Cooperation of Combinatorial Solvers for Air Traffic Management and Control

Ruixin Wang

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. RUIXIN WANG

le jeudi 17 septembre 2020

Titre :

Collaboration de méthodes d'optimisation combinatoire pour la gestion et le contrôle du trafic aérien

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Laboratoire de Télécommunications (TELECOM-ENAC)

Directeur(s) de Thèse :

M. NICOLAS DURAND

M. NICOLAS BARNIER

Rapporteurs :

M. JIN-KAO HAO, UNIVERSITE D'ANGERS

Mme CHRISTINE SOLNON, INSA LYON

Membre(s) du jury :

M. LUIS DELGADO MUÑOZ, Université de WESTMINSTER, Membre

M. NICOLAS BARNIER, ECOLE NATIONALE DE L'AVIATION CIVILE, Membre

M. NICOLAS DURAND, ECOLE NATIONALE DE L'AVIATION CIVILE, Membre

Résumé

Dans le contexte du projet SESAR, le contrôle du trafic aérien (ATC) et sa gestion (ATM) en Europe est en train de changer de paradigme pour être capable de gérer l'augmentation du trafic indiquée par les prévisions actuelles : de nombreux systèmes fondés sur des experts vont être améliorés par des logiciels d'optimisation pour rendre les processus de prise de décision et la planification des régulations plus efficaces. Les techniques actuelles d'optimisation combinatoires qui sont appliquées aux problèmes d'ATM et ATC comprennent des algorithmes d'approximation telles que les métaheuristiques (e.g. Algorithmes Génétiques (AG), recherche taboue, recuit simulé…) et des algorithmes exacts comme la Programmation Par Contraintes (PPC). Cependant, la très grande taille des instances considérées et la gestion des incertitudes inhérentes à ce type de problèmes les rendent très difficiles à résoudre, ce qui peut handicaper fortement les méthodes précédemment mentionnées lorsqu'elles sont utilisées seules.

Afin de surmonter ces difficultés et d'améliorer l'efficacité des algorithmes standards, nous proposons d'étudier la coopération générique d'un ensemble quelconque de solveurs combinatoires, en partageant les solutions découvertes, les bornes d'optimisation ainsi qu'éventuellement d'autres informations pour permettre d'accélérer la résolution. Dans cette thèse, le candidat a spécifié et implémenté un tel système distribué de telle manière qu'il puisse intégrer tout type de solveur combinatoire doté d'une interface adéquate, adapter des solveurs existants pour prendre en compte et fournir des informations sur l'état de la recherche des autres solveurs, et appliquer ce système à la résolution de problèmes d'ATC et ATM tels que la résolution de conflit et l'allocation de porte de vol (GAP).

Pour le premier, nous avons présenté un nouveau cadre générique pour la modélisation et la résolution des conflits en route en trois dimensions, ainsi qu'un grand nombre d'exemples réalistes, qui ont été résolus avec la coopération d'un algorithme mémétique et de la programmation linéaire en nombres entiers (ILP). Pour le GAP, nous avons présenté un nouveau modèle PPC, de nouvelles contraintes d'optimisation et stratégies de recherche, ainsi que leur coopération parallèle, pour maximiser la robustesse de l'allocation. Le solveur, implémenté avec la bibliothèque de PPC FaCiLe, surpasse un solveur ILP à la pointe de la technologie sur des instances réelles.

Mots clés : Optimisation Combinatoire, Hybridation, Métaheuristique, Programmation Par Contraintes, Contrôle et Gestion du trafic aérien.

Abstract

In the context of the SESAR project, Air Traffic Control (ATC) and Management (ATM) in Europe is undergoing a paradigm shift to be able to accommodate the current traffic growth forecast: many expert-based systems will be enhanced by optimization software to improve the decision-making process and regulation planning. Current state-of-the-art combinatorial optimization techniques that are applied to ATC and ATM include approximation algorithms like metaheuristics (e.g. Genetic Algorithm, Tabu Search, Simulated Annealing, etc.) and complete algorithms like Constraint Programming (CP) and Mixed Integer Programming. However, the large scale of the considered instances and the handling of their inherent uncertainties result in very hard problems, which can hinder or even defeat either of the previously mentioned optimization methods alone.

To overcome these difficulties and improve the resolution efficiency of standard algorithms, we propose to study the generic cooperation of any set of combinatorial solvers by sharing solutions, optimization bounds and possibly other information in order to speed up the overall process. In this thesis, we have specified and implemented a distributed system which is able to integrate any combinatorial solver with the suitable interface, adapt existing solvers to take into account and provide information on the state of the search from and to other solvers, and applied this framework to two ATC and ATM problems: the en-route conflict resolution problem and the Gate Allocation Problem (GAP).

For the first one, we have presented a new generic framework for the modeling and resolution of en-route conflicts in three dimensions as well as a large set of realistic instances, which have been solved with the cooperation of a Memetic Algorithm and Integer Linear Programming (ILP) solver. For the GAP, we have presented a new CP model, as well as new optimization constraints to maximize the robustness of the schedule, and search strategies together with their parallel cooperation. The solver, implemented with the FaCiLe CP library, outperforms a state-of-the-art ILP solver on real instances.

Keywords: Combinatorial Optimization, Hybridization, Metaheuristics, Integer Linear Programming, Constraint Programming, Air Traffic Management.

Acknowledgements

First of all, I wish to express my sincere gratitude to my supervisor, Dr. Nicolas Barnier, for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. I can honestly say that I owe it to him that I have come this far. I would like to show my appreciation for my supervisor, Professor Nicolas Durand, who gave me an overall view of the field of aeronautics, supported me throughout my work, and whose invaluable insights improved my work. My sincere thanks also goes to my supervisor Dr. Cyril Allignol. Over the course of my work, we have discussed many scientific ideas, worked together for the Drone project. He has also helped me considerably to organise my publications and thesis better and to create clearer figures to illustrate results of our scientific work. It is worth noting that my supervisors continue to help me to prepare a better thesis from their homes even during the lock down period and alongside their childcare responsibilities. I am greatly touched by their generous help and really want to collaborate more deeply with them in the future. It will be always a big pleasure to participate in very interesting conferences with them!

Many thanks to all members of the jury for agreeing to read the manuscript and to participate in the defense of this thesis. Thanks to the referees, Pr. Christine Solnon and Pr. Jin-Kao Hao, for their thorough reading to indicate every possible improvement to make my manuscript easier to understand. Also thanks to examiner Dr. Luis Delgado Muñoz and all the precious remarks that inspired me to improve my future work.

I also want to thank my friends Dr. Richard Alligier, Dr. Alexandre Gondran, Dr. Jean-Baptiste Gotteland and Pr. David Gianazza, we always had enjoyable lunch time and they never hesitated to share their scientific knowledge with me. Richard also used to teach me one french word per day to help me improve my french. Honestly, I am really happy to have so many lovely workmates and friends during these years, I think I will miss you very much!

I wish to thank other professors and members in the Z building: Catherine Mancel, Marcel Mongeau, Sonia Cafieri, Stéphane Puechmorel, Hélène Weiss, Mathieu Cousy, Georges Mykoniatis.

Many thanks to my labmates at ENAC Lab, Xiao Liang and Zhengyi Wang, our small lovely, funny group; Serge Roux, our best technical support; Ji Ma, Jun Zhou, Hang Zhou, Ying Huo, Xiao Hu, Ruohao Zhang, Shangrong Chen, Alexandre Tran, Lucille Kuhler, Romaric Breil, Vincent Courjault-Radé, Florian Mitjana, Imen Dhief, Sana Rebbah, Sana Ikli, Man Liang, Jérémie Chevalier, Ahmed Khassiba and also my good friends outside of ENAC Lab.

Finally, I would like to thank my parents and my wife for their unconditional love and support.

Contents

Introduction **1**

1 Context **5**
1.1 Parallelization of Combinatorial Algorithms 7
1.1.1 Portfolio of Algorithms 7
1.1.2 Multi-Agent System 9
1.2 Cooperation of Solvers 10
1.2.1 Exchanged Data 11
1.2.2 Communication Points 13
1.2.3 Architecture 14
1.3 Summary 15

2 Generic Framework for Cooperation **17**
2.1 A Generic, Distributed Framework 18
2.1.1 Asynchronous Server-Client Architecture 19
2.1.2 Interprocess Communication Protocol 25
2.1.3 Message Format 27
2.2 Cooperative Algorithms 28
2.2.1 Memetic Algorithm 29
2.2.2 Integer Linear Programming 32
2.2.3 Constraint Programming 38

3 Application to Conflict Resolution **43**
3.1 Related Works 44
3.2 Model 47
3.2.1 Maneuvers and Decision Variables 47
3.2.2 Trajectory Prediction and Conflict Detection 50
3.2.3 Cost of Maneuvers 57
3.2.4 Overall Mathematical Model 59
3.3 Resolution Algorithms 60
3.3.1 Memetic Algorithm 60

3.3.2 Integer Linear Programming . . . 62
3.3.3 Cooperation Between the MA and the ILP . . . 65
3.4 Results . . . 65
3.4.1 Benchmark . . . 65
3.4.2 Experimental Setup . . . 67
3.4.3 Single Algorithms . . . 67
3.4.4 Cooperation . . . 68
3.4.5 Infeasible Instances . . . 72
3.5 Conclusion and Further Work . . . 75

4 Application to Gate Allocation 77
4.1 Related Works . . . 82
4.2 Fixed Job Scheduling . . . 84
4.2.1 Instance . . . 84
4.2.2 Fictive Tasks and Renumbering . . . 85
4.2.3 Decision Variables . . . 86
4.2.4 Non-Overlapping Constraints . . . 87
4.2.5 Transition Cost . . . 87
4.2.6 Global Compatibility Graph . . . 88
4.3 Basic CP Model . . . 89
4.3.1 Constraints on Maximal Cliques . . . 90
4.3.2 Symmetries . . . 90
4.3.3 The IdleCost Constraint . . . 91
4.3.4 Per-Resource Propagation of Transition Costs . . . 92
4.4 Global Propagation of Transition Cost . . . 97
4.4.1 Relaxation of FJS to Path Covering . . . 97
4.4.2 Successor Variables and Reduction to the Linear Assignment Problem . . . 99
4.4.3 The MinWeightAllDiff Constraint . . . 100
4.4.4 Channelling Constraints . . . 101
4.4.5 Search Strategies . . . 103
4.5 Basic ILP Model . . . 107
4.5.1 Tasks and Resources Compatibility . . . 107
4.5.2 Decision Variables . . . 108
4.5.3 Constraints . . . 109
4.5.4 Objective . . . 110
4.6 Flow Model . . . 111
4.6.1 Graph Model . . . 111
4.6.2 ILP Model . . . 115
4.7 Results . . . 116
4.7.1 Data . . . 117

4.7.2 Per-Resource vs. Global CP Models 118
4.7.3 Search Strategies 118
4.7.4 CP vs. ILP Models 121
4.7.5 Robustness of the Schedule 123
4.8 Conclusion 125

Conclusion and Perspectives **127**
Contribution 127
Perspectives 129

Bibliography **132**

Acronyms **145**

List of Figures

2.1 General framework architecture. 19
2.2 Example of a communication sequence between two solvers and the data manager. 21
2.3 Cooperation between a metaheuristic and a CP solver. 24
2.4 Data exchange using ØMQ. 26

3.1 Graph representation of a conflict resolution problem: clique $\langle 1d, 2c, 3c, 4b\rangle$ is a solution. 46
3.2 A traffic scenario on several Flight Levels. 48
3.3 Maneuvers compatible with current ATC practice. 49
3.4 Typical en-route protection volume around an aircraft. 51
3.5 Reaction time uncertainty model with maximal errors E_{t_0} and E_{t_1}. 52
3.6 Heading change uncertainty model with maximal error E_α. . . 53
3.7 Speed uncertainty model with maximal error E_{v_h}. 53
3.8 Climb and descent uncertainty model with maximal error E_{v_v} model. 53
3.9 Flight mode uncertainty model with possible modes "fly by" (F_b, in blue) or "fly over" (F_o, in green). 53
3.10 An example of convex hulls representing a maneuver with uncertainties in the horizontal plane at each time step. 55
3.11 Axis-Aligned Bounding Box (in black) of an aircraft position envelope (in green). 57
3.12 Typical behaviour of the Memetic Algorithm in the landscape of solutions. The inside of green circles corresponds to a subset of admissible solutions. 61
3.13 Geometry of conflict scenario generation. 66
3.14 Comparison of computation times to find an optimal solution for small instances with the MA and ILP w.r.t. the number of aircraft. 68

3.15 Percentage of success for finding an optimal solution within a 300 s time limit with the MA and ILP w.r.t. the number of aircraft. 69
3.16 Percentage of success for finding an optimal solution with a 300 s time limit with MA, ILP and their cooperation w.r.t. the number of aircraft. 70
3.17 Average cost of the best solution found within 300 s, expressed as the ratio of the difference to the optimum w.r.t. the number of aircraft. 70
3.18 Convergence of the cost with a 300 s time limit w.r.t. the execution time. 71
3.19 Comparison of computation times to find an optimal solution for the cooperation and the ILP solver alone w.r.t. the number of aircraft. 72
3.20 Comparison of computation times to prove optimality for the cooperation and the ILP solver w.r.t. the number of aircraft. . 73
3.21 Average of the number of remaining conflicts over 5 instances with 60 aircraft w.r.t. the level of uncertainty l, resolved by the MA with a 300 s time limit. 74

4.1 Paris-CDG 1: terminals T, U, V, W, X, Y, Z. 80
4.2 Paris-CDG 2: terminals A, B, C, D, E, F 81
4.3 The resource and tasks of Example 1 with unbounded lower bound approximation ratio. 95
4.4 A solution to an instance of FJS. 98
4.5 Relaxation of FJS to Minimum Weight Path Cover (MWPC). 98
4.6 A solution to MWPC is not generally a solution to FJS. . . . 99
4.7 If successor variable is assigned on a known resource r_j, then possible in-between tasks t_{i_1}, t_{i_2} and t_{i_3} are removed from the resource r_j. 103
4.8 If no task can be scheduled on resource r_j between two already assigned tasks t_i and $t_{i'}$, then successor variable must be assigned, i.e. $y_i = i'$. 103
4.9 On this partial solution (where assigned tasks are in orange and unassigned ones are in grey), the first resource is currently the least loaded, so the Resource Balancing strategy would select either task RB-EST or RB-BC. 104
4.10 A partial solution (where assigned tasks are in orange and unassigned ones are in grey) with a non-empty hole on resource r_i. 105

4.11 On this partial solution (where assigned tasks are in orange and unassigned ones are in grey), the Critical Hole strategy would either select the hole of resource r_1 (CH-Min) or the one of resource r_2 (CH-Max), then choose among its possible tasks the one with earliest start time (EST) or best cost (BC). 105
4.12 Colored representation of the *compatibility graph* of Example 2 where commodities (or gates) are represented by different colors. Weights (i.e. costs of idle times) are indicated in hundreds of min^2. The flow corresponding to the optimal solution is shown with bold arcs. 113
4.13 Gantt diagram of the optimal solution to Example 2. 115
4.14 Number of backtracks (solid lines) and execution time in seconds (dashed lines) w.r.t. the number of aircraft to prove optimality for ICTAI and MWAD with 7 gates. 119
4.15 Gantt diagram of an optimal solution to an instance with 20 gates and 78 aircraft at terminal J of Paris-CDG airport. . . . 119
4.16 Percentage of instances solved optimally within 60 s by MWAD for all instances at Paris-CDG airport, w.r.t. the strategy. . . 120
4.17 Mean of execution times (in seconds) to prove optimality with the basic ILP model and the Flow ILP model solved by Gurobi, and our new MWAD CP model solved with FaCiLe for instances at Paris-CDG, w.r.t. the terminal. 122
4.18 Distribution of idle times for all terminals over all tested instances. 124
4.19 Distribution of idle times at terminal F over all tested instances. 125

List of Tables

3.1 Maneuver parameters. 50
3.2 Uncertainties on the trajectory parameters. 54
3.3 Value of the uncertainty bounds w.r.t. the error level l. 72

4.1 Number of flights and gates (per day on average) w.r.t. the terminal. The density is the ratio of the average number of flights by the number of gates. 117
4.2 Number of backtracks and execution time w.r.t. the number of aircraft to prove optimality for ICTAI and MWAD with 7 gates. 118
4.3 Average gap between the optimal cost and the one of the best solution found within 60 s w.r.t. the strategy. 121
4.4 Average gap between the cost of optimal solution and the best solution found with various ILP models and MWAD within different time limits. 122
4.5 Average idle time between two consecutive aircraft at the same gate. 123
4.6 Average idle time between two consecutive aircraft at the same gate for terminal F only. 124

List of Algorithms

2.1 Server event loop. 20
2.2 Search state update procedure. 23
2.3 Memetic algorithm 29
2.4 Tabu Search 31
2.5 Collaborative Memetic algorithm (CMA) 33
2.6 Branch and Cut 35
2.7 Cutting plane algorithm 36
2.8 Collaborative Branch and Cut 37
2.9 Branch & Prune. 41
2.10 Collaborative Branch & Prune. 42

Introduction

Air Traffic Control (ATC) and Management (ATM) automation generates many difficult Combinatorial Optimization Problems (COP) that can be challenging for several reasons. First, many uncertainties inherent to the domain must be taken into account to handle complex and changeable aeronautic operations. Then, the need for real-time decisions in Air Traffic Control leads to demanding requirements on the efficiency of solvers. Eventually, the scale of the considered instances according to current traffic growth forecast is very large. Consequently, the difficulty of such air traffic problems has become challenging for human experts and current operational algorithms. However, few attempts have been made to improve the performance of such systems through parallelization, so as to benefit from the increasing computing power offered by current networks of multi-core workstations.

