delay, which is set to be close to the minimization of delay value, so much more time is spent to solve this problem.

6.4 Trajectory groups Kernel Search

Below, the results for the trajectory groups implementation (described in §4.5) are shown, in comparison with the results obtained with the rounding algorithm (described in §3.4.1). The displayed time does not take into account the cost of the sorting

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	48	0,09	2400	0,05
26 Aug	20	0,53	3001	0,04
27 Aug	75	0,96	2400	1,55
28 Aug	31	0,82	2400	0,23
29 Aug	48	0,04	2401	0,03
30 Aug	54	0,15	2400	0,08
31 Aug	22	0,68	3001	0,03
1 Sept	21	0,58	4166	0,05
2 Sept	23	0,56	1106	0,02
9 Sept	31	0,07	2249	0,02

Table 6.7: Comparison between rounding algorithm and standard Kernel Search for the maximization of preferences.

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	7	0,24	717	0,01
26 Aug	11	0,05	736	0,02
27 Aug	350	0,97	3685	0,80
28 Aug	66	0,34	3655	0,36
29 Aug	13	0,25	1914	0,00
30 Aug	25	0,22	2397	0,02
31 Aug	16	0,5	3502	0,02
1 Sept	12	0,07	2712	0,05
2 Sept	4	0,13	638	0,01
9 Sept	12	0,86	531	0,01

Table 6.8: Comparison between rounding algorithm and trajectory groups Kernel Search for the minimization of delay.

algorithm, because it starts when the first subproblem is created: such time has an average value of 40 seconds. This implementation takes more time than the standard one, as displayed in Table 6.8: more specifically the time difference is remarkable in instances where the standard implementation terminates in less than 2 minutes, other instances on average take a similar amount of time.
Compared to standard Kernel Search, trajectory groups provides better results: in particular, also critical instances values are close to the rounding algorithm results (without the timeout change), and for 7 out of 10 instances the gap with the relaxation value is below 0,02%; in this cases the result could be the optimal integer solution, or a solution with a difference of literally one or two units. On maximization of preferences (see Table 6.9), instead, trajectory groups provides the same results as the standard implementation (considering gap approximation at 2 decimals), with the exception of the second critical instance where the gap is a little better, but it takes less time to terminate: on some instances the partial sorting done by the trajectory groups method puts all good variables in the first bucket, where for the standard implementation such variables would be divided in different buckets, thus requiring more iterations (i.e more time) to find the same solution.
Overall, this method is better in any way than the standard one: it provides better results, while times are better or equal, with the exception of very low times on

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	48	0,09	2400	0,05
26 Aug	20	0,53	2400	0,04
27 Aug	75	0,96	2400	1,55
28 Aug	31	0,82	2400	0,20
29 Aug	48	0,04	2401	0,03
30 Aug	54	0,15	2400	0,08
31 Aug	22	0,68	2400	0,03
1 Sept	21	0,58	2400	0,05
2 Sept	23	0,56	1108	0,02
9 Sept	31	0,07	2318	0,02

Table 6.9: Comparison between rounding algorithm and trajectory groups Kernel Search for the maximization of preferences.

minimization of delay that in this methods transforms from a couple of minutes into around 10 minutes, which is still way below the timeout limit.

6.4.1 Integration with the decision tree filter

The three different decision tree filters described in §6.2 have been applied to the trajectory groups implementation. Instead of providing a direct comparison with the rounding algorithm, such executions are compared to the unfiltered version of trajectory groups. For the minimization of delay, the gap (displayed in Figure 6.2) is the same when the filter is applied, with the exception of critical instances, so, by looking at just the results, no improvement is made on this version of the problem, but, considering performances, the method returns the same result in less time. Looking more specifically at execution times, illustrated in Figure 6.3, the application of any filter translates into a reduced Kernel Search time, without impacting the result: even just removing variables below 20% probability of being good, which are on average 500 for the first iteration, is enough to reduce the total time by a considerable amount of seconds; this means the decision tree worked as intended: it removed bad variables to speed up execution times, without affecting the final result. Variables with a bad classification may be used to just perform swaps in the solution, without improving it, so without filter the method was continuing to operate this variable swaps to satisfty the specific bucket constraints, without effectively improving the solution; without these variables, swaps could not be performed, the subproblem does not find an equivalent feasible solution, and so the overall procedure terminates. For the maximization of preferences, filters improve the results, but by a minimal amount, in the order of $0,005\%$, also differences between individual filters do not clearly identify which filter is better, as shown in Figure 6.4.
The trend observed in minimization of delay is inverted for the maximization of preferences, filters lead to greater execution times (see Figure 6.5), because few variables are picked (recall that on average 9.800 out of 10.000 bucket variables have a probability below 10%), resulting in much faster subproblems: with this little amount of new variables inserted, the solution of each subproblem (especially the firsts) is found without reaching the timeout limit, so the procedure can continue with a more refined bound for many additional iterations. This also confirms the hypothesis about the sorting applied on the maximization of preferences: improvements are made even after

