Onboard Flight Dynamic Route Optimization

João Francisco Pereira do Vale Tavares Portugal

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alexandre Paulo Lourenço Francisco
Dr. Ricardo José Nunes dos Reis

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Alexandre Paulo Lourenço Francisco
Member of the Committee: Prof. Luís Jorge Brás Monteiro Guerra e Silva

January 2021

Acknowledgments

First of all, I'd like to thank my family, especially my parents for their hard work and for providing me everything I needed to succeed in life and to reach this point at which I graduate.

I'd like to thank my friends Guilherme, Lucas, Miguel, Nuno, Ricardo and Rita for their help and support through some rough moments I had, and especially for always being there. I would also like to thank Tiago for his tremendous help, patience and time spent in helping me conclude one of the parts of this work.

To my professors, every single one of them that pushed me to be the best, especially my supervisor Alexandre Francisco, Eng. Ricardo Reis, Eng. Ricardo Parizi and Professor José Rui Figueira that were always there to help me if I had any problems or doubts. Their help, support, guidance and patience is why this work was finished.

To each and every one of you – My honest and heartfelt Thank you.

Abstract

Nowadays the flight management systems (FMS) of an airplane can fly it autonomously from takeoff to landing with little intervention from the pilots assuming there are no anomalous events. However, some events require a diversion and subsequent route replanning. This re-plan is non-trivial, subject to a restricted criteria and takes attention away from the pilots and from air traffic controllers, attention that could be spent monitoring other flight systems. Route planning involves finding a new efficient route, communicate it to ATC, receive approval and reprogram the flight computer to follow the new path. Our work presents a module capable of finding a feasible flight path while complying with all the existing restrictions whether they are space restrictions such as weather events or no fly zones, or the airplane movement restrictions, removing the need of pilot intervention in this area lightening up their work as well as the air controller's work. We model the problem as an extension to the knapsack model and we then utilize a modified labeling algorithm created to solve the bicriteria 0-1 knapsack problem and efficiently recalculate the routes while taking into account all existing restrictions. The evaluation made was focused on both the accuracy of the route calculated and the time it took to recalculate the route and present the final result.

Keywords

Multi-objective problem, route planning, FMS, knapsack model, Flight routes

Resumo

Hoje em dia, o computador de bordo de um avião consegue pilotá-lo autonomamente desde a descolagem até à aterragem com pouca intervenção dos pilotos assumindo que não existem eventos anómalos. Contudo, alguns eventos exigem uma diversão e portanto um replaneamento de rotas. Este replaneamento não é trivial visto que é sujeito a critérios restrictos, desviando a atenção dos pilotos e dos controladores de tráfego aéreo para este tarega, atenção que poderia ser dispendida monitorizando outros sistemas de voo no caso dos pilotos. Um planeamento de rota envolve encontrar uma nova rota eficiente, comunicá-la à torre de controlo, receber aprovação desta mesma, reprogramar o computador de bordo para que o avião siga pela nova rota. O nosso trabalho apresenta um módulo capaz de encontrar uma rota de voo viável que respeite todos as restrições, sejam elas de espaço, por exemplo zonas proibidas de voo ou eventos de metereologia ou então restrições específicas à aeronave, removendo a necessidade de intervenção do piloto nesta área, reduzindo a carga de trabalhos tanto deste como dos controladores aéreos. O problema é modelado como uma extensão ao problema da mochila, sendo que depois utiliazmos um algoritmo de labeling modificado para resolver o problema da mochila 0-1 com dois critérios conseguindo assim recalcular eficientemente a nova rota. A avaliação foca-se tanto na precisão da rota encontrada bem como do tempo que levou a encontrá-la e a mostrar o resultado final.

Palavras Chave

Problema multi-objectivo, recalculamento de rota, computador de bordo, problema da mochila, rotas de voo

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Objectives . . . 4

1.3 Outline . . . 6

2 State of the Art **7**

2.1 Pathfinding . . . 9

2.1.1 Introduction . . . 9

2.1.2 Pathfinding State of the Art survey . . . 9

2.1.3 Pathfinding in transportation . . . 10

2.2 Knapsack Problem . . . 15

2.2.1 Introduction . . . 15

2.2.2 Solutions . . . 16

2.3 OODA Automation . . . 19

3 Solution **21**

3.1 System Overview . . . 24

3.2 Input . . . 25

3.3 Objective function estimation . . . 26

3.4 Restrictions . . . 27

3.5 Data Structures . . . 28

3.5.1 Waypoints . . . 28

3.5.2 Data structure S . . . 29

3.5.3 Restrictions . . . 29

3.6 Summary . . . 29

4 The Labeling Algorithm **31**

4.1 The base algorithm . . . 33

4.2 Modifications to the base algorithm . . . 35

4.2.1 Restriction verification 37
4.3 Time and Space Complexities 38
4.3.1 Time Complexity 38
4.3.2 Space Complexity 39
4.4 Output 39
4.5 Summary 40

5 Experimental Evaluation **41**
5.1 Setting and objectives 43
5.2 Results 44
5.3 Discussion 46
5.3.1 Labeling Algorithm 46
5.3.2 Backtracking technique 49

6 Conclusion **51**
6.1 Final Remarks 53
6.2 Future Work 54

A Labeling Algorithm **61**

List of Figures

1.1 Waypoints example. 4

2.1 Path planning levels. Reproduced from Souissi et al. 9
2.2 Goals of guidance. 11
2.3 Refraction of light. 13
2.4 Network Model, adapted from Captivo et. al [1]. 17
2.5 OODA cycle, adapted from “The Essence of Winning and Losing”, JR Boyd [2]. 20

3.1 System architecture overview. 25
3.2 Value calculation for the first objective. 26
3.3 Problem in immediate fuel consumption value. 27
3.4 No-fly zone issued in www.notams.faa.gov 28

4.1 Example reproduced from Captivo et al. 34
4.2 Plane around the world. 37

5.1 An example of a possible map. 43
5.2 Time consumption by the labeling algorithm. 47
5.3 Memory usage by the labeling algorithm. 47

List of Tables

5.1 Average results of the labeling algorithm 44
5.2 Execution time of the labeling algorithm including backtracking 46

List of Algorithms

1 A labeling algorithm to determine efficient paths 62
2 The modified labeling algorithm 64

Acronyms

ATC - Air traffic control
ETOPS - Extended-range Twin-Engine Operational Perfomance Standards
FMS - Flight Management System

1

Introduction

Contents

1.1 Motivation 3
1.2 Objectives 4
1.3 Outline 6

1.1 Motivation

Aviation is growing every year and is expected to double over the next 20 years [3] and more and more challenges surrounding airplanes such as safety, eco-friendliness and increasing air traffic become more and more discussed.

Airplanes are capable of flying autonomously from take-off to landing when considering standard normal operations but when trouble arises, pilot intervention is often needed [4]. Some of these issues require the pilots to adjust the route, sometimes the adjustment ending in a completely new destination.

The issues can be categorized into two different types: internal and external. Internal issues are problems the airplane might be having internally such as engine failure or a medical emergency. External issues are problems outside of the airplane such as weather storms or heavy traffic that might pose danger to the aircraft.

Some of the possible routes are already predetermined by the pilots prior to departure but sometimes a deviation is needed and when that happens the new route found may not be optimal. When considering what is optimal, there is a plethora of possible restrictions and criteria to follow but aircraft safety, fuel efficiency are usually the ones that matter the most.

Flight planning is the process of producing a flight plan that describes a trajectory to be taken from a start point to an end point. It involves calculating how much fuel is needed, the route that takes us to the arrival point safely while complying with ATC regulations and safety rules. Flight planning depends on a lot of factors. Depends on the distance, the weather conditions and the aircraft used. Companies are always required to take a surplus of fuel for safety reasons. This planning is not trivial and is never a one-time process. Varying weather conditions such as tail wind might increase or decrease the fuel needed. Weather storms might block some waypoints requiring the plane to take one another. It is important to note that an airplane has restrictions on its own movement, subject to its own flight dynamic restrictions, meaning that for example it can not change altitude instantaneously or change its course suddenly without affecting either passenger comfort or even the safety of the aircraft itself. This also has to be taken into account in flight planning.

There are already several flight planners able of generating routes, however when the aircraft is already airborne, it becomes dependent on the pilots manually adjusting the course of the route through ATC advice. And this is something that can be theoretically fixed with an efficient on-board flight-planner that can receive input to changing conditions that might require route adjustments and automatically adjust it without taking away attention from the pilots, letting them monitor flight systems more closely.

We propose a flight-planner that can efficiently recalculate a route in the presence of adver-

sarial conditions and automatically adjust the trajectory of the route while maintaining optimality, safety and being compliant with the restrictions imposed.

The rest of this chapter defines the objectives of this thesis, the solution proposed, and the outline of the thesis.

1.2 Objectives

At the present day, routes are generated before the airplane takes off and if there is a problem that surges when the aircraft is already airborne, like a weather storm, the pilots and/or the ATC have to manually re-plan the route to avoid that problem [3] [4] [5]. This can be just a change of one or two waypoints or a much more complex process that can even end in finding a new destination. A waypoint is a pre-determined intermediate point in a route at which course is changed leading to a new waypoint. As seen in Fig. 1.1, KCPS and KMDW are respectively the departure and arrival points while CSX, TEHWY, SPI, ...,etc., are waypoints.

Figure 1.1: Waypoints example.

Our work focus mainly on how to efficiently calculate and adapt routes while the aircraft is already airborne by utilizing the current position of the aircraft and avoid all the obstacles that arise while full-filling the requirements set. Our solution intends to prove that an on-board optimizer is efficient and can be used as a replacement to manual recalculations of the route.

There are already very efficient and fast algorithms such as the randomized potential field [6], the random walk-planner [7] and the rapidly exploring random tree [8], however these solutions are based on random sampling leading to potential non-optimal solutions when subject to a pre-specified criterion.

This problem can be represented as a multi-objective problem and by using well-known solving methods of this type of problem we can reach an optimized route. We convert first approximate the multi-objective problem to a knapsack problem and then use a state-of-the-art algorithm to solve it.

In order to obtain the optimal route, we do the following:

1. Find the optimal geometrical path (trajectory) between 2 points in a 3D space with possible obstacles.

2. Find the optimal trajectory between 2 points in a 3D space with possible obstacles while it being feasible. This means adding restrictions that the plane is subject to such as speed or the rate-of-climb of the aircraft.

While there are numerous situations where route recalculation might be needed, we restrict this type of problems to three situations:

1. Finding the optimal route considering weather problems.

2. Finding the optimal route considering there is a situation that requires the plane to land as soon as possible.

3. Considering both option (1) and (2) and adding ETOPS [1] restrictions to it. ETOPS (Extended-range Twin-engine Operational Performance Standards) functions as a certificate awarded to airplanes that comply to a set of standards allowing twin-engine aircraft to fly routes which at some point are more than 60 minutes of flying time from the closest airport for an eventual emergency landing. There are different levels in an ETOPS certificate, the highest awarded so far was the ETOPS 330 to the Boeing 747-8i as well as the Airbus's A340 and A380.

[1] http://www.boeing.com/commercial/aeromagazine/articles/qtr_2_07/article_02_3.html

These situations are solvable if we can solve the final part consisting in gathering the optimal route in which all restrictions are possible.

To show that our solution is optimal, we ran a multitude of tests which ensures both the efficacy and the efficiency of the tests.

1.3 Outline

The remainder of the Thesis is composed by five additional chapters.

In the next chapter we present the State of the Art where we describe the main concepts and present the background related to pathfinding, route optimization and the knapsack model with special focus on the labeling algorithm we are gonna use in our solution. We also describe existing solutions to route planning.

In Chapter 3 we propose a solution and its architecture with reasoning as to why it will be a good and efficient solution. We also outline the system overview an detail the input treatment as well as the data structures used.

In Chapter 4 we explain the labeling algorithm in our solution and the optimizations made to our system as well as how we generate the final output.

In Chapter 5 we present the results obtained and make a critical discussion of them.

In Chapter 6 we present our conclusions, the limits of our system and future work that could be done to further improve our solution.

2

State of the Art

Contents

2.1 Pathfinding 9

2.2 Knapsack Problem 15

2.3 OODA Automation 19

2.1 Pathfinding

2.1.1 Introduction

Pathfinding has been a problem studied throughout the years. It can be considered as the process of finding the shortest path between two nodes with or without obstacles in between. Djikstra's algorithm invented in 1956 [9] is the simplest form of pathfinding in a weighted graph. Since then, a multitude of pathfinding algorithms has been invented and improved on: A* [10] is an improvement on Dijkstra's algorithm where instead of just using the real costs per node, the search is now conduted based on heuristics. Ford-Fulkerson [11] is an algorithm that computes the maximum flow of a flow network, where we can find optimal paths in arcs that can have at max x flow. All this research has led to each situation requiring a specific algorithm and this algorithm could be adapted into different situations. For example, Bellman-Ford [11] can be used with or without a node queue and could be parallelized if some conditions are verified.