Among various parallelization strategies, like the separation of the search space for a single algorithm or the parallel execution of independent algorithms selected from a portfolio, multi-agent search where a network of optimization agents interact with each other in parallel seem more promising. Indeed, useful information like the current best solution or optimization bounds can be shared between solvers and improve their performance beyond their own abilities if their strengths (and shortcomings) are complementary. Unlike the aforementioned approaches which are limited to execution speed-ups or by the performance of the best algorithm of a predefined subset, multi-agent systems can break the limits of all its contributing algorithms and solve more difficult problems than any solver alone.

To obtain a system which is generic enough to include such diverse and complementary algorithms as local search (or metaheuristics) and tree search (e.g. Constraint Programming or Mixed Integer Programming), with possibly different models tailored to each approach, we decided to study the *cooperation of combinatorial solvers*: we propose a generic framework based on a *server-client* pattern designed to easily plug various combinatorial solvers in and have them cooperate to outperform any single method. We describe the general architecture of our distributed cooperation framework, designed

to be robust to the software or hardware failure of any client, and provide details about the communication scheme and the internal logic of the server, which stands as the "meta-solver" and "data manager". We also show how to adapt diverse combinatorial optimization algorithms to the proposed framework, with a focus on three classic methods: a Metaheuristic, Integer Linear Programming (ILP) and Constraint Programming (CP). Finally, we have applied our framework to two COPs related to ATC and ATM: *en-route conflict* resolution and *gate allocation* at airports.

To increase the level of automation in ATC, one of the key challenges is the resolution of en-route conflicts to avoid losses of separation between aircraft in a given airspace volume. To be able to compare various optimization approaches to this problem, we first present a novel structure which completely separates the modelling of en-route conflicts from its resolution. Though we propose our own realistic 3D maneuvers and conflict detection with uncertainties on the position of aircraft, our structure is able to handle any other kind of maneuver and detection models. By using an efficient GPU-based algorithm, potential conflicts between aircraft can then be detected in less than a few seconds on realistic scenarios. Eventually, we solve this problem with our distributed cooperation framework instantiated with a Memetic Algorithm and an ILP solver. We show how our cooperation approach outperforms individual algorithms by orders of magnitude and can provide in a few minutes optimal solutions to large instances that were out of reach for a single method.

To improve the operations at airports, the Gate Allocation Problem (GAP) focuses on finding and optimizing an assignment of a given set of aircraft with fixed occupancy periods to a number of (non-uniform) gates. As noted by [Bolat, 2001], one of the main objective of the allocation is to optimize the robustness of the overall schedule, in order to absorb possible deviations from the original schedule due to traffic delays, severe weather conditions or equipment failures. We first model gate allocation as Fixed Job Scheduling (FJS): an aircraft with scheduled arrival and departure times can be considered as a task with fixed start and end times, and a gate as a specific resource. Then, we present a new global constraint for CP solvers to propagate the transition costs for FJS. However, the corresponding relaxation is not of good quality w.r.t. the global lower bound, as a task may be simultaneously scheduled on all its compatible resources. To obtain a competitive CP solver, we then introduce a new CP model based on the *MinWeightAllDiff* optimization constraint to compute the lower bound of a Path Cover (PC) of the *compatibility graph* of the problem. This relaxation is much tighter as the constraint directly propagates on the total cost, considering all resources and all tasks simultaneously. We also describe how the optimal PC computed by

the constraint can efficiently guide the search strategy. The resulting CP solver, implemented with FaCiLe [Barnier and Brisset, 2001], using parallel cooperation between strategies, not only outperforms the previous approach by orders of magnitude, but also consistently outperforms a basic ILP model. At the same time, we also point out the problems of the basic ILP model and propose a new Minimum Cost Flow Problem (MCFP) model for the GAP. This MCFP model outperforms the basic one by one order of magnitude and compete well against our parallel CP solver.

This thesis is organized as follows: Chapter 1 presents the literature on the parallelization of combinatorial solvers, then we describe in Chapter 2 the design of our generic framework for algorithm cooperation and detail how to include various combinatorial optimization solvers; Chapters 3 and 4 illustrate the use of our framework with the aforementioned algorithms on en-route conflict resolution and gate allocation at airports. Finally, we conclude and suggest further works in the last chapter.

The work carried out within the scope of this thesis has been published in:

[**1**] Cyril Allignol, Nicolas Barnier, Nicolas Durand, Alexandre Gondran, and Ruixin Wang. Large scale 3D en-route conflict resolution. In *ATM Seminar, 12th USA/Europe Air Traffic Management R&D Seminar*, 2017.

[**2**] Ruixin Wang and Nicolas Barnier. Propagation of idle times costs for fixed job scheduling. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 718–725, Nov 2018.

[**3**] Ruixin Wang, Cyril Allignol, Nicolas Barnier, and Jean-Baptiste Gotteland. Departure Management with Robust Gate Allocation. In *ATM Seminar, 13th USA/Europe Air Traffic Management R&D Seminar*, 2019.

[**4**] Ruixin Wang, Richard Alligier, Cyril Allignol, Nicolas Barnier, Nicolas Durand, and Alexandre Gondran. Cooperation of combinatorial solvers for en-route conflict resolution. *Transportation Research Part C: Emerging Technologies*, 114:36–58, 2020.

[**5**] Ruixin Wang and Nicolas Barnier. Global Propagation of Transition Cost for Fixed Job Scheduling. In *24th European Conference on Artificial Intelligence. Santiago de Compostela, Spain*, 2020.

Chapter 1

Context

Contents

1.1 Parallelization of Combinatorial Algorithms 7

1.1.1 Portfolio of Algorithms 7

1.1.2 Multi-Agent System 9

1.2 Cooperation of Solvers 10

1.2.1 Exchanged Data 11

1.2.2 Communication Points 13

1.2.3 Architecture 14

1.3 Summary 15

With the plateauing of processors frequency and the subsequent advent of multi-core computers during the last decade, there have been a lot of efforts from the scientific community to exploit parallelism for problem solving. Although recent developments on complete methods (e.g. Constraint Programming, SAT or MIP solvers) and incomplete methods (e.g. Local Search or Metaheuristics) have made the resolution of Combinatorial Optimization Problems (COP) more efficient, many instances remain out of reach. Various approaches consist in combining different optimization methods to cope with the largest instances of such problems, like separation of the search space, hybridization, portfolios of algorithms or multi-agent systems.

Among the aforementioned approaches, separation of the search space consists in splitting and distributing the search space among several processes. For example, the resolution of a search problem can be formulated as a search tree in a Constraint Programming (CP) system, so that solving a problem is similar to finding a feasible path in the tree starting from the root node. The children of a given node are a series of subproblems derived from the

subproblem corresponding to each node, and can be dispatched among a set of processors to speed up the search process. Examples can be found in [Rao and Kumar, 1993, Chu et al., 2009]. Note that this approach mostly speed up the search without any improvement on the quality of solutions, as a sequential execution will produce the same result. However, [McCreesh and Prosser, 2015] shows that specifically designed splitting strategies can improve parallel branch and bound beyond the expected speed-up due to the number of cores.

Hybridization of optimization techniques consists in dividing different research tasks into several (generally two) heterogeneous methods. Among successful approaches, CP can be considered as a framework able to integrate various other optimization techniques as a global constraint, like [Focacci et al., 2002] with an LP solver, or the other way around, CP can be used as a component of a metaheuristic, for example to explore ambitious neighborhoods within a Local Search (LS) algorithm, as in Large Neighborhood Search (LNS) [Shaw, 1998]. Metaheuristics can also benefit from the integration of other LS algorithms, like Memetic Algorithms (MA) [Hao, 2012b] which hybridize a population-based evolutionary algorithm with an LS, for example a Tabu Search, to improve new solution candidates. Following a hierarchical classification from [Talbi, 2009], a low level hybridization, also called *integrated approach*, uses an optimization algorithm as an internal operator for a metaheuristics (like LNS and MA), whereas a *high-level hybridization* is the execution of different algorithms in sequence, the output of an algorithm being used as an input to the next one.

The motivation behind high-level hybridization is that the currently executing algorithm can benefit from the output of the previous one. As mentioned in [Talbi, 2009], performances of metaheuristics can be well influenced by initial solution(s), so various techniques can be used at the start to generate solution(s) of good quality. In [Khichane et al., 2010] for example, a CP solver is used to initialize an *Ant Colony Optimization* (ACO) with consistent solutions, then the resulting best solution and pheromone tracks allow to improve a standard CP optimization procedure with the same model by providing a better upper bound and guiding the search strategy.

In our work, we tried to focus more on the parallelization and genericity of the approach whereas the aforementioned hybridization schemes concentrate on a tight integration at a finer grain of (generally) two algorithms to enhance specific phases. Other combinations exist, that better coincide with our goals, such as multi-agent systems [Dorri et al., 2018] and recent extensions of portfolios [Li and Hoi, 2014], two parallel approaches in which different algorithms are running on the complete problem instance individually. In multi-agent systems, a set of solvers attempt to solve the problem in parallel

while sharing useful information, whereas in [Li and Hoi, 2014], a subset of various solvers is selected to run in parallel until one finishes resolution.

We will first present in Section 1.1 the literature on different approaches to parallelization and then concentrate on the main contributions pertaining to our own cooperative approach in Section 1.2.

1.1 Parallelization of Combinatorial Algorithms

As mentioned in [Crainic and Toulouse, 2003], parallelization can be a convenient option to efficiently solve optimization problems with a strict time requirement, like the real-time and dynamic ones occurring in the field of air traffic control and management or vehicle automation. As the performance of a given algorithm can significantly vary depending on the problem, it is a rather intuitive idea to benefit from the growing available parallel computing power to simultaneously execute different algorithms in concurrent processes, either in an independent or cooperative way. Portfolios of algorithms (see Section 1.1.1) address the former approach, while multi-agent systems (see Section 1.1.2) address the latter.

1.1.1 Portfolio of Algorithms

All simple combinatorial solvers experience erratic performances when faced with different problems and instances, but some combinations of algorithms can be complementary, which enables to obtain a robust "meta-solver". To formalize this approach, the problem of "Algorithm Selection" was first introduced in [Rice, 1976]. Taking inspiration from Economics, the authors in [Huberman et al., 1997] present a portfolio method to reduce the risk of having an unpredictable variation in solver performance w.r.t. the instance: computational resources are allocated to a portfolio of algorithms which independently solve the same problem instance. Note that contrarily to multi-agent systems described in Section 1.1.2, there generally is no communication between the algorithms of a portfolio, i.e. they are executed in an independent way.

To solve a combinatorial optimization problem, several elements must be identified to design an efficient portfolio system: algorithms that are the best candidates and features of the problem that should be taken into account to select them. Additional strategies, like the classic *restart* strategy, can also be used to avoid the intensification of the search on a small region of the space only.

The most simple parallel portfolios, like *ppfolio* [Roussel, 2012], are just computer programs which run several SAT solvers in parallel, depending on the number of available processors. Similar to ppfolio, VPS_n (*Virtual n-Parallel Solver*) is introduced in [Amadini et al., 2015]. It runs in parallel with a fixed static selection of n solvers on a machine with n cores. It is a static portfolio because the n solvers are selected in advance, without consideration of the problem to be solved.

Another type of method for portfolio is per-instance algorithm selection based on the idea of a training set as mentioned in [James et al., 2013]. This approach consists in identifying the "features" of the problem (i.e. specific characteristics like the dimensions or constrainedness of an instance, or the performance of its resolution) and, based on a set of training problem instances and available resources, selecting the best algorithms, like the *ASlib* solver for Algorithm Selection described in [Bischl et al., 2016]. [Lindauer et al., 2015] introduces the parallel portfolio *claspfolio2* using Machine Learning (ML) techniques to estimate the top n best candidates to be run in parallel on n processors.

As in Local Search and in some metaheuristics, the *restart* strategy can be used in parallel portfolio, to avoid the search to be stuck in a local minimum and encourage diversification. In a parallel context, restart can be used to execute the same algorithms repeatedly with different starting configurations. A specific restart strategy is introduced in [Arbelaez and Hamadi, 2011] to solve SAT problems with a parallel portfolio of LS algorithms. Instead of randomly generating a new configuration, each algorithm starts with the configuration shared by others, the selected configuration being the one that minimizes the number of conflicting clauses in other LS. Note that this scheme can be seen as a multi-agent system (cf. Section 1.1.2) because the processes communicate with each other during the search.

Another portfolio-based cooperative algorithm, *sunny-cp2*, is presented in [Amadini et al., 2015]. It is the first parallel CP portfolio solver, enabling different CP solvers to execute simultaneously in a multi-core environment. When a solver finds no solution after some time limit, it can restart with the best bound shared by other solvers. Note that it is hard to predict which restart strategy is the best for a given problem as mentioned in [Hamadi et al., 2008].

On the whole, portfolios of algorithms help select the appropriate candidate algorithms for solving optimization problem. However, as cooperation hardly exists among them, the overall performance of the portfolio system can never surpass the performance of its best algorithm on a given instance, so that instances that are not reachable by individual algorithms will not be reachable using the portfolio either. So we choose not to adopt the idea

of portfolio but rather concentrate on how different algorithms may interact with each other during the resolution to enhance the overall performance. The literature on this topic is detailed in the following section.

1.1.2 Multi-Agent System

A multi-agent search is a software system consisting of a network of optimization solvers (agents) that run in parallel and interact with each other. It is designed to solve problems that are difficult or impossible to solve by any individual solver. Each agent works in its own way (e.g. with different algorithms, models, etc) on the entire problem and can communicate with others through a common database.

[Clearwater et al., 1991] is one of the earliest examples of a multi-agent system to solve a Constraint Satisfaction Problem (CSP), with an application to cryptarithmetic puzzles. All the agents can access a common database, in which anyone is allowed to read "hints", i.e. sequences of assignments locally consistent for at least one constraint, and generate new ones which are written back in the database. By sharing these assignments, the agents quickly converge towards tentatives with few unsatisfied constraints.

Multi-agent search can also be applied on distributed constraint satisfaction problem, where each agent only works on part of the problem, so no single agent can solve the entire problem by itself. A complete solution is composed by partial solutions provided by each agent. An example can be seen in [Al-Maqtari et al., 2006], which proposes a CP multi-agent system to solve an agricultural water management problem. The system relies on two types of agents: generic agents and *controllers*. The generic agents work on different clusters of variables and related local constraints, and send the values of their decision variables to controllers which are in charge of verifying whether constraints between the agents are satisfied and inform them back. Note that not all problems are well suited to be partitioned in clusters of variables, as an increasing number of agents may lead to communication overheads that burden the whole system.

As a conclusion, there are two different types of multi-agent systems which differ from each other on whether the agents work on the entire problem or just on part of it. We choose to develop in this thesis the former, as each agent can have a more global view of the problem. Moreover, agents have more freedom to employ different types of algorithm, which is very important as the weaknesses of one algorithm dealing with certain aspects of a problem can be compensated for by other ones. However, multi-agent systems in the literature are mostly tailored to a specific problem, which increases the difficulty of other agents (i.e. other algorithms) to join. We will propose a

more generic cooperation framework in Chapter 2, which can allow various combinatorial solver to be easily integrated and cooperate with each other on a given problem.

Before introducing our framework, we first need to discuss why and how different kinds of algorithm can improve their performance through mutual collaboration. The related literature is presented in the following section.