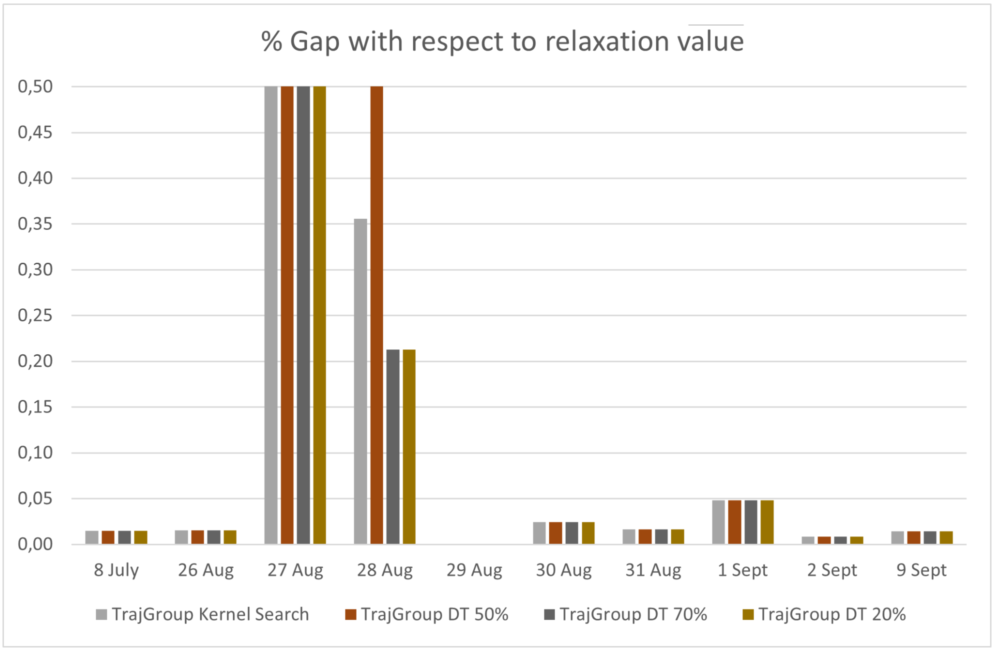

Figure 6.2: % gap comparison of trajectory groups method with different filters (minimization of delay).

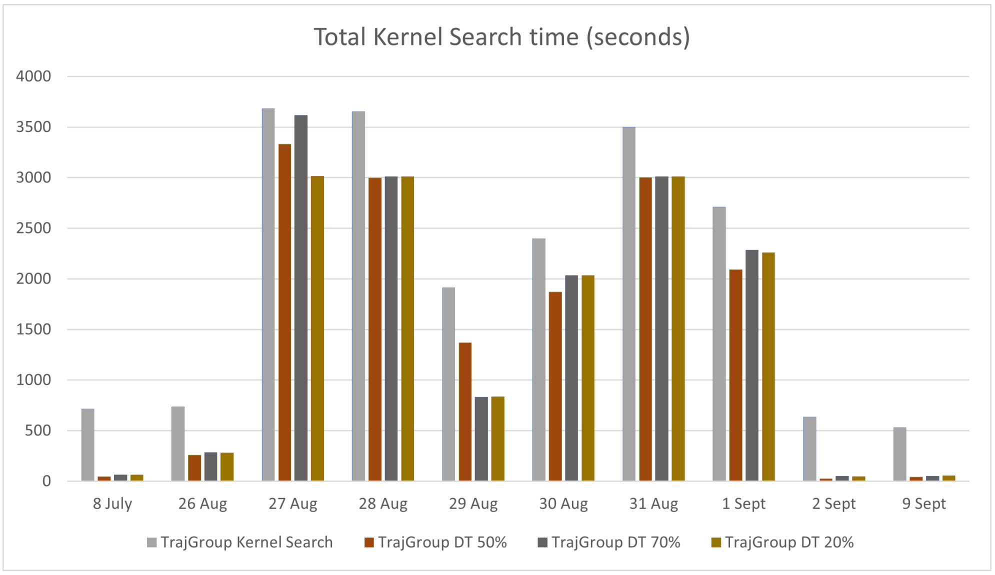

Figure 6.3: Time comparison of trajectory groups method with different filters (minimization of delay).

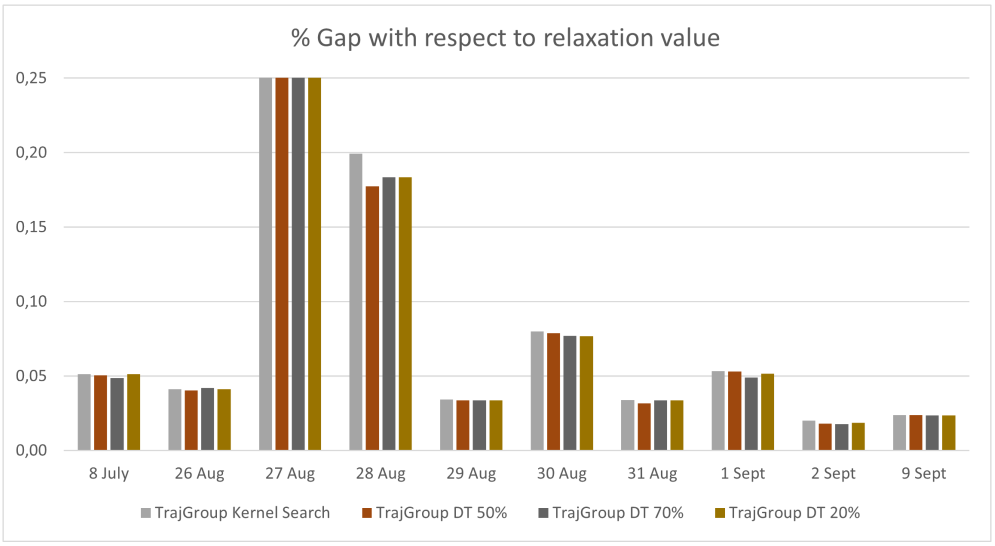

Figure 6.4: % gap comparison of trajectory groups method with different filters (maximization of preferences).

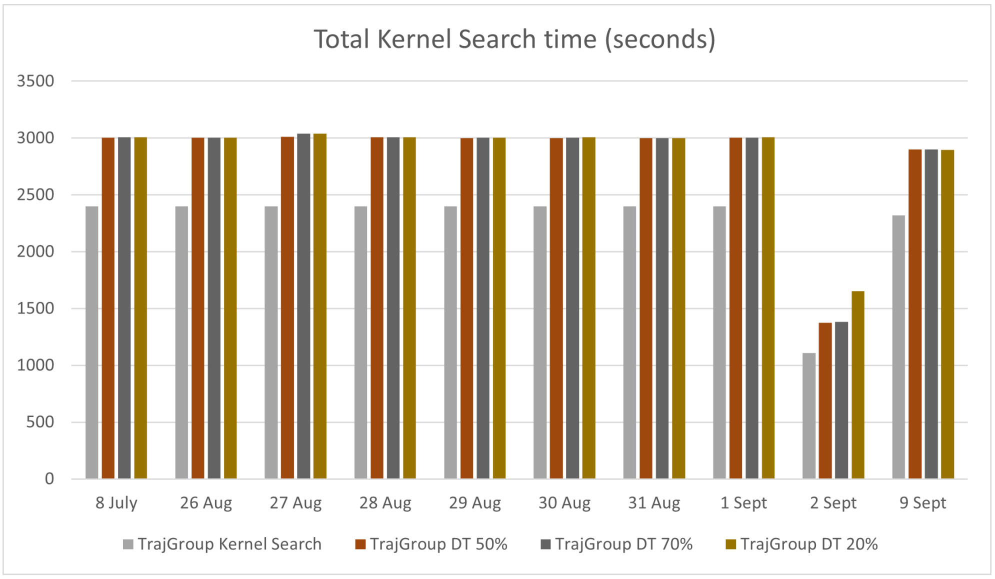

Figure 6.5: Time comparison of trajectory groups method with different filters (maximization of preferences).