2.1.2 Pathfinding State of the Art survey

In 2013, Omar Souissi et al. did a survey and presented a state of the art in path planning in the field of automation, robotics and video-games [12].

Figure 2.1 shows an overall hierarchy of how pathfinding problems can be classified according to these authors.

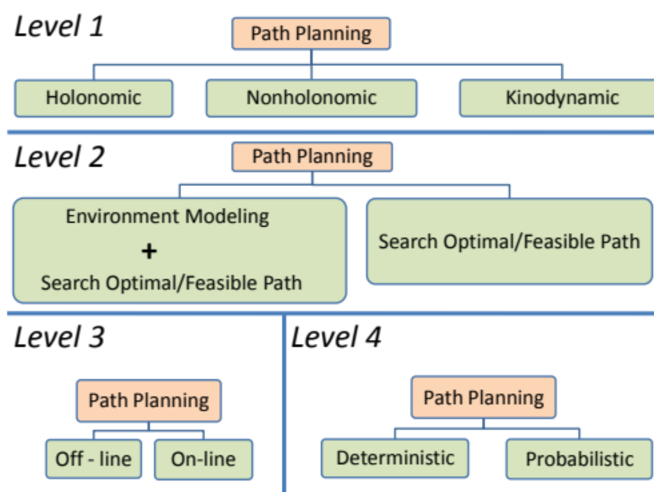


Figure 2.1: Path planning levels. Reproduced from Souissi et al.

Overall, the authors present a lot of different ideas to model a problem and algorithms that could be useful in modern day problems. However, in this paper the authors claim that there is a separation in the second level where one can either have environment modeling and not so optimal solutions or disregard the environment modeling and get optimal or near optimal solutions.

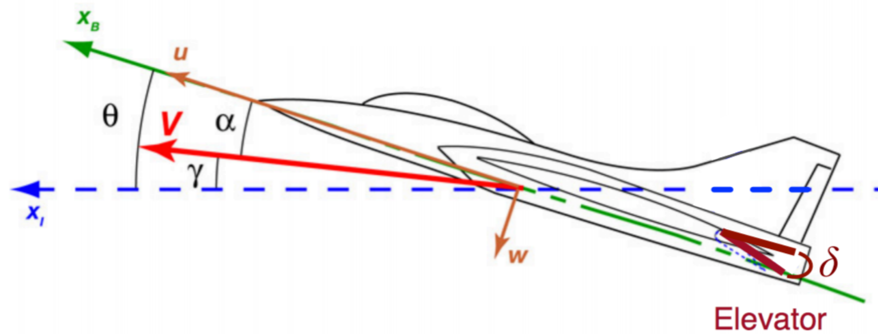
2.1.3 Pathfinding in transportation

In transportation, pathfinding has always been a problem. Everyone wants to get to their destination in the best way possible, but sometimes what is best is subjective. While some prefer to pay tolls to save time, others would rather go through toll-free roads and save money instead of time.

In 1931, the Zermelo's Navigation Problem [13] was the first posed optimal control problem. The problem consisted in finding an optimal path for a boat stuck in water with water currents and wind. If we do not consider the presence of these obstacles, the optimal path is a straight line from start to finish relating to the shortest path. However, when considering these obstacles, the optimal path is usually not a straight line. The same analogy can be used in aviation, obstacles can represent weather conditions or restricted airspace for example, leading to a path that is not a direct line from starting point to end point. Route planning has always been target of research and so there an uncountable number of thesis and works that describe multiple ways of finding the optimal path. We will describe some that pertains to our work the most.

The main goal of guidance applied to navigation is to provide a reference velocity, a path angle and a heading to enable the aircraft to follow waypoints $P_0, P_1, ..., P_N$ as suggested by Figure 2.2. However, there are multiple ways of finding these parameters. The FMS is already programmed so that if provided a waypoint, it can automatically give a path angle and heading into that location. The 3D path planning problem has been shown to be NP-hard [14] but many solutions have already been presented.

In 2003 Myungsoo Jun et al. proposed a method of path planning using a map of threat probabilities made using surveillance data and from there create the optimal route [15]. It starts by determining occupancy values based on sensor readings and then applying the conditional probability of occupancy using Bayes' rule. Then, they generate a digraph from the probability map in order to convert the model to a shortest path problem. To find the optimal path, the authors chose to use Bellman-Ford algorithm [16] because of the flexibility it provides, since changes in the probability map are probable and would change the route and so they could update link lengths without having to stop and restart the algorithm. They also were able to



The other variables and axes in the figure are defined as follows:

α	angle of attack	$\mathbf{V}$	velocity
γ	flight path angle	u, w	axial, vertical components of velocity
$x_{I.}$	x-axis of inertial reference frame	x_B	x-axis of body-fixed reference frame

Figure 2.2: Goals of guidance.

make the Bellman-Ford asynchronous and distributed further improving efficiency. The results showed that the algorithm could generate paths although sometimes it was the safest path and not the optimal one. The main advantage of this solution is that it is very fast in computing a path and is compatible with distributed computation. However, things like frequent acceleration and deacceleration were not considered, which would need to be improved on, since this changes fuel consumption potentially leading to a sub-optimal path in the end.

In 2006 Igor Alonso-Portillo et al. proposed an adaptive trajectory planner capable of adjusting its world model and re-computing feasible flight trajectories in response to adversarial changes [4]. The module proposed was to be included into the FMS and it could transmit information that is not currently being transmitted such as engine failure, control surface jams, anything that could cause flight dynamics to vary and therefore modify the aircraft perfomance. The proposed module was not to be seen as a replacement of the FMS but as improvement to the system robustness. The module when detecting a failure updates the flight dynamic model accordingly, generates a footprint and starts a search for a suitable landing site. This landing site is chosen from an existing database of airports that contains information such as airport location, runway specifications amongst other relevant information. The footprint generated allows to find reachable airports. After this, the module performs a constraint analysis to select minimally safe airports and the constraints are repeatedly relaxed until at least one solution is found. Then an utility function is applied based on the airport characteristics to find the most

suitable airport for an emergency landing. Finally, a trajectory to that airport is created utilizing existing tools on the FMS. Results showed that the module provided robustness to the FMS to different failure modes although more work on the generation of the trajectory was needed including taking into account wind and current weather.

In 2009 David Šišlák et al. proposed a variation of the A*, the AA* (Accelerated A*) [5]. The major draw fo the A* is that this algorithm uses a large space of memory since it stores all the nodes generated in memory.

The main difference between A* and AA* is the reduction of the state space when calculating the trajectory planning. While the precision in finding optimal paths stays around the same as in the A* star algorithm, its memory space is reduced up to 1400 times lower. AA* introduces adaptive sampling: during the expansion, child states are generated by applying vehicle elementary motion actions using adaptive parametrization. These actions are defined by a model of non-holonomic airplane movement dynamics. The parametrization varies based on the distance to the nearest obstacle. The algorithm then finds a path to the airplane which is bounded by a sphere. The results showed that the solution proposed was effective and efficient, providing an acceleration of up to 1400 times in the planning in both the case on where a path existed and in the case of no available path existing, significantly reducing the number of visited states in the search.

Alireza Babaei et al. in 2010 proposed an efficient algorithm for trajectory planning in autonomous unmanned aerial vehicles [17]. A Dubins path is a path that is the shortest path between two points satisfying curvature constraints. The authors propose a Dubins path based algorithm for UAV's in a 3D space considering wind-free space that exploits analytical and differential geometry. The trajectory planning proposed as the form of a closed-loop guidance form that generates commands in terms of waypoint configuration and the maximum curvature allowed. The results showed that the solution was effective and was suggested for dynamic environments. However, further research would be needed, since wind is not taken into account in this work.

In 4D trajectories, waypoints are defined as 3D waypoints but additionally having arrival time at each waypoint specified.

Bousson et al. in 2010 [18] proposed a 4D approach where they transformed the problem in a nonlinear programming problem using pseudospectral integration and Chebyshev polynomials. The method proposed had the advantage of having the expected arrival time calculated at each waypoint in addition to the desired position. They first formulated an optimal control problem and then transformed it into a nonlinear programming problem by parametrizing the state and control. This parametrization is important to the accuracy of the solution and in this

work has been done via pseudospectral methods and then compared to collocation methods, the most known techniques at the time. Results showed that this approach achieved appropriate solutions for the benchmarks set, which were time, path angle, heading, latitude, longitude, height and control, having the collocation methods failed to find solutions in one example while the pseudospectral methods was able to find for both examples provided. Even not considering this, the solutions generated were smoother than the ones presented by the collocation method.

In 2013, Nourelhouda Dougui et al. proposed the light propagation algorithm, an algorithm capable of generating sets of conflict-free 4D trajectories [3]. The algorithm is based on Fermat's principle of least action: *The path of a light ray connecting two points is the one for which the time of transit, not the length, is a minimum.* It finds an optimal path by computing smooth geodesic trajectories in environments with obstacles. Light tends to travel in low refraction index areas where light rays are faster (figure 2.3). Having this in mind, the algorithm programs obstacles as high-index areas, thus mimicking the light propagation behaviour. They were able to successfully apply on three different air traffic management problems. However, the authors noted that the algorithm still needed and could be optimized by computing the cluster resolutions in parallel and that the algorithm needs further improvements in more general restrictions, since it only accounted for temporal congestion and moving weather.

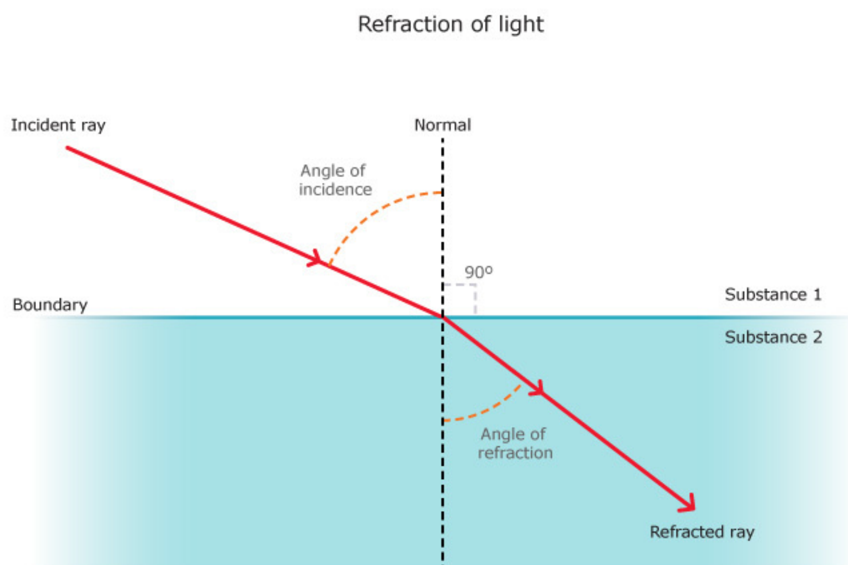


Figure 2.3: Refraction of light.

Arianit Islami et al. proposed in 2016 a large scale 4D trajectory planning system [19] in which they aimed to minimize the global interaction to reduce workload for both pilots and air

traffic controllers. With this work, the authors intended to propose a global deconfliction module that could predict and plan trajectories such a way that two airplanes would never be in conflict even in high density traffic. They proposed a mathematical formulation to achieve a trajectory planning in which they allocate alternative departure times and alternative horizontal and vertical paths. This mathematical formulation was split in three parts. The first one characterizes the uncertainty of aircraft positions and arrival times based on two different models. Then, they define interact-ion between trajectories. Finally, they present a mathematical formulation for the interaction minimization problem. After this section, they present a hybrid metaheuristics approach adapted to handle an air-traffic assignment problem at a continent scale. Since this model is based on uncertainty, there is a trade off between the robustness of the solution and the trajectory modification costs. The authors proposed that the user could consider lower uncertainty levels and iteratively solve the remaining interactions in another phases. However, this is computationally expensive since at each step we would need to recalculate if a space was clear or not and if so recalculate all the possible trajectories.

In 2017 Arno Fallast et al. proposed an automated trajectory generator for an emergency landing procedure for a CS23 aircraft [20]. The finding of an emergency landing includes selecting possible landing sites and from there create feasible trajectories. This procedure is separated in 3 processes: the module responsible for the route generation to the intended landing sites, the module that searches for available landing sites and the last module which is responsible for combining both the search for available landing sites and the route generation scoring each trajectory. The results show that the algorithm is capable of finding an optimal solution if there exists one, this solution being close to the unknown true optimum. The authors note that the work is still on-going and that further testing is necessary if using this generator.

In 2018, Russel Paielli proposed a standard Trajectory Specification Language (TSL) [21] where at any given time, they could specify an aircraft trajectory with tolerances that the aircraft would always be constrained to a precisely defined bounding space. The trajectory is specified using XML language in which everything can be included, from trajectory to a single route element and the cross track bounds.