1.2 Cooperation of Solvers

Based on the observations about the various approaches of parallelization in the previous section, we decided to develop one sub-branch of multi-agent systems, namely the cooperative approach where each agent solves the entire problem and shares useful information with each other. We first discuss the strengths and weaknesses of different optimization algorithms, and how sharing information can improve their performance.

Optimization methods can be either complete or incomplete. Complete methods, like CP or the Branch & Cut (BC) algorithm of MIP solvers for example, are able to find and prove optimal solutions, as they exhaustively explore the search space while reducing its size as much as possible. However, execution times may increase dramatically with the problem size. On the other hand, incomplete methods, such as metaheuristics, can generally provide solutions of good quality in a reasonable time, however they cannot prove the optimality of a solution nor the infeasibility of a problem instance. Population-based metaheuristics (P-MH), like GA, MA, *Particle Swarm Optimization* (PSO) or ACO consist in improving a population of candidate solutions iteratively. Whereas Single-solution-based metaheuristics (S-MH), e.g. LS, TS and *Simulated Annealing* (SA), make iterative local improvements on a current solution to generate another one in the search space. Note that, P-MH is good at exploration, however, weak at exploitation, compared to S-MH.

For large instances of combinatorial problems, complete methods are intractable, and metaheuristics are favored, as they can still produce a valid solution, even though it can be far from the optimum. However, information found by one method might be helpful to the other and vice versa. Cooperative approaches try to benefit from this fact by setting up data exchanges between different algorithms, so that the resulting solver must have better performance than each algorithm considered individually. Designing such a cooperative parallel model requires the following issues to be addressed:

- identify the data that is worth exchanging (see Section 1.2.1), i.e. that can be made available by some algorithm and can benefit to others;

- determine at which pace and at which step for each algorithm the exchanges must occur (see Section 1.2.2);
- determine the network topology and if the involved processes should communicate directly among each other, or *via* a common database. (see Section 1.2.3).

1.2.1 Exchanged Data

The first step to set up an efficient cooperation scheme is to determine what kind of information could be useful for each algorithm, and whether such information can be made available by the others.

In the case of metaheuristics or other local search methods, the principle of maintaining one or several candidate solutions suggests that such solutions could be periodically exchanged to help others. For example, [Diekmann et al., 1996] proposed a parallel simulated annealing, where each algorithm exchanges its current best solution after a fixed number of iterations. In a parallel GA, several good candidate solutions (e.g. a small set of the population) could be migrated at once, as introduced in [Solar et al., 2002]. Information about the search state of each metaheuristic can also be shared. For example, [Cordeau and Maischberger, 2012] implemented a parallel iterated TS to solve a vehicle routing problem. In order to distribute the computational burden among TSs, the maximum number of iterations without improvement is exchanged synchronously. Note that the quantity of exchanged information must be limited to a certain extent, otherwise the communication process may take too much time w.r.t. the search efforts. Moreover, too much incoming information may increase the risk that a metaheuristic abandons its current (potentially promising) search process to redirect itself to a search trajectory which is already explored by others.

On the side of exact methods, the Branch and Cut algorithm (BC), used to solve Mixed Integer Linear Programming (MILP) problems, is a variant of the Branch and Bound (BB) algorithm. [Ralphs et al., 2018] has given a survey on parallel solvers for MILP optimization. As listed in the survey, information that has been considered for exchanges include: global upper bounds, nodes in the search tree, feasible solutions and valid inequalities. Good upper bounds can help other BC to prune their search tree, e.g. if one part of the tree has a lower bound that is already higher than the received upper bound; nodes in the search tree can be shared to dynamically balance workload on cooperative solvers; feasible solutions can enhance primal heuristics of other cooperative BC to find other feasible solutions of good quality (more details about primal heuristics can be found in [Berthold, 2014]); finally, valid in-

equalities (i.e. constraints) can be shared to strengthen relaxations in other part of the search tree. As in the case of metaheuristics, a trade-off is needed between the amount of exchanged data and the communication overhead.

Constraint Programming is another exact method based on the Branch and Prune (BP) algorithm, another variant of BB. Similarly to BC, new upper or lower bound can also help solvers to efficiently reduce the size of the search tree. Such bound exchanges are, for example, implemented in *sunny-cp2*, the parallel CP solver mentioned in Section 1.1.1, where each individual component is working on the entire problem. However the shared bound is only used to restart the solver when no solution is found before a given time limit.

In parallel SAT solvers, learnt clauses can be shared among the processes, as proposed in [Hamadi et al., 2008] and afterwards improved in works like [Guo et al., 2014], to prevent individual agents from encountering the same conflicts as others, hence agents help each other taking efficient decisions during the search and finally improve the overall performance. As stated in [Hamadi and Wintersteiger, 2012], estimating the shared clause quality in terms of its local impact is very hard. Though very promising, we did not implement the sharing of learned constraints in our framework yet, as the first experiments with such solvers (e.g. Chuffed [Ohrimenko et al., 2009]) on our target COPs did not provide good enough results and the CP library we used to develop our CP solvers lacks this feature. Moreover, as the cooperating solvers of our framework may use different internal models, though agreeing on the decision variables and the cost to exchange information, new constraints learned on the internal model variables would have to be translated before being sent, burdening the genericity of our approach.

Different types of algorithms can also search a problem in parallel and exchange information through *shared memory* between *multiple threads* to improve overall performances. A previous work at the ENAC Lab [Vanaret et al., 2013] has proposed a cooperation approach between a Differential Evolution (DE) metaheuristic and an Interval Branch & Contract (IB&C) algorithm to solve continuous optimization problems: the DE provides solutions to the IB&C in order to improve the pruning of its search space, then the reduced search space is given back to the DE to intensify the search on the most promising regions. The resulting cooperation performs better than each individual component. However, in the context of combinatorial optimization, it is much more difficult to derive a favorable region of the search space from a good solution.

We have seen what type of information is worth sharing in a cooperation approach. The next step is to determine the relevant time to proceed to the exchanges, which is discussed in the next section.

1.2.2 Communication Points

Determining when the data exchange have to take place involves taking into account the internals of each algorithm. Considering a single algorithm, sending new information can happen either in a "blind" or adaptive manner, as described in [Talbi, 2009]. With the blind way, the algorithm sends information after every fixed period, e.g. a fixed number of iterations for a metaheuristic. Otherwise, some criteria can be evaluated to decide whether new found information is worthy to be shared, as in [Shi and Zhang, 2018] where only better solution are shared.

Then, considering a set of algorithms running in parallel, the exchanges can be classified into two modes, *synchronous* and *asynchronous*. In a synchronous cooperation, all processes need to choose a fixed communication point in advance. The exchange phase starts only when every algorithm has reached this synchronization point, as mentioned in [Crainic and Toulouse, 2010]. To avoid blocking the search and wasting time at synchronization points while waiting for other algorithms, the asynchronous mode allows any algorithm to acquire and send information at its own pace, which is the most efficient distributed design. Asynchronous cooperative parallelization are well studied, for instance in [Crainic, 2005, Polacek et al., 2008, Ren et al., 2018].

If algorithms of different kinds are involved in the cooperation, an asynchronous communication is likely to be more suitable than a synchronous one, as finding synchronization points that suit every process and works for every instance will be challenging. Moreover, differences in resolution process between constructive, complete tree search algorithms and incomplete ones, which improve one or multiple solutions iteratively, imply that communication should occur at different steps of the algorithms. For incomplete algorithms like LS/MH, with successive iterations applying to candidate solution(s), it is quite natural to share information whenever a new feasible solution has been produced. We can then take advantage of this communication point to also receive new information when available. However, for tree-search-based complete algorithms like the ones of CP or MIP solvers, we can only send new solutions at the leaves of the search tree, whereas acquiring new optimization bounds can be done at any node, i.e. at a much more sustained pace. Therefore, asynchronous communication will be better suited to such a generic cooperative framework involving various types of optimization methods, as described in Chapter 2.

Once the communication points have been set, a topology for the communications must be implemented accordingly. Various topologies are described in the next section.

1.2.3 Architecture

As for any distributed application, setting up a cooperation scheme for optimization algorithms requires to choose a communication architecture between processes. Direct communication can be seen in parallel metaheuristics algorithms [Tanese, 1989, Alba and Dorronsoro, 2009] using topologies like the "ring" graph and the "coarse-grain island" model, where the population is distributed over the nodes of the graph. The advantage of these schemes is that a good level of diversity can be maintained as communication only exists among neighboring solvers and only local exchanges can occur. However, in a generic cooperative framework with any kind of solver, possibly constructive ones like CP or MIP which do not maintain any candidate solutions population, these schemes seem irrelevant.

Indirect exchange means that the algorithms communicate with each other through an in-between process, which often performs like an information pool, accessible by any process. This indirect exchange concept has been used in many asynchronous cooperative solvers. The well-known topology for indirect exchange is the *server-client* design pattern, in which the server plays the role of a *data manager*. [Crainic et al., 1996] proposes a parallel asynchronous TS, where different TSs share their best solution through a server. The shared solution is not necessary the best one so far, but rather a randomly selected solution from the list maintained by the server. This technology can avoid similar TSs, deployed as cooperative processes, from converging to a similar region of the search space.

The issue of using a centralized server and clients rather than decentralized autonomous agents is more an architectural problem which concerns the robustness to hardware or software failure related to one of the framework processes. With such a pattern, our framework can be easily designed to be robust to one (or several) of the client failure, which may happen quite regularly with COP state-of-the-art solvers and development versions, and solvers can be plugged or unplugged at any time to contribute to the overall distributed framework without hindering anything but their own search state. The loss of the server is more problematic and would require to store checkpoints and dynamically reconnect to clients, which was left for future development. Decentralized autonomous agents would be even more robust to any failure, but all processes would have to store the complete state of the distributed algorithm and our non-blocking asynchronous communication scheme along with the update internal logic would be much more cumbersome.

As a result, the choice of an indirect exchange scheme seemed the simplest and most versatile based on the above discussion.

1.3 Summary

In this chapter, we have first presented a literature review on different approaches to parallelization, with a focus on one branch of multi-agent system, namely the *cooperative* approach with every agent working on the entire problem. The reason why we choose to develop a multi-agent method instead of a portfolio is the idea that sharing information among algorithm during their execution can break through the limits of one individual algorithm, whereas portfolios concentrate on which algorithm could be the best single one for each given problem kind or instance.

Then we have discussed why and how different kinds of optimization algorithms can improve their overall performance by collaborating with each other. To design such a cooperative framework, we carefully analyzed three issues: what kind of data is worth exchanging, when the exchange must take place and what is the exchange architecture. According to our discussion, we decide to develop a generic framework with a centralized *server-client* architecture, robust to the loss of any client, in which any optimization solver can easily be integrated and *asynchronously* share *any kind of information* (e.g. cost, solution and proofs in the applications described in Chapters 3 and 4) through the central data manager (i.e. the server). The next chapter describes in details the design of this cooperative framework, then shows how various combinatorial optimization algorithms can be plugged in.

Chapter 2

Generic Framework for Algorithm Cooperation

Contents

2.1 A Generic, Distributed Framework 18

2.1.1 Asynchronous Server-Client Architecture 19

2.1.2 Interprocess Communication Protocol 25

2.1.3 Message Format 27

2.2 Cooperative Algorithms 28

2.2.1 Memetic Algorithm 29

2.2.2 Integer Linear Programming 32

2.2.3 Constraint Programming 38

Many problems in the field of Air Traffic Control (ATC) and Management (ATM) are highly combinatorial, such as conflict resolution, organizing ground operations between aircraft arrival and departure at airports or designing an efficient route network for a given airspace. In many cases, exact algorithms cannot provide solutions within a time-frame that is suitable with operational constraints, and metaheuristics can be the best option. However, such methods provide very few, if any, interesting properties about the fact that a feasible solution can be found or that a solution is optimal. We propose to combine the use of both exact algorithms and metaheuristics to handle such ATC and ATM problems. Although we propose a generic framework, our work focuses on the techniques that are already in use for solving ATM problems in our research team and others. For example, [Durand et al., 2016, Allignol et al., 2012] provide literature reviews about the

uses of metaheuristics and constraint programming respectively, applied to ATM problems. Hopefully, such a framework will help research teams with specializations in various types of algorithm to federate around such problems.

In this chapter, we introduce a generic cooperation framework based on a server-client scheme to integrate various combinatorial solvers in order to have them cooperate on a given problem, with the objective of taking advantage of the strength of each one of them. In Section 2.1, we present the general architecture of the distributed cooperation system, and provide details about the communication between server and clients. Section 2.2 shows how to adapt combinatorial optimization algorithm to this framework, with a focus on three classic methods: a metaheuristic, Integer Linear Programming and Constraint Programming. The use of this framework and adapted algorithms are later illustrated on two ATC and ATM related problems: en-route conflict resolution (Chapter 3) and gate allocation at airports (Chapter 4).

2.1 A Generic, Distributed Framework

We propose a generic framework in which any number of solvers can exchange information to help each other to solve such highly combinatorial problems as the ones encountered in the field of ATC and ATM. The main question that arose was to determine which communication mode to adopt. As described in Chapter 1, optimization algorithms may exchange information either directly or indirectly, and communications can be either synchronous or asynchronous. Given the variety in nature and the potentially large number of optimization algorithms that might cooperate through our framework, it was simpler and more versatile to set up indirect, asynchronous exchanges through a dedicated central process that acts as a data manager.

The general architecture is depicted in Figure 2.1. The data manager holds information about the instance of the problem to solve, as well as on the state of the search for solutions, such as solutions or bounds. Any solver can, at any time, ask for such information or send information they have discovered (e.g. a new solution, a better bound). Currently, only full solutions, upper/lower bounds and status of the optimality proof of the instance are being managed, but the framework is fully generic and could store and share any kind of information, such as global cuts or nogoods, for example.

Given this general scheme, we implemented this framework as a *server-client* architecture, where the central data manager is the server and solvers are the clients. The main advantage of this choice, apart from the fact that it is easy to set-up, maintain and extend, is that it can be distributed

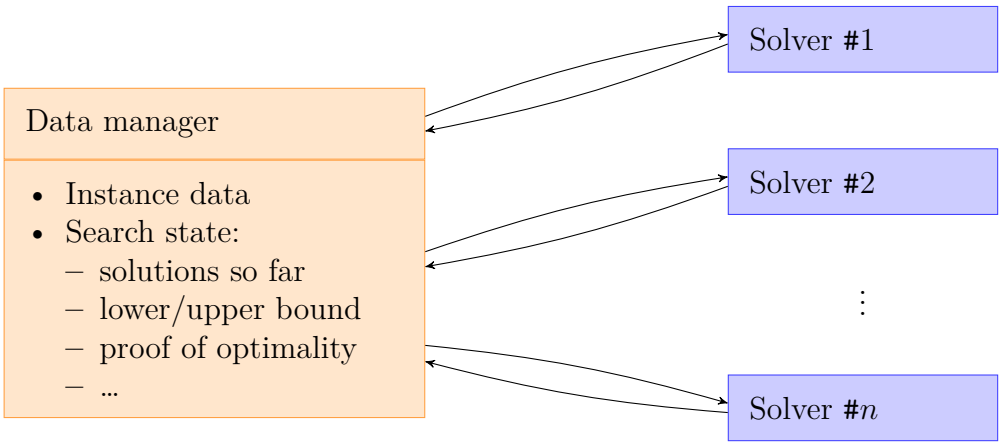


Figure 2.1 – General framework architecture.

over the computers in a network as easily as over the cores of a computer. Also, it is platform- and language-agnostic, as the only requirement is to implement the communication interface. This architecture is also robust to algorithm failures, as processes can connect or disconnect at any time. If one algorithm fails, for any reason, it does not compromise the rest of the resolution by the other algorithms. The main current weakness is the potential failure of the data manager, as in that case, the algorithms would be lacking the communication. Since such a scenario has never been observed in our experimentations, we did not set up any redundancy that would help overcome this issue.

Section 2.1.1 describes the general architecture of our framework as well as the different communications that can occur during the resolution. Section 2.1.2 provides further technical details about the inter-process communication. Finally, Section 2.1.3 describes the format of the messages to be exchanged between solvers and the data manager.

2.1.1 Asynchronous Server-Client Architecture

The central element in our architecture, the server, implements a data and connection manager. It is in charge of setting up the problem instance, accepting connections from the clients and providing them with the instance data, updating data upon reception from a client, and publishing new data. Algorithm 2.1 represents the event loop of the server.