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	7	0,24	745	0,01
26 Aug	11	0,05	745	0,02
27 Aug	350	0,97	3718	0,72
28 Aug	66	0,34	3678	0,36
29 Aug	13	0,25	2497	0,02
30 Aug	25	0,22	2385	0,02
31 Aug	16	0,5	3000	0,02
1 Sept	12	0,07	2491	0,05
2 Sept	4	0,13	613	0,01
9 Sept	12	0,86	529	0,01

Table 6.10: Comparison between rounding algorithm and clustering Kernel Search for the minimization of delay.

many iterations, meaning the decision tree discriminates correctly variables. The issue is that the first buckets mostly contains bad variables, because improvements happen also in the last buckets that should not contain good variables, according to the sorting criterion, that clearly does not identify the correct set of variables to build the first buckets, even if the first iterations bring the biggest improvements (as shown in the performance analysis presented in §6.6).

6.5 Kernel Search with clustering

Below, the results for the clustering implementation (described in §5.1) are shown, in comparison with the results obtained with the rounding algorithm (described in §3.4.1). The displayed time does not take into account the cost of the sorting algorithm, because it starts when the first subproblem is created: such time has an average value of 220 seconds (less than 4 minutes).
In addition to the sorting time, the first time the instance is run the clustering must be performed, this operation adds on average extra 20 minutes (18-19 minutes for all instances, with the exception of critical instances that take 25 minutes) to the full procedure. As shown in Table 6.10, the minimization of delay, on average, requires the same amount of time as the trajectory groups implementation, also solutions found are identical, with the exception of 29th August instance, which takes more time and stops with a worse result, while for the 28th August instance it performs slightly better. For the maximization of preferences, results obtained are comparable to the ones obtained through the trajectory groups implementation, as displayed in table 6.11. Also the time spent on execution is the same, with a difference of some seconds that can be appointed to the specific machine. The two sorting algorithms (trajectory groups and cluster based) produce different buckets, yet the obtained result is the same, the final solution simply has a different set of selected variables, or the gap in the objective function value is so little that the approximation at 2 decimal does not express such difference.
Overall the clustering implementation is not to be considered the best method, because it provides almost the same results as the trajectory groups implementation, but it has additional 20 minutes of clustering for each unique instance, and the sorting takes also more time than the sort on trajectory groups.

Date	Round Time (s)	Round % Gap	KS Time (s)	KS % Gap
8 July	48	0,09	2400	0,05
26 Aug	20	0,53	2401	0,04
27 Aug	75	0,96	2400	1,58
28 Aug	31	0,82	2400	0,20
29 Aug	48	0,04	2400	0,03
30 Aug	54	0,15	2401	0,08
31 Aug	22	0,68	2401	0,03
1 Sept	21	0,58	2400	0,05
2 Sept	23	0,56	1107	0,02
9 Sept	31	0,07	2296	0,02

Table 6.11: Comparison between rounding algorithm and clustering Kernel Search for the maximization of preferences.

6.5.1 Integration with the decision tree filter

The three different decision tree filters described in §6.2 have been applied to the clustering Kernel Search implementation. Instead of providing a direct comparison with the rounding algorithm, such executions are compared to the unfiltered version of clustering Kernel Search. The same trend that is observed in the application of filters in the trajectory groups implementation is represented by diagrams in Figures 6.6 and 6.7: minimization of delay maintains the same gaps with the exception of critical instances, but the 20% and 70% filters finds the optimal solution for the 29^{th} August instance (obtaining the same result as trajectroy groups). The difference in such instance is given by the fact that with the 50% filter and without filter the first subproblem goes on timeout before returning the solution with gap equal to 0. Also, the time analysis is the same as trajectory groups: the decision tree worked as intended, maintaining the same results while reducing the total execution time. Also for the maximization of preferences, the same behaviour of trajectory groups is observed in Figures 6.8 and 6.9: minimal improvements are done at the cost of higher execution times, because many subproblems with few variables are executed, so the clustering based sorting does not put correctly the best variables in the first buckets.

6.6 Performance analysis

In previous sections, data were presented in comparison with the rounding algorithm results, and, in case of the decision tree filters, with the corresponding unfiltered implementation. In this section, the evolution of the objective function value, expressed as the percentage gap with the relaxation value, is analyzed with respect to the time spent, and with respect to the number of total iterations performed; the following conventions are adopted:

- the performance of each method is displayed as the average percentage gap value among all instances. Due to the nature of critical instances, they are not considered for this analysis, because their gap is so different to completely influence the average gap value;
- executions that terminate before the scale limit (both for times and iterations) have their last value propagated to the limit: for example, if an instance terminates

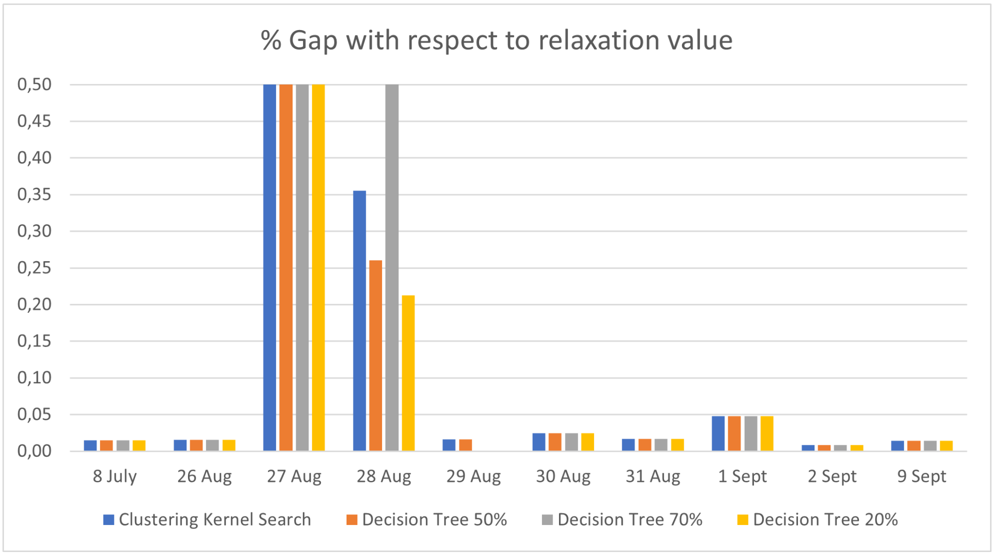

Figure 6.6: % gap comparison of clustering Kernel Search method with different filters (minimization of delay).

