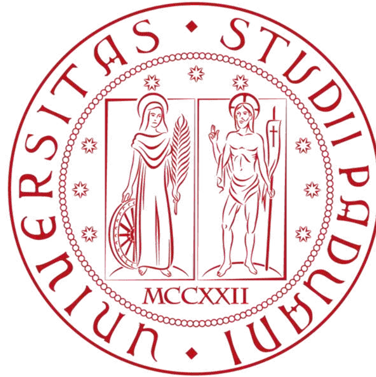

Università degli Studi di Padova

Department of Mathematics "Tullio Levi-Civita"

Master Degree in Mathematics

Il problema della configurazione dinamica dello spazio aereo: un approccio basato su modelli di cammino minimo vincolato

The Dynamic Airspace Configuration Problem: an approach based on constrained shortest paths

Supervisor
Prof. Luigi De Giovanni

Candidate
Luca Dalla Costa
Badge n. 2048573

Academic year 2023/2024

July 19th 2024

Poi, visita al celebre **museo** e incontro con la più emozionante scultura di tutti i tempi, la **Nike di Samotracia!**

Contents

Abstract **2**

1 Introduction **3**

2 Previous works and mathematical formulations **6**
- 2.1 Literature review . . . 6
- 2.2 Mathematical formulations for the dynamic airspace configuration problem . . . 7
  - 2.2.1 Relevant notation . . . 7
  - 2.2.2 Integer Linear Programming model . . . 8
  - 2.2.3 Graph representation . . . 9

3 Theoretical tools **12**
- 3.1 Fundamentals of Graph Theory . . . 12
- 3.2 The shortest path problem . . . 13
  - 3.2.1 Labelling algorithms for the shortest path problem . . . 13
- 3.3 Constrained shortest path . . . 13
- 3.4 Labels . . . 14
  - 3.4.1 Label setting, label correcting . . . 14
  - 3.4.2 Dominance criteria . . . 15

4 Complexity and resource-extended graph approach **16**
- 4.1 Reduction from the TSP . . . 16
  - 4.1.1 Valid approach dropping quiescence . . . 17
- 4.2 Resource-extended graph approach . . . 18
  - 4.2.1 Definition of linear resources . . . 19

5 A label-based approach to the Dynamic Airspace Configuration problem **21**
- 5.1 Label representation of the added constraints . . . 21
  - 5.1.1 Cost and permanence update . . . 23
  - 5.1.2 Quiescence update . . . 23
  - 5.1.3 Components for recovering the optimal path . . . 24
- 5.2 First set of dominance rules . . . 25

5.2.1 Dominance relations between label components 25
5.2.2 Utility of every domination condition 25
5.2.3 Properties of the dominance criterion 27

6 A constrained shorted path approach for dynamic airspace configuration 28
6.1 Description of the proposed method 28
6.1.1 Data preparation 30
6.1.2 Labels initialization 30
6.1.3 Label expansion 31
6.2 Algorithm insights arising from implementation issues 33
6.2.1 A consideration on paths derived from different configurations 33
6.2.2 Some coding issues 34
6.2.3 Focusing on past relations instead of future ones . . . 35
6.2.4 Flagging dominated labels 35
6.2.5 Skipping dominated labels 36
6.2.6 Potential for parallel implementation 36

7 Improved dominance rules 37
7.1 Alternative update of quiescence component 37
7.2 Exploiting graph sparsity 38
7.2.1 Permanence component 39
7.2.2 Quiescence component 39
7.3 Size-scalability and heuristic approach 40

8 Computational results and discussion 43
8.1 Instances generation 43
8.1.1 Merging graph structures 44
8.1.2 Forming compatibility sets 45
8.2 Benchmarking results and observations 46
8.2.1 Results observations 47
8.2.2 Some data on labels 48
8.2.3 Results of a heuristic 49

9 Conclusion and future work 50

A Instance generation 51

B Python code 53
B.1 Exploration block 53
B.2 Label encoding 54
B.3 Labels' domination and expansion 55

References 59

Abstract

The Dynamic Airspace Configuration (DAC) problem consists in the determination of a sequence of airspace partitions, subject to side restrictions regarding steadiness. A recent work illustrates a Integer Linear Programming (ILP) model for the resolution of DAC, also presenting a graphical reformulation of the model on a weighted graph.

In this thesis, after defining the problem in its operational context, we use the graph formulation of DAC to discuss its computational complexity.

We then propose a new exact method for DAC based on constrained shortest paths, making use of labels that are suited to express the side constraints.

The proposed method has been implemented in Python and tested on realistic instance created on purpose. Computational results have been collected and compared to the performace of the ILP model solved by a general purpose mathematical programming solver, namely Cplex.

After extensive testing on small and medium-sized instances, we outlined two main scenarios. When the graph is sparse, our method results more effective than Cplex. On the contrary, when an instance presents a large number of feasible solutions, our method performs worst, due to the large number of non-dominated labels to store and process. In order to tackle the latter case, we describe a heuristic domination strategy, that grants fast solving times, without any guarantee on the quality of the solution, but with solutions that, in our tests, are close to the optimal ones.

Chapter 1

Introduction

Aviation plays an increasingly important role in the global transportation network. As such, it becomes apparent the importance of understanding what problems could arise, and develop ad-hoc methods to approach them.

The first topic to face is the estimated increase in number of flights [14]. Different studies underline the need to reconcile a high volume of traffic with a limited air space capacity. The constraint on capacity derives from the way Airspace Traffic Control (ATC) entities guarantee a certain degree of security: a team of operators is lined up, and is responsible of defining traffic regulations, such as rerouting or planning delays on departures. These regulations intend to limit and keep under control phenomena as conflicting routes and hotspots, i.e. portions of the airspace with high-density demand.

The notion of capacity is associated to the concept of sector. We derive from [9] an approach based on the elementary and collapsed sectors. An elementary sector is a $3D$ portion of the airspace. A collapsed sector is defined by the union of one or more adjacent elementary sectors. An airspace configuration is a partition of the collection of elementary sectors, each element of the partition will be a different collapsed sector. Starting from the same group of elementary sectors, different configurations can be achieved. For a clear view on configurations and collapsed sectors, we refer to Figure 1.1. It is obtained by [7] as result of their work on methods for the formation of configurations.

The capacity of a collapsed sector is the maximum number of flights that the collapsed sector controllers can monitor without exceeding a default maximum workload. The capacity of a sector depends, among other things, on its size and geometry, but the capacity is dynamic and changes over time to allow ATC to operate in safe conditions. Thus, the set of configurations that can be achieved in a specific time of the day is also dynamic.

Another aspect we must take into account regards transitions, and plausible moves between subsequent configurations. The term transition refers to the process of abandoning a configuration in favor of another one; it is

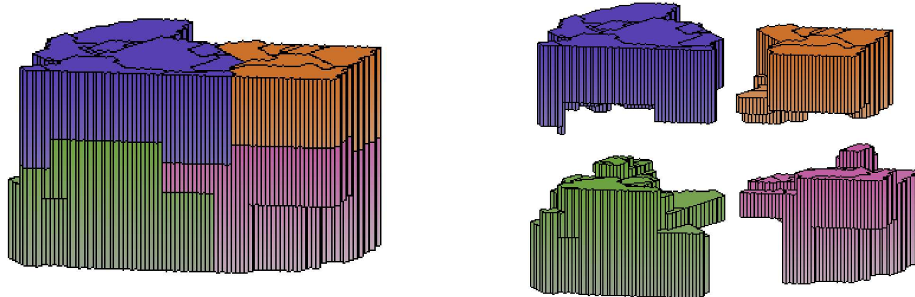

Figure 1.1: An example of configuration, and its decomposition into collapsed sectors [7].

an operation that both requires time and coordination, for this reason we must control the number of transitions and a degree of similarity between configurations involved. We can not impose a severe twist on the shape of sectors in a short time window, it would affect the performance of the operators. To this end, we focus on the concept of compatibility: we consider two configurations *compatible* when they present only small differences between each other. Thus, a major change could be performed only through a series of smaller ones.

Further constraints to be taken into account are related to the steadiness of the sequence of configuration to be operated, which limit frequent changes, as discussed in [9, 16]. We do not provide the details here, instead we report their considerations in Section 2.2.

We are ready to state a formulation for the Dynamic Airspace Configuration (DAC) problem. Assume to know in advance the air traffic demand, i.e. the list of routes and timetables of flights that demand to cross the airspace volume. Given the limited capacity, DAC asks to determine a sequence of compatible configurations that responds best to the distribution of routes, thus minimizing the excess of demand, i.e. the difference between the demand and the total capacity provided by a configuration.

In Chapter 2, we present different takes on DAC, given how the literature on the problem flourishes with diverse approaches. In particular, we will focus on [9], in which the authors describe the problem through a Integer Linear Programming (ILP) formulation, and touch on the possibility to represent it as a constrained shortest path problem on a weighted graph.

In Chapter 3, we describe some notions related to graph theory: we present the Resource Constrained Shortest Path Problem (RCSPP), and describe the basics about labeling methods.

In the following chapters, we collect our contributions. We devote Chapter 4 to shed some light on the complexity of the graph-based problem presented by [9]. In particular, we show that general values for the parameters it uses make the problem NP-hard, by reduction from the Travelling Sales-

man Problem (TSP). Additionally, we illustrate how the drop of a specific subsets of constraints leads to a polynomial resolution process, thanks to a graphical reformulation. Finally, we present a first adaptation of a method proposed in the literature for the RCSPP, with the intent to solve the graph formulation for the DAC problem. We conclude by highlighting the limitations of this solution method.

In Chapter 5, we improve the graph and label representation towards a more efficient optimization method. In particular, we focus on the relation between paths' labels on the graph and the DAC problem constraints, and we define dominance criteria between labels.

Chapter 6 is devoted to the implementation of the proposed optimization approach for DAC. It presents the pseudo-code of the required procedures and discusses some implementation details, in particular about the representation of the labels and its impact on the overall efficiency.

In Chapter 7, we suggest and present a refinement of the previous dominance criteria; additionally, we touch on possible heuristics.

In Chapter 8, we discuss the results from the implementation of our method in Python; we offer a detailed view on how we generated some medium-sized instances, starting from a collection of realistic ones proposed in [9]. We reserve Chapter 9 for conclusions. In the appendix, we provide the code of our work in Python.

Chapter 2

Previous works and mathematical formulations

Scientific literature outlines two main classes of problems: the first regards static formation of sectors, while the second focuses on dynamic alternation of configurations.

2.1 Literature review

We refer to [16] for an in-depth analysis of the literature regarding static and dynamic airspace configuration problems.

The static problem of designing sectors, also referred to as sectorization problem, involves designing sectors with constraints on shape, such as compactness, connectedness, and convexity, posing computational challenges. Two main approaches are used: the graph-based approach, which uses graphs to model intersections and segments of trajectories with Voronoi diagrams [21] to define sector borders, and the cell-based approach, which partitions airspace into smaller blocks combined into sectors [7]. Various optimization methods, including metaheuristics (e.g., genetic algorithms, local search [7]) and machine learning techniques (e.g., random forests [17]), are used to solve the problem and assess workload.

After determining the sectors, the configuration problem involves deciding which sectors to open to partition the airspace. Solutions include classifying airspace elements and using models like Integer Programming within a State-Task Network framework [14]. Methods to solve the configuration problem include genetic algorithms [18], machine learning (e.g., Neural Networks), and enumeration algorithms, such as branch-and-price [20] and coin-or-branch methods [13].

2.2 Mathematical formulations for the dynamic airspace configuration problem

Here we report two mathematical formulations for the Dynamic Airspace Configuration (DAC) problem, as proposed in [9], and partially in [16]. These formulations are respectively based on mathematical programming and graph theory. In this introduction we enlist the components common to the two approaches.

We presented informally the DAC problem in Chapter 1, introducing the the concepts of (elementary or collapsed) sector, capacity, traffic demand, configuration and compatible transitions.

We recall that the intent is to compute a sequence of compatible airspace configurations, minimizing the need for interventions (delays, deviations). Thus the objective is to minimize the excess of demand, with respect to the limited capacity of the airspace volume. We assume to know the traffic information in advance, and to possess a finite collection of achievable configurations. Reference [16] underlines the main advantage of considering a predefined collection of configurations: it grants the proposed approach the independence from the method used for the formation of configurations.

2.2.1 Relevant notation

Reference [16] considers a time-horizon discretized into small intervals, called time periods. The problem consists in the determination of a configuration to implement for each time period. This configuration is referenced as the *active* one. The decision process is also discretized: A configuration change is permitted only at discrete times, in correspondence of a time period.

We report the relevant notation:

- T is the set of decision time periods in which the time horizon is discretized,
- C is the family of airspace configurations,
- E_t^c is a non-negative parameter that measures the overload (or excess) of traffic demand in configuration c at decision time period t. If the demand is lower than the capacity, we set the value to zero.
- C^t denotes the set of configurations available at time period t
- C_c^{t+1} is the set of configurations that can be reached at time period $t+1$ if at time period t the active configuration is c.

The authors consider a 24 hours time horizon, divided into 5 minutes intervals. As a result $T = \{1, \ldots, 288\}$.

We now cover two aspects regarding steadiness. In order to limit frequent changes, the authors introduce the concept of *permanence*. Following a configuration change, the configuration must remain active for t_p instants. It satisfies $t_p \geq 1$, reflecting that a configuration can not remain active for less than 1 instant. Additionally, Reference [9] introduces the concept of *quiescence*. It represent a cool-down time to wait, before an abandoned configuration could be reactivated. It is referred by the symbol t_q. We ask that any interval of length $t_q + 1$ does not contain the same configuration in non-adjacent times. It satisfies $t_q \geq 2$, since a configuration can not be abandoned and reactivated in less time.