The server is first initialized (line 1) by creating the instance data (e.g. reading them from a file) and opening the required communication channels (see Section 2.1.2 for details). An initially empty state is created to later

```
1: (instance, state) ← initializeState()
2: while termination criterion is not met do
3:     (client, request, data) ← receive()
4:     switch request do
5:         case first connection
6:             send(client, instance, state)
7:         case new search data
8:             update(data, state)
9: return best_solution(state)
```

Algorithm 2.1 – Server event loop.

store information about the search for solutions. Then, the server enters a main loop, in which it awaits for clients requests (received at line 3) and processes them with one of the two following procedures:

- If the client connects for the first time, register the client and provide it with the instance data and current search state (line 6).
- Else the client has some new information to share. This information is used to update the search state (line 8) and, if it lead to any progress (e.g. there was a higher lower bound), then the new search state is shared among all clients (line 8), which is to be described in Algorithm 2.2.

As soon as a pre-defined termination criterion is met (e.g. one of the solvers proved the optimality of a solution, or a time limit has been reached), the best solution is returned (line 9).

The clients, i.e. the solvers, have to connect to the server to start solving the instance. After the first connection, they might subscribe to future updates if they want to benefit from the information found by other solvers. Also they can post new information to the server during the search. As explained in Chapter 1, we opted for an asynchronous communication scheme for this framework. In such a scheme, each individual process implements a message queue, where messages addressed to it are temporarily stored. The underlying program can access that message queue at any time (in a first-in-first-out manner), providing high flexibility for the adaptation of well-known optimization algorithms to the cooperation, as we can choose, for each kind of solver, the suitable point in the code to either send or receive information. Further details about how we adapted various kinds of solvers to the framework are provided in Section 2.2.

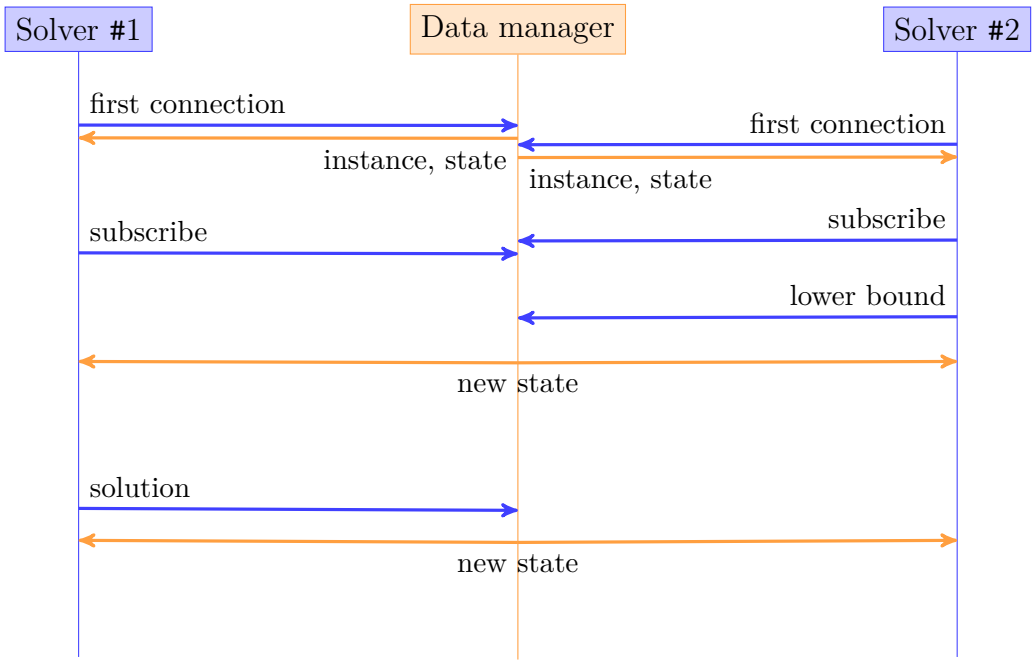


Figure 2.2 – Example of a communication sequence between two solvers and the data manager.

Figure 2.2 illustrates a communication sequence involving the data manager and two solvers. The vertical rules represent the timelines for each process (the upper the earlier), and the blue (resp. orange) arrows represent the messages from a solver to the data manager (resp. from the data manager to a solver). Both solvers, whenever they are ready, start by registering to the data manager, which in return sends the instance data and current search state. In this example, both solvers also subscribe to further updates to the search state (the subscription was made optional to handle algorithms that might only provide data without requiring any). Later on, solver #2 computes a new lower bound, and sends it to the data manager. This lower bound, in our example, is better than the one currently held, thus the search state is updated and broadcasted to every solver by the data manager. From now on, solver #1 can use the lower bound found by solver #2 to speed up its internal search algorithm. The same process is repeated later, when solver #1 finds a solution to the instance.

There is no theoretical limitation regarding the type of knowledge that can be exchanged through this framework. However, as stated in Section 1.2, a suitable trade-off must be found between the quantity of information and its relevance in the context of the cooperation. We have currently implemented

and experimented the following knowledge exchanges:

- complete solutions;
- lower and upper bounds;
- proofs of optimality or infeasibility.

Our implementation of the data manager maintains a state that contains the above information (for a minimization problem: the solution with lowest cost so far, lower bound and a boolean corresponding to the current status of optimality proof), with an history of the received data, each one tagged with the identifier of the process that sent it. The underlying internals of the `update` function are detailed in Algorithm 2.2. Depending on the type of data that it receives, the procedure:

- updates the search state with the new information, either a better solution (line 24) or a better lower bound (line 16);
- concludes on optimality and terminates; this is the case when a received lower bound is equal to the cost of the current solution (line 12) or conversely when a received solution has a cost equal to the current lower bound (line 20);
- broadcast the new search information among all clients.

Whenever information is broadcasted, the identifier of the process that initially provided the information is associated to the message, so that a process might easily ignore information that were produced by its own underlying algorithm.

In our experiments, complete solutions are incorporated by population-based, cooperative metaheuristics, leading to a faster convergence. Bounds have been used by complete algorithms to help prune the search space and thus speed-up the discovery of new solutions or the proof of optimality (or infeasibility). The exchange of such proofs does not help the resolution per se; they are used to tidy up the cooperation process by informing all other algorithms that they can stop. An example of communications between a data manager, a metaheuristic and a CP solver is depicted in Figure 2.3. At the beginning of this sequence, the search state contains a solution of cost 158 and a lower bound of value 82. After a few data exchanges that consecutively update the search state, the data manager arrives at a state where both the cost of the best solution and the lower bound have value 117, which ends the search (the optimality has been proved at this point) and triggers the sending of termination messages to all solvers.

```
1: function update(data, state)
2:     switch data do
3:         case infeasible
4:             state.infeasible ← true
5:             broadcast(stop)
6:         case optimal solution
7:             state.optimal ← true
8:             state.best ← optimal solution
9:             broadcast(stop)
10:        case lb                                    // new lower bound
11:            if lb = state.best.cost then
12:                state.optimal ← true
13:                broadcast(stop)
14:            else
15:                if lb > state.lb then
16:                    state.lb ← lb
17:                    broadcast(state.lb)
18:        case solution                              // a new solution
19:            if solution.cost = state.lb then
20:                state.optimal ← true
21:                broadcast(stop)
22:            else
23:                if solution.cost < state.best.cost then
24:                    state.best ← solution
25:                    broadcast(state.best)
```

Algorithm 2.2 – Search state update procedure.

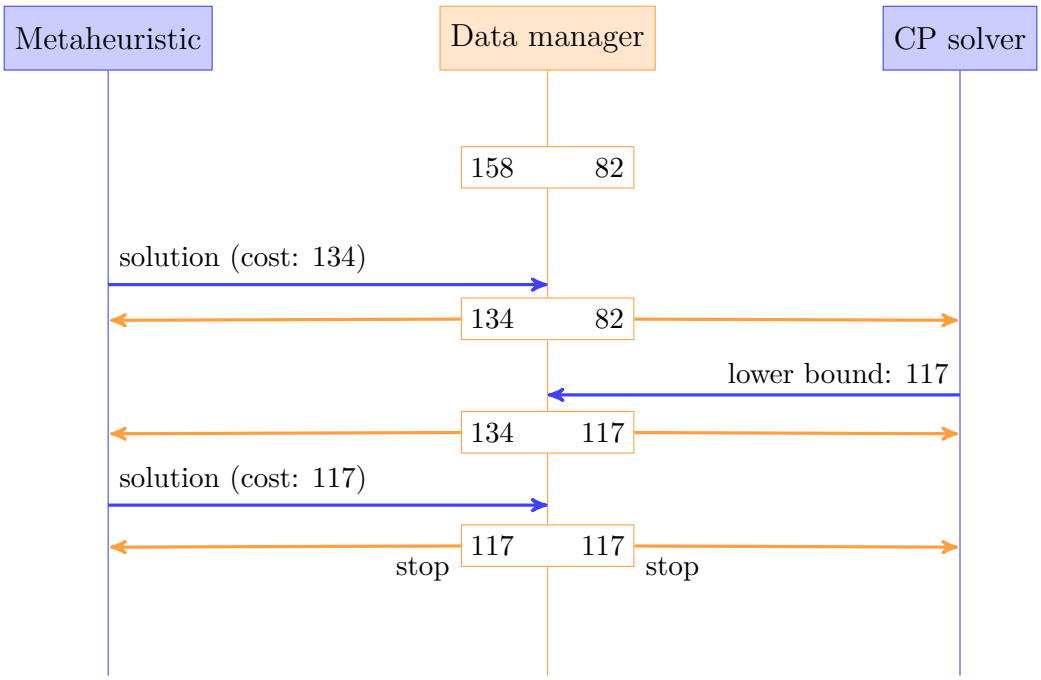


Figure 2.3 – Cooperation between a metaheuristic and a CP solver. In the orange boxes are displayed the cost of the best current solution (left) and the lower bound (right) from the search state.

The next section provides technical details about the implementation of the framework, focusing on the communication aspects.

2.1.2 Interprocess Communication Protocol

One of our objectives in the design of our cooperation scheme was to ensure that it would be as generic as possible, either in terms of the data that is exchanged or of the solvers that can connect to it. To handle the latter concern, we needed a software component which could transfer messages across processes, either on a single computer or a network. Other important criteria for our choice were performance, an open license and the possibility to use the software with many programming languages, so as to be able to plug in many existing solvers. We decided to use ZeroMQ (ØMQ) [Hintjens, 2013], which meets all the above criteria:

- it has bindings to more than 50 programming languages (among which 13 are officially supported), including the ones that we mainly use: OCaml for CP programs using the FaCiLe library and C/C++ for MILP programs using Gurobi;
- it is distributed under GPLv3 open license;
- it is (self-)presented as "a high-performance asynchronous messaging library" and is widely used.

This library is also well-known for being simple to use in any context (hence the "Zero"). Alternatives could have been to directly use sockets, however ØMQ handles the low-level parts, which is more convenient. Also, any message passing technology (e.g. MPI) could be used for implementing such a framework

Various communication schemes can be implemented using ØMQ, depending on the topology (one-to-one, one-to-many, many-to-many, etc.) and the required message synchronicity (e.g. with or without acknowledgement). Our framework uses two schemes called request-reply and publish-subscribe that are described next, and are illustrated in Figure 2.4.

Request-Reply This is a one-to-one communication scheme, where one process makes a request to the other, and awaits for a reply (no other requests can happen before the reply has been received). In our setup, this will be used for the first connection of solvers to the data manager, and for solvers to send new information. In both cases, the solver is performing the request and the data manager produces a reply. In practice, the data manager has a REP socket that is bound to a given port, and each solver has a REQ socket that is connected to this port on the server.

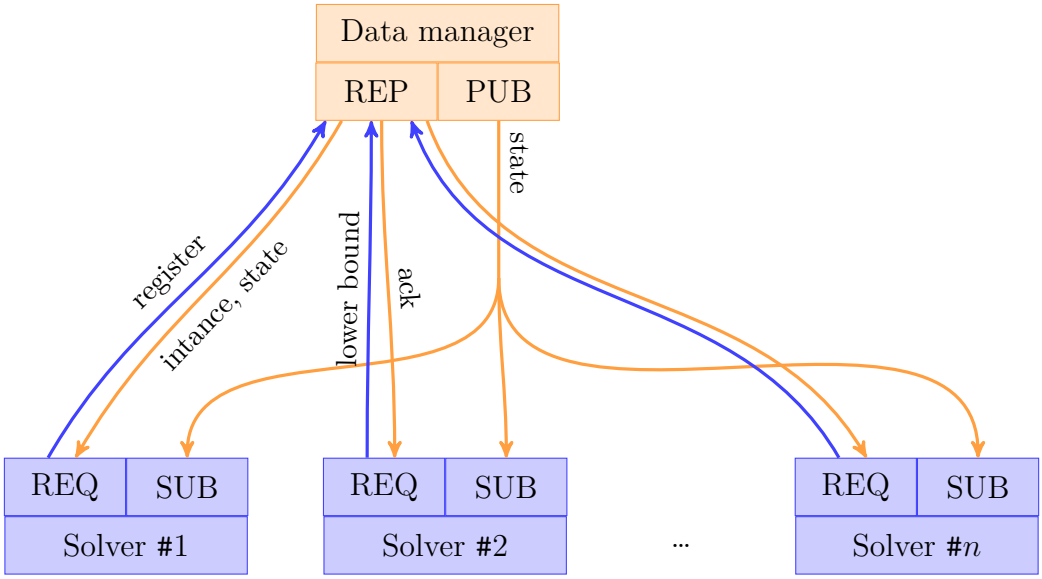


Figure 2.4 – Illustration of exchanges between the data manager and solvers using ∅MQ socket pairs.

Publish-Subscribe In this one-to-many mode, one of the processes (the publisher) sends messages to any process that has subscribed. This is used in our framework to publish the new search state any time it has been modified. Only the data manager uses such broadcastings; it has a PUB socket bound to a port (different from the one used by the REP socket). Each subscribing solver has a SUB socket connected to that port on the server. This communication scheme is non-blocking, which is a particularly interesting feature to maximize the use of the computation time: each algorithm checks its messages, but if there is none, it can continue its computations.

In Figure 2.4 (with the same color scheme as in the previous figure), we see that each solver can communicate individually with the server using the REQ-REP socket pair. For example, solver #2 sends a new lower bound, and gets acknowledgement for its reception. The data manager can broadcast a new search state to all solvers using the PUB-SUB socket pairs.

We have described the architecture of our framework, and detailed the communication protocols involved. The next section focuses on the contents of the messages, by defining a common interface between the various clients embedding solvers of different types and the server.

2.1.3 Message Format

The most efficient way (in terms of message size) to pass data among processes is serialization. However, as we want our framework to operate with programs written in any language, running on different hardware with different operation systems, this can become particularly difficult to set up. We thus chose to exchange data in a textual format, which is a bit more costly, but much more versatile. To help define an interface for exchanges, we use Extensible Markup Language (XML [Bray et al., 2006]) to add a clear structure to the messages. The advantages of the XML format are twofold:

- it is both human- and machine-readable;
- it is widely used in many kinds of applications, thus someone writing a new solver for our cooperation framework is likely to know the format already.

We use an XML structure with various types of sections to distinguish message types (e.g. first connection, or search state broadcasting) and carried information (e.g. client identifier, solution to the instance). In the case of a solver first connecting to a running data manager, the solver sends a very basic message consisting only of a single, empty section of type `first`. The reply from the data manager, shown in Listing 2.1, contains information that can be split into two categories:

- Connection information indicate to the solver how to set up further communication with the data manager. It contains a unique identifier (line 2) that the solver must indicate when performing requests, and a connection port number (line 3), i.e. where to subscribe to get up-to-date search data.
- Instance information tell the solver which instance is being solved (here, at line 4, it references a file name, however richer structures can be used), and, if other solvers have already made some progress, the current search state. In our example, the search state (from line 5 to 14) already contains a non-optimal solution with cost 20, which is described by a sequence of values.

Given this format, all cooperative algorithms are required to use the same main decision variables and cost, even though each one can freely use any additional variable for its own use.

The server part of the framework we have described in this section was implemented with the OCaml language. As we have also implemented solvers

Listing 2.1 – Reply from the data manager to a newly registered solver.

```
<f_reply>
  <id_client>6</id_client>
  <pub_port>5556</pub_port>
  <data_file>cluster_10ac_2err_6</data_file>
  <sol_option>
    <instance>
      <sol>
        <sub>
          <cost>20</cost>
          <values>1;2;10;6;59;30;16;26;96;66</values>
        </sub>
      </sol>
    </instance>
  </sol_option>
</f_reply>
```

for this framework with C, C++ and OCaml, we have produced wrappers to help connecting new solvers written with one of those languages. In the next section, we describe how we adapted classic optimization algorithms to our cooperation framework.