The main benefits of using Trajectory Specification is in case of a system outage, risks are minimized since even if ATC fails, safety must be guaranteed and so, if a explicit bounding deviation from the trajectory assigned is computed, then a safe separation can be assured within a time threshold previously defined known as the conflict-free time horizon. TSL is proposed as a way to achieve these specifications and to communicate them via air/ground datalink. From air to ground, the language would be used as a trajectory request service while from ground to air, the language would be used as a trajectory assignment.

2.2 Knapsack Problem

2.2.1 Introduction

The knapsack problem is a combinatorial optimization problem and it concerns the common problem of packing the most valuable things without overloading the luggage. Formally, it seeks to select from a finite set of items, the subset that maximizes the linear function of the items chosen, subject to constraints [1]. It can be described as:

$$max f_1(y) = \sum_{j=1}^{n} v_j^1 y_j$$

$$\vdots$$

$$max f_r(y) = \sum_{j=1}^{n} v_j^r y_j$$

$$s.t. \sum_{j=1}^{n} w_j y_j \leq W$$

where:

- n is the number of items,
- r is the number of criteria,
- v_j^k is the k*th* profit of the j*th* item, for $k = 1, ..., r$,
- w_j is the weight of item j,
- W is the capacity of the knapsack.

We can define dominance using the following definition [22]:

Let z' and z" $\in \mathbb{R}^n$ denote two criterion vectors. z' dominates z" if z' $\geq$ z" and z' $\neq$ z". z' strongly dominates z" if and only if z' $>$ z".

Non-dominance can be defined as:

Let z,$\overline{z} \in \mathbb{R}^n$. z dominates $\overline{z}$ if there does not exist z $\in$ Z such that z $\geq \overline{z}$ and z $\neq \overline{z}$. Otherwise, $\overline{z}$ is a dominated vector.

The concept of dominance is very useful because it is the definition of an optimal solution in a solution set. A non-dominated vector represents a vector that is better or worse, depending on the viewpoint, in at least one of the components when compared to another set of vectors. A set of non-dominated solutions or vectors consists in a Pareto front.

A knapsack model can modeled into a pathfinding problem [1] and thus an optimal trajectory can be found using this model. Indeed, we can choose as objectives what we want to maximize or minimize in our set, such as fuel costs or travel time and set the restrictions as to things like no-fly zones, passenger comfort, etc. The time complexity of solving the single-objective is $\mathcal{O}(nW)$.

2.2.2 Solutions

In 2003, M. Eugenia Captivo et al. proposed a method in *Solving bi-criteria 0–1 knapsack problems using a labeling algorithm* [1] where we can formulate the knapsack problem as a shortest path problem [23] by transforming the multiple criteria knapsack problem into a multiple criteria shortest path problem over an acyclic network and then proceed to label it. It first converts a knapsack model to an acyclic network model.

To convert the model to an acyclic network we first need to determine the set of nodes and the set of arcs and respective arc costs. The set of nodes can be found by using a layer technique. The first layer has only the starting node, each progressive intermediate layer can contain several nodes and the last layer only has one node. Layer j can be directly obtained from layer $j-1$ where each layer has at most $W+1$ nodes.

The starting node s always has 2 outgoing arcs, one with cost 0 and the other with cost $-v_1$. The first arc represents the decision of not including item 1 in the knapsack while the second one represents the decision to include it. After that, in each layer, each node j^a where $a = 0, ..., W$ has at most 2 arcs. The first arc has cost 0 meaning that item $j+1$ is not included in the knapsack. The second arc has cost $-v_{j+1}$ if $a + w_j + 1 \leq W$ meaning that, node $j+1$ is included in the knapsack if and only if the knapsack has capacity to contain this item. All nodes from the final intermediate layer are connected to the final node t. The network model can be seen in figure 2.4 which was generated from the following example:

max $f_1(y) = 8y_1 + 9y_2 + 3y_3 + 7y_4 + 6y_5$

max $f_2(y) = 3y_1 + 2y_2 + 10y_3 + 6y_4 + 9y_5$

s.t. $3y_1 + 2y_2 + 2y_3 + 4y_4 + 3y_5 \leq 9$

where $y_j \in \{0, 1\}$ for $j = 1, ..., 5$.

The network generated by this algorithm has no cycles, every feasible solution of the knap-

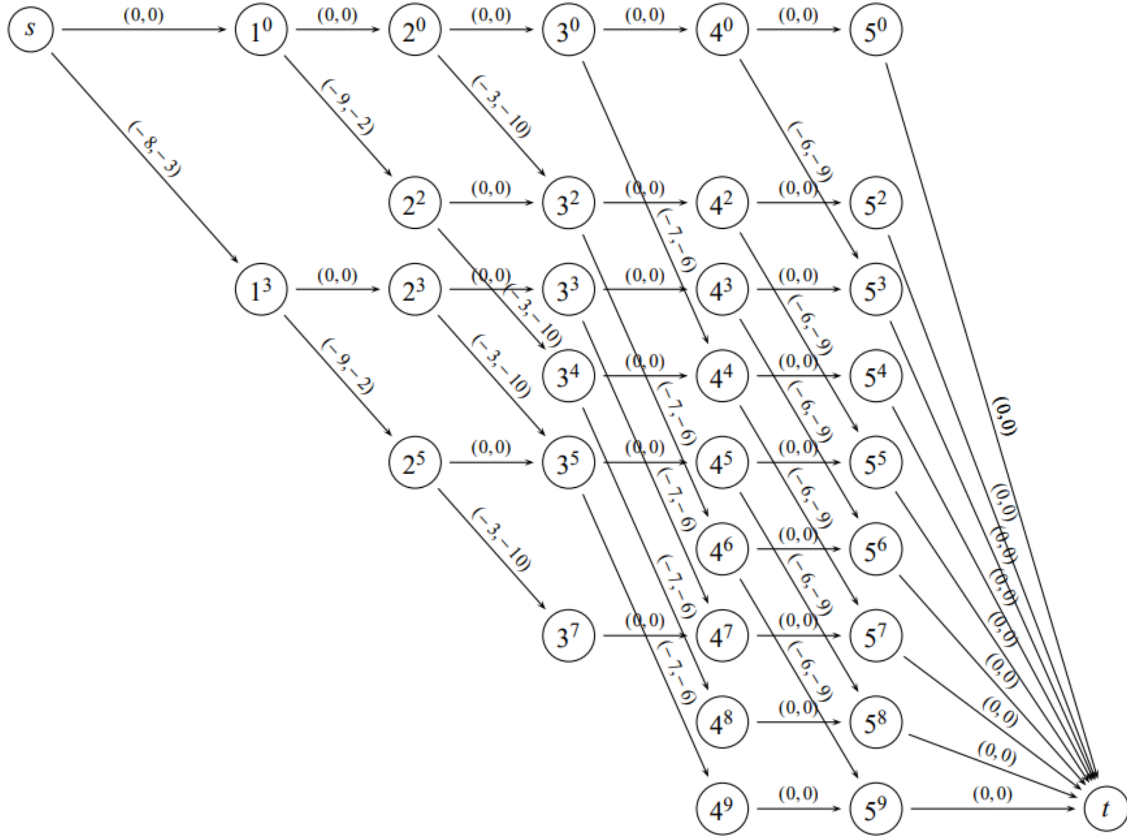


Figure 2.4: Network Model, adapted from Captivo et. al [1].

sack problem has a corresponding path in the network generated from starting node s to end node t and the shortest path from s to t represents the optimal solution for the knapsack model and the value is the negative cost of the shortest path in the network.

This transformation is only for a single criterion but the generalization is very easy to do. Instead of just assigning one cost per arc, we assign a vector of costs per arc, each cost corresponding to one different criterion.

After converting it, to label it, it is ordered lexicographically but adding an additional property that other labeling algorithms do not have: the values concerning the first criterion are placed in non-decreasing order, while the values of the second one are placed in non-increasing order. So, to see if a new given label is dominated or not, it is only necessary to compare this new label with the last non-dominated label determined. For more than two criteria the new label must be compared with all the labels already determined.

Algorithm 1 (Appendix A) shows the labeling algorithm proposed by the authors where T and V are two lists of superscripts related to the nodes in layer j and $j+1$ and $S(j^a)$ is the set of non-dominated labels concerning the set of paths from s to j^a. The time complexity of this

algorithm is $\mathcal{O}(nW)$.

In 2009 Cristina Bazgan et al. proposed an algorithm to efficiently solve the 0-1 multi objective knapsack problem [24]. The main idea of the algorithm was to disregard partial solutions that would lead to new non-dominated criterion vectors by using several complementary dominance relations.

It was claimed to be the most efficient algorithm existing with up to 4000 items in less than 2hrs in the bi-objective case comparing the results obtained with Captivo et al. [1], the most efficient method at the time.

In 2013, J. Figueira et al. in *Algorithmic improvements on dynamic programming for the bi-objective 0,1 knapsack problem* [25] proposed an optimization to the at the time state-of-art algorithm which was the one proposed by Bazgan [24]. The optimizatin was based on applying several dominance relations to discard elements from the solution pool. It proposed 3 techniques to do this:

1. Computing a small set of efficient solutions by solving a sequence of weighted scalarized bi-objective 0, 1 knapsack problem with dichotomic search.

2. Computing the nondominated extreme vectors of the relaxation of the problem solved by a bi-objective simplex algorithm and then using the *improvement* method [26] to restore feasibility.

3. Generating feasible extensions for each partial solution at a given stage k by adding its profit to the profits obtained through dichotomic search on the reduced bi-objective $0, 1$ knapsack problem with the remaining $n - k$ objects.

The experiments led using these techniques resulted in average of a 20% CPU time improvement and significant less memory used than in the original method. Results showed improvement even in the most difficult problems in the literature.

In multi-objective problems its very rare to reach only one optimal (or nondominated) solution. Instead, we reach what is called a Pareto front, a set of nondominated solutions, being chosen as optimal, if no objective can be improved without sacrificing at least one other objective.

There are techniques designed to pick one of these solutions. We will talk about two of the most known ones: TOPSIS [27] and AHP [28]

The Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) was proposed by Yoon and Hwang in which the ideal solution would be the one to have the shortest distance to the ideal solution but also the farthest distance from the negative-ideal solution.

TOPSIS assumes that each attribute in the decision matrix takes either monotonically increasing or monotonically decreasing utility. With this in mind, it is easy to locate the ideal solution, this is the one being the closest to the optimum but farthest from the non-optimum. For example, if a solution S_1 has the closest distance to the optimum O but also the closest distance to the non-optimum NO while a solution S_2 is close to O but the farthest away from NO then it gets really hard to justify the selection of S_1.

The Analytic Hierarchy Process (AHP) is a technique for organizing and analyzing complex decisions proposed by Thomas Saaty. Here, factors relevant to the decision are arranged in a descending hierarchy from an overall goal to criteria, sub-criteria and alternatives in successive levels.

When constructing the hierarchy is important to include enough relevant detail to the environment surrounding the problem, the correct identification of both the issues and the attributes that surround the problem and to identify the participants associated with the problem. The hierarchy serves two purposes: an overall view of the problem and the quantification of the magnitude of the issues; if two are on different levels then it is easier to compare them accurately.

In AHP a scale of measurement consists of three elements: a set of objects, a set of numbers and a mapping of the objects to the numbers. The numbers are measures, and while in standard scales measures could be standard units such as meters or radians, AHP generates relative ratio scales of measurement, these ratios being obtained through normalization. However, the normalization and composition of weights of alternative with respect to more than one criterion measure on the same standard scale it leads to nonsensical numbers since normalizing separate sets of numbers destroys the linear relationship between them. In order to be able to use AHP, the weights must be first composed and then normalized. After building these sets, we can get the priorities by adding benefits and subtracting costs of decisions, getting the optimal decision.

2.3 OODA Automation

OODA (observe-orient-decide-act) is an information strategy concept developed by Colonel John Boyd [2] for information warfare. It was created for military purposes, although nowadays is used for business and comercial strategies. According to Boyd, decision-making occurs within a cycle: firstly, make observations on the world around the system (the enemy). After that, an orientation would follow. This consisted on the formulation of a decision hypothesis. This decision would take things like cultural tradition and previous experience as basis for the hypothesis. After that, the decision making would take place and finally it would be execution

of the decision.

Boyd argued that whoever completed the OODA cycle would win, since if we executed the decision before our enemies we would lengthen their decision making since they would have to process our movement.

Figure 2.5 shows an overall view of the OODA cycle.