We note that imposing $t_p = 1$ and $t_q = 2$ would result in the constraints being automatically satisfied. For the purpose of our study, we consider $t_p = 3$ (15 minutes) and $t_q = 12$ (one hour), representing realistic time periods. Nonetheless in the following we are going to make some observations valid for general values.

We underline that the two constraints cover different aspects, but they are nonetheless related. In fact, whenever $t_q \leq t_p$ the quiescence constraint would drop, since it would be satisfied automatically. Imagine to abandon a configuration c at time t^*, in favor of c' at time $t^* + 1$; then for t_p instant a feasible solution will maintain active c', and the first return to c can occur starting from

$$t \geq t^* + t_p + 1 > t^* + t_q.$$

2.2.2 Integer Linear Programming model

Following [9, 16], here we state a formal ILP model for the problem at hand.

The decision variables form the following groups:

- x_t^c is a binary variable taking value 1 if configuration c is active at time period t, and 0 otherwise;
- s_t^c is a binary variable taking value 1 if there is a transition to configuration c at time period t (i.e., c is active at time period t and not active at time period $t-1$), and 0 otherwise.

The objective function sums the traffic overload over each sector. In [16] they consider a second term in the objective function, which penalizes configurations that use a large number of sectors, in order to penalize underload phenomena. We did not consider this aspect.

The model is the following.

$$\min \sum_{t\in T} E_t^c x_t^c$$

$$s.t. \sum_{c\in C_t} x_t^c = 1 \qquad \forall t \in T \qquad (2.1)$$

$$x_t^c - \sum_{c'\in C_c^{t+1}} x_{t+1}^{c'} \leq 0 \qquad \forall t \in T, \forall c \in C^t \qquad (2.2)$$

$$\sum_{\tau=t}^{t+t_p-1} \sum_{t\in C^\tau} s_\tau^c \leq 1 \qquad \forall t \in T \qquad (2.3)$$

$$x_t^c - x_{t-1}^c \leq s_t^c \qquad \forall t \in T, \forall c \in C^t \qquad (2.4)$$

$$x_t^c + \sum_{\tau=t+1}^{t+t_q} s_t^c \leq 1 \qquad \forall t \in T, \forall c \in C^t \qquad (2.5)$$

$$x_t^c, s_t^c \in \{0,1\} \qquad \forall t \in T, \forall c \in C^t$$

Constraint (2.1), combined with the hypothesis $x_t^c \in \{0,1\}$, imposes a feasible solution to achieve exactly one configuration per instant. (2.2) restricts the choices for the next configuration, it has to be chosen within those compatible to the current one. Constraint (2.4) state a connection between the variables x_t^c and s_t^c.

Equations (2.3) and (2.5) formalize the requests on permanence and quiescence.

2.2.3 Graph representation

As suggested in [9], the problem at hand can be represented by the means of a weighted graph, on which we aim to determine a path of minimum weight, subject to additional constraints.

Start by constructing the graph. Consider a pair (t, c), where time $t \in T = \{1, \ldots, 288\}$, and the configuration c is picked among C^t, i.e. those available at time t. For any formed pair, place a vertex, and arrange them in columns, by increasing time coordinate. We assume to identify as the same configuration (t, c) and (t', c), for any $t' > t$. Hence, in the following figures, same row will mean same configuration. In Figure 2.1, we give an idea on the shape of a transition graph.

Arcs connect exclusively compatible configurations, belonging to adjacent columns; also, arcs are always directed towards increasing times. Formally, we place an arc starting from (t, c_1) to $(t + 1, c_2)$ only if c_1 and c_2 are compatible. A natural consequence is that the resulting graph is acyclic. The cost of an arc represents the excess of demand, associated to the activation of the configuration the arc enters. Thus, every arc entering the same node shares the same cost.

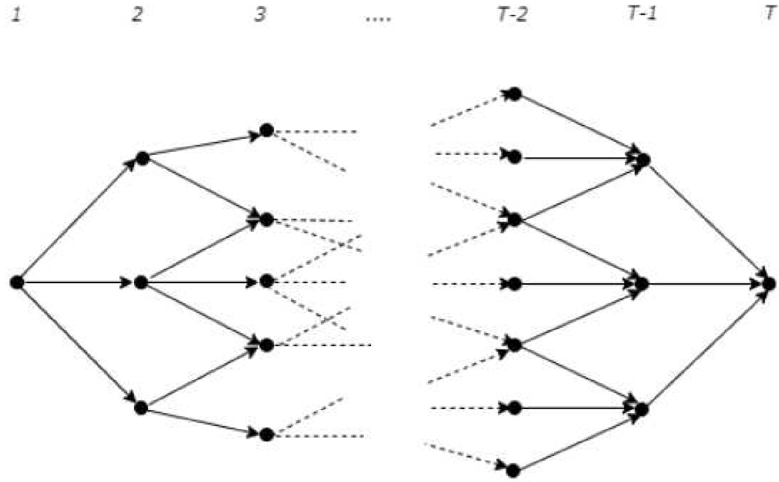

Figure 2.1: Example of a transition graph [16]

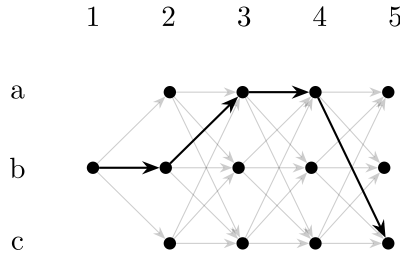

Figure 2.2: Assuming $t_p = 3$, permanence is violated

Assuming to arrange the same configuration at same height, one could easily represent which paths are feasible and which are not. Figures 2.2, 2.3 work to this end; we imagine grey arcs representing the transition graph and we highlight the arcs included in a certain path; letters and numbers placed, respectively, to the left and above the graph represents a coordinate system for the transition graph, made of, respectively, configurations and discretized time intervals.

Recalling the role of permanence and quiescence, respectively, one could reinterpret those in terms of the graph. Any time a path includes an arc that connects two different configurations, both the following must occur: the next $t_p - 1$ arcs have to connect the newly activated configuration to itself, Figure 2.2 violates it by changing too soon configuration at time 4 (assuming $t_p = 3$); the following $t_q - 1$ arcs can not reenter the abandoned configuration, this time Figure 2.3 repeat an abandoned configuration before the end of the period of quiescence (assuming $t_q = 12$).

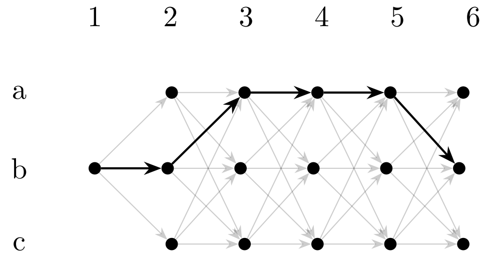

Figure 2.3: Assuming $t_q = 5$, quiescence is violated.

We observe that, ignoring constraints on permanence and quiescence, we can solve the DAC problem as an instance of the weighted shortest path problem, with non-negative cost on every arc.

Most importantly, we note that the addition of these sets of constraints make the resolution by SPP algorithms unfeasible: a said algorithm would consider only information on the cost of an arc, and could not include information on future behaviour of a partial path. It could be shown, by constructing ad-hoc instances, that any algorithm would violate at least one side constraint, by creating a graph with a zero-cost unfeasible path, see Section 3.2.1.

One could think of the possibility to transform the arcs of the graph, or the graph itself, in order to reduce the problem to a SPP instance. We cover some details about this topic in Section 4.1.1.

Chapter 3

Theoretical tools

We collect here some basics definitions from [2, 3, 19].

3.1 Fundamentals of Graph Theory

An *undirected graph* is defined as a pair of finite sets (V, E); where V is non empty, and E is a collection of unordered pairs $\{u, v\} \in V \times V$. The elements of V are called *vertices* or nodes, and *edges* those of E. $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.

Whenever needing to model a direction within a movement, one should make use of the notion of *oriented* graphs (digraph). in that case, we speak of *arcs*, instead of edges. A is formed by ordered pairs (u, v); u will be the *tail* of the arc, and v its *head*.

We define the *degree* of a vertex as the number of edges incident with that node. In case of ordered graphs, one should differentiate between *in*-degree, and *out*-degree.

Two nodes u, v in a graphs are said to be *connected* when there's a $u - v$ path in G. Connection forms an equivalence relation on V, thus a partition on the vertices. If G has exactly one equivalence class, the graph itself is said to be *connected*. Two nodes are defined as *adjacent* if there's an edge connecting them. The *adjacency list* of a node refers to the set of nodes adjacent to a given one.

A *path* is defined as a sequence of adjacent nodes $v_1 v_2 ... v_k$. The *length* of a path is defined as the number of nodes it connects. We define a *cycle* as a path $v_0 v_2 \dots v_k$, with $k \geq 3$, v_0,...,v_{k-1} all different nodes, and $v_k = v_0$. When a graph does not contain cycles, it is called *acyclic*. Any subgraph of an acyclic graph, is itself acyclic.

3.2 The shortest path problem

From [3] we recover the following description of the Shortest Path Problem (SPP). Associate to each edge e of G a real number $w(e)$, called *weight*. Given a subgraph H of a weighted graph, the weight $w(H)$ of H is the sum of the weights $\sum_{e \in E(H)} w(e)$ on its edges. Many optimisation problems amount to finding a subgraph of a certain type with minimum (or maximum) weight.

The SPP problem is well studied, and there exists numerous variations in the literature [2]. We will report exclusively an approach we found more related to our work.

3.2.1 Labelling algorithms for the shortest path problem

We want to focus on algorithms that update labels to solve the SPP, like the Dijkstra or the Bellman-Ford algorithms. We report here an informal explanation. We suggest [3] for a in-depth analysis.

Starting from a node s, we iteratively consider paths of increasing length. We associate to every node a numerical value, called a label, corresponding to the distance associated to the best path we found so far, since there could be more than one. We couple the nodes that are still to be reached with the value $+\infty$.

Up to completion, this process stores only an upper bound on the minimum distance; and the label associated to each node could vary during the process, when a better path is determined. Reference [19] states and proves an optimality certificate of an optimal path computed by the Bellman-Ford algorithm. Reference [3] does a similar thing with the Dijkstra method. Both these certificates obtained optimality of a partial path, based on optimality of the previous steps performed. They are proved by induction: we obtain optimality at step k, considering labels obtained at step $k-1$ and evaluates the effects the addition of an arc.

It is important to note that, at current stage, this method do not store any information about the path associated to the minimum cost. For this reason, it is important to associate every best value to a *predecessor* node. Doing so, we obtained a *one to all* best shortest path.

3.3 Constrained shortest path

The Resource Constrained variation of the shortest path problem (RCSPP) was studied by Desrocher in 1986 [11]. It finds application, for example, in the framework of the Vehicle Routing Problems (VRP). It is not uncommon to find papers that intertwine the two topics, e.g. [4, 5, 11].

The main side constraint, added to the original problem, come from the definition of additional cost components, each conditioned to a given bound.

In this context, an arc has associated multiple weights, one could express the collection of them with a vector.

The typical way these costs add up is linear, but [11] consider the possibility to use non-decreasing, or non-linear functions. We do not report the details; we cite it to justify the use of *max* and *min* functions, that we are going to make in Section 5.1.1, when defining the resource consumption update functions.

Even asking for the graph to be acyclic, and focusing on just one resource, the problem is known to be NP-hard [6, 8]; we report that [5] describes a pseudo-polynomial label-setting approach, valid whenever the update function for the resource consumption are linear.

The vast majority of literature seems to focus on the 1-resource problem, or on a generic set of resources with linear update functions. In a recent study [15], the most common techniques are presented and compared, those are: dynamic programming, Lagrangian relaxation and labeling methods. We focus on the latter, and give a brief presentation.

Literature also presents a refinement of the RCSPP, in which the optimal solution must not contain subcycles. This problem is called the Elementary Resource Constrained Shortest Path (ERCSPP). In general, it results harder to solve, confronted with the RCSPP, nevertheless, ERCSPP and RCSPP problems coincide whenever defined on an acylic graph.

3.4 Labels

Even limiting the scope to the SPP framework, there are different meaning for the expression *labeling*.

A *label* is a compact way of representing a path, it sums up the information about the consumption of each of the resources [4, 5]: specifically we identify a label l with the consumption of each resource $(cost, r1, \ldots, r_k)$. All the labels, associated to paths that reach a generic node v, starting from a given source, are collected in a list; every node has such a list, possibly empty. A label does not constrain the associated path in any way, it just collects data, such as unreachable configurations [4]. One could define a method that ignores or discards those labels that do not satisfy given criteria, when exploring the graph.

When proposing a labeling algorithm one should disclose the criteria regarding: the exploration of the graph, how those labels update when crossing an arc, and most importantly how those labels relate to each other [11].

3.4.1 Label setting, label correcting

We recall that even within the framework of the SPP the terminology "labeling method" is not a novel thing. For example, the Dijkstra algorithm

or the Bellman-Ford algorithm for the shortest path is often referenced as a *labelling* algorithm.

We derive from [2] a clear statement, regarding the difference between label setting and label correcting methods. Typically, the algorithmic approaches for solving shortest path problems are subdivided into two groups: label setting and label correcting. Both are iterative; and they vary on the method used to update the labels. The first class treats labels as permanent, once set in the course of iterations. In contrast, the other considers labels as temporary, until the final step iteration, when they all become permanent.

In short, label setting assigns a set of labels and an increasing subset of them will not be modified along the duration of the method; contrary to that, a label correcting could modify any list of labels, when a *better* one comes up.

Regarding the theme of better labels, and in general the process of ranking them, we should introduce the topic of dominance criteria between labels.

3.4.2 Dominance criteria

It should be natural to assume that storing all the labels is not an effective way to solve any problem, even if possible. One should define some criteria which grant the possibility to (at least partially) rank the labels [5]. Ranking the labels translates to the possibility of expanding exclusively those that are more promising [12].