2.2 Cooperative Algorithms

The cooperation framework we have described in the previous section is generic as regards the type of information to exchange, and relies on asynchronous communications, meaning that solvers are free to send or receive up-to-date information about the search state whenever they decide to. Thus writing a solver for this framework requires the following:

- Identify the information that it can provide to others. For example, a local search could provide its current solution with its cost; an integer linear program could provide solutions and bounds.
- Identify the information that could benefit the underlying algorithm. For example, a population-based metaheuristic could integrate solutions to its population to enhance the convergence; a Branch and Bound algorithm would benefit from upper or lower bounds to prune the search tree.

- Determine the suitable timing for data exchanges, e.g. after a fixed or adaptive number of iterations of a metaheuristic, or upon backtracks in a CP solver.

In this section, we present the various optimization algorithms used in the applications described in Chapters 3 and 4, i.e. a metaheuristic, namely a Memetic Algorithm in Section 2.2.1, and two exact methods, Integer Linear Programming in Section 2.2.2 and Constraint Programming in Section 2.2.3, and show for each of them how they can be adapted to our cooperation framework.

2.2.1 Memetic Algorithm

A *Memetic Algorithm* (MA) [Hao, 2012a], is a combination of an evolutionary algorithm and a local search algorithm. In this section, we first present the principle of the Memetic Algorithm used in the application presented in Chapter 3. Then, we provide further details about a few aspects of the MA used: construction of an objective function, use of a Tabu Search intensification, and crossover operator for diversification. Finally, we show our collaborative version of the algorithm.

Principle

The MA used in this thesis, described in Algorithm 2.3, is a hybridization of an Evolutionary Algorithm and a Tabu Search (TS). It is thus a population based algorithm, where each individual is a possible solution. The main feature of our MA is that each element of the population is a local minimum of the objective function.

```
1: function MA(P)
2:     population ← initializePopulation(P)
3:     while termination criterion is not met do
4:         (parent_1, parent_2) ← select(population)
5:         child ← crossover(parent_1, parent_2)
6:         (child, c_c) ← TabuSearch(child)
7:         population ← replace(population, child, c_c)
8:     return the best element of population
```

Algorithm 2.3 – Memetic algorithm

First (line 2), a population of candidate solutions is randomly initialized from the problem P, and a TS (described in Algorithm 2.4) is applied to each

candidate. Then (line 4), we randomly select two elements called *parents* in the population and generate a new element called a *child* through a standard *crossover* operation between the two parents that recombines their solutions (line 5). Afterwards (line 6), the child is improved by applying a TS until a local minimum is found (with cost c_c). Then the worst element of the population is replaced by this *child* if its cost is lower and if it does not already belong to the population (line 7). We iterate this procedure until a given time limit (line 3) or when no improvement is made for a given number of iterations.

Objective Function

We presented the general principle of our MA with a minimization problem in mind. The minimization criterion is related to some performance of the solution with respect to the problem being solved (e.g. minimize the departure delays when computing a take-off sequence, or minimize the fuel consumption when providing maneuvers to avoid conflicts). However, with an MA, as with most metaheuristics, it is often difficult to maintain a population of candidate solutions that respect all the constraints of the problem. In such a situation, it is convenient to allow for some constraint violations and to account for such violations in the minimization objective. To do so, we generally define two terms for the minimization:

$$f_{\text{sat}}(sol) = \sum_{\forall i} C_i(sol) \tag{2.1}$$

$$f_{\text{cost}}(sol) = \text{cost}(sol) \tag{2.2}$$

where Equation (2.1) represents the number of unsatisfied constraints for a given solution, while Equation (2.2) represents the real cost of a solution. Then for the MA, we can define the objective function as the following linear combination:

$$f(sol) = M \times f_{\text{sat}}(sol) + f_{\text{cost}}(sol)$$

where M is a big enough integer to guarantee that, given two candidate solutions, the one with more constraint violations will have a higher value for f, no matter the real optimization cost.

Intensification with Tabu Search

Each generated candidate solutions (either at initialization or after a crossover operation) is improved with respect to the objective function by a Tabu

Search (TS). TS is a local search algorithm, and as such operates by making successive, small changes (or *moves*) to a solution, trying to keep only the best moves. As an example, when trying to find the best schedule for pilots in an airline, a solution is described by an assignment of pilots to flights, and a move could be to exchange the flights for two pilots. The main feature of TS is that it maintains a *tabu list* of moves, which is a list of the moves that are (temporarily) forbidden, in order to be able to escape from local minima. Algorithm 2.4 describes a generic Tabu Search.

```
1: function TS(s)
2:     tabuList ← ∅
3:     while termination criterion is not met do
4:         mv ← selectBestMove(s, tabuList)
5:         s ← move(s, mv)
6:         tabuList ← update(tabuList, reverse(mv))
7:         (s_best, c_best) ← saveBest(s_best, s)
8:     return (s_best, c_best)
```

Algorithm 2.4 – Tabu Search

The input of the algorithm is a candidate solution s, and the tabu list is initially empty (line 2). First (line 4), the set of all possible moves from s (called the *neighborhood* of s) is computed, and the best move mv from this neighborhood that is also *not* in the tabu list is selected. The current solution s is updated using this move (line 5). Notice that the selected move does not necessarily improve the cost of the candidate solution s. In order to avoid coming back on our steps in a near future (and thus be able to escape from local minima), the reverse of mv is added to the tabu list for a pre-determined number of iterations (line 6). It will not be possible to select this move as long as it is in this list. Finally (line 7), the best solution (and its associated cost) is updated. This procedure is iterated until a fixed termination criterion has been met (line 3).

In the Memetic Algorithm, TS ensures the *intensification*, that is the convergence of the population towards a locally minimal solution. Another mechanism is required to provide *diversification* and make sure that we do not let large parts of the search space unexplored.

Crossover for Diversification

The crossover is an operation that combines two or more candidate solutions into one or several new candidates. The recombination can take various

forms, and mostly rely on a random scheme that selects parts of the solution from one parent or the other, an example is detailed in Chapter 3.

In most cases, such combinations are interesting to gather interesting parts of several solutions. However, the resulting element(s) are likely to violate many constraints, as the recombination does (generally) not take them into account. In our Memetic Algorithm, the Tabu Search that is applied to every newly created candidate solution helps mitigate their poor quality by providing some sort of repair. This operation largely contributes to diversification for the MA.

Collaborative Version of Memetic Algorithm

As described above, the MA is an iterative process in which exploration (*via* crossover) and intensification (*via* Tabu Search) are combined in each step. The idea behind the adaptation of our MA to the cooperation framework is simply to communicate at each iteration. As the MA essentially handles solutions, it can hardly provide nor benefit from lower bounds. However, it is able to provide solutions, and more interestingly its best solution so far. It can also integrate some solution from other algorithms to enhance its own population. The resulting Collaborative Memetic Algorithm (CMA) is described in Algorithm 2.5.

Before entering the main loop, the CMA registers to the server (line 1) to get the instance P, and possibly a solution s_r or a proof of termination p (either a proof of optimality, or a proof of inconsistency). If there is indeed a solution s_r available, it is integrated into the initial population and it is the best solution so far (lines 5 to 7). If there is no proof of termination, then the main loop starts as in our standard MA, with selection, crossover and TS (lines 9 to 12). If the child solution that has been produced is the new best solution so far, and if all constraints are satisfied (i.e. $f_{\text{sat}} = 0$), it is shared to the others by sending it to the server (lines 13 to 15). Before the next iteration, the mailbox is checked, and if any new information is available (i.e. a solution or a proof of termination), it is integrated (lines 16 and 18). A Cooperative Memetic Algorithm is used in Chapter 3 to solve a problem of air conflict avoidance.

2.2.2 Integer Linear Programming

Integer Linear Programming (ILP) [Jünger et al., 2009] has become a very powerful tool for modeling and solving real-world combinatorial optimization problems, like planning and scheduling. In this section we describe the general model for ILP problems and a common algorithm, namely Branch

```
1: (P, s_r, p) ← register(server)  // get problem and current best solution
2: function CMA(P)
3:     population ← initializePopulation(P)
4:     c_best ← +∞                          // to hold the cost of the best solution
5:     if s_r ≠ ∅ then
6:         population ← replace(population, s_r)
7:         c_best ← cost(s_r)
8:     while termination criterion is not met and ¬p do
9:         (parent_1, parent_2) ← select(population)
10:        child ← crossover(parent_1, parent_2)
11:        (child, c_c) ← tabuSearch(child)
12:        population ← replace(population, child)
13:        if c_c < c_best then
14:            c_best ← c_c
15:            send(server, child)                    // send better solution
16:        (s_r, p) ← receive(server)      // try to receive useful information
17:        if ¬p and s_r ≠ ∅ then
18:            population ← replace(population, s_r)
19:    return the best element of population
```

Algorithm 2.5 – Collaborative Memetic algorithm (CMA)

and Cut, for solving such problems. We then present how this algorithm is modified to be integrated to our cooperation framework.

ILP Model

Linear programs (LPs) are a mathematical model for problems where the constraints and optimization criterion are all expressed as a linear relationship. Such problems are solved efficiently using the simplex algorithm. Integer linear programs (ILPs) are a variant of linear programs where variables take their values in $\mathbb{Z}$ rather than in $\mathbb{R}$. ILPs are inherently combinatorial and thus much harder to solve than continuous linear programs.

Given a combinatorial problem for which we want to use ILP techniques, we must first model it in a linear form, compliant with the ILP paradigm.

Such models can be expressed as the following standard form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq 0, \\ \text{and} \quad & \mathbf{x} \in \mathbb{Z}^n, \end{aligned}$$

where $\mathbf{x}$ is the vector of (integer) variables, $\mathbf{b}$ and $\mathbf{c}$ are vectors, and A is a matrix with integer values. The expression $\mathbf{c}^T\mathbf{x}$ represents the optimization criterion, and the expression $A\mathbf{x} \leq \mathbf{b}$ represents the constraints of the problem. An additional requirement for such a standard form is that $\mathbf{x}$ must have only positive values. ILPs can be solved using various heuristic methods or exact algorithms. In this report, we focus on the Branch and Cut algorithm, which we describe next.

Branch & Cut Algorithm

The *Branch and Cut* algorithm (BC) [Mitchell, 2002] makes use of *continuous relaxations* and of the search for *cutting planes* within a standard *Branch and Bound* algorithm.

Relaxation A continuous relaxation of an ILP P simply consists in considering its continuous version, i.e. where variables have continuous values instead of integer values. As previously stated, this LP P' can be quickly solved using a simplex algorithm, providing an optimal, but not integral, solution $\overline{\mathbf{x}}$. If all values in $\overline{\mathbf{x}}$ are integers, then it is also a solution to P. Otherwise, as $\overline{\mathbf{x}}$ is optimal, its optimization cost is a lower bound (in the case of minimization) of the optimal solution P.

Cutting-plane From the non-integral solution $\overline{\mathbf{x}}$ to the relaxation P' of P, it is possible to deduce new linear constraints to refine P, by the means of a cutting-plane algorithm. Such constraints, also called *cuts*, must be violated by the solution $\overline{\mathbf{x}}$ (otherwise it has no effect) and verified by all feasible solutions to the integer problem P (otherwise, we might miss the integer-optimal solution).

Branch and Cut The branch and cut algorithm, detailed in Algorithm 2.6, consists in recursively splitting the search space into smaller spaces and search each of these smaller spaces using relaxation and a cutting-plane algorithm. It maintains the following structure and data:

- the set E of spaces to explore;
- the current best solution s_{best} and its cost c_{best};
- a global lower bound lb.

```
1:  function BC(P_0)
2:      (s_best, c_best) ← (∅, +∞)
3:      (s_0, c_0) ← LPrelax(P_0)
4:      E ← {(P_0, s_0, c_0)}
5:      lb ← c_0
6:      while E ≠ ∅ and c_best − lb > ε do
7:          take (P, s, c) from E
8:          (P_c, s_c, c_c) ← searchCut(P, s, c)
9:          if s_c ≠ ∅ and c_c < c_best then
10:             if s_c is integral then (s_best, c_best) ← (s_c, c_c)
11:             else
12:                 (P_1, P_2) ← branch(P_c)
13:                 (s_1, c_1) ← LPrelax(P_1)
14:                 (s_2, c_2) ← LPrelax(P_2)
15:                 E ← E ∪ {(P_1, s_1, c_1), (P_2, s_2, c_2)}
16:                 lb ← min_{∀P∈E} c_P
17:     return (s_best, c_best)
```

Algorithm 2.6 – Branch and Cut

To obtain an initial lower bound, we first solve the continuous LP relaxation of initial problem P_0, thanks to a simplex algorithm embedded in function `LPrelax` (line 3). The solution s_0 and its cost c_0 are stored with P_0 in the (initially empty) set E of active subproblems (line 4). After this initialization steps, we loop over the list of active subproblems E (line 6) until it is emptied (the search space has been exhausted) or the difference between the cost of the current best solution c_{best} and the lower bound lb is smaller than a given tolerance ϵ ($\epsilon = 0$ to obtain the optimal solution). If not, one subproblem P (line 7) is selected and removed from E. We then apply procedure `searchCut` (described afterwards and shown in Algorithm 2.7) to P (line 8), returning a new solution s_c, its cost c_c and a problem P_c enriched with new cutting planes. If P_c does not have a feasible solution or its cost c_c is greater than the current best cost, this means that a better solution cannot be found and the problem P is discarded. Otherwise, we check whether s_P is integral to replace the current best solution and best

cost by s_c and c_c (line 10). If not, problem P must be explored further to find its optimal solution, so procedure `branch` chooses a branching variable to produce two subproblems P_1 and P_2 (line 12) on which `LPrelax` is applied to tighten their formulation (lines 13 and 14), prior to their addition to the set of active subproblems (line 15). The global lower bound lb can then be updated with the minimum of the relaxed costs of the active subproblems (line 16). Eventually, the best solution and corresponding cost are returned (line 17).

The `searchCut` procedure (Algorithm 2.7) iteratively searches for cutting planes in a problem P until a fixed point is reached. It takes as input an ILP problem P and the solution s (of cost c) to its continuous relaxation. If the relaxation is infeasible or its optimal solution has a lower cost than the best solution and is integral (line 2), which is seldom the case, it is immediately returned. Otherwise, we search for useful cutting planes (line 3), if any, which are then added to the problem (line 5). In this case, the optimal solution of the continuous relaxation of the enriched problem is computed 6 and `searchCut` is called recursively on the result (line 7) until no cutting plane can be added anymore.

```
 1: function searchCut(P, s, c)
 2:     if s ≠ ∅ and c < c_best and s is not integral then
 3:         cuts ← searchCuttingPlanes(P)
 4:         if cuts ≠ ∅ then
 5:             P_c ← addCuts(P, cuts)
 6:             (s_c, c_c) ← LPrelax(P_c)
 7:             (P_c, s_c, c_c) ← searchCut(P_c, s_c, c_c)
 8:         else
 9:             (P_c, s_c, c_c) ← (P, s, c)
10:     else
11:         (P_c, s_c, c_c) ← (P, s, c)
12:     return (P_c, s_c, c_c)
```

Algorithm 2.7 – Cutting plane algorithm

Collaborative Version of Branch & Cut

As regards cooperation, the Branch and Cut algorithm is able to share solutions (if any) at each iteration after refining the problem with cutting planes. If no solution is found at a given iteration, the BC branches on the current problem, and can deduce a (possibly) better lower bound by the use of relax-

ation. This new lower bound can be shared with other solvers as well. BC can also benefit from any solution found by other algorithms, as they will enable further filtering of the set E of problems to explore. These modifications are detailed in the Collaborative Branch and Cut (CBC) Algorithm 2.8.

```
1:  (P_0, s_r, c_r, lb_r) ← register(server)          // get problem, current best solution and bounds
2:  function CBC(P_0)
3:      (s_best, c_best) ← (∅, +∞)
4:      if s_r ≠ ∅ then (s_best, c_best) ← (s_r, c_r)          // update internal data
5:      (s_0, c_0) ← LPrelax(P_0)
6:      E ← {(P_0, s_0, c_0)}
7:      lb ← max(c_0, lb_r)
8:      while E ≠ ∅ and c_best − lb > ε do
9:          take (P, s, c) from E
10:         (P_c, s_c, c_c) ← searchCut(P, s, c)
11:         if s_c ≠ ∅ and c_c < c_best then
12:             if s_c is integral then
13:                 (s_best, c_best) ← (s_c, c_c)
14:                 send(server, s_best)                    // send better solution
15:             else
16:                 (P_1, P_2) ← branch(P_c)
17:                 (s_1, c_1) ← LPrelax(P_1)
18:                 (s_2, c_2) ← LPrelax(P_2)
19:                 E ← E ∪ {(P_1, s_1, c_1), (P_2, s_2, c_2)}
20:                 lb ← min_{∀P∈E} c_P
21:                 send(server, lb)
22:         (s_r, c_r, lb_r) ← receive(server)          // receive useful information
23:         if s_r ≠ ∅ then
24:             (s_best, c_best) ← (s_r, c_r)
25:             E ← {(P, s, c) ∈ E | c < c_r}
26:         if lb_r ≠ ∅ then lb ← lb_r
27:     return (s_best, c_best)
```

Algorithm 2.8 – Collaborative Branch and Cut

As for the CMA, the CBC algorithm first registers to the server and gets the problem and the current state of the search (line 1). If any solution s_r or lower bound lb_r have been received, they are used to update the internal data (lines 4 and 7 respectively). During the resolution process, any better feasible solution, once found by CBC, is sent to the server (line 14). Then

we check for any updates in the search state from the server (line 22) and, if applicable, the best solution and cost (line 24) and lower bound (line 26) are updated. Better feasible solutions and costs are always useful for the BC algorithm because they provide efficient cutting planes. If the LP relaxation cost c_P of some active problem in E is even greater than the received better cost c_r (better upper bound), these active problems are removed from the set at line 25. As a result, the search space can be drastically reduced in some cases. Finally, any better lower bound can also be systematically sent to the server (line 21). Implementations of this CBC algorithm are described in chapters 3 and 4.