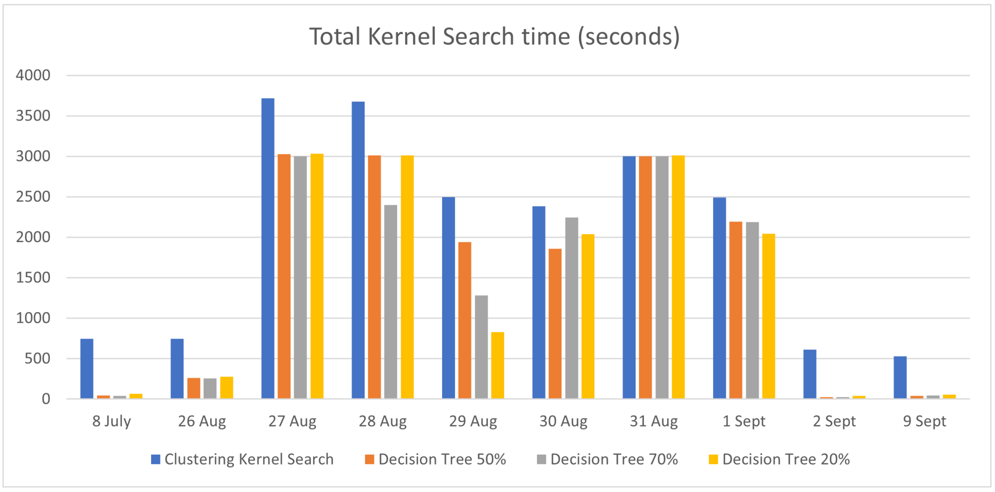

Figure 6.7: Time comparison of clustering Kernel Search method with different filters (minimization of delay).

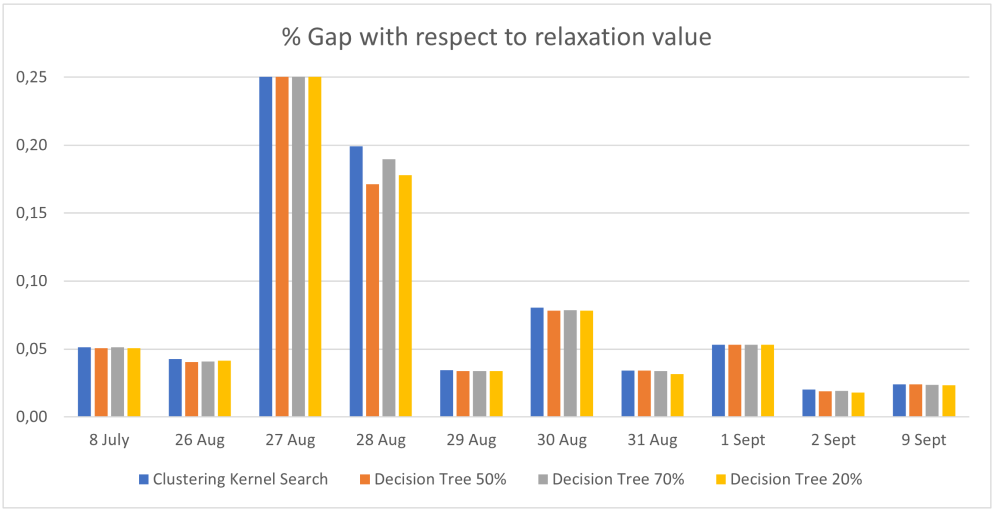

Figure 6.8: % gap comparison of clustering Kernel Search method with different filters (maximization of preferences).

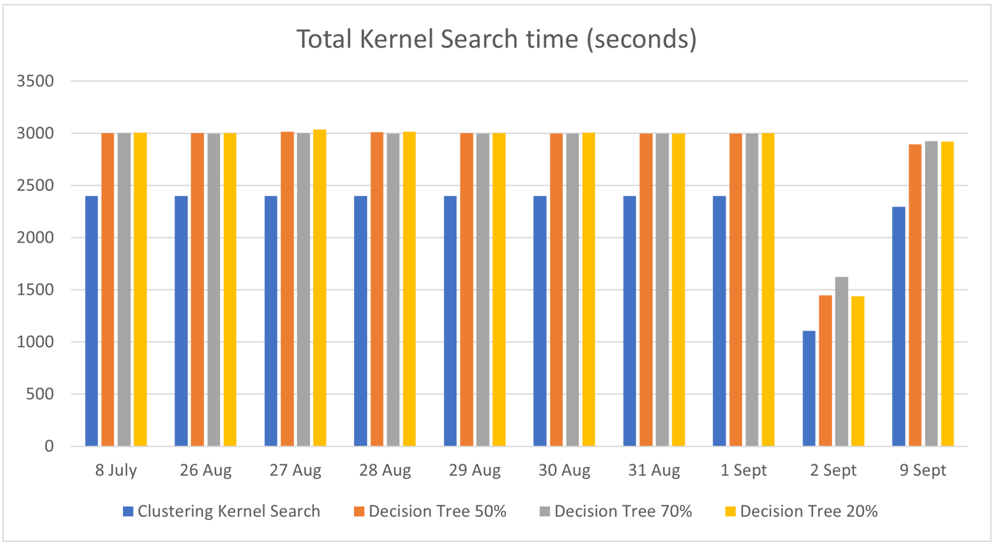

Figure 6.9: Time comparison of clustering Kernel Search method with different filters (maximization of preferences).