In general, one should state how the labels should compare, for one to be more *desirable* than one another, thus we say for one to *dominate* the other. As an example, consider a minimization problem, and a single resource, with linear non-decreasing update rule, bounded to respect an upper limit. Assume now to possess two labels l_1, l_2; consider the relative pair $(cost, resource) = (c_i, r_i),\ i \in \{1, 2\}$. We prefer l_1 over l_2, or say that l_1 dominates l_2, if both the following are satisfied: $c_1 \leq c_2$ and $r_1 \leq r_2$.

Typically, when working with more than one resource, is not possible to think of a total relation order. Both [5, 12] define a lexicographical order, which is a total order. Most often than not, some pairs could not be compared, and to describe this situation we use the term *Pareto optimality.* Formally, it is the set of labels that are not dominated by any other. Other documents use the definition of *efficient* labels. This event of incomparability arises commonly, since it could happen that, given two labels l_1, l_2, each has a non-dominated component: e.g. $c_1 < c_2$ but $r_1 > r_2$, thus none is univocally better than the other.

We can also differentiate between *strict* or *loose* dominance, with the latter typically referring to cases in which two labels perform equally. We do not make this distinction, even in case of equality our method keeps just one of the two labels, i.e., we only consider strictly dominating labels.

Chapter 4

Complexity and resource-extended graph approach

The objective of the chapter is to begin tackling the graph formulation presented in Section 2.2.3. Our first task has been to determine the complexity of the problem, without any assumption about the *shape* of the graph: what configuration are present, for how long, how they relate etc.

4.1 Reduction from the TSP

We now show that, the Dynamic Airspace Configuration problem is NP-Hard. We will obtain this by reduction from the Traveling Salesman Problem (TSP).

Consider to this end a directed graph $G = (V, A)$, on which we aim to determine a minimum cost Hamiltonian cycle, according to an arc metric where c_{ij} is the cost a an arc (i, j) in A. Let us build a DAC instance as follows. Consider $T = \{1, ..., |V|, |V| + 1\}$, and $C_t = V$ as the set of configurations present for every $t \in T$. Suppose that the compatibility conditions, thus C_c^{t+1} for $\forall t \in T$, are defined by the arcs present in G. The cost of every transition is defined as the cost of an arc in the original graph. Ask now for $t_p = 1$, and $t_q = |V| - 1$.

Note that, in the DAC instance, a configuration, thus a node of G, could be maintained in consecutive times only if self-loops are present in G. This scenario could be avoided through a preprocess. Thus, once the method chooses a node at time t, it must select a different one at time $t + 1$. By the definition of quiescence provided in 2.2, any configuration visited at time $t > 1$ can not be reactivated. The unique node that could be repeated is the first one, at times 1 and $|V| + 1$.

As a consequence, if a solution were to exist for the DAC reformulation, we would obtain a Hamiltonian cycle: it can not contain sub-cycles due to the quiescence constraint. We underline the fact that in the event of every partial path being expandable exclusively towards prohibited configurations, closing a sub-cycle, the method would determine the absence of a feasible solution; that is, if TSP is infeasible, the corresponding DAC will be infeasible as well and viceversa.

We remark that the operation of reducing the TSP on the graph G to its DAC reformulation is a polynomial operation: it only needs to compute T, C_t, C_c^{t+1} and E_t^c. We imagine a black box computing the solution for DAC; finally, the solution for DAC is directly encoded into a solution for the TSP, without any computation in between.

4.1.1 Valid approach dropping quiescence

We disclose here what would be a graphical reformulation for the Problem presented in Section 2.2.3, if we enforce exclusively permanence.

For any arc of the graph, connecting (t^*, c) to $(t^* + 1, c')$ with $c' \in C^{t^*+1}$, we substitute the arc with a set of them, such that they satisfy by construction the permanence constraint, as detailed in Section 2.2. Consider an arc, between (t^*, c) and $(t^* + 1, c')$, and drop it. Now, starting from $t = t^* + t_p$, if c' is present (i.e. $c' \in C^t$), place an arc from (t^*, c) to (t, c'). Now consider $t + 1$, if c' is present in $C^{t+1} \cap C_{c'}^{t+1}$, place an arc from (t^*, c) to $(t+1, c')$. Continue this procedure, progressively augmenting t, up to the first moment in which $c' \notin C^t$. See Figure 4.1, where we highlighted an arc before the transformation, and the set of arcs we added to replace it; grey arcs can be ignored, they represent a portion of the other arcs of the graph.

The interpretation of an arc between (t^*, c) and (t', c') is: activate c' at $t^* + 1$, maintain it up to time $t' \geq t^* + t_p$. Thus, permanence is satisfied. It can happen to drop an arc from a node (t, c), without adding any back. It represent the impossibility for any partial path to spread and keep feasibility, changing configuration at (t, c).

The importance to insert a *set* of arcs lies in the necessity to model the possibility for a path to keep active a configuration longer than t_p instants. Meanwhile, we insert arcs at consecutive times, and stop when the configuration disappears, given the interpretation we stated before: it would not make sense to maintain a configuration active for a period of time, and in that interval having the configuration not achievable.

While the number of nodes would remain the same, we could deduce a large upper bound on the number of arcs that can be possibly added. Recall T from the notation of Section 2.2, let $n = \sum_{t \in T, c \in C^t} |C_c^t|$; we can not add more than $n \cdot |T|$ arcs, obtained by replacing every arc with $|T|$ copies of it. In reality, the number of additions would be much smaller, since the amount of arcs to add depend both on the time coordinate of a node, and the size

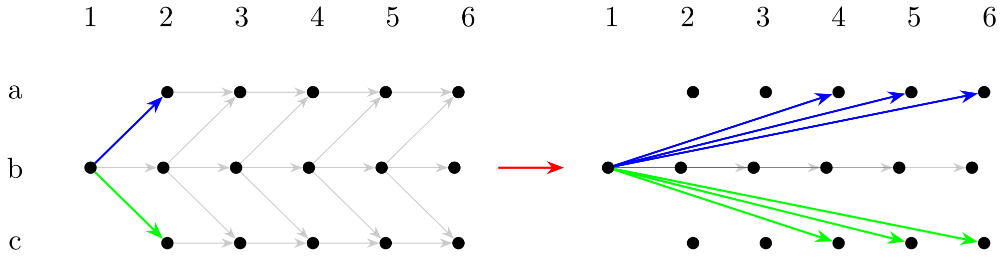

Figure 4.1: Example of reformulation of the arcs, when only permanence is enforced (not all arcs represented).

of its neighbourhood.

Concluding, The procedure adds at most a quantifiable number of arcs, and that it can be solved by an algorithm for the Shortest Path Problem in polynomial time. Thus, if we enforce exclusively permanence the problem is still polynomial solvable.

4.2 Resource-extended graph approach

Literature does not provide ready-to-use methods that could force a path to repeat a node, it would not make sense on a general graph, especially in the VRP framework.

Reference [12] studies a method to discard short enough cycles in the context of the $RCSPP$. It also recall previous studies, as [5]. This approach compromises between the $ERCSPP$ and the $RCSPP$ (see Section 3.3).

Our interest to study literature regarding cycle-elimination, comes from the possibility to drop the time component in the DAC formulation (as defined in Section 2.2.3), and imagine to work on a different graph. Nonetheless, this method works exclusively considering a set of resources with linear updates, and would not apply directly to our problem. Most part of literature focuses on linear update rules, thus making the suggested methods often inapplicable.

In the framework of vehicle routing problems, it is defined the concept of *mutual exclusion*: two nodes could not belong to the same path, for it to be feasible. We can not apply this concept, in order to mimic quiescence, as defined in Section 2.2, we would need a triplet of nodes. Excluding (t, c) and (t', c) for $t < t' < t + t_q$ is not sufficient, it includes the feasible scenario of a path keeping active c in the interval $[t, t']$; also, we cannot exclude (t, c') and (t', c) with $c' \neq c$, since it includes the feasible scenario of a path changing the active configuration. We need to forbid paths to assume a triplet of nodes: (t, c), $(t + 1, c')$ with $c' \neq c$, and (t', c) with $t + 1 < t' < t + t_q$.

4.2.1 Definition of linear resources

Here we define some resources in details, specifying the respective cost for every arc of the graph. We will show the limitation of this approach and abandon it shortly. Consider the following: $perm_{t,c}$ and $quies_{t,c}$, that represent respectively permanence and quiescence, see Section 2.2. Focus on the first collection of resources, we set the resource cost to be: $t_p - 1$ for any arc that enters the node (t, c) from a different configuration, i.e. any node of the type $(t-1, c')$ with $c' \neq c$; -1 for every arc connecting (t, c) to its $t_p - 1$ future copies, i.e. $(t+s, c)$ with $0 \leq s \leq t_p - 1$; finally 0 elsewhere. The upper bound we fix for this resource is 0, forcing a path to cross all the $t_p - 1$ arcs with cost -1, whenever a new configuration is achieved. Refer to Figure 4.2, we highlight the arcs that have an impact (positive or negative) on the resource consumption; we assume $t_p = 3$, thus only two arcs connecting b to itself are highlighted. We imagine grey arcs to be all the arcs of the transition graph, with zero cost.

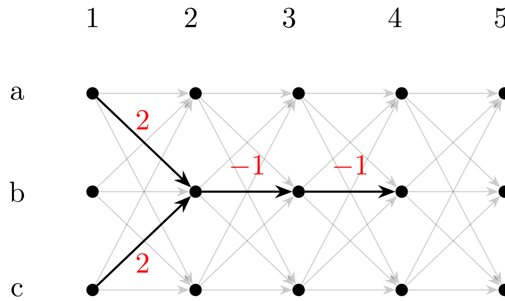

Figure 4.2: Linear resource consumption - permanence. $t_p = 3$.

The second family of resources to be defined will have cost +1 for any arc that connects (t, c) to any other configuration $\tilde{c}$ at $t + 1$, and also cost +1 for any arc that in $\tilde{t} \in \{t+1, \ldots, t+t_q-1\}$ enters c from a different configuration, 0 otherwise. The bound value is +1, forcing any feasible path to include at most one of these arcs, see Figure 4.3.

We want to underline the importance of time dependence of these resources: we distinguish multiple resources for the same configuration at different times. If we were to drop this consideration, the consumption check would be performed on the *totality* of a path, eventually leading to undesired scenarios. It could happen that a path violates a constraint, while managing to satisfy the cost condition, by some opportune choice in multiple times. An example could be a path that abandons prematurely a configuration, thus violating permanence constraint, however managing to satisfy the cost condition, by keeping the configuration active more than needed in a later moment, as shown in Figure 4.4; we imagine the black arcs representing said unfeasible path.

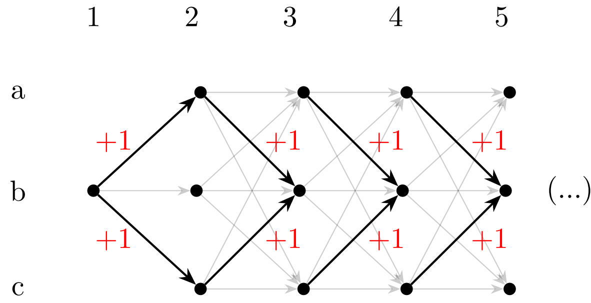

Figure 4.3: Linear resource consumption - quiescence. $t_q \geq 4$.

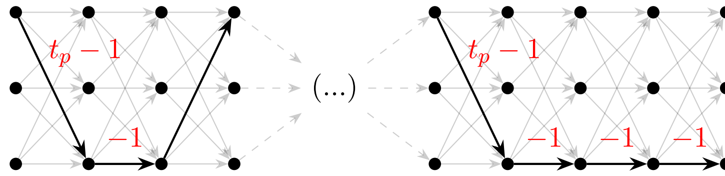

Figure 4.4: unfeasible scenario, accepted by cost conditions, if time component were to be discarded.

On the contrary, the distinction based on time does enforce the contributes to be adjacent.

Chapter 5

A label-based approach to the Dynamic Airspace Configuration problem

In Chapter 2, the DAC problem has been formulated as a shortest path with additional constraints on an acyclic graph. In this chapter, we explore a method based on defining resource-related labels. Labels plays the role of storing all necessary information to build feasible and optimal paths. Imagine to follow a partial path and ask what knowledge is vital, in order to respect the constraints? We remark that we have to answer given only information collected up to the current node, and keeping in mind that the more data is stored, the more expensive the computation becomes.

This approach is not necessarily better than the one presented in Section 4.2.1, we are trading a large number of resources with a huge number of partial paths that spread from a initial node. Our objective is to develop and test some effective pruning strategies, that will permit to solve the problem efficiently.

5.1 Label representation of the added constraints

As introduced in Section 3.4, in order to talk about labels optimization, we have to define what is a label for the problem at hand, and how to compare them.

Referring to the notation of Section 2.2, every label is associated to a partial path on the transition graph. We enlist the components of a label:

- *cost* is the total exceeded demand, we sum the cost on the arcs forming the partial path associated to the label;
- *perm* $\in \{0, \dots, t_p - 1\}$ represents the number of times a partial path

has to repeat the last node visited, before a feasible change of configuration;

- $qu \subseteq T \times C$, is the collection of deactivated configuration in the last t_q moments; it stores pairs formed by time of deactivation and abandoned configuration.

We present an example of a label, associated to a feasible path highlighted in Figure 5.1. We start from a label with zero cost, maximum permanence minus one ($t_p - 1$, reflecting the fact that we already kept the current configuration active for a time interval) and empty quiescence set; additionally we assume $t_p = 3$, $t_q = 12$ as in Section 2.2. In short, we represent the label with $l = (0, 2, \emptyset)$, associated to the node $(1, b)$.

Since we are considering a single partial path, we refer to the the label associated to any stage of the path with l; thus we present how l changes when the partial path traverses an arc. We state here any time a partial path crosses an arc, the label representing the path is associated to the last node of the path, with a suitable time component, i.e. the length of the corresponding partial path.