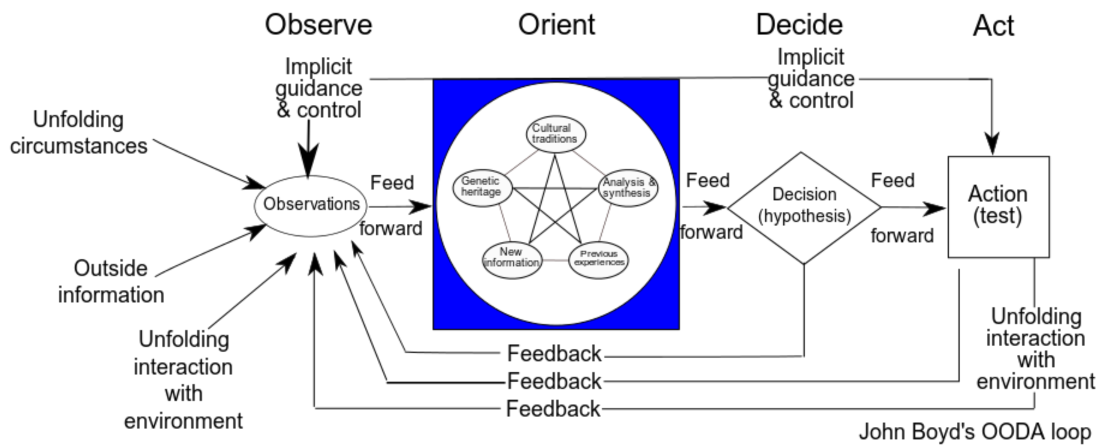


Figure 2.5: OODA cycle, adapted from “The Essence of Winning and Losing”, JR Boyd [2].

In this cycle, feedback is the concept that decisions do not have to necessarily become actions. The optimal course may remain the same, even if some condition changed. The implicit guidance and control means that the process occurs in the background and so the course only changes when the decision is final and not before that.

3

Solution

Contents

3.1 System Overview 24
3.2 Input 25
3.3 Objective function estimation 26
3.4 Restrictions 27
3.5 Data Structures 28
3.6 Summary 29

All the previous work, while achieving relevant results to their own studies, none presented a clear module that worked efficiently that could be used online in an autonomous non-tripulated vehicle considering a wide number of restrictions. While some proposed a module for an emergency landing [20], it would be limited to certain types of aircraft (CS23). Others, do not consider wind variations or another type of restrictions in their planning.

Overall, the solutions proposed were designed for efficacy and not efficiency. Even if some factors such as wind or other factors were not considered in the solution, the main problem resides in that it would take too long for the current proposed flight-planners to find a feasible path for an on-going flight. The objective of this work would be to correct this issue that most flight-planners have while adding the dynamism that most flight-planners lack.

Our module receives as input the current trajectory in the form of: current position, optionally the waypoints that must be followed after and the destination point. These positions are represented in the form of *Latitude Longitude Altitude*. In addition to the trajectory, we receive a series of restrictions as well.

These restrictions are split in two big groups: Aircraft restrictions and route restrictions. Aircraft restrictions are constraints that the aircraft has such as speed, turn-rate and fuel consumption. These restrictions are variable and depend on which aircraft model is being used. The route restrictions can be divided into two sets: dynamic and static. The static restrictions are set restrictions by the ATC known prior to departure to the pilots such as restricted airspace, also known as no-fly zones, or zones with limited speeds. Dynamic restrictions are restrictions that occur when the aircraft is already airborne. Weather storms, medical emergencies, all of these constitute dynamic restrictions.

As output our module returns the replanned route in the form of a start point, waypoints to be followed and the destination point. Each waypoint is represented as in the input.

In terms of space search, we decided that a straight line could be followed from point A to point B as long as the segment line formed by these points didn't break any of the restrictions imposed.

We reached the conclusion to use Captivo et al. labeling algorithm [1]. This algorithm is simple to use and is by itself very time efficient despite potentially using a large portion of space and ready to make updates in real time if adversarial changes happen.

In our problem we will inevitably reach situations where we will have to choose a solution from this set. A potential solution is to use TOPSIS, AHP or to find the knee-point of the Pareto front and use it.

It is important to refer that our paths are not just in 2D meaning that a 2D graph is not

enough to find the optimal path since altitude variations are possible, thus making the problem more difficult.

Overall, we intend to provide a solution that companies are able to pick up and adapt easily to their needs and their priorities, giving flexibility in the restriction set and their values. This is made possible by not restricting our algorithm to a fixed set of restrictions but instead make it dynamic and adaptive to sudden changes. For example, before taking off, the weather was clear, but after departure a weather-storm developed. Then it could be modeled as a restriction and from there we could make changes in the route. The other way around could also be used as an example, if a weather-storm was taking place and then it disappeared.

Our module will only act if conditions change, meaning that only if we receive a signal that a condition has changed, only then we would verify the current route and changed it if needed be.

We used the OODA cycle as guidance to what the solution is able to do. The module observes that conditions were changed, makes an hypothesis on a new route (it could be the same route), tests it and then forwards it, sending feedback back to another modules on the change that happened.

This cycle could be used in pathfinding problem and specifically in our case because:

1. In the observing phase, the changing conditions would be the restrictions: whether they appear or disappear, they change the environment possibly making new routes.

2. If conditions change and new routes are possible, then it makes sense to verify effectively if a better route exists and if the current course (orientation) should be changed.

3. A different module would then perform a test on its feasibility and if it is feasible it would act upon it and change current course.

For the rest of the chapter, we will provide a high-level overview of our system, the input treatment, how we define the objective functions, how are restrictions set and the data structures used to store information. The latter is rather important since the time it will take to run the algorithm will greatly depend on how the system is implemented and how we manipulate the existing the data. The final section makes a summary of this chapter.

3.1 System Overview

Our system can be subdivided into 3 big phases: the input phase, the algorithm phase and the output generation where we run a backtracking technique to find the waypoints that lead to the

solution found by the labeling algorithm. Figure 3.1 gives a logical insight on how the module works.

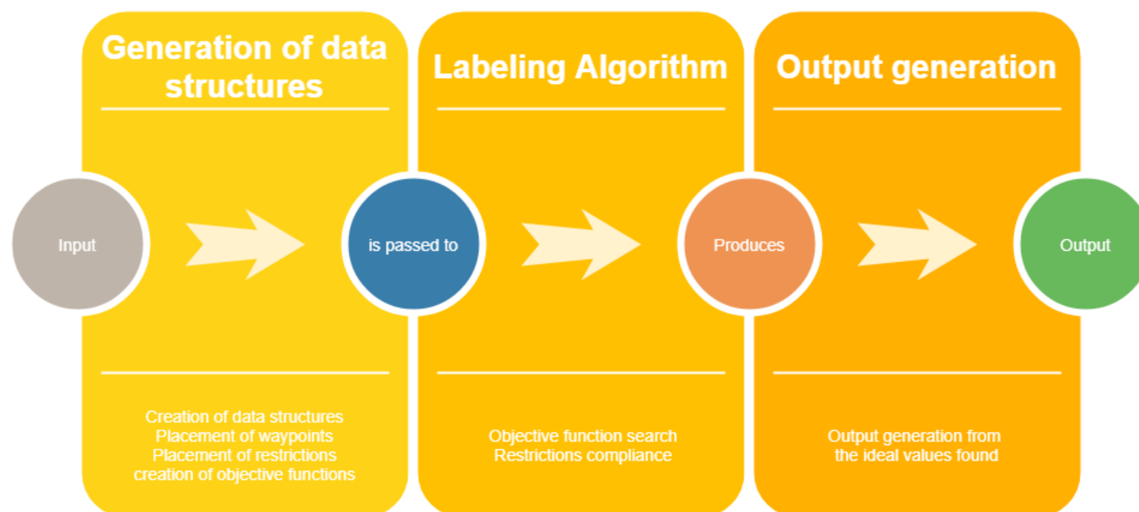


Figure 3.1: System architecture overview.

The input phase consists in how we process the data that we will need later on to run the algorithm, initializing the data structures where we store the data, calculating the first part of the objective function and where we process the restriction set.

In the algorithm phase, we run all the necessary data through the algorithm in order to search for a feasible and optimal solution that complies with all the restrictions given.

The output generation consists in picking one solution from the optimal set and running a backtracking technique to find the set of waypoints that lead to the labels obtained by the labeling algorithm, these waypoints being the solution to the problem and then returning them.

3.2 Input

The input consists in a starting point, then a number that represents the number of optional waypoints to be passed, the corresponding number of waypoints, and the destination point. This destination point is saved as the last member of the list of all waypoints that could be chosen in the solution path;

After, we receive another number that contains the number of restrictions we will receive, the restrictions and in the end, a few aircraft parameters such as speed, fuel and weight.

The restrictions are set by a type and then the corresponding parameters. A type 0 restriction expects a sphere, so the 4 following values will be the center of the sphere in form of latitude

longitude, altitude and then its radius in kilometers.

The set of all existing waypoints is expected to exist prior to run-time and is loaded when the algorithm begins.

3.3 Objective function estimation

For this problem, we decided that having two objective-functions was enough to reach optimal solutions. The first one is to minimize the time needed to reach our destination and the second one is to minimize fuel consumption.

In order to find the first objective-function, we defined a line segment between our starting point and our destination point and the distance between the selected waypoint and the line segment - which forms a perpendicular line to the initial segment - is used to calculate the value. The farther the distance of the waypoint to the line segment, the lower the value as shown in Figure 3.2, where s and t are our start and end points respectively and A and B are waypoints. Therefore, the value of B will be higher because it is closer to the line segment than A.

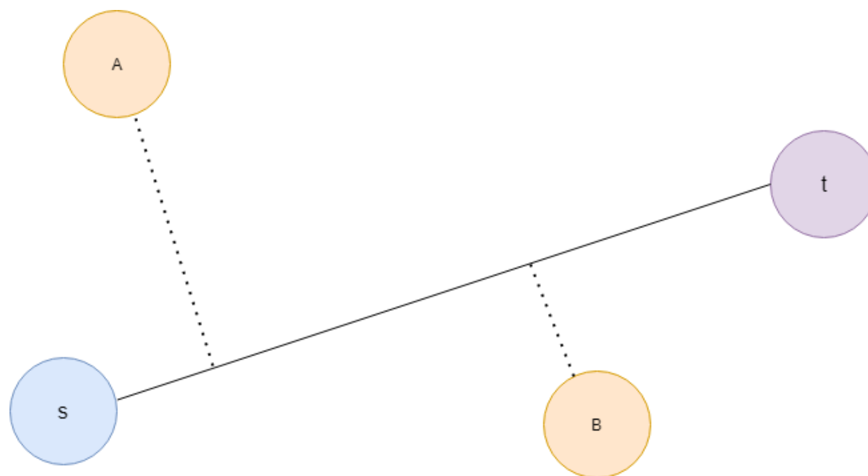


Figure 3.2: Value calculation for the first objective.

To calculate the second-objective function, we could not immediately calculate the value of a waypoint because it depends on where the plane comes from. Figure 3.3 exemplifies this problem. We can not set the value of waypoint A or waypoint B because it depends on where it comes from and where it goes to. For example, if waypoint A goes to waypoint B, it requires one amount of fuel but if it goes to waypoint C it requires a different amount. We then decided to leave this determination to when we run the labeling algorithm.

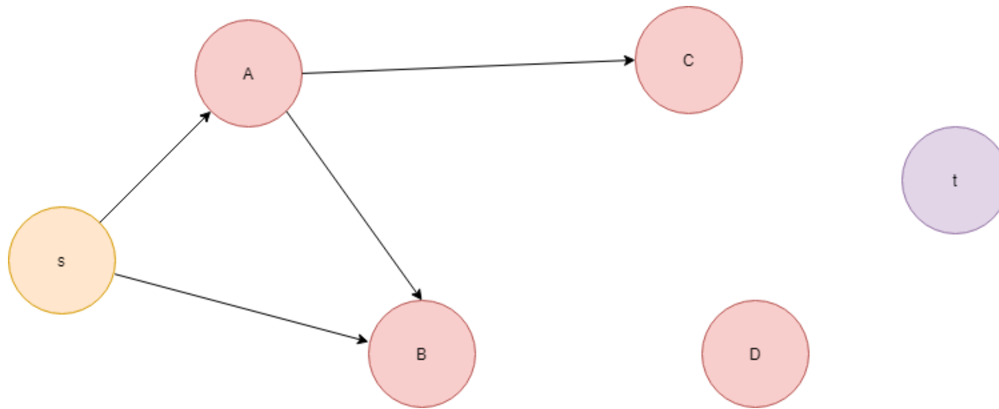


Figure 3.3: Problem in immediate fuel consumption value.

3.4 Restrictions

For the restrictions, there are two types of restrictions: the space restrictions and the airplane restrictions.

Space restrictions were defined as spheres that could be placed anywhere.

They have a radius and a center with coordinates in latitude, longitude and altitude. It becomes easy then to calculate if the arc formed by the waypoints intersects with a sphere. Figure 3.4 represents a no-fly zone issued around the house of at the time president-elect of the USA, Joe Biden, 7th November 2020.