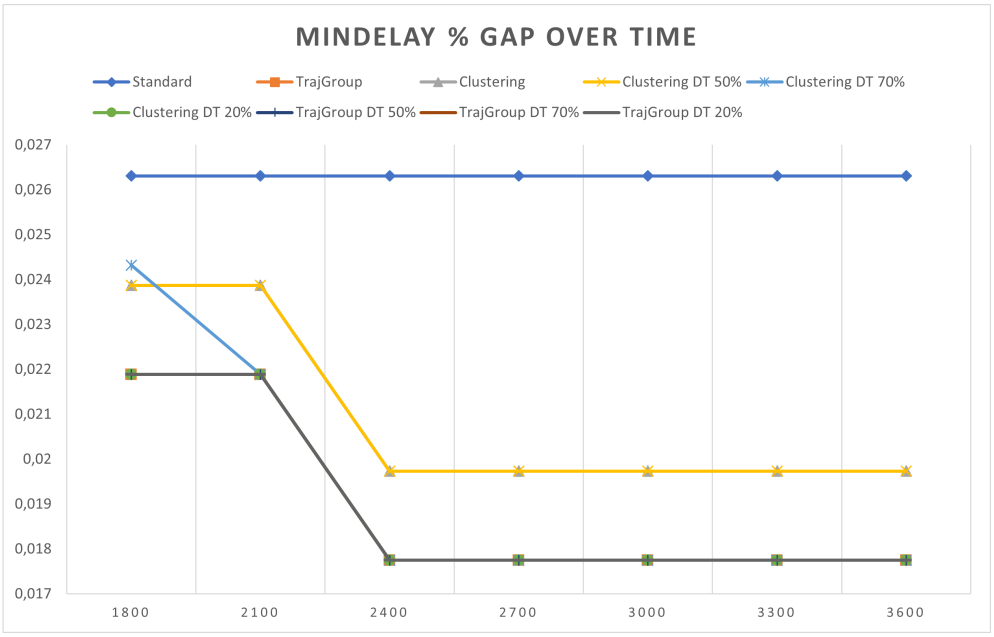

Figure 6.10: Average % gap over time (elapsed seconds) for the minimization of delay.

in 7 seconds with 2 iterations and returns a gap of $0,01\%$, then it is considered as if it has the same gap after 30 seconds, 4 minutes or after 5 iterations, 30 iterations, and so on. This is done to avoid having only few instances that contribute to the average on higher times/iterations, that would otherwise be confusioning: assume the only instance that takes more than 300 seconds is the one that has a final gap of 3%, while the average of all instances after 270 seconds is $0,5\%$, by considering only this instance for the time 300 seconds the curve would register an increase in the gap (and so, a worse solution value) with more time spent, which is absurd, because the Kernel Search method can only improve or maintain the objective function value;

- for the same reason described above, the graphics on execution time start with a value that is shared by all instances, i.e. the time spent to solve the hardest kernel problem, while for iterations the axis simply starts from 1.

In addition to that, maximization of preferences methods display two different starting points, whose gap difference is $0,0003\%$, that could be appointed to the single machine performance, that stopped the execution of the first subproblem on the fixed timeout of 30 minutes, but with literally one less simplex iteration. For the minimization of delay, the standard implementation has the worst performance, that is formed by a flat line, as visible in Figure 6.10, showing that the method found a solution and never improved it. On the other hand, all trajectory groups implementations (with all decision tree filters) and the clustering method with filters at 20% and 70% have the best performance over time, with some differences before 2.100 seconds, but such difference is negligible. The average gap difference with the clustering method and the clustering with a decision tree filter at 50% can be appointed to the solution obtained for the instance relative to the planning of 29^{th} August, that was already discussed in §6.5.1, the difference is of one integer point.

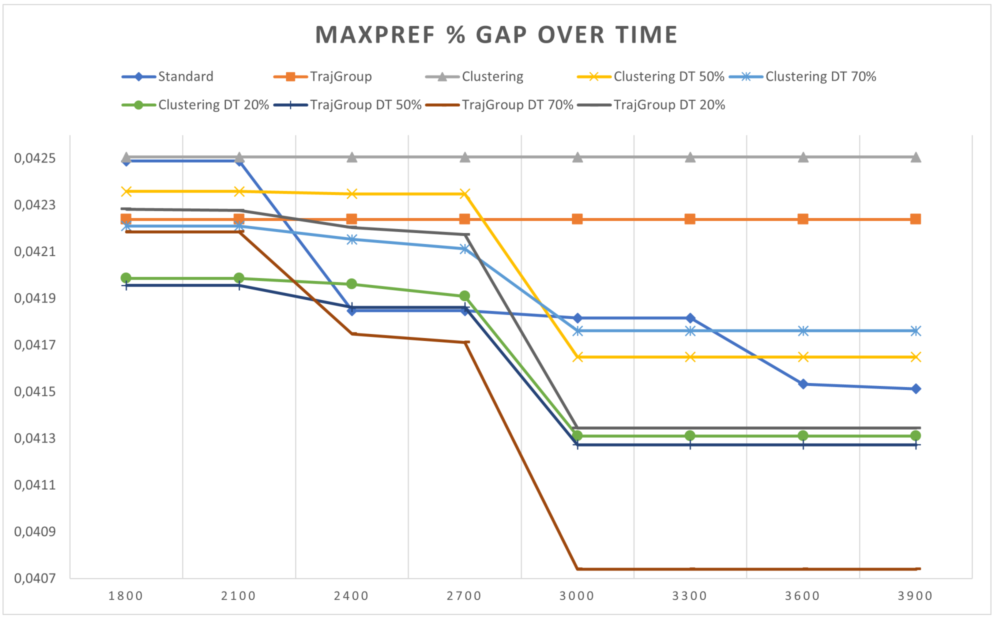

Figure 6.11: Average % gap over time (elapsed seconds) for the maximization of preferences.