We consider to traverse the arc from $(1, b)$ to $(2, b)$, l becomes $(3, 1, \emptyset)$. We note that the other arcs, connecting $(1, b)$ to $(2, a), (2, c)$ can not be traversed, since we have a non-null value for permanence.

We repeat the operation with the second arc of the path, obtaining $(5, 0, \emptyset)$. Now the label have a zero value for permanence, and it is eligible for expansion towards different configurations.

We decide to traverse the arc that connects $(b, 3)$ to $(a, 4)$. Permanence is set again at $t_p - 1 = 2$, and now the quiescence set equals $\{(3, b)\}$. In short, $l = (10, 2, \{(3, b)\})$. The remaining operations are similar to the first two. We conclude with the label $l = (17, 0, \{(3, b)\})$ associated to the node $(6, a)$.

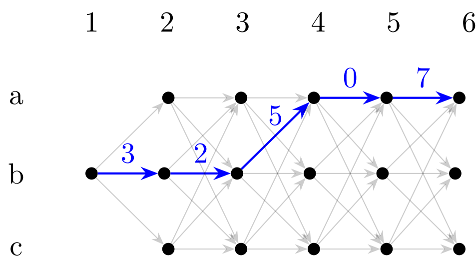

Figure 5.1: Example of a feasible path

The idea of defining sets in labels is inspired by the work of [4]. In it, the authors define sets of unreachable nodes, each associated to a different path. Ultimately those paths are compared, preferring those that has less

unreachable nodes. A similar approach is suggested in [11, 12] for effective pruning.

5.1.1 Cost and permanence update

The cost update is simple: sum to the current cost component the weight of the arc to traverse. The other two components have a non-linear update rule, as considered by [11]. Precisely, let $l = (cost, perm, qu)$ a label associated to the node (t, c), consider firstly the permanence update.

$$perm_{t+1,c'} = \begin{cases} \max\{perm_{t,c} - 1, 0\}, & \text{if } c' = c \\ t_p - 1, & \text{if } c' \neq c \end{cases} \tag{5.1}$$

This way, we obtain a deceasing effect when a partial path keeps a given configuration, and we reset the timer whenever a transition occurs. We recall that from the moment in which the value reaches zero, the partial path could spread to different configurations. It is irrelevant to know exactly how much time has elapsed from the activation of the current configuration, as long as the permanence constraint is satisfied; thus, it is natural to make use of the function *max*.

5.1.2 Quiescence update

For the quiescence, we add to $qu_{t,c}$ the pair (t_{exit}, c), whenever we abandon a configuration in favor of a different one. While a node is present in the set, the associated configuration can not be reactivated, thus the path could not expand towards the corresponding nodes. Clearly, it is of no use to maintain the list of *every* configuration a label has came across, and abandoned. After some time, some old configurations could be legitimately reactivated, and it may become disadvantageous to drag outdated information.

We want to determine a formula, which can describe the *obsolescence time* of the quiescence constraint, i.e. the moment in which such constraint is validated, from that moment onward it will always be valid. The use of this concept will be determinant in future parts of the work, as we will see in Section 7.2.2.

The determination of configurations towards which a partial path could sprout is carried out at time t. Thus, for a path to enter a precise node at time $t + 1$, it is necessary for the (eventual) related quiescence constraint to be considered obsolete, hence discarded, up to time t. At given time t, a constraint associated to the node $(t_{exit}, *)$ is to be considered obsolete if and only if

$$t \geq t_{exit} + t_q \tag{5.2}$$

Let us follow through with an example. Assume t_{exit} of a given configuration is 0. Then a path could revisit that configuration at $t = t_q + 1$. As a

consequence, the formula assess the quiescence constraint to be obsolete at time $t = t_q$, and valid for any $t \leq t_q - 1$.

Furthermore, what does it mean to update the quiescence, and when does it happen? We speak about updating quiescence, when we perform a check aimed to determine if some quiescence constraints have become obsolete; consequently they are discarded. This operation is performed for every label active at a given time. In the following definition it will be implicit.

Finally, imagine to traverse an arc connecting (t, c) to $(t+1, c')$; we can write

$$qu_{t+1,c'} = \begin{cases} qu_{t,c} & \text{if } c' = c \\ qu_{t,c} \cup \{(t,c)\}, & \text{if } \tilde{c} \neq c \end{cases} \tag{5.3}$$

5.1.3 Components for recovering the optimal path

Similar to section 3.2.1, we do not store in every label the associated path traversed, rather just a few information about its parent label. Given the fact that every node has attached a list of labels, we need to store multiple parameters to identify univocally the parent. We make use of three components: the position *pos* in the list of labels, the parent node $prtNode$ and the position of the parent label $prtLbPos$, within the list of labels associated to $prtNode$. The position *pos* of a label is used for identification.

The way the optimal path is recovered is simple: once arrived at termination, sort the labels by cost. Then, construct a path backwards, following the directions of $prtNode$ and $prtLbPos$. Up to now, there are no rule on which path to prefer in case of ties; Reference [16] defines a secondary cost, preferring configurations formed by less collapsed sectors.

One could ask if it is possible to recover the position of a label, based exclusively on the set of information stored in the labels. In fact, one could travel backwards the graph, subtracting every time the cost of a node. We recall that the cost of a node does not depend on the arc chosen. And one could observe that the permanence attribute would communicate when a transition occurred, and the quiescence would reveal the abandoned configuration. The real problem arises when we want to reconstruct a path that kept configuration active for a long period of time. Both permanence and the quiescence would not provide information after some time. Thus, traversing backwards we could have to manage a label that does not communicate the exact moment of transition, and its origin, thus we would be open to interpretation on the source of the transition.

5.2 First set of dominance rules

Surely one could not expand to termination all partial paths, due to obvious limitations of the computational resources (testing showed that the code could not reach termination even after eight hours of computation); it is thus fundamental to determine some rules which grant the possibility to discard non-optimal paths, and preserve the optimality of the solution. Here few simple criteria will be discussed.

5.2.1 Dominance relations between label components

Since labels are composed of multiple components, in order for a label l_1 to dominate another label l_2, a series of conditions must occur.

The cost condition is the easiest to allocate in this context: we ask for $cost_1 \leq cost_2$. A similar condition describes the rule for the permanence: we demand

$$perm_1 \leq perm_2 \tag{5.4}$$

The relation portrays the possibility for label l_1 to possibly spread earlier, having to wait less time. The conditions on quiescence are double, and the latter explains the need to store the deactivation time of each configuration. Firstly, we ask

$$C(qu_1) \subseteq C(qu_2) \tag{5.5}$$

where $C(qu_i) = \{c \in C : (t, c) \in qu_i \text{ for some } t \in T\}$; it translates to the request of l_1 having the same, or smaller, set of configurations that can not be reactivated. Then we observe that the time of exit is a crucial information, since it determines the moment a given configuration could be reactivated. Thus we determine the following condition

$$time_{j,1} \leq time_{j,2}, \ \forall j \in C(qu_1) \cap C(qu_2) \tag{5.6}$$

where $time_{j,i}$ refers to the time of deactivation of the configuration j, according to the label i. Since $C(qu_1) \subseteq C(qu_2)$, it would be equivalent to consider $j \in C(qu_1)$.

5.2.2 Utility of every domination condition

We go through a few toy-examples, in order to give a visual grasp on the role of each of these rules. These scenario are unavoidably artificial, but they aim to show how certain paths could become unreachable, following an incorrect pruning, due to imprecise rules.

Imagine to compare two labels l_1, l_2 with $cost_1 < cost_2$. If we were to drop one between the permanence (5.4) or first quiescence checks (5.5, we would obtain the undesired scenarios depicted in Figures 5.2, 5.3.

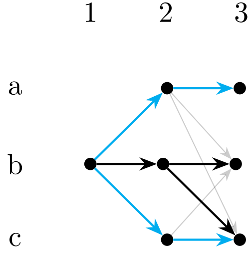

Figure 5.2: permanence check dropped

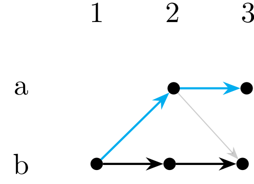

Figure 5.3: quiescence check dropped

In Figure 5.2, we imagine $cost_1 < cost_2$, and $0 = p_2 < p_1 = 1$, $qu_1 = qu_2 = \emptyset$. Thus l_2 could spread from the initial node, and l_1 can not. Without the check on permanence (5.4), l_2 would be considered dominated, thus discarded. We highlight in cyan the arcs that the partial path exclusive for the partial path associated to l_2. We note that with following these pruning, node $(3, a)$ is non reachable by the method.

In Figure 5.3, we imagine the two labels satisfying $cost_1 = cost_2$, $perm_1 = perm_2 = 0$ and

$$\emptyset = C(qu_2) \subset C(qu_1) = \{1\}$$

without the check on quiescence (5.5), l_2 would be considered dominated and discarded. thus the solution could not reach the node $(2, a)$ and $(3, a)$.

Finally, and most importantly, consider two labels such that $cost_1 = cost_2$, $p_1 = p_2$, $C(qu_1) = C(qu_2)$ but there exists an element $c \in C(qu_1)$ such that $t_{c,1} > t_{c,2}$. Without the third check, the two labels would be considered equal, and one of the two would be discarded based on arbitrary criteria. We can construct instances that (after some time) recreate a scenario in which $C(qu_1) \nsubseteq C(qu_2)$, thus the label we kept has a strictly smaller set of configuration to expand towards.

This observation demonstrates that if two labels do not satisfy all the conditions of the dominance criteria, even if they performs equally at a certain node, there exists examples in which one label could result strictly better than the other in future times; thus, it underline the importance to consider collectively all the conditions of the dominance criterion.

These rules check the *possibility* for a partial path to conveniently spread in the future, and are extremely conservative. In Chapter 7, we state some observations that permit to form more grounded domination criterion: we use little information about the future of a partial path, in order to investigate the real possibility for a label to produce optimal paths.

5.2.3 Properties of the dominance criterion

Implemented collectively, these rules grant that whenever a label is discarded, there is at least another label with minor cost, possibility to spread sooner, and equal destinations to spread towards. This consideration follows as a consequence of the domination criteria being a partial relation order.

First thanks to the following lemma, one obtains that deactivation times are univocally identifiable, given a label.

Lemma 5.1. *A configuration could not appear more than once in the quiescence component of a label.*

Proof. Suppose that a configuration appears twice; hence the configuration has been deactivated two times, before the previous of the two occurrence could be eliminated by obsolescence. This means that the configuration has been reactivated, while still being classified as prohibited; concluding an absurd. □

Lemma 5.2. *Furthermore, the relation dominates forms a partial order on the set of labels associated to any node.*

Proof. The relation is clearly *reflective*. We prove *anti-symmetricity*. Let l_1, l_2 two labels, such that $l_1 \leq l_2$ and $l_2 \leq l_1$. Then $cost_1 = cost_2$, since both $cost_1 \leq cost_2$ and $cost_1 \geq cost_2$ are true. The same could be showed for permanence. Regarding quiescence, similarly it follows $C(qu_1) = C(qu_2)$; for every $c \in C(qu_1)$ it again follows $t_{c,1} = t_{c,2}$ $\forall c \in C(qu_i)$. The two labels equate each other on any component, thus $l_1 = l_2$ by the definition we stated in Section 5.1. Focusing on *transitivity*, let l_1, l_2, l_3 labels associated to the same node, such that $l_1 \leq l_2 \leq l_3$. It follows $cost_1 \leq cost_2 \leq cost_3$, and a similar argument will prove both $perm_1 \leq perm_3$ and $C(qu_1) \subseteq C(qu_3)$. Finally, for every $c \in C(qu_1)$, it is true that $t_c, 1\ leqt_c, 2 \leq t_c, 3$. Thus $l_1 \leq l_3$.

Regarding the partiality of the order, one could think of the following example. Consider two labels, l_1 and l_2, with $cost_1 < cost_2$ and $perm_1 > perm_2$. Regardless from the quiescence, they are not comparable. □

An important consequence of Property 5.2 is that the order in which one compares the Labels is irrelevant, at least from a theoretical point of view.

The authors of references [5, 12] suggest the definition of a *lexicographic* total ordering of the labels. This ordering consists in asking equality for all resources up to a certain index i^*, where one is strictly smaller than the other, and no requests for $i > i^*$. This could not apply directly in our case, since it would contradict the second condition on quiescence.

Chapter 6

A constrained shorted path approach for dynamic airspace configuration

In this chapter, we propose an algorithmic approach to DAC that stems from graph formulation presented in Section 2.2.3, the representation of permanence and quiescence constraints discussed in Section 5.1 and the dominance rule described in Section 5.2.1.

6.1 Description of the proposed method

We present the algorithm we implemented for the resolution of the constrained shortest path problem that formulates DAC, as detailed in Section 2.2.3.

Reference [11] details the components of a general dynamic programming RCSPP algorithm. In particular it enlists the following steps: initialization, path extension, dominance step, and finally a filtering step. *Initialization* refers to the definition of a single label from which the method starts; *path extension* means to consider all the partial paths, in some order, and expand them towards any feasible destination; the *dominance* step is a process subsets of labels are compared, consequently only the most promising are maintained. Finally the filtering step recovers a feasible solution. Furthermore, reference [11] collects several remarks, e.g. the importance of effective dominance criteria, how their definition strongly depends from the description of the problem at hand.

In Algorithm 1, we present our scheme to approach the problem. First of all, we compute the transition graph (thus C^t, C_c^{t+1} $\forall t \in T$) and costs for every transition (E_c^t). This operation depends on two parameters: *graph*, *traffic*; given the fact that the construction of transition graph is independent from cost of transitions; see Section 8.1 for a detailed description on how

instances are built.