But, in terms of relating these restrictions in function of the waypoints, to put them as weight restrictions in the knapsack model, it was impossible to do that because for each new restriction added as a knapsack weight restriction, the labeling algorithm would need to be adapted, so that it ensured all the weights were being respected and also because the weight of each waypoint depended on where it came from. So we decided to create only one restriction function that would serve as a weight limit to the functions and the rest to be accounted for in run-time of the labeling algorithm.

The restriction we decided to use in the knapsack model was the maximum of fuel reserves that the airplane had baring 30 minutes of fuel and the problem here was the same to the problem we faced in the objective function calculation where we could not calculate the fuel consumption solely based on a waypoint, only on a set of waypoints. And so what we ensured was that the sum of all waypoints considered could not be higher than the fuel capacity of the airplane defined above.

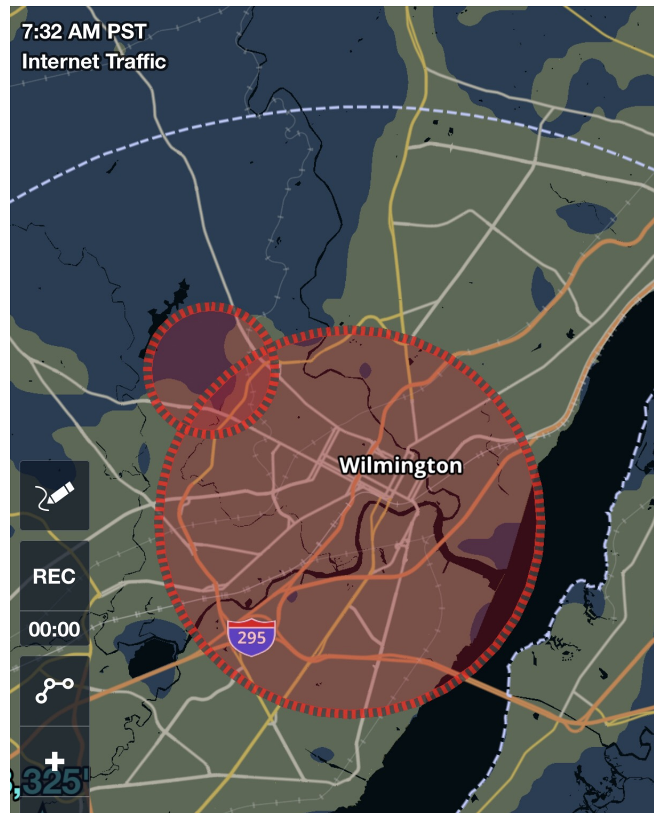


Figure 3.4: No-fly zone issued in www.notams.faa.gov .

3.5 Data Structures

Data structures are particularly important in our problem since we require that operations need to be as fast as possible. This includes finding $S(j^a)$, the restrictions set and the waypoints in an efficient way. To do this, we chose different implementations for each data type, that would fit best or at least not compromise the running time of the algorithm.

3.5.1 Waypoints

Since waypoints are always directly accessed, we have them stored in an array. Before running the algorithm, we run a merge-sort algorithm to order them based on distance to the starting point. Merge-sort has a time complexity of $\mathcal{O}(n \log n)$. Since we access directly the waypoints, then the time complexity here is simply $\mathcal{O}(1)$ for each access.

3.5.2 Data structure S

We denominate S as the data structure represented in the labeling algorithm that will always be accessed through indexes. We always know its key, for example in $S(j^a)$ the key will be j^a, which is unique for each node. So, we decided to model S as an hash table. Collisions in this hashtable are solved using linear probing. If the number of elements present in the hashtable is half of the maximum number of elements allowed in the hash table, we rehash the old table into a new one with double the size. This further prevents collisions. The amortized cost of inserting an element into an hashtable, even considering the eventual rehash is $\mathcal{O}(1)$.

3.5.3 Restrictions

The restrictions are saved using a linked list and for each arc AB, we see if it breaks any of the restrictions stored in the linked list. Since we have to verify them all and considering we have r restrictions then the cost of this operation will be $\mathcal{O}(r)$. Although we do not expect a high number of restrictions, this is something we have to take in mind in testing and if needed further improving since it could bottleneck our module.

3.6 Summary

Our module receives an input and using it and a pre-existing database of waypoints, it runs the algorithm. The first objective is set before running the algorithm, the value of it being the distance from the waypoint to closest point in the segment formed by the starting point to the end point. The second objective is dynamic and will be the fuel consumed by going to the waypoint in question from the last waypoint in the potential path solution. It outputs the set of optimal paths or none if none exists.

We retrieve the waypoints using an array which yields very little time consumption since we directly access it, data structure S is an hash table which is also very efficient and the restrictions are saved using a linked list which may not be optimal for a high number of restrictions, but is not expected to cause significant addition to the running time.

4

The Labeling Algorithm

Contents

4.1 The base algorithm 33

4.2 Modifications to the base algorithm 35

4.3 Time and Space Complexities 38

4.4 Output 39

4.5 Summary 40

In this chapter, we explain how the base labeling algorithm works, what modifications we made to making it capable of handling our problem and what optimizations we made in order to increase its efficiency and efficacy. Section 5.1 describes the base labeling algorithm, section 5.2 the modifications we made to the algorithm. Section 5.3 makes a summary of this chapter.

4.1 The base algorithm

The base algorithm as seen in Appendix A (Algorithm 1) was proposed in 2003 by Captivo et al [1].

This algorithm intends to maximize N objective functions subject to one restriction with a maximum capacity.

It starts by initializing S, where $S(j^a)$ is the set of non-dominated labels concerning the set of all paths from s to j^a, setting $S(1^0)$ with label $(0, 0, ..., 0)$ and $S(1^{w_1})$ with label $(-v_1^1, ..., -v_r^1)$ where r is the number of . It also creates two lists, T and V, where V starts out empty and T with two values, 0 and w_1, the weight value of item 1. These two sets contain all the labels of layer 1.

Then, for the remaining layers j $(j = 2, ..., n)$, the set $S(j^a)$, where $a \in 0, ..., W$ is built as the following:

1. If j^a has only one arc incoming then consider two different cases:
   (a) if the arc $((j-1)^a, j^a)$ belong to T but $a - w_j$ does not, then $S(j^a) = S((j-1)^a)$
   (b) if the arc $((j-1)^{a-w_j} j^a$ exists, i.e a does not belong to T but $a - w_j$ does then labels in $S(j^a)$ can be defined by summing $(-v_j^1, ..., -v_j^r)$ and $S(j-1)^{a-w_j}$
2. If two incoming arcs, a and $a - w_j$ then the labels in $S(j^a)$ by choosing the non-dominated labels simultaneously from the labels in $S((j-1)^a)$, and from the labels obtained by summing $(-v_j^1, ..., -v_j^r)$ to each of the labels in set $S((j-1))^{a-w_j}$

Because this last case is rather confusing, we present an example extracted from Captivo et al. 2003 paper to clarify it.

Consider the following example:
max $f_1(y) = 8y_1 + 9y_2 + 3y_3 + 7y_4 + 6y_5$
max $f_2(y) = 3y_1 + 2y_2 + 10y_3 + 6y_4 + 9y_5$
subject to: $3y_1 + 2y_2 + 2y_3 + 4y_4 + 3y_5 \leq 9$
where $y_j \in \{0; 1\}$ for $j = 1; ...; 5$.

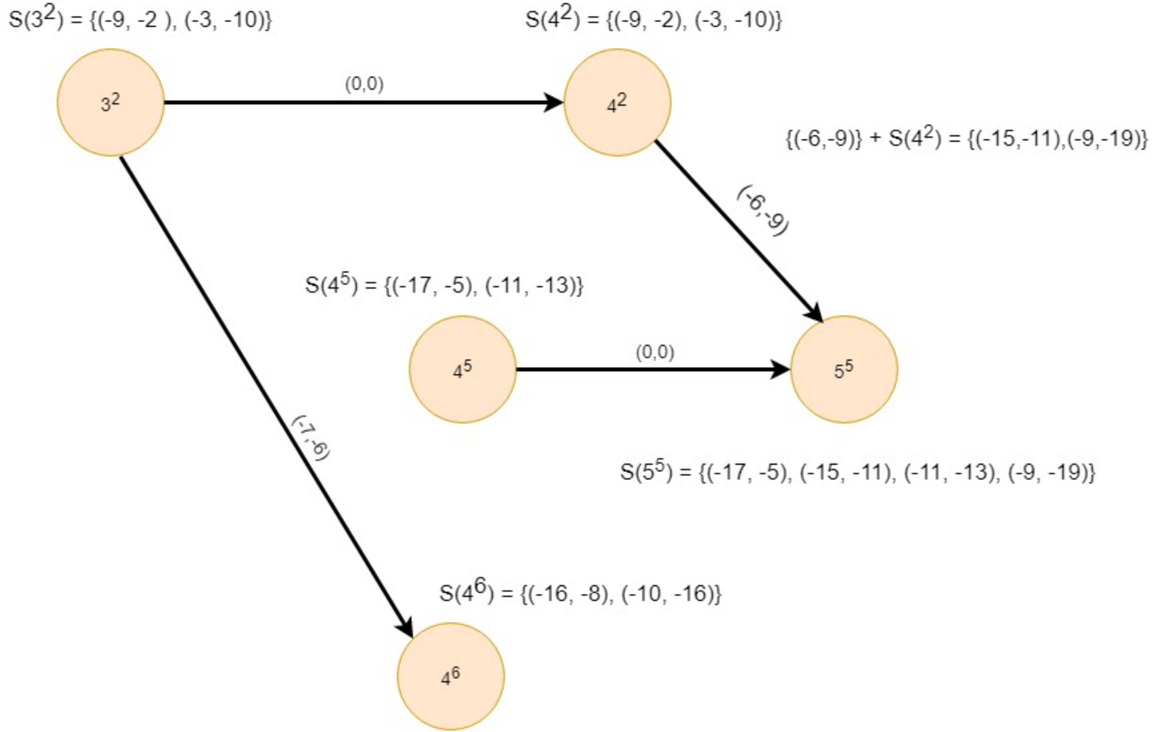


Figure 4.1: Example reproduced from Captivo et al.

In this example we have that:

$S(4^2) = \{(-9,-2),(-3,-10)\}$,
$S(4^5) = \{(-17,-5),(-11,-13)\}$,
$c^1(4^2,5^5) = -6$, $c^2(4^2,5^5) = -9$,
$c^1(4^5,5^5) = c^2(4^5,5^5) = 0$.

The labels in $S(5^5)$ are determined in the following way (figure 4.1):

1. Compare the first label of $S(4^5)$ with the label obtained from the sum of the first element of $S(4^2)$ and $c^1(4^5,5^5), c^2(4^5,5^5)$, $(-17,-15)$ with $(-15,-11)$. The lexicographic minimum between these two elements is $(-17,-15)$. So this is the first label of set $S(5^5)$. This label can be removed from $S(4^5)$, and so the first element of this set becomes $(-11,-13)$

2. Similarly, compare $(-11,-13)$ to $S(4^2)$ plus $c^1(4^5,5^5), c^2(4^5,5^5)$ - $(-15,-11)$. The lexicographic minimum here is $(-15,-11)$. To know if this label is dominated or not, compare it to the last label inserted in $S(5^5)$. $(-15,-11)$ is non-dominated, so we add it to $S(5^5)$ and remove label $(-9,-2)$ from $S(4^2)$, where the first label is now $(-3,-10)$.

3. Compare the first label of $S(4^5)$ with the first label of $S(4^2)$ plus $c^1(4^5,5^5), c^2(4^5,5^5)$, i.e

$(-11, -13)$ with $(-9, -19)$. Here, the lexicographic minimum is $(-11, -13)$. Comparing this label with the last label of $S(5^5)$, we can see that it is also non-dominated. Then, this label is added to $S(5^5)$ and $S(4^5)$ becomes an empty set.

4. Lastly, compare the last label of (4^2), $(-9, -19)$ with the last label of $S(5^5)$. This label is also non-dominated and so it is added to $S(5^5)$ and $S(4^2)$ becomes an empty set as well. So $S(5^5) = \{(-17; -5); (-15; -11); (-11; -13); (-9; -19)\}$

The problem solutions, $S(t)$ can be given as the set of non dominated labels of $\bigcup_{a=0}^{W} S(n^a)$ and the items chosen can be found by using a backtracking technique.

Since we are in a multi-objective problem, there can be more than one solution to choose from. We arbitrarily decided to choose the one that minimizes flight-time.

Backtracking is an algorithmic technique to solve a problem by recursively try to build a solution incrementally, one piece at a time, removing pieces that fail to satisfy the constraints given at any point in time. In this case, we have two objective functions and so we could backtrace both functions to find the items that satisfy both conditions. Since recursive calls are computionally expensive, what we do is to find all solutions that satisfy the first objective function, save the indexes of the items that lead to the solution and then run a loop where we try to satisfy the second objective by calculating $f_2(y)$ with the items we considered.