Considering the extra time required to execute clustering, the best method for the minimization of delay is trajectory group, with a time limit of 2.400 seconds (40 minutes), with any decision tree filter. About the maximization of preferences, whose performance over time are displayed in Figure 6.11, without any decision tree filter the best method is standard Kernel Search, the difference is not noticeable with the gap approximation at 2 decimals, in fact it is negligible: it is in the order of 0,001%; this diagram shows that a decision tree applied to the standard implementation might be the best choice (because it is the best unfiltered method), but further test are required to verify this claim. The best results are obtained by the trajectory groups implementation, with a decision tree filter of 70% applied, after 3000 seconds (50 minutes), which is chosen to be the best method, considering the extra time required by clustering.
The same evolution that happened in the time analysis is represented for the iteration analysis: for the minimization of delay, the standard implementation stops improving after the first iteration, while all other methods find the best solution after the second iteration, with the exception of trajectory groups with a decision tree filter of 50%, that stops improving at the third iteration. This is illustrated in Figure 6.12: flat lines show that the respective method didn't find any other improving feasible solution. Together with the previous time analysis, the best method is trajectory groups, without any filter, or with a 20% or 70% decision tree filter applied, with a limit of 2 iterations. Similar to the gap evolution over time, for the maximization of preferences, unfiltered implementations do not make any improvement over successive subproblems (as shown in Figure 6.13), with the exception of the standard method that provides the best result at the second iteration. All methods that involve a decision tree filter keep improving until reaching a number of iterations equal to 200, which corresponds to a potential total number of $200 * 10.000 = 2.000.000$ examined variables, but most of them had been removed by buckets through the filter; this differences in gap after

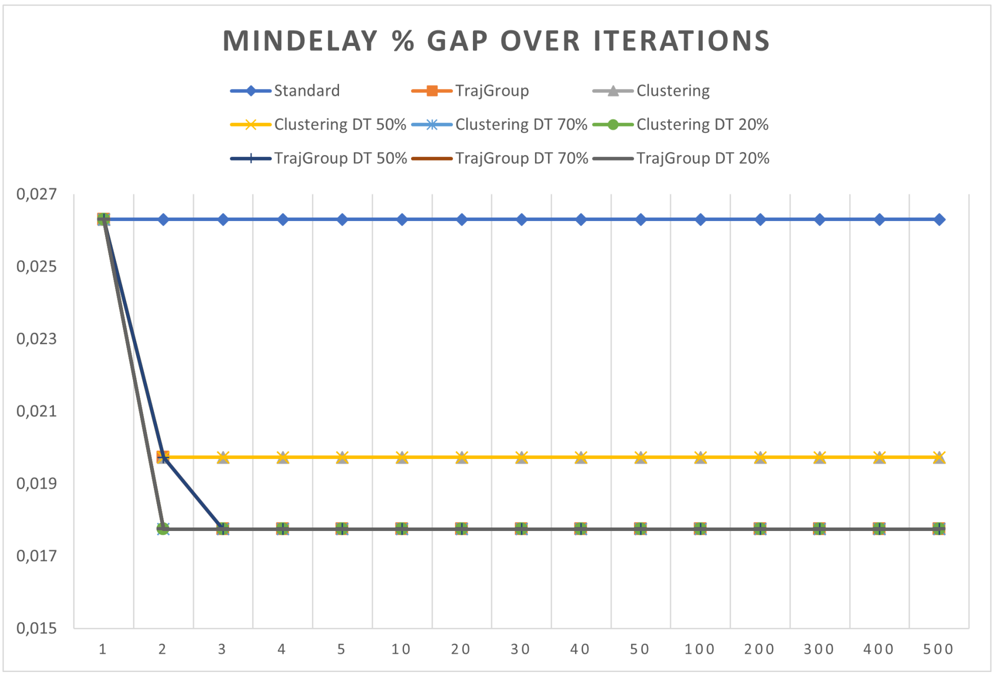

Figure 6.12: Average % gap over iteration for the minimization of delay.

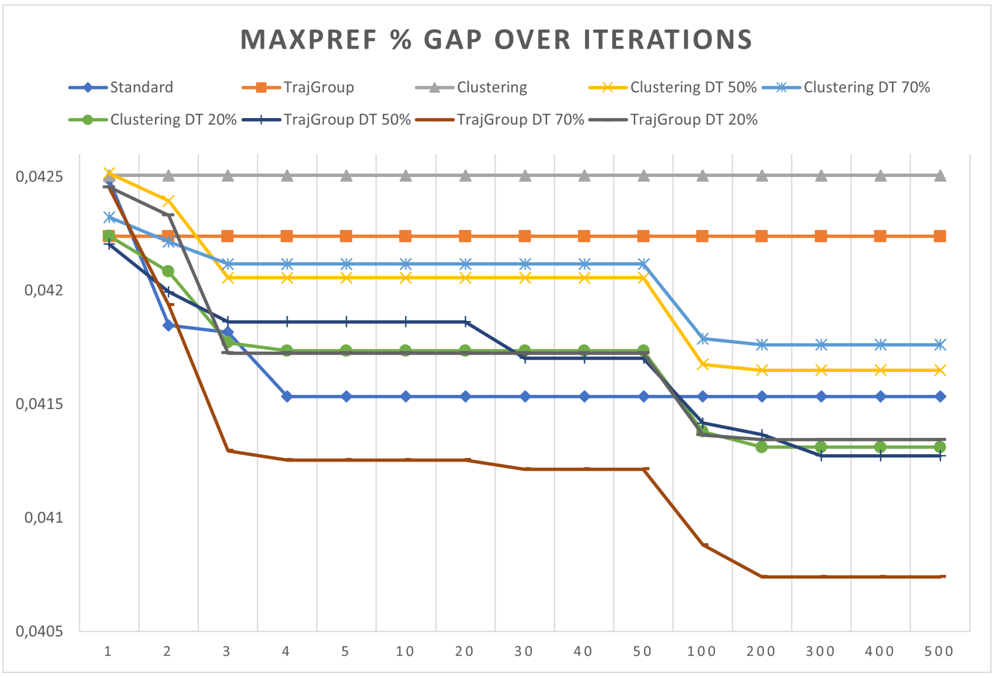

Figure 6.13: Average % gap over iteration for the maximization of preferences.

such number of iterations is negligible, in the order of $0,001\%$. Given the extra time required for clustering, and the previous time analysis, the best method is trajectory groups Kernel Search, with a decision tree filter of 70% and an iteration limit of 200; to obtain a good result (not the best) in less time, the iteration limit could be fixed at 3.