2.2.3 Constraint Programming

Constraint Programming (CP) [Rossi et al., 2006] is a versatile optimization technology based on the Constraint Satisfaction Problem (CSP) formalism, which emphasizes the satisfaction of combinatorial constraints. The problem must be defined by its decision variables and a set of constraints, i.e. arbitrary relations between variables. Then a constraint solver is used to find a feasible assignment for all decision variables while holding constraints satisfied among related variables during the search. A formal CSP is described in Definition 1, it consists of a set of variables; a finite set of possible values (its domain); and a set of constraints restricting the values that related variables can be assigned to simultaneously.

Definition 1 (Constraint Satisfaction Problem (CSP))**.** *A CSP (or* Constraint Network*) is defined by a triplet* (X, D, C)*:*

- $X = \{x_1, \ldots, x_n\}$ *is the set of unknown* variables*.*
- *Each variable* $x \in X$ *is associated with its* domain $d_x \in D$ *of possible values.*
- C *is the set of* constraints*. Each constraint* $c \in C$ *is defined over a subset of variables* $X_c \subseteq X$ *by a relation* $R_c \subset \bigtimes_{x \in X_c} d_x$ *that specifies the set of allowed tuples (combinations) for* X_c*.*

A solution *to a CSP is an assignment of all variables to a value in its domain* $\phi : X \mapsto \bigcup_{x \in X} d_x$ *s.t.* $\forall x \in X, \phi(x) \in d_x$*, and s.t. all constraints are satisfied:* $\forall c \in C, \phi(X_c) \in R_c$*.*

Note that the assignment function ϕ is overloaded to also apply to subset of variables, in which case it returns the tuple (or vector) of the corresponding subset of values: $\phi(\{x_{i_1}, \ldots, x_{i_k}\}) = (\phi(x_{i_1}), \ldots, \phi(x_{i_k}))$.

With such a scheme where the CSP formalism and constraint solver are independent, the problem modeling and its resolution can be tackled separately, thus giving the opportunity to test various search strategies. Also, most constraint solvers provide high-level constraints, leading to concise models that are easier to produce and understand. Unlike ILP, most constraints are not linear. This is a huge advantage as many real-world problems are inherently non-linear and modeling is more natural and straightforward using CP. However, CP solvers cannot benefit from the efficiency of linear relaxations and cuts from ILP solvers and thus mainly perform an exhaustive exploration of the search space, which can be too time-consuming if the filtering is not successful enough at reducing its size.

To find a feasible solution to a CSP, typical CP solvers implements a backtracking algorithm which iteratively extends a *partial* assignment (i.e. assignment of a subset of the variables) by providing a value to a yet unassigned variable. At each step, a typical backtracking algorithm at least checks the relevant constraints to avoid exploring unfeasible subspaces, as defined by Definition 2.

Definition 2 (Partial Assignment and Constraint Check)**.** *A* partial assignment $\phi_V : V \mapsto \bigcup_{x \in V} d_x$ *is an assignment defined over a subset of variables* $V \subseteq X$*. If* $V = X$*, the assignment is said to be* total*.*

A constraint check *of partial assignment* ϕ_V *is a predicate which returns true iff each constraint covered by assigned variables is satisfied:*

$$\forall c \in C,\ s.t.\ X_c \subseteq V, \quad \phi_V(X_c) \in R_c$$

otherwise, it returns false.

Whenever an assignment violates a constraint, the last value-to-variable association is removed and an untried other one is performed. A solution corresponds to a total assignment. Such a general procedure is quite inefficient to explore large instances, as it might take many steps before discovering that a given constraint is not satisfied. To improve this backtracking scheme, CP solvers mostly rely on *look-ahead* techniques which attempt to remove inconsistent values in the domains or detect inconsistent constraints before the assignment of the concerned variables, as explained in the next section.

Constraint Propagation

As previously mentioned, the set of constraints in a CSP must hold during the resolution. In order to prune the search tree and thus reduce the time needed for resolution, every time the domain of a variable is modified or,

more generally, a constraint is added in a branch of the backtracking algorithm, all constraints associated to this variable execute their consistency algorithms until a fix point is reached, i.e. no more deduction can be done. This phase, called *constraint propagation*, aims at maintaining stronger levels of *consistency* (cf. [Rossi et al., 2008]) than the basic constraint check described in Definition 2.

Ensuring local consistency properties can help reduce the search space by filtering out values in the domains of some variables for which there is no possibility to find any solution, or precociously detecting more subtle cases of inconsistency (like the pervasive AllDifferent constraint for example). This can be performed using various algorithms that can enforce different types of local consistencies, like arc consistency, bound consistency, path consistency, etc. Constraint propagation is one of the main concepts behind the Branch & Prune algorithm, which is widely used to solve CSPs and is described in the next section.

Branch & Prune Algorithm

Branch & Prune (BP) is an algorithm based on a depth-first exploration of the search space. The branching part refers to the assignment of a variable to a value taken from its current domain (or more generally any constraint added to the CSP), at each step of the search, thus creating a branching point. Pruning refers to the constraint propagation as described in the previous section, which leads to domain reductions and the pruning of the current branch whenever a domain becomes empty.

BP is detailed in Algorithm 2.9. The algorithm maintains the set V of unassigned variables and the partial assignment ϕ, which are respectively initialized to the set of variables of the CSP being solved and the empty set with the first call BP$(X, \emptyset)$, for a CSP (X, D, C) (the other parameters being considered accessible from a global state of the solver). Note that in this algorithm, we represent the mapping of a partial assignment ϕ_U by a set of couples: $\{(x, \phi(x)), \forall x \in U\}$ instead of a function.

First (line 2), if all decision variables have been assigned, it means a feasible solution was found. If not, an unassigned variable x is selected from V (line 4), and a branching process is iterated on every value α in its domain. For each such value, two branches are implicitly created (line 6): one where $x = \alpha$, in which this assignment is added to the partial assignment ϕ, and the other where $x \neq \alpha$. More generally, any constraint and its negation can be used to represent an alternative in the search tree.

This decision is propagated using a local consistency algorithm, as described in the previous section, and BP is recursively called with the new

subproblem (line 7). The `propagate` function must return `false` if at least one variable gets its domain wiped out by the constraint propagation (meaning no solution can be found in the subsequent branch). If the exploration of this branch is unsuccessful, the domains modifications due to the constraint propagation must be undone (line 10) before exploring a new branch.

```
1:  function BP(V, φ)
2:      if V = ∅ then return true
3:      else
4:          x ∈ V
5:          for α ∈ d_x do
6:              φ' ← φ ∪ {(x, α)}
7:              if propagate(x ← α) and BP(V \ {x}, φ') then
8:                  return true
9:              else
10:                 undo( )
11:         return false
```

Algorithm 2.9 – Branch & Prune.

Next section describes how we adapted the BP algorithm to our cooperation framework.

Collaborative Version of Branch & Prune

With respect to cooperation, the Branch & Prune algorithm can provide newly found solutions (if any) during the resolution. This will indicate to other algorithms where to find some feasible regions of the problem; also the cost of the solution is a useful upper bound. Conversely, BP can benefit from any optimization bounds found by other algorithms, as they will help to further prune the search tree if any inconsistency is induced. These modifications are shown in the Collaborative Branch and Prune (CBP) Algorithm 2.10.

First (line 2), as for CMA and CBC, the CBP registers to the server and gets the instance data and the current state of the search. A CSP is generated (line 3) based on the instance P_0 and upper and lower bounds c_{ϕ_r} and lb_r. If a new solution is found during research, it is sent together with its cost to the server (line 6). Otherwise, the search continues as in the standard BP. Upon a failure during the constraint propagation on the current branch, the CBP first backtracks (line 15) as in the BP, and then gets the latest search data available from the server if any (line 16). That information is used to

update the bounds of the cost (line 17) in a monotonic way (i.e. not subject to backtracking, as the standard bounding scheme of a Branch and Bound algorithm), possibly triggering a failure in the current branch if it returns false. Note that the server could be regularly queried at other control points of the BP algorithm (e.g. just before the call to the `propagate` function), but the update of the cost bounds was more convenient to integrate within the similar mechanism used by the Branch and Bound algorithm of FaCiLe, which occurs after each backtrack.

```
1:  function CBP(V, server)
2:      (P_0, c_{φ_r}, lb_r) ← register(server)
3:      (V, φ) ← generateCSP(P_0, c_{φ_r}, lb_r)
4:      function CBP_rec(V, φ)
5:          if V = ∅ then
6:              send(server, φ, c_φ)
7:              return true
8:          else
9:              x ∈ V
10:             for α ∈ d_x do
11:                 φ' ← φ ∪ {(x, α)}
12:                 if propagate(x ← α) and CBP_rec(V \ {x}, φ') then
13:                     return true
14:                 else
15:                     undo( )
16:                     (c_{φ_r}, lb_r) ← receive(server)
17:                     if not update(c_{φ_r}, lb_r) then
18:                         return false
19:             return false
20:     CBP_rec(V, ∅)
```

Algorithm 2.10 – Collaborative Branch & Prune.

In this chapter, we have proposed a generic, distributed framework for the cooperation of various optimization algorithms, and provided details about its implementation. We also presented three classical optimization algorithms together with their cooperative versions which can benefit from and provide information to each other through the framework. Next chapters describe how we used this framework and algorithms to solve optimization problems linked to air traffic management: en-route conflict avoidance (Chapter 3) and airport gate allocation (Chapter 4).

Chapter 3

Application to Conflict Resolution

Contents

3.1 Related Works 44
3.2 Model 47
3.2.1 Maneuvers and Decision Variables 47
3.2.2 Trajectory Prediction and Conflict Detection . . . 50
3.2.3 Cost of Maneuvers 57
3.2.4 Overall Mathematical Model 59
3.3 Resolution Algorithms 60
3.3.1 Memetic Algorithm 60
3.3.2 Integer Linear Programming 62
3.3.3 Cooperation Between the MA and the ILP 65
3.4 Results 65
3.4.1 Benchmark 65
3.4.2 Experimental Setup 67
3.4.3 Single Algorithms 67
3.4.4 Cooperation 68
3.4.5 Infeasible Instances 72
3.5 Conclusion and Further Work 75

One of the key challenges towards more automation in Air Traffic Control (ATC) is the resolution of en-route conflicts to avoid hazardous losses of separation between aircraft in a given airspace volume. In this chapter, we present a generic framework for *conflict resolution* that clearly separates the trajectory and conflict models from the solver technology. It is able to handle any kind of maneuver and detection models, though we propose our own realistic 3D maneuvers and conflict detection that takes into account uncertainties on the positioning of aircraft. Based on these models, realistic scenarios are built, for which potential conflicts are detected using an efficient GPU-based algorithm. The resulting instances of the conflict resolution problem are provided to the community as a public benchmark.

To efficiently solve this problem, we have used the generic framework for the cooperation of optimization algorithms described in Chapter 2. The framework benefits from the various plugged algorithms by sharing relevant information among each other, and is implemented as a distributed application for better performance. We illustrate its behavior on the conflict resolution problem with the cooperation between a Memetic Algorithm and an Integer Linear Program which consistently outperforms previous approaches by orders of magnitude. Instances with up to 60 aircraft are optimally solved within a few minutes using this framework, while each algorithm taken individually only provides sub-optimal solutions. This cooperative approach thus seems appropriate for application in a real-time context.

This chapter is organized as follows. Previous research works related to the conflict resolution problem are presented in Section 3.1. Section 3.2 describes a maneuver model allowing heading or Flight Level changes and a 3D-trajectory model that takes into account realistic uncertainties to detect all potential conflict. Section 3.3 details a Metaheuristic approach and a Integer Linear Programming method, as well as the framework for their cooperation. Section 3.4 compares both optimization techniques and their cooperation on our benchmark in order to assess the performance of these different approaches, showing the advantage of the cooperation method for large instances. We then conclude and present future research directions in Section 3.5.

The work carried out with the en-route conflict resolution problem has been published in [Allignol et al., 2017, Wang et al., 2020].

3.1 Related Works

Research on automated aircraft conflict resolution started in the 1980s. Many different models were introduced to comply with existing resolution tech-

niques. Some studies like [Erzberger et al., 1997], conducted by Air Navigation System Providers, offer realistic models but do not focus on the resolution methods. Other approaches, like [Durand et al., 1996, Granger et al., 2001] which use uncertainty models and the Base of Aircraft Data (BADA, developed and maintained by Eurocontrol [Nuic et al., 2010]), studied both the model and the resolution algorithms. However, they were completely tailored to the underlying traffic simulator (CATS [Alliot et al., 1997]), which prevents the scientific community from comparing different resolution methods.

Many mathematical models have led to specific resolution algorithms able to deal with very complex situations, but which require specific characteristics for trajectory prediction. This is the case for [Pallottino et al., 2002] which uses Mixed Integer Linear Programming (as [Vela et al., 2009, Alonso-Ayuso et al., 2011, Rey et al., 2012]). Theses models rely on constant-speed trajectories and assume that all maneuvers are executed simultaneously, which is not realistic. They cannot deal with trajectory models able to handle descending or climbing aircraft, nor with complex trajectory uncertainties. Other authors like [Alonso-Ayuso et al., 2016a,b] proposed different Mixed Integer Non-Linear models that can deal with horizontal and vertical maneuvers, taking multi-objective criteria into account, though uncertainties are not included in the trajectory prediction.

Moreover, conflict resolution is known for being highly combinatorial [Durand and Granger, 2003] and large instances can therefore be very difficult to solve, but the optimal solution (or at least a good enough one) is needed very quickly for real-time en-route conflict resolution. So assessing the relative merits of different solvers is very useful to pave the way to future automation tools. To be able to fairly compare the performance of various solvers, [Allignol et al., 2013] proposed a framework to separate the trajectory model from the resolution algorithm, as well as a Constraint Programming (CP) approach to solve the problem with 2D maneuvers only. This framework was extended to scenarios involving several Flight Level with 3D maneuvers in [Allignol et al., 2017], and a second approach using a Memetic Algorithm (MA) was proposed.

However, in these studies, optimal solutions could only be found on small instances. On larger instances, the MA was able to find good solutions without reaching the optimal, while CP was occasionally able to prove optimality or infeasibility for highly constrained instances. The strict separation between the model and the resolution made it possible for us to publish the instances of the problem for the scientific and ATM community. This offered the opportunity to test different algorithms on various problems without investing effort in the model. With such a framework, resolution times and

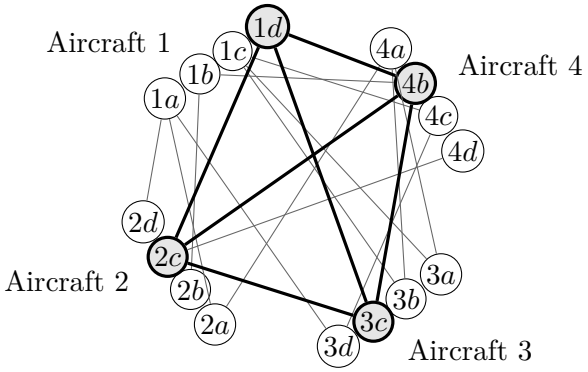


Figure 3.1 – Graph representation of a conflict resolution problem: clique $\langle 1d, 2c, 3c, 4b \rangle$ is a solution.

costs of different solvers can be fairly compared. Instances can be downloaded from the dedicated website: `clusters.recherche.enac.fr`.