For example, using the example above, one of the solutions is $\{(16, 25)\}$. Easily, we can see that in $f_1(y) = 8y_1 + 9y_2 + 3y_3 + 7y_4 + 6y_5$, the solutions that lead to 16 are $\{y_2, y_4\}$ or $\{y_3, y_4, y_5\}$. So the indexes we save are $\{(0, 1, 0, 1, 0), (0, 0, 1, 1, 1)\}$. Then, to obtain the final solution we try to calculate $f_2(y) = 25$. $0 \times 3 + 1 \times 2 + 10 \times 0 + 6 \times 1 + 9 \times 08 \neq 25$ but $0 \times 3 + 2 \times 0 + 10 \times 1 + 6 \times 1 + 9 \times 1 = 25$. We can then conclude that the solution that satisfies the label is $\{(0, 0, 1, 1, 1)\}$ and retrieve the waypoints the indexes refer to.

4.2 Modifications to the base algorithm

The base algorithm was made with a very specific problem in mind, the knapsack problem. However, since our problem is not directly related to knapsack, we need to adapt it in order to function with our problem.

The algorithm is made considering only one single capacity restriction and not multiple restrictions that could hinder movement, like no fly zones, but that are in no way related to the item set and still affect the order in which items are selected. We thus decided that the capacity restriction, W, should be the fuel the airplane was carrying, leaving the rest of the restrictions to run-time verification.

Since we have more than one restriction in our problem, we had to change the algorithm. Algorithm 2 shown in Appendix A shows the pseudo-code of the base algorithm with the changes we made.

We start by running merge-sort on the waypoints, ordering them by distance to starting point s.

As referenced in chapter 4, it is impossible to estimate a weight cost of an item, solely based on it. It has to take into account the waypoint it comes from. We then added the change, that every time we change the value being assessed a, we recalculate all the values that w_j can become because we know where it comes from if $S(j-1)^a$ is defined. We add a last item parameter to every $S(j)^a$, referencing the last item added to the set at that point. For example, if j is added to the possible solution set, then the last item parameter of $S(j)^a$ is j. If j is not added to the solution set but $S(j)^a$ exists, then the last item in $S(j)^a$ is equal to the last item contained in $S(j-1)^a$. The last item in $S(1^0)$ is the starting point s and the last item in $S(1^{w_1})$ is 1.

Since the weight of an item can then assume multiple values,we calculate all those values and introduce a new third loop, where we iterate through all the possible weights of this item.

Every time we consider adding an item to the solution set, we have to consider all the existing restrictions and if they are broken if we add this item. Since the restrictions rely on knowing where the plane is coming from, we fetch the last item where the plane was headed from $S(j-1)^a$.

The first situation in the algorithm where we create $S(j^a)$ is if only $a \in T$. We then verify if the restrictions are broke from $S(j-1)^a$ to $S(j^a)$. If not, we copy the labels from $S(j-1)^a$ to $S(j^a)$.

The second situation is if $a - w_j \in T$ and $a \notin T$. Then, we have to verify if going from the last item in $S(j-1)^{a-w_j}$ to j results in breaking a restriction. If it does, we discard the arc and nothing is created. If not, we add the labels in $S(j-1)^{a-w_j}$ plus the values of flying from the last item to that item.

The third situation is if both $a - w_j \in T$ and $a \in T$. We separately verify if j is achievable by both $S(j-1)^a$ and $S(j-1)^{a-w_j}$ Then we proceed as follows:

1. If only $S(j-1)^a$ complies with the restriction set, we proceed as if we were in the first situation.

2. If only $S(j-1)^{a-w_j}$ complies with the restriction set, we proceed as if we were in the second situation.

3. If both cases comply with the restriction set, we then choose the non-dominated labels

from both $S(j-1)^a$ and $S(j-1)^{a-w_j}$ plus the sum of the values of flying from the last item to the new item.

4. If no case complies with the restriction set, we discard the arc and continue.

For simplicity we consider that only sphere restrictions are possible in the set of space of restrictions, and since we have two waypoints (origin and destination) we can calculate if the arc composed by those two points intersects any of the existing space restrictions.

To do this, we introduce a change in the base algorithm, where if the condition $a - w_j$ was verified, meaning that an arc where the item was included, we first confirm that this item can be added to the item set considering where it came from $S(j-1)^a$.

4.2.1 Restriction verification

When running the algorithm, restrictions need to be verified in case a trajectory is not possible. As suggested by figure 4.2, the plane does not travel through a flat surface, it travels in arc around Earth, which can be either considered an ellipsoid or a sphere. For this work, we considered it a sphere, though it is trivial to convert the calculations into an ellipsoid which is by adding a second radius.

Figure 4.2: Plane around the world.

To see if the arc that forms the path from A to B, and considering C as the center of the sphere that represents the restriction we do the following:

1. Convert A, B, and C coordinates to Cartesian coordinates. Note that, when converting to Cartesian coordinates, R will be the radius of the earth and we will have to sum the altitude of the waypoint considered.

2. A, B and O, where O is the center of the Earth define a plane that contains arc AB. We calculate the normal vector normalized to this plane, that is given by $(A \times B)/N(c)$ where $\times$ represents the cross product and N(c) is the norm of the cross product between A and B.

3. With the norm, we can now calculate the distance from C to the plane given by $C \cdot n$ where $\cdot$ represents the dot product.

4. Now we can compare the distance obtained with the radius of the sphere. If this distance is smaller or equal to the radius, then the arc intersects the sphere. If the distance is bigger, then the arc does not intersect the sphere.

For the aircraft restrictions, a very simple model was created in which we set the max bank angle as 30º, which will affect the turn time, and the maximum vertical rate of the plane was 3000 feet per minute. This is trivial to calculate, all we have to do is to see if the time it will take to change altitude at the maximum rate surpasses the time it will take to travel from point to point and if it does then the airplane would need a higher rate which is not possible.

4.3 Time and Space Complexities

4.3.1 Time Complexity

The upper bound time complexity of the algorithm can be estimated considering the following:

1. It needs to run through all n waypoints

2. It needs to run through the capacity W

3. The third loop depends on the number of weights that the item chosen can be which will be of D dimensions

4. The restriction set is a linked-list with R items and the algorithm needs to verify them all

   Having this, we have that the time complexity at worst can be $\mathcal{O}(nWRD^2)$

4.3.2 Space Complexity

The space complexity of this algorithm is not trivial to calculate, since it depends greatly on how restrictions affect the potential solutions. The higher number of restrictions, the higher the likelihood of fewer labels being created. If no restrictions are in place, the labeling algorithm will create one label for each possible combination of waypoints resulting in a $\mathcal{O}(\sum_{j=0}^{n} j!)$ of total nodes which results in a **very** large space. On the other hand, if the restrictions are very well placed and hinder a lot of movement, there could be a situation in which only one path could be created, minimizing the space usage.

4.4 Output

The labeling algorithm outputs a set of solutions which is a Pareto front, or none if no path to t is feasible. Considering that a set of solutions existed, we needed to pick a solution from this set. We arbitrarily chose to pick the one that minimized time spent.

From there, we can then run a backtracking technique to find all the solutions for objective-function 1. The backtracking technique consists in a dynamic-based approach where we check if it's possible to reach the solution from the current element. Fundamentally, this consists in the well-known subset-sum problem in which the target is the first-objective and the set, the first objective function. The dynamic approach to the subset sum problem is made over a integer set. We multiplied our value function by 10 and discarded the rest of the decimal part to convert it from a real set to a integer set.

If a solution set is found, it verifies if that set of waypoints chosen complies with objective-function 2 and if it does, we have found the solution. If not, it continues to exhaust every option until it finds one.

After all this, we can now produce the final output. This final output consists in the list of waypoints that forms the optimal path from s to t in the form of latitude longitude altitude, that begins with the starting point, the waypoints to be followed and the destination point.

There were two ways of implementing the backtracking tehcnique. The naive one which is a recursive function and using dynamic programming.

The dynamic programming implementation of the backtracking technique has a $\mathcal{O}(n * target)$ exploration time complexity. However, this only finds if a sum is achievable in a subset, not which elements do achieve it. Since we have to find which set we have to create a function that will be recursive and it works in the following way:

1. We will traverse the boolean matrix M filled.

2. if we reached the end of the matrix and the current sum is non-0 then we can obtain the final set if and only if the current element can be added to the final set making the current sum 0.

3. Else if the current sum is 0 then we have obtained the set that sums to the target.

4. If the given target can be obtained by ignoring the current element, we make a recursive call ignoring the current element.

5. Else if the given target can be obtained by considering the current element, we make a recursive call considering the current element, meaning that the current sum in the recursive call will be the former sum minus the value of the current element.

The time complexity of this algorithm is $\mathcal{O}(2^n)$ since at worst all elements will be considered and so every possible branch will be created.

4.5 Summary

The initial labeling algorithm only considered 1 restriction at a time to obtain the solution set. We expanded that algorithm to fit multiple restrictions, where the main restriction tied to the knapsack model was the fuel capacity of the airplane and the rest of the restrictions, both space and airplane restrictions, were calculated in run time. We added a third loop to calculate all the possible weights of an waypoint since its weight depends on the previous waypoint considered.

The algorithm returned a set of non-dominated labels to choose from. We arbitrarily chose one and from there we ran a backtracking technique where we first run a dynamic programming based approach to see from which elements we can achieve the target from and then recursively tried to find the final solution which consist in the starting point, waypoints to follow and destination point in the form of latitude, longitude and altitude.

5

Experimental Evaluation

Contents

5.1 Setting and objectives 43
5.2 Results 44
5.3 Discussion 46

In this chapter, we define what we are looking for when testing the module, the setting where we tested and how our tests were generated. We also present and discuss the results obtained.

5.1 Setting and objectives

For this study, we consider that the space was restricted to the Iberian Peninsula and that the altitudes could vary randomly between 0 feet and 30000 feet. Figure 5.1 represents a simple example of a generated 2D map where the green triangles represent the start point and destination, the blue squares represent the waypoints and the red circles represent restrictions.

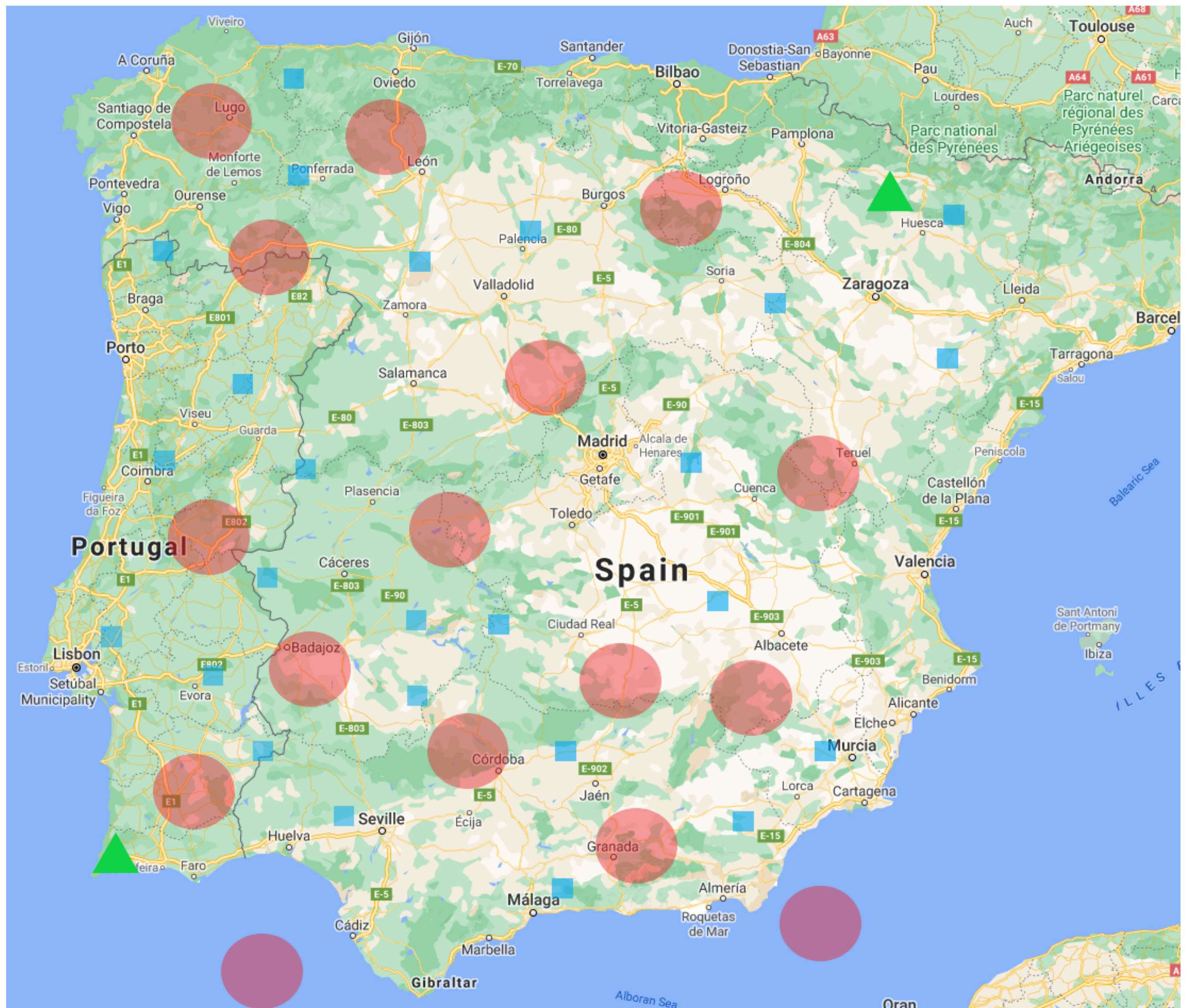


Figure 5.1: An example of a possible map.