This analysis shows that the procedure is spending time and iterations without producing any considerable improvement, stop conditions can be improved to save computational time:

- a limit to the number of iterations can be set at 2 for minimization of delay and at 200 for maximization of preferences, without losing anything on the result value; alternatively, the maximization of preferences limit can be set to 3 iterations, in order to obtain a good result in a faster time, because improvements made in successive iterations are minimal;

- the total time limit can be set to 40 minutes for the minimization of delay, and to 50 minutes for the maximization of preferences; combined with the previous iteration limit, the procedure can end before this timeout, once it reached the limit number of iterations.

To sum up, the overall best Kernel Search implementation to solve the ATFM problem is trajectory groups, with or without any decision tree filter (with the exception of the 50% filter if the limit at 2 iterations is set) for the minimization of delay, and with a decision tree filter of 50% for the maximization of preferences: clustering implementations, on the other hand, require a minimum of 20 additional minutes, and do not even provide better results.

Chapter 7

Conclusions

In this chapter, final remarks are presented, focusing on the results provided by the proposed implementations of the Kernel Search matheuristic for the ATFM problem, and, in particular, on the role of the integrated Machine Learning tchniques. Possible future improvements are also outlined.

We considered the ATFM problem and a solving procedure based on the ILP model proposed in [10] and described in §3.3.2. This model is very large and existing literature proposes dynamic column insertion to solve its continuous relaxation and a rounding heuristic based on Branch & Cut, to obtain near optimal integer solutions.
The thesis proposes procedures based on Kernel Search to improve existing gaps, that are already good, with respect to the bound given by the relaxation value. Different implementations have been proposed, which represent the methodological contribution of this thesis. In particular:

- basic Kernel Search (described in §4.4) implements the Kernel Search as described in [19], [16] and [15], by adapting the proposed algorithm to the ATFM problem;
- trajectory groups Kernel Search (described in §4.5) uses an alternative sorting algorithm to determine "good variables": it consists in grouping all trajectories and related decision variables by *"origin-destination"*, and each group is sorted internally by increasing reduced cost. Finally, groups are sorted by the average reduced cost of the first six elements, and buckets are built using a round robin schema;
- clustering Kernel Search (described in §5.1) takes the trajectory groups implementation, and adds a clustering filter to it. In particular, a KMeans clustering is executed for each *"origin-destination"* group on a normalized description of flight trajectories; cluster labels are used in the round robin insertion to avoid picking variables with equal labels (within the same *"origin-destination"* group) for the same bucket, thus guaranteeing more diversity in each bucket;
- each of the previous implementations can be integrated with a decision tree filter (described in §5.2), that further reduces the size of each bucket. Decision trees taken from previous works on ATFM are used to compute the probability of each bucket variable to provide improvements with respect to the current solution, and filters discard from buckets all variables whose probability of being good is below a fixed threshold.

Tests have been executed on real-world instances: results show that the Kernel Search approach brings better result than the existing approaches, for all presented implementations. In particular, the integration of a decision tree filter drastically reduces execution times for the minimization of delay, without affecting the solution value; for the maximization of preferences, the filter brings better results at the cost of higher computational times.
Concerning clustering, as integrated in the Kernel Search, results show that this addition does not seem to bring improvement in the solution value with respect to the implementation that does not use it. Furthermore, this integration requires, on average, further 23 minutes (20 for clustering and 3 for buckets creation), which turns to be an additional computational cost that brings negligible advantages in terms of quality of the obtained solutions.

7.1 Possible improvements of the procedure

Results presented in the previous chapter show that the Kernel Search implementations can be improved with additional stop conditions to save computational time, also there could be improvement to the returned result, according to the following observations:

- decision tree filters work as expected for the minimization of delay, while for the maximization of preferences they show that few good variables are available after the first iteration, so that a possible improvement is to analyze the solutions given in the last iterations, identifying variables in the solution, especially in relation with variables included in the first bucket, understand their features, and redefine the sorting criteria for the maximization of preferences, as to effectively have the best variables in the first buckets.
  It is reasonable to have different sorting criteria for different problems, because the preference for a trajectory does not consider the delay, that appears only in one constraint. Since it is not included as coefficient in the objective function, the delay is not impacting the reduced cost, and so variables with a high delay are put at the same level with variables related to the same flight and plan, but with a lower delay; it is thus likely that this variables with high delay will not be inserted into the solution because they would violate the maximum delay constraint. On the other hand, the decision tree has the delay as one of its features, and recognizes that only few variables of the first bucket can be a good insertion. As a consequence, the possible improvement for the sorting criteria could be to take into account the delay assigned to each variable. Summarizing, the design of the sorting procedure should take the following considerations, related to each variable, into account:
  - value: this feature has no impact, because all variables with a non-zero value are already inserted in the kernel;
  - reduced cost: order variables by "promising impact on the solution" (this is the sorting criterion already implemented);
  - delay: for the maximization of preferences, in case of equal reduced cost, give precedence to the variable with lower delay; this does not apply to the minimization of delay, because in such problem the delay appears in the objective function and so the reduced cost is already affected by that;
- for further improvement, we observe that all kernel subproblems, with one exception, go on timeout for the maximization of preferences, and even in this

case such problems provide a good gap: a way to improve the result could be to set a higher timeout for the first iteration, in order to allow kernel problems finding their optimal solution, so to provide the best possible bound for the following subproblems.

7.2 Is Machine Learning worth the effort?

According to our computational experience, the Kernel Search integrating clustering provided worse results with respect to the one including trajectory groups. The difference is negligible but, in any case, the important information is that additioanl clustering does not seem to provide improvements, and has a time difference of 20 minutes for executing the clustering and 3 more minutes to sort variables. Clustering is executed once per instance, thus avoiding to repeat it when running an already processed instance, but even in this case, the sorting time is always sustained, resulting in overall higher execution times. On the other hand, the trajectory groups sorting takes less than a minute, and provides slightly better results, so that the proposed clustering technique does not seem to be a good choice to integrate Kernel Search for solving the ATFM problem.
The decision tree filters, instead, bring better results with almost no additional computational cost, because once trees are trained, they can be used on any instance to reduce the actual size of subproblems. Solutions are not improved for the minimization of delay, but overall execution times are reduced. Improvements on the objective function value have been observed only for the maximization of preferences, because filters remove all bad variables, that are usually the majority of the first buckets variables, thus avoid spending computational time on subproblems with a very low probability of yielding an improvement. This Machine Learning technique seems to be a good integration to the Kernel Search algorithm for solving the ATFM problem, because it brings improvement by removing bad variables, thus allowing the execution of more subproblems in less time. It can potentially be better if applied to a better sorting criterion that puts many good variables in the first buckets (in particular, for the maximization of preferences).