In 2015, [Lehouiller et al., 2015] also proposed a general framework by modeling the problem with a graph where the vertices are the trajectories and the edges connect compatible trajectories. This is possible because the problem only involves binary constraints, i.e. constraints that specify the allowed combination of maneuvers for exactly two aircraft. The problem can thus be viewed as minimizing the cost of a maximum clique. For example in Figure 3.1, each aircraft must choose between four maneuvers $\{a, b, c, d\}$. The edges represent compatible maneuvers. There is only one maximum clique representing a solution: $\langle 1d, 2c, 3c, 4b \rangle$. Lehouiller obtains good results using this model on problems involving up to 20 aircraft with a small number of maneuver options. Then in 2017, [Lehouillier et al., 2017] proposed two decomposition algorithms to enhance the resolution. This graph model can easily be generated with our framework, but uncertainties are not taken into account. The same year, [Rey and Hijazi, 2017] proposed a new complex number formulation and convex relaxation for the centralized problem and showed that it could reduce the resolution time.

As observed from previous studies, classic algorithmic approaches of the en-route conflict resolution problem have not proved to be effective as soon as instances are of large dimension. In the last ten years, combining various algorithmic strategies, such as mathematical programming techniques and metaheuristics, has proved powerful in many domains [Raidl and Puchinger, 2008]. Problems for which pure traditional approaches were ineffective could be successfully solved by exploiting synergies between different techniques [Blum et al., 2008, 2011]. Even among metaheuristics, some are better at local search [Li et al., 2017], while others cope well with global search [Mirjalili

and Lewis, 2016]. Therefore, the hybridization of optimizers can alleviate the intrinsic drawbacks of basic algorithms [Awad et al., 2017], leading to reduced calculations, improvement of the precision of results and more stable convergence behaviors [El-Abd and Kamel, 2005, Heidari et al., 2019]. The purpose of this chapter is to show that combining different algorithms on the conflict resolution problem can improve the results obtained with pure approaches alone.

3.2 Model

In this section, we present a generic approach to the conflict resolution problem which can be based on any conflict prediction system that takes as input a discrete set of possible maneuvers for each aircraft.

We first describe a possible model of realistic 3D maneuver options compatible with current Air Traffic Control (ATC) practice. Then we present a trajectory prediction model that approximates an aircraft possible positions at each time step as a convex polyhedron, according to a set of uncertainty parameters, as described in [Allignol et al., 2017]. Possible conflicts are thereupon detected by an efficient GPU-based parallel algorithm described in [Alligier et al., 2018], which improves the execution time by orders of magnitude compared to our previous sequential approach (cf. [Allignol et al., 2017]). Eventually, the resulting *conflict matrix* is used to specify the constraints of the instance and a simple cost function is defined to favor later, shorter and least-deviating maneuvers.

3.2.1 Maneuvers and Decision Variables

In the traffic scenarios used for our benchmark, aircraft are initially leveled on consecutive Flight Levels (FL) spaced-out by 1000 ft, as sketched in Figure 3.2. On each FL, the routes of the flight plans are defined by a sequence of waypoints (specified by their coordinates in the horizontal plane).

In our trajectory model, a maneuver could be a *heading* change (cf. Figure 3.3a), a *FL* change (cf. Figure 3.3b) or a speed change (cf. Figure 3.3c). These types of maneuvers are representative of ATC practice and can be easily implemented by pilots and current Flight Management System (FMS) technologies (cf. [Granger et al., 2001]). But we do not allow to combine different types in order to keep them simple enough.

The first phase of a maneuver begins at a discrete given time t_0, when it deviates from the initial trajectory, and its second phase starts at a later given time t_1, when the aircraft returns to its initial trajectory, as depicted in

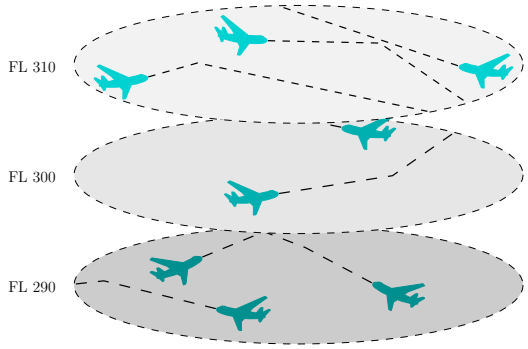


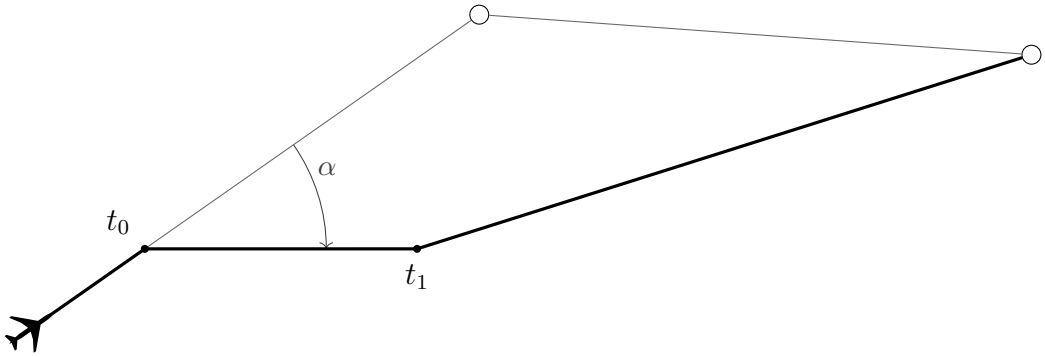
Figure 3.2 – A traffic scenario on several Flight Levels.

Figure 3.3. To recover the trajectory, described as a sequence of waypoints (depicted by large white dots in Figure 3.3a), while implementing an heading change maneuver, the first of the next waypoints that can be reached with an acceptable turning angle (i.e. $\leq 60°$) is selected; therefore, some waypoints might be skipped as illustrated by Figure 3.3a. For vertical maneuvers, the aircraft begin to climb or descend at a standard rate at time t_0 towards its assigned FL and starts to return to its original FL (also at a standard rate) at time t_1. Note that the time spent by an aircraft to climb or descend only depends on the climbing or descending rate and is not related to t_0 or t_1.

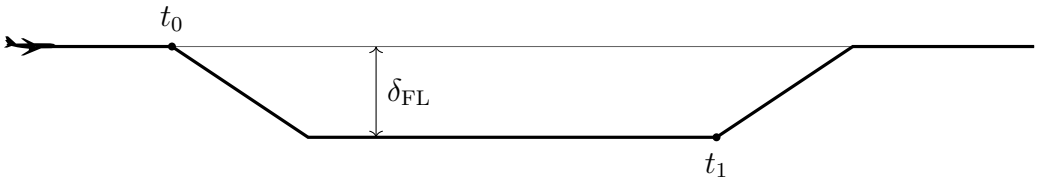
Heading changes α can take $n_\alpha = 6$ different values in our benchmark (see Section 3.4.1), i.e. 10°, 20° or 30° degrees to the left or the right of the current heading. Vertical moves δ_{FL} can take $n_{\text{FL}} = 4$ values, i.e. climb or descend 1000 ft or 2000 ft (i.e. one or two FLs) from the current level. Speed changes σ can take $n_\sigma = 2$ different values, i.e. $-6\,\%$ slowdown or $+3\,\%$ acceleration (w.r.t. the current speed), which corresponds to the speed adjustment range deemed acceptable w.r.t. pilots and air traffic controllers constraints by the ERASMUS project [Averty et al., 2007]. The number of maneuver types is thus $n_k = n_\alpha + n_{\text{FL}} + n_\sigma = 12$. We limit the number of possible maneuvers by choosing t_0 among n_0 values (typically $n_0 = 4$ in our experiments). The number of values for t_1 is also chosen among a limited set of n_1 values (typically $n_1 = 4$).

If we combine n_0 values for t_0 and n_1 for t_1 with n_α possible angles, n_{FL} vertical maneuvers or n_σ possible speed modifications, plus one maneuver for unaltered aircraft (as a null heading, level change or speed change correspond to the same trajectory, regardless of t_0 and t_1), the number of maneuvers per aircraft is:

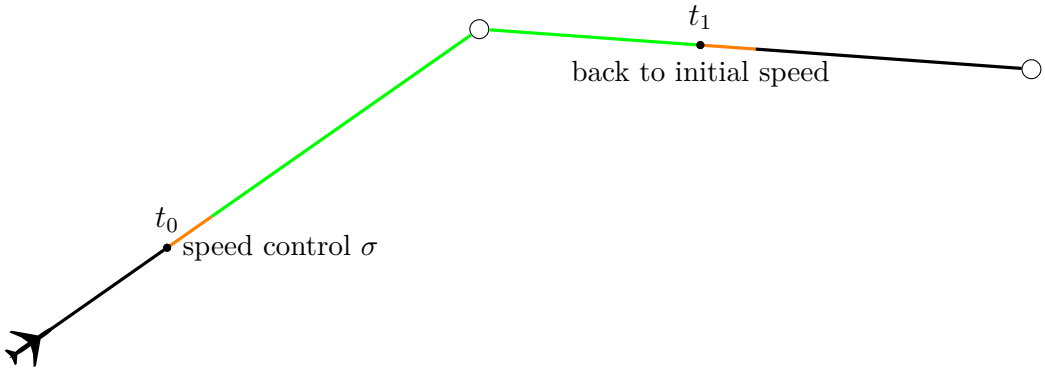
$$\begin{aligned} n_{\text{man}} &= n_0 \times n_1 \times (n_\alpha + n_{\text{FL}} + n_\sigma) + 1 \\ &= n_0 \times n_1 \times n_k + 1 \end{aligned}$$



(a) Horizontal maneuver.



(b) Vertical maneuver.



(c) Speed maneuver. The black parts of the trajectory correspond to the nominal speed, the green part corresponds to active speed control. The orange parts represent transition phases during which the aircraft speeds up or slows down.

Figure 3.3 – Maneuvers, beginning at t_0 and returning at t_1, compatible with current ATC practice. Large white dots correspond to waypoints of the initial trajectory, which is itself depicted in light gray.

Table 3.1 – Maneuver parameters.

Parameter		Size (value)		Typical values
t_0	start time	n_0	(= 4)	0, 1, 2 and 3 min
t_1	return time	n_1	(= 4)	5, 6, 7 and 8 min
α	heading change	n_α	(= 6)	-30, -20, -10, +10, +20, +30°
δ_{FL}	FL change	n_{FL}	(= 4)	−20, −10, 10 and 20 FL
σ	speed change	n_σ	(= 2)	−6 and +3 %

Table 3.1 sums up the maneuver parameters and their respective values in the benchmark presented in Section 3.4.1, which amounts to $n_{\text{man}} = 4 \times 4 \times (6+4+2)+1 = 193$ combinations. For a conflict cluster with n aircraft, the search space size is then:

$$n_{\text{man}}^n$$

therefore $193^{20} \approx 5.14 \times 10^{45}$ for a 20-aircraft instance and up to 1.36×10^{137} for 60 aircraft.

To provide a generic view of the maneuver model, we restrict the number of parameters for each aircraft i to a single decision variable m_i that aggregates variables t_0, t_1 and the heading change α, the FL change δ_{FL} or the speed change σ, thanks to a bijection from the valid 5-tuples to interval $\{1, \ldots, n_{\text{man}}\}$. We call $\mathcal{M}$ the set of decision variables of the problem:

$$\mathcal{M} = \{m_i \in \{1, \ldots, n_{\text{man}}\},\ \forall i \in \{1, \ldots, N\}\} \tag{3.1}$$

3.2.2 Trajectory Prediction and Conflict Detection

To compute possible conflicts between maneuvers within the time frame of the resolution, we first model various sources of uncertainty, then describe how our trajectory prediction take them into account to detect incompatible maneuvers and build the *conflict matrix*.

Conflict

For ATC, a conflict occurs between two aircraft if there is a simultaneous loss of horizontal and vertical separation according to some distance thresholds (called *separation norms* in the following) which depend on the airspace considered:

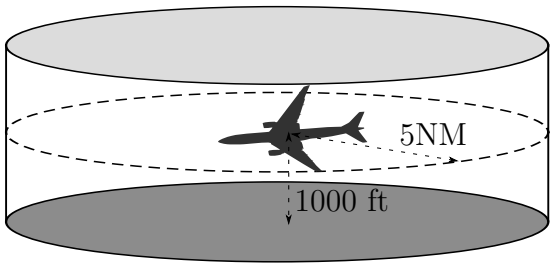


Figure 3.4 – Typical en-route protection volume around an aircraft.

Definition 3 (Conflict Between Aircraft)**.** *Aircraft i and j are in* conflict *iff* $\exists t$ *s.t.:*

$$dist_h(p_i(t), p_j(t)) \leq \text{norm}_h \quad \wedge \quad dist_v(p_i(t), p_j(t)) \leq \text{norm}_v$$

where:

- $p_k(t) = (x_k^t, y_k^t, z_k^t)$ *is the position of aircraft k at time t;*
- $dist_h(p_i(t), p_j(t)) = \sqrt{(x_i^t - x_j^t)^2 + (y_i^t - y_j^t)^2}$ *is the distance in the horizontal plane;*
- $dist_v(p_i(t), p_j(t)) = |z_i^t - z_j^t|$ *is the distance between altitudes.*

For en-route traffic, the separation norms are usually $\text{norm}_h = 5\,\text{NM}$ and $\text{norm}_v = 1000\,\text{ft}$ as illustrated by Figure 3.4 showing the *protection volume* around an aircraft. A conflict occurs whenever another intruding aircraft enters the protection volume.

Uncertainties on Trajectories

To model the inaccuracy of realistic trajectory prediction systems, we model the six following sources of uncertainties, associated to the implementation of the maneuver or to the state of the aircraft (as in Figure 3.3, large white dots represent waypoints of the aircraft flight plan, while small black dots correspond to turning points of maneuvers in Figures 3.5–3.9):

- When instructed to maneuver, a pilot can react more or less quickly. Uncertainty $\varepsilon_{t_0} \in [0, E_{t_0}]$, which represents the maximum reaction time to start a maneuver, is associated to time t_0 (see Figure 3.5).
- Uncertainty $\varepsilon_{t_1} \in [0, E_{t_1}]$, which represents the maximum reaction time for ending a maneuver, is associated to time t_1 (see Figure 3.5).

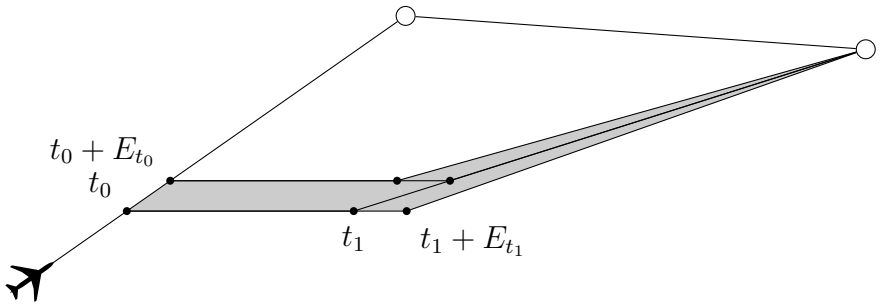


Figure 3.5 – Reaction time uncertainty model with maximal errors E_{t_0} and E_{t_1}.