The experiments were led on a i5-8400 running at 3.8GHz with 8GB RAM available. The whole module was written in C language (C11) and both the input and the waypoint database were randomly created using a python script using the random.py library.

The objectives for these experiments were to find if the algorithm could find solutions in efficient time when varying the number of waypoints, the number of restrictions and the capacity, these three factors being the main sources of time consumption in the labeling algorithm.

The timed tests were made first solely on the labeling algorithm and then including the backtracking technique.

When doing the tests, only the waypoint database was constant when it didn't change its size throughout the segment, meaning that in every segment of X waypoints, the restrictions were randomized in between each set. This can affect the time that the labeling algorithm will take to execute.

The restrictions were generated using a maximum altitude change rate of 3000 feet and with each circle having a random number between 0.5 and 5 kilometers.

5.2 Results

The results can be seen in table 5.1 and table 5.2. For the first part, the factors changed were the number of waypoints, restrictions and the capacity. For each test, 10 samples were ran and the time median of those 10 samples was chosen. The memory usage of the labeling algorithm is also displayed.

For the second stage of testing, since the backtracking technique solely depends on how many waypoints there are, we tested based on changing this number, while maintaining the capacity and restriction numbers, factors that influence the labeling algorithm, the same at 100 and 5000 respectively.

We also present plotting for easier visualization of the data obtained. In the first plot, the number of restrictions is 100, while in the second the number of restrictions is set to 1000.

Table 5.1: Average results of the labeling algorithm

Waypoins	Capacity	Restrictions	Time(s)	Memory used (MB)
100	5000	100	0.023	2.5
200	5000	100	0.025	2.6
500	5000	100	0.035	5.7
1000	5000	100	0.050	17.5
2000	5000	100	0.063	64.8

Continued on next page

Table 5.1 – *Continued from previous page*

Waypoins	Capacity	Restrictions	Time(s)	Memory used (MB)
5000	5000	100	0.204	384.1
10000	5000	100	0.657	1530
100	5000	500	0.007	2.5
200	5000	500	0.008	2.6
500	5000	500	0.010	5.7
1000	5000	500	0.020	17.5
2000	5000	500	0.045	63.5
5000	5000	500	0.207	384.6
10000	5000	500	0.750	1529.5
100	5000	1000	0.004	2.7
200	5000	1000	0.006	2.7
500	5000	1000	0.010	5.7
1000	5000	1000	0.020	17.3
2000	5000	1000	0.044	63.5
5000	5000	1000	0.206	384.4
10000	5000	1000	0.701	1529.7
100	30000	100	0.020	2.5
200	30000	100	0.074	2.5
500	30000	100	0.120	5.7
1000	30000	100	0.194	17.4
2000	30000	100	0.213	64.2
5000	30000	100	0.392	384.1
10000	30000	100	1.120	1529.6
100	30000	500	0.014	2.4
200	30000	500	0.017	2.9
500	30000	500	0.032	5.6
1000	30000	500	0.058	17.5
2000	30000	500	0.112	63.3
5000	30000	500	0.385	384.4
10000	30000	500	1.013	1529.7
100	30000	1000	0.012	2.5
200	30000	1000	0.017	2.5
500	30000	1000	0.033	5.7
1000	30000	1000	0.055	16.8
2000	30000	1000	0.115	63.4
5000	30000	1000	0.381	384.5
10000	30000	1000	1.035	1529.5
100	50000	100	0.046	2.6
200	50000	100	0.064	2.6
500	50000	100	0.072	5.6

Continued on next page

Table 5.1 – *Continued from previous page*

Waypoins	**Capacity**	**Restrictions**	**Time(s)**	**Memory used (MB)**
1000	50000	100	0.150	17.6
2000	50000	100	0.265	63.4
5000	50000	100	0.554	384.1
10000	50000	100	1.306	1530.1
100	50000	500	0.025	2.4
200	50000	500	0.028	2.9
500	50000	500	0.042	5.6
1000	50000	500	0.103	17.4
2000	50000	500	0.184	63.4
5000	50000	500	0.520	384.6
10000	50000	500	1.286	1529.3
100	50000	1000	0.025	2.6
200	50000	1000	0.040	2.7
500	50000	1000	0.055	5.6
1000	50000	1000	0.098	17.2
2000	50000	1000	0.193	63.4
5000	50000	1000	0.504	384.3
10000	50000	1000	1.310	1529.7

Waypoints	Time(s)
100	0.125
200	0.351
500	0.537
1000	1.007
2000	5.633
5000	11.164

Table 5.2: Execution time of the labeling algorithm including backtracking

5.3 Discussion

5.3.1 Labeling Algorithm

The labeling algorithm executes in a very fast time, yielding promising results even in the highest numbers tested. In terms of real time efficiency, 1 second for the labeling algorithm

Execution time of the labeling algorithm depending on the number of waypoints

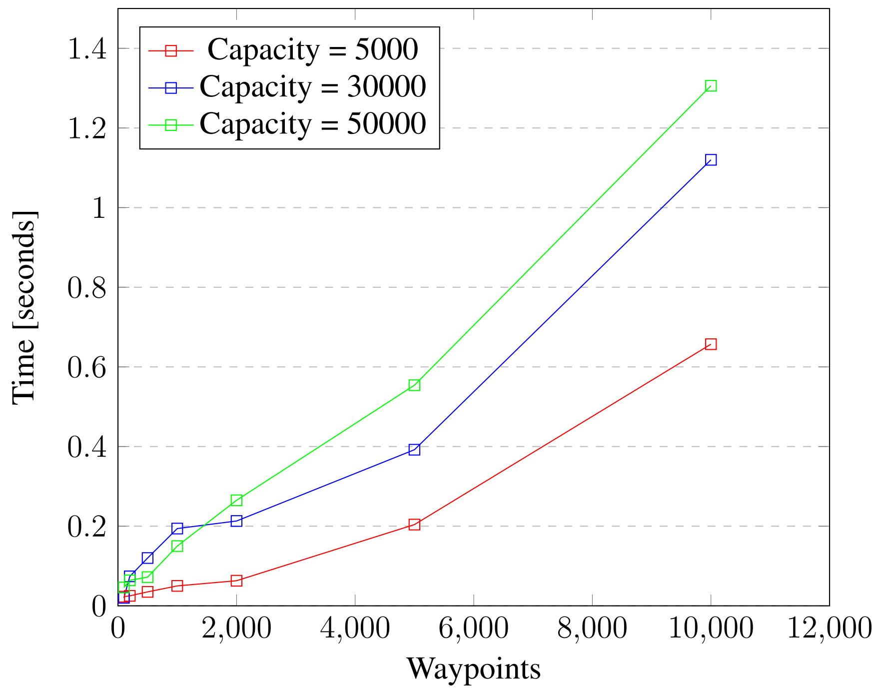


Figure 5.2: Time consumption by the labeling algorithm.

Memory usage of the labeling algorithm depending on the number of waypoints

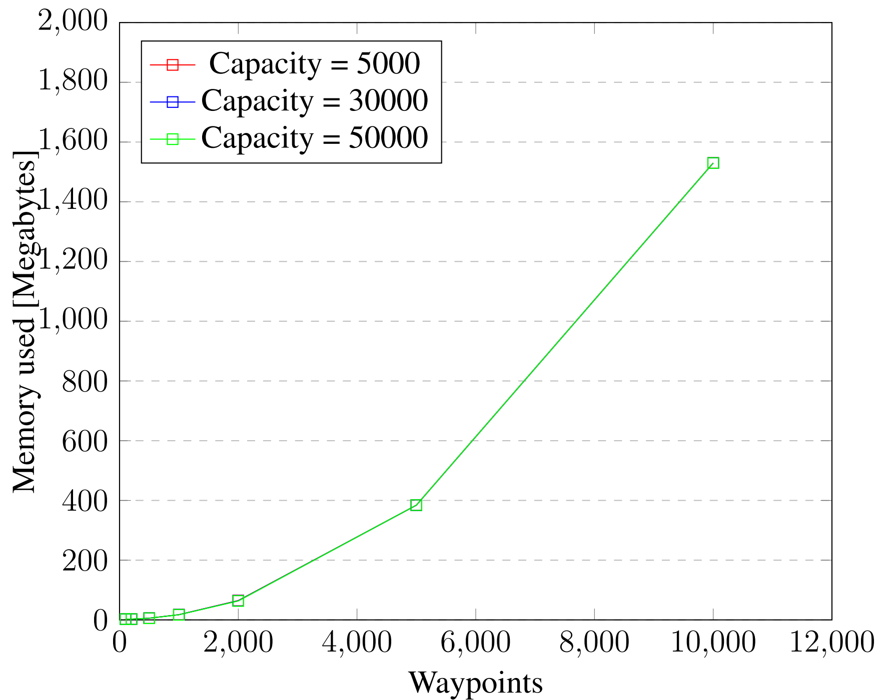


Figure 5.3: Memory usage by the labeling algorithm.

considering a plane at 400 knots, consists in approximately 200 meters, which is a very small change causing little to no change if the algorithm was reran at that point.

Despite, the number of restrictions rising from 100 to 1000, little change was verified. This is due to the fact that, while in the worst case scenario the time could go substantially up, in average, the restrictions limit the number of labels being created, thus severely limiting the third loop.

The capacities tested can be considered for low flight fuel in the form of 5000, a flight fuel that could be considered for a plane to be in mid-flight with 30000 fuel left and a flight fuel that could exist in the departure stage of the travel in 50000. Noticeably, the capacity is the most impactful factor since when it changes, it usually changes in the order of tens of thousands, from 50000 to 60000. If we consider a Boeing 747-400, it can take up to 216840 litres of fuel, a big change from the 50000 maximum considered.

The number of waypoints influence the time as well despite not so much, the highest number, 10000, being a obscene number of waypoints, tested just to prove that the algorithm could handle such a number.

In terms of space, the algorithm requires a lot of space, especially if the number of restrictions are not suited. For example, if the restrictions are all set in America and the waypoints are all set in Europe, the algorithm will generate every single possible combination of waypoints granted the capacity allows it. So, if both conditions are true, we then have a $\mathcal{O}(\sum_{j=1}^{n} j!)$ of nodes, which is a insurmountable number. Since our tests were randomly generated, some of them were bound to fail, since this possibility existed. As seen in table 5.1, the memory usage of the labeling algorithm in some cases reaches 1500 megabytes, which is already a considerable amount.

In terms of efficacy, a small number of tests was made to ensure the results were right. But even so, no real input was introduced and compared with real cases. So, in the long range, some problems might arise that the solution is not optimal.

Further testing should be made using real world cases, where waypoints have a very limited number of waypoints they can go to, the other part being limited by the restriction set, making the space limit as low as possible. If the number of restrictions is low and the restriction dimension is low so that it covers a low area, a lower number of waypoints should be considered to ensure that the algorithm does not run out of memory.

5.3.2 Backtracking technique

For the backtracking technique, up to 5000 waypoints were tested. The results were good, in the highest number of waypoints, the total time was 11.164 seconds, which considering a plane at 400 knots it is about 2.2 kilometers.

For a short to mid-haul flight, a very low number of waypoints in 2000 can be considered, making the time around the 5 second mark. The long haul can take more than 5000, although in the time it would take to calculate it, the conditions would change, so the alternative could be to search the solution space from the current point, to a point where pilots know they will cross.

In this algorithm, we intend to minimize the objective-functions and so, the sum the backtracking technique will be searching for, will be as low as possible, making it as efficient as it can be, both in terms of search time and search space. This problem is an on-going investigation and if found that the classical problem can be solved in polynomial time, that $\mathbf{P} = \mathbf{NP}$, or at least a better approximation can be made, then this part should be updated so that it can solve the problem faster.