7.3 Final Remarks on Kernel Search

In a direct comparison with the rounding algorithm described in [18] and mentioned in §3.4.1, the Kernel Search algorithm takes, in all its implementations, more time, but always have a better results (with the exception of critical instances, as discussed in chapter 6), which is the purpose of this work: we allow to spend more time than the existing rounding procedure, as to find better solutions. The standard Kernel Search implementation, which is already giving good results (actually excellent results, for the maximization of preferences), has been further improved with different sorting methods (trajectory groups and clustering based) and, in order to reduce execution times on each subproblem, the bucket size has been reduced dynamically by applying a filter, based on the classification performed by a decision tree. Results obtained by this final methods are close to the optimal relaxed solution, which means they are also close to the optimal integer solution, and in some cases they are the exact optimal solution, making Kernel Search a good choice to obtain an excellent approximated solution to the ATFM problem.

References

[1] A. Agustín et al. “Mathematical Optimizationg models for Air Traffic Flow Management: A review.” In: *Studia Informatica Universalis* 8 (Jan. 2010), pp. 141–184 (cit. on p. 16).

[2] F. Aiolli. *Material of the course Machine Learning*. Università degli Studi di Padova, Dipartimento di Matematica. 2021 (cit. on p. 10).

[3] D. Bertsimas, G. Lulli, and A. Odoni. “An Integer Optimization Approach to Large-Scale Air Traffic Flow Management”. In: *Operations Research* 59 (2011), pp. 211–227 (cit. on p. 16).

[4] D. Bertsimas and S. Patterson. “The Air Traffic Flow Management Problem with Enroute Capacities”. In: *Operations Research* 46.3 (1998), pp. 406–422 (cit. on p. 16).

[5] D. Bertsimas and S. Patterson. “The Traffic Flow Management Rerouting Problem in Air Traffic Control: A Dynamic Network Flow Approach”. In: *Transportation Science* 34 (Aug. 2000), pp. 239–255. DOI: 10.1287/trsc.34.3.239.12300 (cit. on p. 16).

[6] *C++ Reference Manual*. URL: https://en.cppreference.com/w/ (cit. on p. 12).

[7] *CMake documentation*. URL: https://cmake.org/documentation/ (cit. on p. 12).

[8] *CPLEX Callable C API Reference Manual*. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=cplex-callable-library-c-api-reference-manual (cit. on pp. 8, 34).

[9] L. De Giovanni. *Material of the course Methods and Models for Combinatorial Optimization*. Università degli Studi di Padova, Dipartimento di Matematica. 2021 (cit. on pp. 8, 34).

[10] L. De Giovanni, G. Lulli, and C. Lancia. *Data-driven optimization for Air Traffic Flow Management with trajectory preferences*. 2022. DOI: 10.48550/ARXIV.2211.06526. URL: https://arxiv.org/abs/2211.06526 (cit. on pp. 16–19, 21, 28, 29, 43, 44, 49, 53, 71).

[11] F. Djeumou Fomeni, G. Lulli, and K. Zografos. “An optimization model for assigning 4D-trajectories to flights under the TBO concept”. In: 2017, pp. 1–10 (cit. on p. 16).

[12] M. Echerle. *Algoritmi euristici di ottimizzazione per la gestione del traffico aereo basati su modelli di programmazione matematica e machine learning*. Master Degree Thesis, Università degli Studi di Padova, Dipartimento di Matematica. 2021 (cit. on pp. 21–24, 43, 48, 49).

[13] A. Gelmi. *Data analytics e matheuristics per la gestione del traffico aereo con un'applicazione allo spazio aereo europeo*. Master Degree Thesis, Università degli Studi di Padova, Dipartimento di Matematica. 2020 (cit. on p. 22).

[14] *GitHub*. URL: https://github.com (cit. on p. 14).

[15] G. Guastaroba and M.G. Speranza. "A heuristic for BILP problems: The Single Source Capacitated Facility Location Problem". In: *European Journal of Operational Research* 238 (2014), pp. 438–450 (cit. on pp. 9, 27, 71).

[16] G. Guastaroba and M.G. Speranza. "Kernel search for the capacitated facility location problem". In: *Journal of Heuristics* 18 (2012), pp. 1–41. DOI: 10.1007/s10732-012-9212-8 (cit. on pp. 9, 32, 71).

[17] M.P. Helme. "Reducing air traffic delay in a space-time network". In: *1992 IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 1. 1992, pp. 236–242. DOI: 10.1109/ICSMC.1992.271770 (cit. on p. 16).

[18] T. Loss. *Gestione del traffico aereo con metodi basati su data analytics e column generation: un'applicazione allo spazio aereo europeo*. Master Degree Thesis, Università degli Studi di Padova, Dipartimento di Matematica. 2020 (cit. on pp. 8, 21, 28, 29, 43, 44, 49, 53, 73).

[19] V. Maniezzo, M. Boschetti, and T. Stützle. *Matheuristics: Algorithms and Implementations*. 2021. Chap. Kernel Search, pp. 189–197. ISBN: 978-3-030-70276-2. DOI: 10.1007/978-3-030-70277-9_9 (cit. on pp. 8, 9, 29, 71).

[20] *NumPy Documentation*. URL: https://numpy.org/doc/ (cit. on p. 13).

[21] A. R. Odoni. "The Flow Management Problem in Air Traffic Control". In: *Flow Control of Congested Networks*. Ed. by A. R. Odoni, L. Bianco, and G. Szegö. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 269–288 (cit. on p. 16).

[22] *Python 3.6 reference documentation*. URL: https://docs.python.org/3.6/ (cit. on p. 12).

[23] *Python scikit-learn library*. URL: https://scikit-learn.org (cit. on pp. 13, 43).