After the initialization of the first label, we scroll the values for time $t > 1$ (line 4), here we recall $T = \{1, \dots, 288\}$. For each value of t considered, we skim the achievable configurations, i.e., C^t. For every configuration $c \in C^t$, we decide to expand the labels associated to $(t-1, c)$, when present and connected to (t, c). Thus we add the expanded labels to (t, c). In Algorithm 1, this process corresponds to Lines $5 - 9$. Consequently, consider every configurations $c \in C^{t-1}$. We want to expand the labels associated to c, towards every compatible configuration $c' \in C^t_c$. We recall that the process of expansion depends on the destination (see Section 5.1.1,5.1.2): the cost varies with c', and not every transition is feasible for every label. We have perform two checks: if $c' = c$ we already computed the expansion operation; additionally, inside the expansion process, we are going to determine the list of labels feasible for expansion towards c'. Refer to Lines 9-17.

In Line 12 of Algorithm 1, we use the command *skip*. It commands to ignore the following operations, and jump to the next iteration of the *for* cycle; in this case it means to consider the next c'.

Algorithm 1 Master procedure

Require: graph,traffic $\in \{0, \dots, 10\}$

Ensure: optimal solution for the RCSPP, or certificate of unfeasibility

```
1: Prepare data: depending on the values of graph and traffic, recover
   C^t, C^t_c ∀t, c. Compute cost (excess of demand) for each transition.
2: Initialize the labels.
3: for Every instant t > 1 do
4:     for Every conf c ∈ C^t do
5:         if c ∈ C^{t−1} and is connected to (t, c) then
6:             Expand labels from (t − 1, c) and add them to (t, c)
7:         end if
8:     end for
9:     for Every conf c in C^{t−1} do
10:        for Every compatible destination c′ ∈ C^t_c do
11:            if c′ = c then
12:                Labels already expanded. skip
13:            else
14:                Expand labels from (t − 1, c) and add them to (t, c′)
15:            end if
16:        end for
17:    end for
18: end for
19: Recover the minimum cost path, feasible by construction.
```

Thus the exploration method is breadth first: it performs every possible operation at time t, before considering $t + 1$. The expansion procedure

consider every possible destination. The domination step is included in the expansion procedure. Finally, a optimal path is recovered with the standard procedure described in in Section 3.2.1.

In the following paragraphs, we describe the components regarding the computation of the necessary data, thus the initialization and expansion processes for the labels.

6.1.1 Data preparation

The block of functions corresponding to Line 1 of Algorithm 1 has been provided by the authors of [9]. We recall that configurations are composed of smaller components, namely elementary and collapsed sectors.

We imagine to know in advance the capacity and traffic for each collapsed sector. From this knowledge, the excess of demand of each configuration is computed, at every time, as the linear sum of the excesses for each collapsed sector composing it. How capacity and excesses are built from elementary sectors up to collapsed sectors is thoroughly explained in Appendix A.

6.1.2 Labels initialization

Aiming to develop a general algorithm, we considered the possibility for the graph to start (resp. end) with more than one configuration. For this reason, we define two *dummy* configurations, that have no real-world translation. Their only purpose is to connect multiple nodes to a single source (resp. sink).

We start from the node $(0, InitialDummy)$, considering a single label with: zero cost, zero permanence, and empty quiescence set. The first action to perform is to add and traverse an arc, from the initial dummy configuration, to every adjacent node. The cost of this newly added arcs corresponds to the excess of demand at time one of each configuration. Similarly, at the end of the day, we want to collect all labels in one place, in order to compare them. Thus, we will repeat the process with the final dummy configuration, however now the cost will be 0 on every arc.

We note that between time 0 and time 1, the dummy configuration will enter the quiescence set. This will produce no effect on the path search, since the dummy configuration does not appear in the near future, and it appears in all the label. Meanwhile, the permanence value will be set to $t_p - 1$, coherently with the request to wait an appropriate time before the first transition.

Up to now there are no constraints regarding the final values of permanence or quiescence, so we sort labels based exclusively on cost.

6.1.3 Label expansion

Given (t, c) as starting point, and $(t+1, c')$ as destination, what we ask the expansion function is to expand and subsequently funnel the labels, through a series of controls, returning only those non dominated. Label expansion is given by Algorithm 2.

Algorithm 2 Expansion process

Require: time, cc: current conf, nc: next conf
Ensure: A list of expanded, non dominated, labels

```
1: Make a copy of the labels
2: Discard labels that could not be expanded towards nc, recalling function
   IsExpandible (Algorithm 3)
3: Expand remaining labels, by recalling for each an expansion function,
   pass as parameters t, cc, nc; define ExpandedList the list of updated la-
   bels.
4: if |ExpandedList| > 1 then
5:     for l_1 ∈ ExpandedList do
6:         for l_2 ∈ ExpandedList, pos(l_2) > pos(l_1) do
7:             Confront the two
8:             if l_1 ≤ l_2 then
9:                 Discard l_2; skip to the next l_2
10:            else
11:                if l_2 < l_1 then
12:                    Discard l_1; skip to next l_1
13:                end if
14:            end if
15:            The two are incomparable; next l_2
16:            Execute isPareto(t+1. l_1, nc)
17:            if Label is Dominated then
18:                Discard l_1
19:            end if
20:        end for
21:    end for
22: end if
23: Return the (possibly empty) Pareto efficient list of labels
```

The first inspection Algorithm 2 performs is to control which labels are eligible for expansion, refer to Line 2. This operation is performed for each label individually, running Algorithm 3 on the label. We recall that a label is always expandable towards its current node at time $t+1$, when present. Thus, we apply Algorithm 3 by passing as argument the next configuration and time. We note that whenever the function *return* is executed, further operation are not considered. With this fact, we simplified the if-else state-

ments, e.g. from Line 2, it follows that $nc \neq cc$.

Algorithm 3 Check feasibility for expansion

Require: cc: currconf, nc: next conf
Ensure: True if the label could expand towards nc, False otherwise.
1: **if** Next and current configurations coincide **then**
2: Return $True$
3: **end if**
4: **if** $Perm(label) > 0$ or $nc \in Quies(Label)$ **then**
5: Return $False$
6: **end if**
7: Return True

We return to Algorithm 2. Line 3 will perform operation described in Section 5.1.1. Starting from Line 4, we compare every pair of labels eligible for expansion, to spot eventual dominated ones. We note that, since the domination criteria are well posed, we can skip a portion of operations whenever a label is classified dominated. In fact, it is not useful to determine what other labels it does dominate, or is dominated by, since we already found a better one. Thus, we detail the scenarios of Lines 9, 12 of Algorithm 2. If l_1 dominates l_2, we discard it; equivalently, we can ignore any possible future operation regarding it: such as checking if $l_2 \leq l_1$. If l_2 dominates l_1, we discard l_1, and do not compare the remaining labels against l_1; those labels are going to confronted against the current l_2 in the future.

In Lines 8,11 of Algorithm 2, we perform a check on the domination between l_1 and l_2. In the implementation we will se in B this check is performed by a function *Dominates*. We do not detail its definition, since we have already extensively discussed the domination criteria in Sections 5.2.1, 7.1, 7.2. We defined the function *Dominates* to exclusively check if the first dominates the second, and not viceversa. The motivation is that, in some cases, we are almost certain that a label could not dominate the other, and we prefer to avoid computing operations of which we anticipate the result.

Brief mention goes to the function *isPareto* called by Algorithm 2 at Line 16, its role is to confront a label in hand with all the labels that have been already added to a node. In such a way, we could decide to expand exclusively labels that we know already determined being non dominated.

We note the independence between the checks performed on Pareto efficiency, Lines 16-19 of Algorithm 2, and the one performed by scrolling in pairs all the labels, Lines 5-15. they could be rearranged, with equal (theoretical) results. In our code, due to implementation needs, we decided to separate the two in different cycles.

6.2 Algorithm insights arising from implementation issues

The goal of this section is to detail the decision process regarding the formation of the code we presented. The order of operations may have resulted unintuitive, thus we want to enlist some consideration related to implementation issues.

6.2.1 A consideration on paths derived from different configurations

We had to balance opposite approaches as: update all the labels, and let the dominance check (Algorithm 2, Lines 4-15) take care of it later, opposed to update a single label at a time, and checking right away if it is non dominated by any label on the associated node. We opted for a hybrid approach, having in mind both these goals: bound the number of checks regarding dominance, and keep the code readable, without jumps between operations.

We recall that when a path changes configuration, the permanence is reset to $t_p - 1$, and the abandoned configuration is added to the quiescence set, as stated in Section 5.1.1. Thus, we outline two scenarios that will lighten the domination step, to consider when a general algorithm compares two labels, associated to the same configuration c, with different parent nodes.

Assume to possess l_1, l_2 associated to the same node (t, c). Assume that the parent node c' of l_1 is different from c, i.e. the path associated to l_1 just changed configuration. Instead, Assume that l_2 is derived from a path that repeated the configuration c, i.e. the parent node of l_2 is the current configuration. As a consequence, the permanence value of l_2 is strictly smaller than the one associated to l_1. Thus, l_1 can not dominate l_2.

Consider again l_1, and assume to possess a new label l_3, both r to related (t, c); furthermore, assume l_3 is derived from $c'' \neq c'$. By recalling the rules for expansion, it follows that both labels contain, in their respective quiescence set, the parent node associated to time $t - 1$. As a consequence, neither of the two can dominate the other, since they could either: contain different elements in the quiescence set, or the same elements at different times. The explanation is simple: in the first case, they would violate the first condition on quiescence (5.5); meanwhile, in case the quiescence sets equal each other, they would contain both c', c'', and violate the second condition on quiescence (5.6).

In our algorithm these consideration are implicitly included in the way the functions are placed. We decided to first expand paths (labels) that maintain the configuration (Algorithm 1, Lines 5-8), and only subsequently proceed with other operations. In particular, we compare labels with the

same origin in Algorithm 2, Lines 4-15,20-22; subsequently, the labels that we want to place in (t, c) and are derived from $c' \neq c$, are compared with labels derived from $(t-1, c)$ in Algorithm 2, Lines 16-19.

It is important to remark that this observation does not affect the exactness of the approach. Consider two labels not compared by this observation at time t, and associated to the same configuration c; they are going to be compared at $t+1$, as labels originated from the same configuration. Reference [11] makes a couple of remarks on waiting some time before starting comparing labels, thus postponing operation as in our case. In particular, it states that it may be convenient to compute it exclusively when there is a guarantee on the number of labels that would be discarded; we preferred to compute domination check at each moment, but find this suggestion important for future development.

6.2.2 Some coding issues

During the phase of coding, we encountered some obstacles, due to Python's nature.

Recall Algorithm 2 line 1, we asked to make a copy of the labels, we want to cover the motivations behind the operation. In general, we do not want the list of labels associated to (t, c) to be modified, by any of the operations involving them, e.g. the expansion procedure. Instead, We would like to compute and work on a second support list. By python construction, this operation is possible through the use of the function *deepcopy*, which create an exact replica of any object; meanwhile other methods did not work as intended.

A major drawback of the deepcopy function is the time it asks to perform a task, especially on large objects. Since the expansion of a label varies with the destination, one could have to repeat this operation as many times as the number of possible transitions.

Fortunately, recalling how the update method works, we should note that the update of permanence and quiescence does not depend on where an arc is directed; the process of expansion cares exclusively to differentiate if the current configuration is repeated or not.

Thanks to this observation, one could solve the inconvenience by defining up to two replicas of the list of labels associated to a node (t, c): one time if the current configuration $c \in C_c^{t+1}$, and another time for all the remaining configurations, i.e. $C_c^{t+1} \setminus \{c\}$, considered collectively; Performing such approach, the method would not have to repeat the process of deep-copying for each destination. The scheme would be the following: starting from (t, c), choose any compatible destination $(t+1, c')$ with $c' \neq c$; derive the labels with new components (cost, permanence, quiescence), and add the updated labels to the considered configuration c'. Then, subtract to all these so-formed labels the cost of the transition. Repeat the operation with

a new compatible configuration. Note that quiescence and permanence do not change.

6.2.3 Focusing on past relations instead of future ones

Following the considerations of Section 6.2.2, we want to discuss our choice to focus on past compatibilities.

It would be natural to develop a method that, given a node (t, c), consider its labels and expand them to any future neighbour $(t + 1, c')$ $\forall c' \in C_c^{t+1}$, without considering -for the moment- the domination checking. Nonetheless, this approach does not take in mind the consequences of the Section 6.2.1. To this end, we want to collect some considerations on the advantages of an *elitist* strategy when expanding the labels. Thus to define a preference on which labels to prioritize in the expansion process.

A future-oriented method may not have derived the *best* labels for a given configuration, and have already added labels that may end discarded in a few moments. By the contrary, an elitist perspective would avoid this scenario, considering as first thing more promising collections of labels over others.

In order for a method of this type to cohabit with the considerations on deepcopying, we underline the importance of defining two different for cycles, as in Algorithm 1 (the first: Lines 5-9, the second: Lines 10-18). In the first one, a proposed method would have to deepcopy exclusively labels on arcs with destination equal to origin. Then, in the second cycle, the method could make use of the procedure described in Section 6.2.2.

6.2.4 Flagging dominated labels

In Algorithm 2 we proclaimed the intent to discard a label that has been classified dominated. In practice, we implemented a list *isDominated*, composed of *Bools*, of length equal the number of labels to consider. In the i-th position, it stores the information regarding the respective label. We are going to outline in a short while our take on labels structures. For now, just consider that the code stores them in a ordered lists, thus the concept of position is well defined.

Following the definition of such list, it arises the necessity to compute and store the position of a label; this could be performed in two ways. The first is to cycle through the labels and *compute* the position with builtin commands, e.g. *index.* in Python. The second is to *store* the value, using a counter that has to be updated accordingly to each plausible scenario.