6

Conclusion

Contents

6.1 Final Remarks 53

6.2 Future Work 54

6.1 Final Remarks

Flight planning is becoming more and more demanding in terms of time efficiency since airspace is becoming more and more overcrowded. While the effects of the on-going COVID19 pandemic will certainly make its mark in the aviation world, aviation was expected to double over the next 20 years. Pilots need to rely less on air traffic tower and be able to quickly re-plan their route if an issue arises.

The current system for pilots to change the current flight path is to search where the adversarial condition is when informed by the air traffic controller and check which viable waypoints are closer to that point so that they can travel to that instead. After that, they still have to inform ATC about the new route they intend to take and await confirmation. This process is not trivial and takes a lot of time both from the pilots and from the air traffic controller, time that could be well spent doing other tasks.

We propose a flight planner that is efficient and effective in finding a route while avoiding restrictions, which can replace the process mentioned above. The algorithm used can work with any type of restrictions since it does not depend on them. It also can take any number of value functions making it desirable for companies that want to maximize the value of a route, to their own standards of value and spending. Whether they wish to minimize fuel consumption or flight time or passenger comfort, all they have to do is create a mathematical function for that function, and run the algorithm over it.

The algorithm used consisted in an adaptation from the labeling algorithm proposed in 2001 by Captivo et al. [1]. This base algorithm worked by modeling a knapsack model into a shortest path problem, by layering the items and creating two arcs between them if possible, one when the item was added into the solution set and one where the item was not in the final solution set.

The modifications made were designed so that multiple restrictions could be fitted and still respected, since the base algorithm only supported one restriction at a time. The algorithm verifies if all the restrictions are not broken, and if they are not, then the corresponding arc can be created. The main factors influencing the execution time are the number of waypoints, the current fuel of the airplane and the restriction set.

The labeling algorithm returns the set of non-dominated labels from the starting point to the destination point. In this work, we arbitrarily chose one, since they are all optimal. From there, a backtracking technique using dynamic programming was used to find the waypoints, the first value of the label corresponded to. Then, we calculated if the second value corresponding to the value in the second-objective function. If it did, the final solution was found. If not, the backtracking technique would continue until it found the solution set. The final output consisted in the starting point, waypoints to follow and the destination point.

The results of running tests over the labeling algorithm and the backtracking technique were good where the times were not big enough so that the distance flown by the airplane while the result was being generated created enough change for the result to be different if calculated again at that point in time. Results showed that it is possible for an on-board flight-planner to exist and dynamically adjust flight routes if initial conditions change.

6.2 Future Work

Despite good initial results, several improvements can be made and should be explored. Initially, it would be very beneficial to have feedback from flight-planning specialists. This would allow the validation of the current system in place, and further improvement could be made over it.

As mentioned before, the algorithm can take up to a lot of space, especially when the number of waypoints is high and the number of restrictions is low, since a very high number of labels will be created. A potential solution to this problem would be the longest common sub-sequence in which only states near the optimum would be created. This could reduce the number of labels created and massively improve the space usage of the algorithm.

The algorithm was setup by always having 2 objective functions. It would be interesting to explore the cost of having 3 or more objective functions which would lead to a maximization of profit by companies that want it. However, for more than two criteria, it is important that it is not enough to compare the new label with the last label added and instead compare it to all others added to the node, which is a very time consuming operation.

When choosing a solution from the solution set, in this work we arbitrarily did so. It would be a good change to explore ways of picking a solution, whether is using an algorithm that finds the knee point in the Pareto front or creating a framework where the user can express their preference for which objective to prioritize.

Although the results did not suggest the restrictions were causing a huge time factor in the system, the current method in place is not the most efficient one. Even if the waypoints being analyzed are not close to a space restriction, they can even be on one side of the map and the restriction on the other, the restriction will still be checked. Implementations of new data structures that can efficiently obtain which restrictions are close to the waypoints should be researched and explored.

Currently, despite the algorithm being able to receive any type of restrictions, the restrictions themselves are currently just the spheres and the altitude rate change . It would be interesting to modify those to more different types of, to really cement the model as a very adaptive module.

In this work, a very simple airplane model was used. Better flight dynamics should be considered and implemented, despite the labeling algorithm being independent from this. The cost-functions are simple and do not require a lot of operations. If a lot of trigonometry is needed, this could impact the execution time, so this should be taken into account

When obligatory waypoints exist, concurrent programming could improve the running time by running two or more algorithms concurrently between this waypoints to reach a faster solution.

Turn time and speed variation were not considered in this work which could change the solution in some cases, making the solution less optimal that what it should be. Frequent altitude changes were also not considered for the second objective which can greatly influence the fuel consumption.

Testing should be made using real world cases and real world restrictions. For this to happen, a rigorous flight dynamic model should be in place, to ensure the solution set found by the labeling algorithm is the right one. These tests will serve to confirm and ensure the efficacy of the algorithm.

Bibliography

[1] M. E. Captivo, J. Clímaco, J. Figueira, E. Martins, and J. L. Santos, "Solving bicriteria 0-1 knapsack problems using a labeling algorithm," *Computers and Operations Research*, vol. 30, no. 12, pp. 1865–1886, 2003.

[2] J. R. Boyd, "The Essence of Winning and Losing."

[3] N. Dougui, D. Delahaye, S. Puechmorel, and M. Mongeau, "A light-propagation model for aircraft trajectory planning," *Journal of Global Optimization*, vol. 56, no. 3, pp. 873–895, 2013.

[4] I. Alonso-Portillo and E. M. Atkins, "Adaptive Trajectory Planning for Flight Management Systems," University of Maryland, Maryland, Tech. Rep., 2006.

[5] D. Sislak, D. Sislak, P. Volk, P. Volk, M. Pechoucek, and M. Pechoucek, "Flight Trajectory Path Planning," *2009 ICAPS Scheduling and Planning Applications woRKshop (SPARK)*, pp. 76–83, 2009.

[6] S. Carpin and G. Pillonetto, "Merging the adaptive random walks planner with the randomized potential field planner," *Proceedings of the Fifth International Workshop on Robot Motion and Control, RoMoCo'05*, vol. 2005, pp. 151–156, 2005.

[7] S. Carpin, "Motion planning using adaptive random walks," *IEEE Transactions on Robotics*, vol. 21, no. 1, pp. 129–136, 2005.

[8] S. LaValle, "Rapidly-Exploring Random Trees: Progress and Prospects."

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] B. R. Peter E. Hart, Nils J. Nilsson, "A Formal Basis for the Heuristic Determination of Minimum Path Costs," pp. 100–107, 1968.

[11] L. Ford and D. Fulkerson, “Maximal Flow Through a Network.”

[12] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau, “Path planning: A 2013 survey,” *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management, IEEE - IESM 2013*, vol. 2013, no. October, 2013.

[13] E. Zermelo, *ber die Navigation in der Luft als Problem der Variationsrechnung*, 1930.

[14] J. Canny and J. Reif, “New Lower Bound Techniques for Robot Motion Planning Problems.” *Annual Symposium on Foundations of Computer Science (Proceedings)*, pp. 49–60, 1987.

[15] M. Jun and R. D’Andrea, “Path Planning for Unmanned Aerial Vehicles in Uncertain and Adversarial Environments,” pp. 95–110, 2003.

[16] R. Bellman, “On a routing problem,” *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[17] A. R. Babaei and M. Mortazavi, “Three-dimensional curvature-constrained trajectory planning based on in-flight waypoints,” *Journal of Aircraft*, vol. 47, no. 4, pp. 1391–1398, 2010.

[18] K. Bousson and P. Machado, “4D Flight trajectory optimization based on pseudospectral methods,” *World Academy of Science, Engineering and Technology*, vol. 70, no. May 2016, pp. 551–557, 2010.

[19] A. Islami, S. Chaimatanan, D. Delahaye, A. Islami, S. Chaimatanan, D. Delahaye, L. Scale, and T. Planning, “Large Scale 4D Trajectory Planning,” pp. 27–47, 2016.

[20] A. Fallast and B. Messnarz, “Automated trajectory generation and airport selection for an emergency landing procedure of a CS23 aircraft,” *CEAS Aeronautical Journal*, vol. 8, no. 3, pp. 481–492, 2017.

[21] R. A. Paielli, “Trajectory Specification Language for Air Traffic Control,” *Journal of Advanced Transportation*, vol. 2018, 2018.

[22] A. Jouglet and J. Carlier, “Dominance rules in combinatorial optimization problems,” *European Journal of Operational Research*, vol. 212, no. 3, pp. 433–444, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.ejor.2010.11.008

[23] J. B. Orlin, *Ahuja, Magnanti, Orlin - Network flows Theory, algorithms and applictions*, 1993.

[24] C. Bazgan, H. Hugot, and D. Vanderpooten, "Solving efficiently the 0-1 multi-objective knapsack problem," *Comput. Oper. Res.*, vol. 36, no. 1, pp. 260–279, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2007.09.009

[25] J. R. Figueira, L. Paquete, M. Simões, and D. Vanderpooten, "Algorithmic improvements on dynamic programming for the bi-objective {0,1} knapsack problem," *Computational Optimization and Applications*, vol. 56, no. 1, pp. 97–111, 2013.

[26] C. Gomes Da Silva, J. Clímaco, and J. Figueira, "A scatter search method for bi-criteria {0, 1}-knapsack problems," *European Journal of Operational Research*, vol. 169, no. 2, pp. 373–391, 2006.

[27] C.-L. Hwang and K. Yoon, *Multiple Attribute Decision Making Methods and Applications*, 1981, vol. 618.

[28] T. L. Saaty, "What is the analytic hierarchy process?" pp. 577–591, 2011.

Labeling Algorithm

Algorithm 1: A labeling algorithm to determine efficient paths

```
Set S(1^0) ← {(0, ..., 0)} and S(1^{w_1}) ← {(−v_1^1, ..., −v_1^r)};
Set T ← {0, w_1} and V ← {};
for j=2 to n do
    for a=2 to W do
        if (a ∈ T) then
            V ← V ∪ {a};
            if (a − w_j ∈ T) then
                S(j^a) ← Set of ND labels of (S((j − 1)^a) ∪
                {(-v_j^1, ..., −v_j^r)} + S((j − 1)^{a−w_j}));
            else
                S(j^a) ← S((j − 1)^a)
        else
            if (a − w_j ∈ T) then
                S(j^a) ← {(−v_j^1, ..., −v_j^r)} + S((j − 1)^{a−w_j}));
                V ← V ∪ {a};
    end
    T ← V and V ← {}
end
S(t) ← Set of ND labels of ⋃_{a=0}^{W} S(n^a)
```

Algorithm 2: The modified labeling algorithm

```
MergeSort(WaypointList)
Set w_1 = c(s, 1)
Set S(1^0) ← {(0, ..., 0)} and S(1^{w_1}) ← {(−v_1^1, ..., −v_1^r)};
Set lastitem(S(1^0)) ← s and lastitem(S(1^{w_1})) ← 1
Set T ← {0, w_1} and V ← {};
for j=2 to n do
    for a=0 to W do
        LW = ⋃_{k=0}^{TSize} LastWaypoint ∈ S(j − 1)^k, k ∈ T:
        Set arrayofweights ← CalculateAllWeights(LW, j)
        for el in arrayofweights do
            w_j ← el;
            if (a ∈ T) then
                V ← V ∪ {a};
                if (a − w_j ∈ T) then
                    LI1 ← LastWaypoint ∈ S(j − 1)^a
                    LI2 ← LastWaypoint ∈ S(j − 1)^{a−w_j}
                    FlagA ← VerifyRestrictions(j, LI1)
                    FlagB ← VerifyRestrictions(j, LI2)
                    if (FlagA = True and FlagB = True then
                        lastitem(S(j)^a) ← j;
                        S(j^a) ← Set of ND labels of
                          (S((j − 1)^a) ∪ {(−v_j^1, ..., −v_j^r)} + S((j − 1)^{a−w_j}));
                    else if FlagA = True and FlagB = False then
                        lastitem(S(j)^a) ← lastitem ∈ S(j − 1)^a;
                        S(j^a) ← S((j − 1)^a)
                    else if FlagA = False and FlagB = True then
                        lastitem(S(j)^a) ← j;
                        S(j^a) ← {(−v_j^1, ..., −v_j^r)} + S((j − 1)^{a−w_j}));
                else
                    lastitem(S(j)^a) ← lastitem ∈ S(j − 1)^a;
                    S(j^a) ← S((j − 1)^a)
            else
                if (a − w_j ∈ T) then
                    LI ← LastWaypoint ∈ S(j − 1)^{a−w_j}
                    Flag ← VerifyRestrictions(j, LI)
                    if (Flag = True then
                        lastitem(S(j)^a) = j;
                        S(j^a) ← {(−v_j^1, ..., −v_j^r)} + S((j − 1)^{a−w_j}));
                    V ← V ∪ {a};
        end
        T ← V and V ← {}
    end
end
S(t) ← Set of ND labels of ⋃_{a=0}^{W} S(n^a)
```