We implemented the second approach, since we thought the use of the function index to be too slow. Between one iteration and the next, we know which label (thus position) we are considering, and the command would not make use of this information; rather, it would search the entire list every

time. In Algorithm 2 we did not report the use of this counter as a matter of readability. Note that also the function *isPareto*, Algortihm 2 Line 16, would have to work with this list, instead of discarding labels.

Once Algorithm 2 terminates, we return exclusively labels whose corresponding value in the *isDominated* list is False.

6.2.5 Skipping dominated labels

As a consequence of maintaining all the dominated labels, it is important to decide when it is *possible*, and when it is *necessary* to avoid some operations.

Let us go through a scenarios of possibility. Recall Algorithm 2 line 12, and the explanation provided in Section 6.1.3. Imagine to have flagged a label l as dominated at a early stage of the process, it is not necessary to perform any other operation regarding it. The transitivity of the domination criteria guarantees that: any label dominated by l is also dominated by the labels l' that dominates l. Paired with the knowledge that Algorithm 2 consider all the labels, a method without immediate eliminations does not differ from said Algorithm.

In practice, consider l to be a label, classified dominated in a early stage of the method. Referring to Algorithm 2 line 5, consider $l_1 = l$; we can skip an entire portion and consider the next value for l_1. it is important to place a check on the related value of *isDominated*, before the l_2 for cycle, i.e. Line 6 of Algorithm 2. Similarly, when cycling l_2 we should place there another check on *isDominated*. There is no motive to determine if a label is dominated by just one or multiple labels.

Consider now the scenario of necessity. Imagine to possess two identical labels, there is no necessity to keep the both of them; at the same time, the dominance criteria must not discard them both. Thus, once we found that l_1 dominates l_2, we do not check if also l_2 dominates l_1. And we skip to the next l_2.

6.2.6 Potential for parallel implementation

We want to note here that some components of the algorithm could be computed in parallel. Different destinations does not share information, so one could compute those labels independently. Keep in mind that, before the procedure could advance to future times, all the processes has to be concluded.

Chapter 7

Improved dominance rules

> Quel motore doveva essere un rottame, a meno che... Montandolo a casaccio, i nostri meccanici ne abbiano azzeccato uno migliore del **vero straturbo Perrari!**
>
> *Zio Paperone e l'avventura in formula 1*

The collection of rules, presented in Section 5.2.1, has been implemented and allow solving the DAC problem. However, after scaling up the size of the instances, they become nonetheless unpractical. The performance testing showed sensibly larger running times with respect to solving ILP model for the DAC problem (see Section 2.2.2) with a state of the art mathematical programming solver.

The reason has proven to be rooted in the large number of labels that it had to treat at every instant. With the aim of reducing the number of active labels, we tried to invert the order of operations within the Algorithm 1 (see Section 6.1), trying to early generate dominating labels, obtaining only marginal gains.

In this section, we describe more grounded domination rules, that make use of some additional observations regarding the future of a partial path: how permanence and quiescence relates; or *reachable configurations* starting from a given node. In general, we will discuss how to loosen up the original dominance criteria that tested to be too strict, resulting in a weak pruning.

7.1 Alternative update of quiescence component

Despite the fact that the model we considered have the values of t_p and t_q set, we discussed what would happen arranging differently the values of

these parameters.

In general, we recall that for any value of t_q such that $t_q \leq t_p$, the quiescence constraint will be automatically verified, see Section 2.2. We want to adapt this observation to scenarios in which $t_q \geq t_p + 1$, in order to obtain in advance the validation of some quiescence constraints, equivalently their elimination, see Section 5.1.2.

Assume to follow a partial path, referred by a label $l = (cost, 0, qu)$, and to enter c' at time $t^* + 1$; now the related label would become $l' = (cost_l + E^t_{c'}, t_p - 1, qu_l \cup \{c\})$. Then, ranging from $t^* + 1$ to $t^* + t_p$, the path can only repeat the configuration c', as observed in Section 2.2.

In these first part of a path, quiescence does not play a role in the definition of what configurations are reachable or not, it is sufficient to know the value of permanence, see Algorithm 3. Thus, it would be a good fit to update ahead of time the quiescence set qu, as in Section 5.1.1, imagining to be at $t = t^* + t_p$. Performing this operation could possibly relax a portion of the constraints active on l' and, as a consequence, l' could perform better in the domination procedure at node $(t^* + 1, c')$, as explained in Algorithm 2.

Formally, let $l = (cost_l, perm_l, qu_l)$ be a label with $perm_l > 0$ at time $t = t^*$; then the next $perm_l$ moves are set: the partial path related to l must repeat the last visited configuration. Thus, we should update quiescence as if $t = t^* + perm_l$. The new time of obsolescence of a quiescence constraint related to a label l, as defined in Section 5.1.2, becomes

$$t \geq time_{c,l} + t_q - perm_l \tag{7.1}$$

7.2 Exploiting graph sparsity

The first crucial fact to remember is the following: pruning is best when operated before a label could spread. The more we carry a label, the more it spreads and adds up to the total process time.

We came to the realization that the dominance criterion we presented, see Section 5.2.1, does not take into account if distinguishing, thus maintaining, different labels serves a *real* purpose. It could happen that two labels could produce the same outcome, even when evaluated as *non comparable*, making us carry twice the same potential solution.

Consider two labels l_1, l_2, associated to the same node (t, c), and assume they resulted non comparable, see Chapter 5 and Algorithm 2. This response from the domination criteria does not provide, for neither of the two, any certificate regarding future expansions, e.g. future compatibilities and, as a consequence, exclusive feasible solutions. In particular, it is not guaranteed that abandoning one of the two would result in a strict loss, regarding the optimal value for the problem.

To mitigate this behaviour, we want to take in account more information, especially regarding the near future of a path, hoping to catch a more grounded motivation to the usefulness to keep multiple labels.

7.2.1 Permanence component

Starting with the permanence, though it is correct that a strictly smaller value leads to the ability to spread sooner, this property has to meet the condition of having in reach configurations different from the one active.

Imagine to compare two partial paths, related to labels l_1, l_2, with same costs, same quiescence sets, but different permanence values; if, from the moment the first one acquires the ability to transition, up to the moment the second gain the same ability, there is no possibility to change active configuration, then there is no advantage to distinguish the two labels. In fact, the first partial path would continue repeating the active configuration. Then the partial paths would both reach a node (the only one reachable) where two labels would compare equal. We suggest to include this information in advance into the domination criteria; in order to avoid making a distinction that would not result in any advantage.

Formally, let l_1, l_2 be two labels, both associated to the node (t^*, c); with perm values of p_1, p_2, such that $p_1 < p_2$. l_2 could still be compared to l_1 if

$$C_c^{t'} = \{c\},\ \forall t' \in \{t^* + p_1, \ldots, t^* + p_2\}. \tag{7.2}$$

7.2.2 Quiescence component

Let us focus now on quiescence. Again, let l_1, l_2 be two labels, and assume that l_1 does not dominate l_2 by quiescence. By recalling the domination criteria of Chapter 5, implemented in Algorithm 2, we conclude that the cause is either one of the following:

$$C(qu_1) \setminus C(qu_2) \neq \emptyset \tag{7.3}$$

or

$$\{i \in C(qu_1) \cap C(qu_2) : \mathit{time}_{i,1} > \mathit{time}_{i,2}\} \neq \emptyset. \tag{7.4}$$

Considering $\mathit{time}_i = 0$ if $i \notin C(qu)$, we could merge the two scenarios within the latter expression.

Similarly to Section 7.2.1, we want to identify two labels if the condition that makes them non comparable does not have an effect on the expansion of the partial paths.

Let c' any configuration arising from (7.4), if c' is not reachable from the current node, up to the moment both quiescence constraints become validated, see Sections 5.1.2 and 7.1, then the two labels are still comparable.

Formally, let l_1, l_2 two labels associated to a node (t^*, c); let c' be a configuration arising from (7.4), assume $c' \in c \in C(qu_1) \cap C(qu_2)$, define the obsolescence time for l_1, l_2 as follows:

$$obsTime_i = t_i^{c'} + (t_q - 1) - perm_i \text{ for } i \in \{1, 2\} \tag{7.5}$$

we note that $t_i^c \leq t^* - 1$. Observe that in the scenario $c' \notin C(qu_2)$, the expression (7.5) is not useful; however, enforcing permanence, we recall that the partial path associated to l_2 can not transition for $perm_2$ time intervals. Thus, in the case $c' \notin C(qu_2)$ we set $obsTime_2 = t^* + perm_2$ and $obsTime_1$ as the result of the expression (7.5).

Finally, we can state:

$$c' \notin Reachable_{t,c}, \ \forall t \in \{obsTime_2, \ldots, obsTime_1\} \tag{7.6}$$

where the term *reachable* hide the necessity to compute all the nodes that can be reached from the node (t^*, c) up to time $t \in \{obsTime_2, \ldots, obsTime_1\}$. Thus, one should compute it by merging subsequent C_c^t sets.

In our implementation, we wrote another expression, simpler to handle, but more strict: we asked $c' \notin C^t \ \forall t \in \{obsTime_2, \ldots, obsTime_1\}$.

The two conditions presented in Sections 7.2.1 and 7.2.2, implemented together, grant a significant drop in computation times, and keep the the algorithm exact. We want to emphasize the fact that the validity of these condition is certainly intertwined with the sparsity of the graph at hand, see Section 8.1: typically, a configuration that appears in the first moments of a day is not present after few hours. Thus, the quiescence constraints are automatically satisfied, by the shape of the instances. Nonetheless these domination criteria could be applied as well to a general graph, they just would not be as effective in that case. In 8, we the results of the testing we performed.

7.3 Size-scalability and heuristic approach

The current procedure is a breadth first approach, or equivalently we collect labels in ordered lists, and process them with a First-In First-Out (*FIFO*) queue method. Thus we are certain that before reaching a given time t^* we have explored, thus expanded, all best paths up to t^*. A limitation to this method is the fact that before it could compute a feasible solution, it has also process all the others. In this section we propose an informal analysis of possible ideas towards a heuristic search on the graph that provides good, although non provably optimal, solution to the DAC problem..

The literature does not offer simple effective solutions, reference [15] presents two insights, that fall short in our context. The first aims to discard all nodes that, once reached, could not permit the partial path to

be completed maintaining feasibility. This analysis is performed operating a one-to-all shortest path instance, starting from the destination, in order to compute all the minimum resource consumption to reach each node. Nonetheless, our consumption update rule is non linear, and thus it would not be effective.

The other suggested method is to compute a double search in both directions, but experiments show it does not offer any time advantage. By the contrary, a recent paper [1] suggests to launch multiple parallel searches, which start from different *depots*. It would be somewhat applicable, if we could define those depots as *stopping points* within the graph. We could rank them based on the possibility to spread towards other configurations. Sadly, the shape constraint it would ask on the graph is too restrictive: in our datasets, configuration disappear at the end of each time period, and would possibly lead to the definition of depots at the end of each hour. This approach could not be generalized, thus we have to abandon it.

Reference [18] strongly advocates against computing one solution at a time, while reference [15] speaks about the *curse of dimensionality*, when considering a dynamic programming approach with a large size network. We did not consider classical metaheuristics, in fact it would not be an easy task to define a good criterion for a neighbourhood of a solution.

We found that the major reason of large computation times is the size of labels to manage, if we were to define *heuristics domination criteria* we would certainly obtain better results. A first idea is to define a *elitist rule*, which maintains the best k labels ordered by cost. We opt for this method, for a simple effective solution. Nonethless, One has to keep in mind the limits of this approach, we saw earlier that cost alone can not guarantee any information about the future.

We thought to adapt the best insertion approach, in order to consider *some* information about the expansion of a path. We started by considering the following observation: changing configuration constrains the path for a period of time, while keeping the current node at a major cost could lead to better option by the next turn. So, what we opted for is to rank the destinations at each time. Then, let the labels both expand towards the current configuration (when possible), and towards the best n reachable nodes, where n is a parameter.

The ranking method works as follows: order the configurations $c \in C^{t+1}$ based on the average cost over the next t_p moments. If a configuration could not be kept for this amount of time, assign a very large cost, so that the ranking process would punish it. We observe that small n values could lead to unfeasibility, while larger values for the parameter would make this heuristic ineffective, since we expand all the paths.

The pseudocode is reported in Algorithm 4.

Then, we would have to adapt the cycle presented in Algorithm 1 Lines 9-17, in order to include the rank we performed. A good way to perform it,

Algorithm 4 heuristic expansion

```
for c ∈ C^{t+1} do
    if c ∈ C^{t+1+s} ∀s ∈ {0, t_p − 1} then
        AvgCost_c = Σ_{τ=t+1}^{t+t_p} e_c^τ / t_p
    else
        AvgCost_c = very large cost
    end if
end for
Rank c ∈ C^{t+1} based on AvgCost
```

in Python at least, consists in not reordering each set C_c^{t+1}, as it would be a time consuming procedure, rather we would scroll C^{t+1} -already ordered- skipping configurations that are not present in C_c^{t+1}.

This process is still a breadth first approach, but the testing showed fast enough performances, even on large datasets.

Chapter 8

Computational results and discussion

We collect in this chapter the result of the implementation in Python of our label-setting algorithm for the DAC problem, as presented in Chapter 6. We collect the solving times computed on different groups of instances, divided by size and type. We underline the presence of two classes of data: sparse and dense; we decided to test our method on datasets characterised, respectively by, a large number of transitions opposed to a sparse transition graphs.

8.1 Instances generation

An instance of the DAC problem is comprised of three parts: the composition of the configurations, the compatibility graph, and the traffic data. The graph implicitly include the configurations and stores the information regarding which configurations could be achieved at a given hour, and which transitions are possible between those. Meanwhile, the traffic is used to compute the costs, precisely the excess of demand, related to every elementary sector, for every hour. Thus, given a set of transition graphs and a set of traffic demand, any two elements of the corresponding sets can work interchangeably, making possible a lot of combinations.

The authors of [16] kindly provided a set of ten both graphs and traffic files. Those has been used as ground-zero for the testing of the algorithm. Soon enough, it arose the necessity to come up with richer datasets including large-size instances, in order to deeply test the limits of our code. To have an idea on the dimension of these instances, one could refer to the first row of Table 8.1.

The provided datasets has been crafted with the aim to mimic realistic conditions, and could not be replicated by simplistic rules, see Appendix A. We could, in fact, decide to create a configuration that partitions randomly

the airspace, but that would loose the intent to respect, e.g., compactness and convexity conditions on collapsed sectors.

We warn the reader about a specific structure of these instances: the configuration set C^t is defined at the start of an hour, and kept unchanged for the rest of that hour, i.e. $C^{12 \cdot t+1} = \cdots = C^{12 \cdot (t+1)}$ for $t \in \{0, \ldots, 23\}$; we recall that every C^t is defined over a 5 minutes interval, see Section 2.2. We accept overlaps for achievable configurations, if we consider consecutive hours. Thus, going through the instances, they satisfy the following:

$$C^1 = \cdots = C^{12} = \{1, \ldots, n\},$$

$$C^{13} = \cdots = C^{24} = \{k, \ldots, m\}, \text{ with } k \leq n+1$$

We limit to consider the first two hours, i.e. from C^1 to C^{24}, for simplicity. Also C_c^t are determined at the start of every hour, and kept constant for that hour.

Most of all, one should amend the absence of configurations that appears and disappears in a short time window, without being achievable for the full hour. This addition does not have a realistic counterpart, in fact it would go against the interest of controllers to have short-lived configurations, thus in contrast with the requests explicitly stated in [9, 16]. Nonetheless, their eventual presence in a generic instance is what make so difficult coming up with effective domination criteria. It is the quiescence that fuels the large number of labels, while also prohibiting the activation of an extremely convenient configuration, if it were to appear.

8.1.1 Merging graph structures

We briefly comment on dimensions of ground-zero instances, in order to appreciate better the size of the datasets we are going to build. They start, and end, with a unique configuration achievable, composed of a unique collapsed sector. We then subsequently split this sector in various way, up to obtaining less than a dozen of configurations, all achievable in the middle of the day; configurations are compatible with at most 6 others.

In order to obtain realistic large-size instances, we propose to merge graph structures from different ground-zero instances, forming larger pools from which the algorithm could choose the active configuration. We describe compatibility rules between these nodes in Section 8.1.2. At the same time, we observe there is no need to touch the traffic data, since these configurations are computed starting from elementary sectors, see Appendix A.

Thus, we merged $k \in \{3, 5, 7, 9\}$ graphs from the ground-zero instances, meaning that we chose k instances $\forall t$ we computed the union of C_i^t for $i \in \{1, \ldots, k\}$. The combinations one could produce are numerous, quantifiable by the bynomial function.

Alternatively, we propose another method of creating configurations: augment the number of configurations available at each time; i.e. if at time t there are n configurations, add at each moment of that hour k configurations, with $k \in \{1, 5, 9\}$. This process wanted to mimic the possibility to start and end the procedure with more choices. Nonetheless, the results are compatible by the ones we obtained through merging. Both in terms of instances' size and solving times. For this reason, we decide absorb together the solving times we collected.

8.1.2 Forming compatibility sets

We ditched the idea to determine compatibilities based on random criteria, since all of the realistic intents would be lost. Thus we inferred some rules, giving a deep look to how different configurations relate to each others.

We thought of a *sparse* approach. It maintains exclusively the compatibilities present beforehand. The result is a truly sparse graph, given the fact that, during the hour, some configurations may be related only to themselves, or to an additional one; this description is not exhaustive, since some other configurations have a net of links that connects them to the majority of C^{t+1}.

Secondly, we thought of a *dense* criterion. This time, we aimed to approximate how the process works. During the first twelve hours, we accept, hence possibly move to, more complex configurations: we want them to use a larger number of (collapsed) sectors; while we bound how much they can change, in order to define it a viable transition. By the contrary, in the second half of the day, we move to simpler configurations; here we could decide how quickly we want this process of shrinking to happen.

n order to give a clear grasp on magnitude of the instances we formed, consider that now the peak of achievable configuration reaches 65 nodes (instead of 9), and the maximum number of compatible transitions is 30, enforcing sparsity, and 55 choosing the dense criterion.

Now we describe technically what the measure does, when deciding if configurations c_1, c_2 are compatible. We remember that a configuration is composed of numerous collapsed sectors, each formed by a collection of elementary sectors. Thus, the condition is described by a request on the collapsed sectors.

During the hour, we want to connect configurations as similar as possible, bounding with a constant the number of collapsed sectors that changes between the two. We experimented with the value of this constant, in order to have a rich graph, while avoiding it having every configuration compatible with every others. We observe that the condition reflects the part of the day: in the morning, it counts the number of collapsed sector that would get added; while in the afternoon it counts the ones which would get removed.

A more precise criterion should take in mind the composition of each

Sparse	Graph Stats				Solving Times (s)		
Name	$\|V\|$	$\|A\|$	$\|V\|/\|T\|$	$\|A\|/(\|V\| \cdot \|T\|)$	Cplex	Dom 1	Dom 2
sparse 1	1356	2965	4,7	1,91	0,68	1,61	0,12
sparse 3	4258	9701	14,8	2,03	7,98	3,57	0,4
sparse 5	6914	15625	24	2,11	32,12	6,07	0,64
sparse 7	9453	21542	33,1	2,13	75,45	5,92	0,64
sparse 9	11440	26007	39,7	2,13	122,34	8,74	1,11

Table 8.1: Results and dimension on sparse instances.

collapsed sector, but that would again result in a sparse graph. Thus we decided to keep this simplistic rule, which provided instances with the desired properties..

At the end of each hour, the condition slightly changes, now becoming the following: two configurations are compatible with each other, if every collapsed sector of one configuration is obtained through an operation of splitting or merging applied to any collapsed sector in the other configuration.

Finally, we collect these instances in groups, depending on the value of k, number of graphs we merged, and the criterion used to determine the compatibilities.

8.2 Benchmarking results and observations

We encoded the algorithms presented in Section 6.1, with domination criteria presented in Section 7.1, using the Python programming language, version 3.10. We ran experiments on a Windows machine, with 8Gb of Ram and a 11*th*-Gen i5 Intel processor. We also solved the model presented by 2.2.2, through a general purpose mathematical programming solver, namely Cplex [10]. We acknowledge that both the model and its expression in python has been provided by the authors of [9].

We collected solving times on the method detailed in Chapter 6 with both our domination criteria: the basic one of Chapter 5, referenced by *Dom1*; and the improved version of Chapter 7, referenced by *Dom2*. For each value of k, and type of compatibility criterion, we tested the algorithm on ten instances and three values of traffic. We recall graphs and traffic data are independent and can be used interchangeably. We collected the results and computed the average for each group.

In the following tables, refer to 8.1 and 8.2, each row corresponds to a different sized collection of graphs. There are collected average measures on the size of these graphs. In particular, the total number of nodes and arcs, the average size of C^t and C^t_c. Thus, it is natural to observe the similarity within the average measurements that does not depend on the set of arcs A.

Dense	Graph Stats				Solving Times (s)		
Name	$\|V\|$	$\|A\|$	$\|V\|/\|T\|$	$\|A\|/(\|V\| \cdot \|T\|)$	Cplex	Dom 1	Dom 2
Dense 1	1356	3635	4,7	1,91	0,26	44,73	0,32
Dense 3	4258	21094	14,8	4,08	2,69	1803	9,09
Dense 5	6914	48566	24	6,65	9,24	>16000	56,62
Dense 7	9453	88033	33,1	8,54	41,94	-	231,05
Dense 9	12121	134876	42,1	10,93	73,05	-	565,82

Table 8.2: Results and dimension on dense instances.

We focus for a moment on the first set of domination criteria. Given the definition of sparse compatibility conditions, the first row of Table 8.1 represents exactly the original instances from [9].

The results seems promising: our basic approach has been able to compete with Cplex. This clearly motivated the need to create additional -larger-instances.

Tested on different datasets, see for example the first row of Table 8.2, we can note a difference that begins to form between the solving times of the two methods. This difference can only exacerbate when considering larger, denser, instances.We observe that the very large number of labels to manage, leads the the graph based approach to DAC to running time sometime over one hour. For this reason, we soon decided to not test further the first set of rules.

We observe that our basic method is faster then Cplex when working on a large sparse graph. the reason would lie in the presence of a huge number of variables taking value zero (i.e. all the non activated configurations). Nonetheless, this success does not outweigh the impracticality that is shown when working with any other type of graph.

8.2.1 Results observations

Through an intense testing, we checked that both the domination criteria find the optimal value, accordingly to the theory, though spending significantly different amounts of time.

It seems that the second set of dominance rules (refer to Chapter 7) outperforms vastly the first (refer Chapter 5) with both compatibility criteria; additionally, it does not fall behind heavily when scaling the problem. It is true that the method could not keep up with Cplex, but the solving time are somewhat acceptable, given the fact that it asks for a few minutes. All depends on the concept of applications: some papers approached the problem of sectorization (refer to Chapter 1) as a quick tool to use, from which we would expect fast solving times. By the contrary, we do not consider this aspect of the problem, so 10 minutes could be an acceptable waiting time when planning the operations for the following day or week.

Part of the credits goes to the formulation of the method, in particular its component to control the domination before the addition of a label to the group. We repeated some tests using old criteria for domination, confronting the old implementation code (which firstly adds every labels, from all origins, and only then applies domination), to the one we presented in 6.1. We could appreciate a clear gain in solving times.

8.2.2 Some data on labels

In previous sections, we stated our suspicion towards a dependence between high solving times, and a large number of labels to treat. We decided to shed some light on the phenomenon, computing some indicators regarding the number of labels processed. It would be interesting to compare the different exact methods presented, in order to highlight the difference in the size of labels computed, thus the effectiveness of the domination criteria.

	First set			Second set		
Name	Tot	Avg	Max	Tot	Avg	Max
sparse 1	15475	9	31	3578	2	10
sparse 3	53049	11	38	10902	2	20
sparse 5	53049	11	38	19125	2	34
sparse 7	99401	13	70	24899	2	42
sparse 9	143692	12	56	28091	2	27
Dense 1	75302	40	1146	5846	3	78
Dense 3	806585	186	4871	33008	7	284
Dense 5	-	-	-	156416	19	1538
Dense 7	-	-	-	380566	35	2366
Dense 9	-	-	-	500151	41	2908

Table 8.3: Data on labels processed.

In Table 8.3 we read the total number of labels managed by the algorithm, then the average over the totality of nodes, and finally the max value of labels a single node had stored in at least one moment. We do not report the minimum amount of labels processed, since it would be constantly 1: the single label processed at the beginning. We want to insist instead on the concept of *average*: the graphs present nodes behaving differently, even within the same time coordinate. We started by considering exclusively the average on time, as to represent the total number of labels treated at a given time. But that result did not consider that at different times different amounts of nodes are present. Thus we opted for this measure. Reporting the max wants to give a measure that could effectively represent the concept of a graph being dense, and how well the domination criteria perform: a low max value implies that the domination criteria are counterbalancing the size of possible expansions.

By juxtaposing the Tables 8.1, 8.2 and 8.3, one could see a clear trend with the number of labels skyrocketing with the first criterion for domination, and the solving times behaving accordingly. We could appreciate that when considering sparse instances, both domination criteria do an acceptable job, but it is just a consequence of the shape and sparsity of the datasets. Rather, when considering the dense graphs, the criteria presented in Chapter 7 truly shines. And a solving time so high is justified by the fact that, even if the number of labels is relatively low, there is a huge amount of nodes.

8.2.3 Results of a heuristic

We want to recall that our approach suffer from the fact that before the method ends, there is no feasible solution, as anticipated in section 7.3. We also observe that Cplex works formulating almost instantly a solution, then spending a lot of time making it better or proving it is optimal through advanced methods. If one necessitates a solution in short amount of times, recurring to a heuristic would be fundamental.

It is necessary to mention the fact that the shape of the instances surely play a role: we suspect that if we were to drop the majority of labels and keep the *first* label to arrive to a node, we could obtain good-enough results, in sensibly less time.

We suggest to repeat the tests on different instances, in order to highlight consequences of the performed choices. Additionally, one should try testing the method on real world instances, as done by the authors of [16], and others.

We implemented the heuristic presented in Section 7.3. We could appreciate solving times being near 1 second, for both $Dense7$, $Dense9$. A gap is present between optimal values, but too small to be concerning: we could see different solutions having values like 20782 as optimum, and 20815 from heuristic, an 0.15% increase.

Once again we note that we could not advocate much for this method, since it does not guarantee anything on feasibility and optimal value. Its major strength is the possibility to launch different values for the parameter, knowing that it would imply much smaller solving times, than solving the instance with exact methods.

Chapter 9

Conclusion and future work

We considered the *Dynamic Airspace Configuration*(*DAC*) problem, a problem from arising in Air Traffic Management, in which it is asked to determine an optimal sequence of configurations, in order to minimize the excess of demand, interpreted as the number of flights that exceeds a limited capacity. The problem present side constraints regarding steadiness, we focused on a model presented by [9], in which the concepts of *permanence* and *quiescence* are introduced.

We proved the DAC problem being NP-hard, following the addition of quiescence constraints alone, thanks to a reduction from the Travelling Salesman Problem (TSP); we also proved that the DAC problem can be solved in a polynomial time, if we were to enforce exclusively permanence constraints.

We modeled the problem as an instance of the Resource Constrained Shortest Path Problem (RCSPP), defining an appropriate set of resources and related labels. Additionally, we proposed an exact method based on labels optimization, defining the components that form a label, how to expand and compare them. We focused, in particular, in the definition of two valid dominance criteria.

We implemented the proposed method, and tested it on large instances we generated starting from realistic ones we were provided. We compared the results of our method against a Integer Linear Programming formulation for DAC proposed by [9], solved by a general purpose ILP solver, namely Cplex, interfaced by Python programming language. We showed the effectiveness of our method in the case the graph is sparse; on the contrary, Cplex performs better when our proposed approach has to compute a large number of non-dominated partial paths.

We implemented a fast heuristics that performed well on the test instances, however it does not guarantee any optimality gap.

As future work, the graph representation of DAC and the RCSPP approach proposed in this thesis can be expanded to accommodate for robustness issues, in order to take traffic demand uncertainty into account.

Appendix A

Instance generation

We go through some of the work performed by one of the authors of [9].

The airspace is discretized as a $60x100$ grid. A set of points scattered on thee grid are considered as origin and destination of all the flights. The points are chosen with the following criteria: 4 fixed points in the middle of the grid, 3 random points on both the bottom and top rows, and 5 random points on both the leftmost and rightmost columns.

A route can connect two interior points, or an interior point to one on the border. In order to generate the routes, the author uses a parametric equation, representing a section of an ellipse, of which we can control the curvature. For each pair of points, two values of the parameter are considered, thus generating two different routes. Once the set of routes is formed, the author picks a sample of it, keeping 1500 of them.

The total time frame is one day, subdivided in five-minutes intervals. For each flight, it is chosen a random time of departure, and a permanence time on every element of the grid is assigned. Once the traffic is known, the grid is partitioned in 60 elementary sectors, formed taking in consideration the location of the points in which traffic tends to accumulate.

The **capacity** of **elementary sectors** is defined starting from the traffic demand within the day. It considers 85% of the highest value of occupation, plus a small random component, that could vary at each time interval. Meanwhile, the **capacity** of a **collapsed sector** is computed following this idea: take the max of the elementary sectors capacity, and add three additional flight for every other elementary sector. The increment is bounded by a 30% factor. This means that a different arrangement of elementary sectors could lead to different capacity configurations. It is not a linear function ruling how the capacities adds up.

The **transition graph** is a 288-partite graph, given the 5 minute time windows. It starts considering a single collapsed sector, formed by the union of every elementary sector. For every hour, it considers a group of configurations. Every hour the number of feasible configuration changes, increasing

in the first 12 hours of the day, and decreasing after. At the end of the day, again only one configuration is permitted: the one formed by the union of all elementary sectors.

The compatible configurations are determined by operations of splitting and merging, the number of derived configurations is random. These configuration are collected in different groups, depending on the origin. Transitions are acceptable exclusively between configurations of the same group.

Appendix B

Python code

Here we report the code we implemented.

B.1 Exploration block

The following functions corresponds to Algortihm 1 presented in Chapter 6.

```
def Algo():
    starttime=time.time()
    initLabels()
    for t in T[:-1]:
        for conf in CONFt[t+1]:
            #expand first native labels
            if conf in PASTtc[t,conf]:
                Labels[t+1,conf]+= updateLabels((t,conf,conf))

            #collect foreigners labels in a list
            LbToAdd=[]
            for PrevConf in PASTtc[t,conf]:
                if PrevConf == conf: continue
                LbToAdd += updateLabels((t,PrevConf,conf))
            Labels[t+1,conf]+=LbToAdd

            #refresh labels position after pruning
            counterPos = 0
            for l in Labels[t+1,conf]:
                l.pos=counterPos
                counterPos+=1
        #print statistics
        print(f'{t,[len(Labels[t,conf]) for conf in CONFt[t]]}'
                                                )
    conclude()
    endtime=time.time()
    if Labels['finalDummy'] != []:
        Labels['finalDummy'].sort(key=lambda l: l.cost)
        optPath=RecoverPath()
        return (Labels['finalDummy'][0].cost, optPath, float(
                                        endtime-starttime))
```

```
    return
```

This code is different from the pseudocode presented in Chapter 6: it does not use a double for cycle, thus it will operate more than once the deep copy operations. As previously stated, that code should grant marginal gains, but we reported here the code that we used for testing. Also the pseudocode does not define the function *Conclude*, useful to collect in one place all the labels and compare them. It is our filtering step [11].

What follows is a collection of the function we used. Also *Algo()* is defined as a function, since we are going to recall it multiple times, aiming to automate the benchmarking process on multiple graphs and traffic.

We follow with the operations of initialization and conclusion.

```
def initLabels():
  '''Initialize the labels'''
  for conf in CONFt[0]:
    Labels[(0,conf)]=[lb(e[conf][0])]
  for t in T[1:]:
    for c in CONFt[t]:
      Labels[(t,c)]=[]
  Labels['finalDummy']=[]
  return

def conclude():
  for c in CONFt[T[-1]]:
    suppLbList=copy.deepcopy(Labels[T[-1],c])
    Labels['finalDummy']+=[lb(cost=l.cost,prtNode=c, prtLabPos=
                                l.pos) for l in suppLbList]
  return
```

B.2 Label encoding

We opted to store labels using a class.

```
global Labels #dict : key = (time, conf) tuple. element: list
                             of labels associated
Labels={}

class lb: #label
    '''singular label'''
    def __init__(self, cost = 0, permanescence = 1, quiescence
                                    = {}, LabPos = 0, prtNode =
                                     'dummy', prtLabPos = 0):
        self.cost        = cost
        self.perm        = permanescence
        self.qu          = quiescence
        self.pos         = LabPos #ID of a label.
        self.prtNode     = prtNode
        self.prtLabPos   = prtLabPos
```

In a previous version, we represented them in a list, but that resulted in a more convoluted notation, and some difficulties when it came to printing information. This time, using some class methods, we could decide what attribute to return when a label is printed.

```
    def __repr__(self):
        return f'(cost: {self.cost}, perm: {self.perm}, quies:
                                        {self.qu}, prt: {self.
                                        prtNode})'
```

B.3 Labels' domination and expansion

```
    def dominates(self,adversaryL,CurrTime,CurrConf):
        if self.cost > adversaryL.cost:
            return False
        if self.perm < adversaryL.perm:
            for s in range(tp-adversaryL.perm,tp-self.perm):
                if bool(set(CONFtc[CurrTime+s+1,CurrConf]).
                                                difference(set(
                                                [CurrConf]))):
                    #adversaryL can spread before Self.
                    return False
        QuarantinedKeys=[key for key in self.qu.keys() if key
                                        not in set(adversaryL.
                                        qu.keys())] #keys that
                                        potentially make
                                        uncomparable the two
                                        labels
        if bool( QuarantinedKeys ):
            for key in QuarantinedKeys:
                ObsTime=self.qu[key]+(tq-1)-(tp-self.perm)
                for s in range(CurrTime+1,ObsTime+1):
                    if key in CONFt[min(s,T[-1])]:
                        #adversaryL could enter more node
                        return False
        QuarantinedKeys=[key for key in set(self.qu.keys()).
                                        intersection(set(
                                        adversaryL.qu.keys()))
                                        if self.qu[key]>
                                        adversaryL.qu[key]]
        if bool(QuarantinedKeys):
            for key in QuarantinedKeys:
                ObsTime=self.qu[key]+(tq-1)-(tp-self.perm)
                AdvObsTime=adversaryL.qu[key]+(tq-1)-(tp-
                                                adversaryL.perm
                                                )
                for s in range(AdvObsTime,ObsTime+1): #col +1 o
                                                 senza? CON.
                                                vedi sopra
                    if key in CONFt[min(s,T[-1])]:
                        return False
```

```
        return True
```

We want to comment the fact that we used a large number of list descriptors, they have been extremely useful to describe concisely the condition we wanted. Also, we expressed the if statements in nested fashion, in order to squeeze out as much time gains as possible.

Regarding the other, simpler, modules.

```
    def newQu(self,CurrTime,CurrConf, ConfMantained: bool =
                                        False ):
        newQu=copy.deepcopy(self.qu)
        if not ConfMantained: newQu[CurrConf]=CurrTime
        for obsoleteKey in [key for key in self.qu.keys() if (
                                        self.qu[key] <=
                                        CurrTime - (tq -1) + (
                                        tp - self.perm ))]:
            newQu.pop(obsoleteKey)
        return newQu

    def newPerm(self,ConfMantained = False):
        return min(self.perm+1,tp) if ConfMantained else 1

    def isExpandible(self,CurrConf, NextConf):
        '''return a Bool value representing if a given label
                                        can be expanded'''
        if NextConf==CurrConf:
            return True
        if self.perm < tp or NextConf in self.qu.keys():
            return False
        return True

    def expanded(self,CurrTime,CurrConf,NextConf):
        '''return a label transformed'''
        return lb(self.cost + e[NextConf][CurrTime+1], self.
                                        newPerm(NextConf==
                                        CurrConf), self.newQu(
                                        CurrTime,CurrConf,
                                        NextConf==CurrConf),
                                        len(Labels[CurrTime+1,
                                        NextConf]), CurrConf,
                                        self.pos)
```

References

[1] Saman Ahmadi, Guido Tack, Daniel D. Harabor, and Philip Kilby. "A Fast Exact Algorithm for the Resource Constrained Shortest Path Problem". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.14 (May 2021), pp. 12217–12224. URL: https://ojs.aaai.org/index.php/AAAI/article/view/17450.

[2] Ravindra K. Ahuja, James B. Orlin, Thomas L. Magnanti, and Ravindra K. Ahuja. *Network flows : theory, algorithms and applications / Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin.* eng. Harlow: Pearson, 2014. ISBN: 9781292042701.

[3] John Adrian Bondy, U. S. R. Murty, and John Adrian Bondy. *Graph theory / J.A. Bondy, U.S.R. Murty.* Graduate texts in mathematics. New York: Springer, 2008. ISBN: 1846289696.

[4] Luigi De Giovanni, Nicola Gastaldon, and Filippo Sottovia. "A two-level local search heuristic for pickup and delivery problems in express freight trucking". In: *Networks* (2019).

[5] Martin Desrochers and Francois Soumis. "A Generalized Permanent Labelling Algorithm For The Shortest Path Problem With Time Windows". In: *INFOR: Information Systems and Operational Research* (1988).

[6] Irina Dumitrescu and Natashia Boland. "Algorithms for the Weight Constrained Shortest Path Problem". In: *International Transactions in Operational Research* (2001).

[7] Patrik Ehrencrona Kjellin. *Airspace Sectorisation Using Constraint-Based Local Search.* 2014.

[8] Michael R. Garey, David S. Johnson, and Michael R. Garey. *Computers and intractability : a guide to the theory of NP-completeness / Michael R. Garey, David S. Johnson.* eng. ¡¡A ¿¿series of books in the mathematical sciences. San Francisco: Freeman, 1979. ISBN: 0716710455.

[9] De Giovanni, Galeazzo, and Lulli. "An IP approach for dynamic airspace configuration". In: (2024).

[10] IBM. *Cplex library for Python implementation.* https://ibmdecisionoptimization.github.io/docplex-doc/mp/index.html.

[11] Stefan Irnich and Guy Desaulniers. “Shortest Path Problems with Resource Constraints”. In: *Column Generation.* Ed. by Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. Boston, MA: Springer US, 2005, pp. 33–65. ISBN: 978-0-387-25486-9. URL: https://doi.org/10.1007/0-387-25486-2_2.

[12] Stefan Irnich and Daniel Villeneuve. “The Shortest-Path Problem with Resource Constraints and k-Cycle Elimination for k $\geq$ 3”. In: *INFORMS Journal on Computing* 18.3 (2006), pp. 391–406. URL: https://doi.org/10.1287/ijoc.1040.0117.

[13] Sameer Kulkarni, Rajesh Ganesan, and Lance Sherry. “Dynamic Airspace Configuration Using Approximate Dynamic Programming: Intelligence-Based Paradigm”. In: *Transportation Research Record* 2266.1 (2012), pp. 31–37. URL: https://doi.org/10.3141/2266-04.

[14] M Florencia Lema-Esposto, Manuel Ángel Amaro-Carmona, Natividad Valle-Fernández, Enrique Iglesias-Martínez, and Adrián Fabio-Bracero. “Optimal dynamic airspace configuration (DAC) based on state-task networks (STN)”. In: *Proceedings of the 11th SESAR Innovation Days, Online* (2021), pp. 7–9.

[15] Leonardo Lozano and Andrés L. Medaglia. “On an exact method for the constrained shortest path problem”. In: *Computers & Operations Research* 40.1 (2013), pp. 378–384. ISSN: 0305-0548.

[16] De Giovanni Luigi, Galeazzo martina, Lulli Guglielmo, and M. Florencia Lema-Esposto. “An Integer Programming approach to Dynamic Airspace Configuration”. In: 2024.

[17] Francisco Pérez, Victor Gomez Comendador, Raquel Jurado, María Zamarreño Suárez, Dominik Janisch, and Rosa Valdés. “Dynamic model to characterise sectors using machine learning techniques”. In: *Aircraft Engineering and Aerospace Technology* ahead-of-print (Jan. 2022).

[18] Marina Sergeeva, Daniel Delahaye, Catherine Mancel, and Andrija Vidosavljevic. “Dynamic airspace configuration by genetic algorithm”. In: *Journal of Traffic and Transportation Engineering (English Edition)* 4.3 (2017), pp. 300–314. ISSN: 2095-7564. URL: https://www.sciencedirect.com/science/article/pii/S2095756417301927.

[19] Marco Di Summa. *Appunti di Ottimizzazione Discreta.* Chap. 6,7.

[20] Tambet Treimuth, Daniel Delahaye, and Sandra Ulrich Ngueveu. “A branch-and-price algorithm for Dynamic Sector Configuration”. In: 2016.

[21] Min Xue. “Airspace Sector Redesign Based on Voronoi Diagrams”. In: *Journal of Aerospace Computing Information and Communication* (2009).