- Uncertainty $\varepsilon_\alpha \in [-E_\alpha, E_\alpha]$ is also associated to the heading change angle α (see Figure 3.6).
- Horizontal speeds v_h are subject to relative error $\varepsilon_{v_h} \in [-E_{v_h}, E_{v_h}]$ (expressed as a percentage) such that future positions of aircraft are spread over a range which grows with time (see Figure 3.7).
- Climbing and descending rates v_v are also subject to relative error $\varepsilon_{v_v} \in [-E_{v_v}, E_{v_v}]$ (as a percentage) as illustrated in Figure 3.8.
- The *fly mode* f_m can be chosen among two values $f_m \in \{F_b, F_o\}$ as an aircraft can "fly by" (F_b) or "fly over" (F_o) a waypoint, depending on the pilot practice or the airline rules, and we consider both options to build the future trajectory (see Figure 3.9). More precisely, when an aircraft must turn at a waypoint, it cannot strictly fly linear segments with instant turning points. Flight Management Systems or pilots can either "fly by" or "fly over" the turning point: when the pilot anticipates the turning angle before arriving at the waypoint, she flies by the waypoint, and when the pilot turns once she has reached the waypoint and heads back to the initial trajectory after it, she flies over the waypoint. Because we do not know which choice is going to be made by the pilot, we take the so-called "fly mode" uncertainty into account in our model.

Table 3.2 sums up these uncertainties, related to the maneuver parameters at the top of the table, and to the aircraft characteristics only at the bottom. Note that there is no uncertainty on the lateral position when an aircraft is heading toward a waypoint as its FMS is able to dynamically correct the lateral error. Accordingly, no uncertainty is considered for the maneuver parameter specifying the FL change δ_{FL} as current FMS are able to precisely

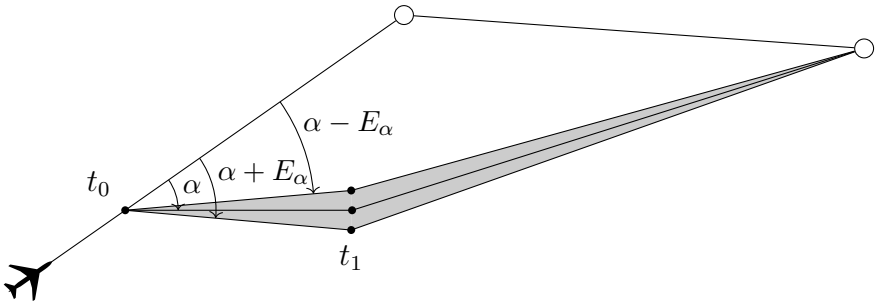


Figure 3.6 – Heading change uncertainty model with maximal error E_α.

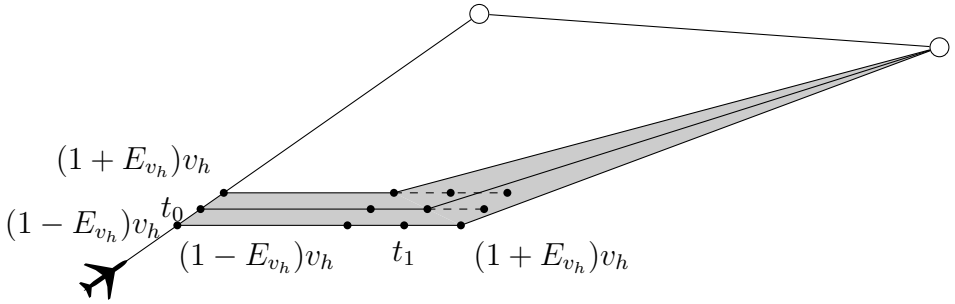


Figure 3.7 – Speed uncertainty model with maximal error E_{v_h}.

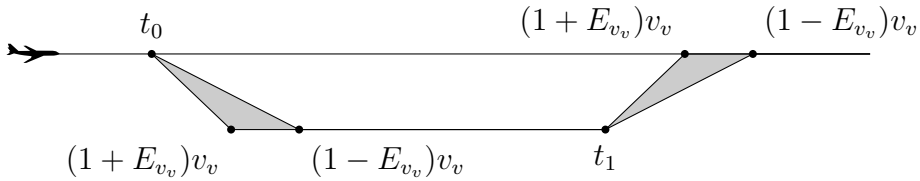


Figure 3.8 – Climb and descent uncertainty model with maximal error E_{v_v} model.

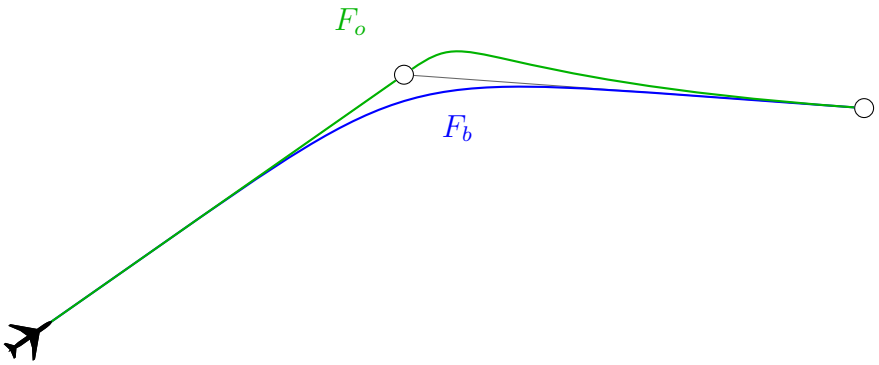


Figure 3.9 – Flight mode uncertainty model with possible modes “fly by” (F_b, in blue) or “fly over” (F_o, in green).

Table 3.2 – Uncertainties on the trajectory parameters.

Parameter		Error	Typical values
t_0	start time	$\varepsilon_{t_0} \in [0, E_{t_0}]$	10, 20 and 30 s
t_1	return time	$\varepsilon_{t_1} \in [0, E_{t_1}]$	10, 20 and 30 s
α	angle	$\varepsilon_\alpha \in [-E_\alpha, E_\alpha]$	1°; 2°; 3°
v_h	horizontal speed	$\varepsilon_{v_h} \in [-E_{v_h}, E_{v_h}]$	2, 4 and 6 %
v_v	vertical speed	$\varepsilon_{v_v} \in [-E_{v_v}, E_{v_v}]$	5, 10 and 15 %
f_m	waypoint fly mode	$f_m \in \{F_b, F_o\}$	F_b, F_o

level out at the specified FL. Uncertainty on the vertical profile is therefore taken into account by the error on the vertical speed ε_{v_v} alone.

Aircraft Position Envelope

To detect the conflicts, time can be discretized into regular steps, provided their duration τ is small enough to avoid missing even the shortest conflicts. In Section 3.4.1, we fix $\tau = 3\,\text{s}$ because two facing aircraft flying at 600 kn (maximal speed for airliners) get only 1 NM closer every 3 s, which ensures the detection of such a conflict, as the target separation distance is 5 NM (see [Barnier and Allignol, 2012] for a more in-depth discussion on this topic).

We assume that at a given time step, the position of an aircraft belongs to the set of positions allowed by any combination of the uncertainty parameters with a uniform probability distribution (but this assumption could be refined with a more realistic distribution model as described in [Erzberger et al., 1997]). We therefore build the envelopes of possible positions at each time step for all maneuvers before checking their distance to detect a possible conflict (see Section 3.2.2).

As described in Section 3.2.1, we have defined six uncertainty parameters for our trajectory prediction: ε_{t_0}, ε_{t_1}, ε_α, ε_{v_h}, ε_{v_v} and f_m. In order to take into account every possible trajectory, we test every combination of the extreme values of these parameters: $2^6 = 64$ trajectories are computed to built the geometric boundaries of a single maneuver over time. Once these extreme trajectories are built, we compute for each time step t a 3D polyhedral convex envelope to safely approximate the possible positions of an aircraft. We use the well-known Graham's algorithm [Graham, 1992] to build the corresponding smallest convex hull in the horizontal plane, and take the minimum and maximum altitudes in the vertical plane. The resulting prism is the smallest convex orthogonal cylinder that includes the set of the possible positions of

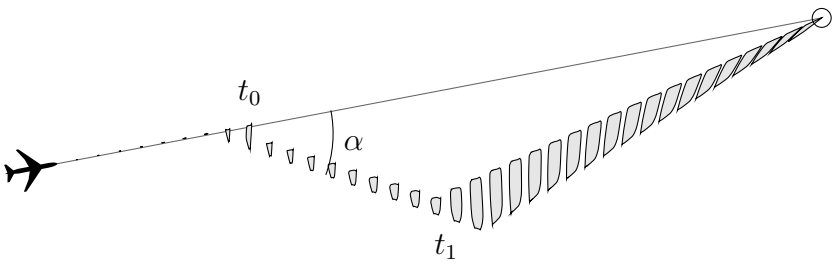


Figure 3.10 – An example of convex hulls representing a maneuver with uncertainties in the horizontal plane at each time step.

the aircraft at step t.

Figure 3.10 gives an example in the horizontal plane of a 30° heading change maneuver starting at $t_0 = 5\,\text{min}$, lasting $10\,\text{min}$ (therefore $t_1 = 15\,\text{min}$), with $E_{t_0} = E_{t_1} = 60\,\text{s}$, $E_\alpha = 5°$ and $E_{v_h} = 5\,\%$. The original route of the aircraft is represented as a gray solid line and the maneuver is depicted every minute as a polygon corresponding to the convex hull that includes the possible aircraft positions.

Conflict Detection

In the previous section, each predicted trajectory is modeled as a sequence of aircraft position envelopes. As the notion of conflict is only defined pointwise between two perfect aircraft trajectories, we extend the notion of conflict to trajectories modeled as a sequence of aircraft position envelopes.

Definition 4 (Conflict Between Predicted Trajectories)**.** *Predicted trajectories p and q are in* conflict *iff $\exists t$, $a \in p(t)$ and $b \in q(t)$ s.t.:*

$$\text{dist}_h(a,b) \leq \text{norm}_h \quad \wedge \quad \text{dist}_v(a,b) \leq \text{norm}_v$$

where $p(t)$ and $q(t)$ are aircraft position envelopes at time t, and a and b are 3D positions.

We assume that the actual future trajectory will always be inside the aircraft position envelopes of the predicted trajectory. Thus, with Definition 4, if the predicted trajectories are not in conflict then this property will hold for the actual future trajectories as well.

Using this definition, we can compute all the conflicts between all the predicted trajectories considered for the resolution and store them in a 4D boolean matrix. We combine the maneuver parameters t_0, t_1, α, δ_{FL} and β thanks to a bijection from the valid 5-tuples to interval $\{1, \ldots, n_{\text{man}}\}$ in

order to reduce them to a single number and provide a generic access to the conflict matrix. Then, for each pair of aircraft (i, j), with $i < j$ as the conflict relation is symmetric, and each pair of maneuver options (k, l), where k is a maneuver option for aircraft i and l for aircraft j, we test if maneuvers k and l are in conflict to set the coefficients $C_{i,j,k,l}$ of the conflict matrix:

Definition 5 (Conflict Matrix)**.** *The coefficients of the symmetric conflict matrix of dimensions $n \times n \times n_{\text{man}} \times n_{\text{man}}$ are defined by:*

$$C_{i,j,k,l} = \begin{cases} 1 & \textit{if maneuver } k \textit{ and } l \textit{ conflicts according to Definition 4} \\ 0 & \textit{otherwise} \end{cases}$$

$\forall i \in \{1, \ldots, N\}, \forall j \in \{i+1, \ldots, N\}, \forall k \in \{1, \ldots, n_{\text{man}}\}, \forall l \in \{1, \ldots, n_{\text{man}}\}.$

The calculation of such a conflict matrix requires to compare $n_{\text{man}}^2 \frac{n(n-1)}{2}$ pairs of predicted trajectories. In the end, it leads to compare $n_{\text{man}}^2 \frac{n(n-1)}{2} T$ pairs of aircraft position envelopes where T is the number of time steps and therefore of aircraft position envelopes per trajectory. This number can be huge even with a limited number of aircraft. For the smallest scenario in this paper with $n = 15$ aircraft, $n_{\text{man}} = 193$ possible trajectories and $T = 150$ time steps, we have to check the separation for 586 672 750 envelope pairs. If we were to integrate our conflict solver in a traffic simulator or operational system, we would have to solve iteratively a sequence of conflict resolution problems over a Rolling Horizon (RH) with the fastest possible update rate allowed by the running time of the resolution process (see [Granger et al., 2001]). Therefore, the conflict matrices must also be built as fast as possible.

In order to avoid numerous time-consuming distance computations, we can use simple bounding volumes with cheap intersection checks. As depicted by Figure 3.11, for each aircraft position envelope we can compute an Axis-Aligned Bounding Box (AABB) increased by half the separation norm (cf. Definition 3). The intersection test between AABBs is very fast as it requires six floating-point number comparisons at most. If the AABBs of two aircraft position envelopes do not intersect then the envelopes cannot conflict. Otherwise, we use the ISA-GJK algorithm [van den Bergen, 1999], a more time-consuming test, to check if the envelopes intersect. It is a variant of the GJK algorithm [Gilbert et al., 1988] which is widely used in robotics and computational geometry to determine the distance between two convex shapes. This algorithm runs in linear time w.r.t. the number of vertices of the aircraft position envelopes.

To reduce even more the running time of our solver, the computation of the conflicts can be parallelized. A parallel implementation on Graphics Processing Unit (GPU) described in [Alligier et al., 2018] using the AABBs

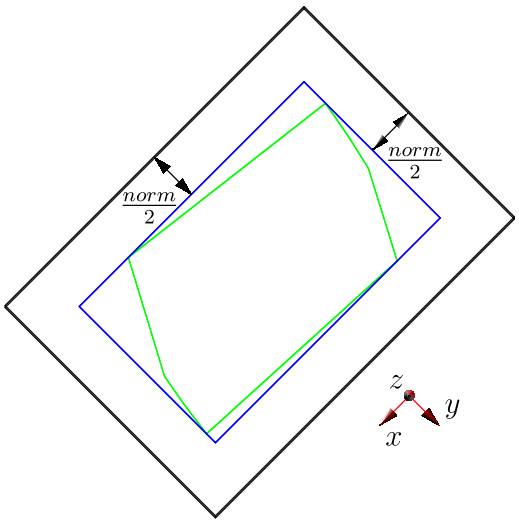


Figure 3.11 – Axis-Aligned Bounding Box (in black) of an aircraft position envelope (in green).

and ISA-GJK has been successfully tested to compute conflict matrices with a decrease of two orders of magnitude for the total running time of the detection phase. For instance, the conflict matrix of the largest scenario available in our online benchmark (see Section 3.1) with $n = 100$, $n_{\text{man}} = 193$ and $T = 150$ is computed in 1 s only. On average, the trajectory pairs are processed at a rate of 117 000 pairs per millisecond.

3.2.3 Cost of Maneuvers

To discriminate among the solutions to feasible instances, we build an arbitrary cost function for the maneuvers so as to ensure the following properties which characterize efficient solutions from an operational point of view:

1. Any maneuver is more costly than no maneuver.
2. Maneuvers should start as late as possible: because uncertainties are reduced over time for successive resolution problems in a RH solver (cf. Section 3.2.2), the cost of a delayed maneuver could be reduced when the problem is updated.
3. Maneuvers should be as short as possible.
4. The angle should be as small as possible.
5. FL change should be as small as possible.

6. A $-6\,\%$ slowdown or $+3\,\%$ acceleration is here equivalent to a 10° heading change for the sake of simplicity.

7. A 1000 ft vertical maneuver is deemed equivalent to a 20° heading change, and a 2000 ft one to 30°.

To compute the cost of a maneuver, values of t_0 are enumerated by an index k_0 varying in $\{1, \ldots, n_0\}$, values of t_1 by index k_1 in $\{1, \ldots, n_1\}$ and angles α, of value 10, 20 or 30° right or left, are respectively indexed by k_α in $\{1, \ldots, \frac{n_\alpha}{2}\}$. Speed changes σ are enumerated by an index k_σ in $\{1, \ldots, \frac{n_\sigma}{2}\}$. Similarly, FLs are indexed by k_{FL} in $\{1, \ldots, \frac{n_{\mathrm{FL}}}{2}\}$. For our benchmark, the cost of a maneuver is then defined as follows:

Definition 6 (Cost of a Single Maneuver)**.** *The cost $c(m_i)$ of a single maneuver $m_i \in \{1, \ldots, n_{\mathrm{man}}\}$ for aircraft i is:*

$$c(m_i) = \begin{cases} (n_0 - k_0)^2 + k_1^2 + k_\alpha^2 & \text{if } \alpha \neq 0 \\ (n_0 - k_0)^2 + k_1^2 + (1 + k_{\mathrm{FL}})^2 & \text{if } \delta_{\mathrm{FL}} \neq 0 \\ (n_0 - k_0)^2 + k_1^2 + k_\sigma^2 & \text{if } \sigma \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where k_0, k_1, k_α, k_{FL} and k_σ are the indices corresponding to the values of t_0, t_1, α, δ_{FL} and σ for maneuver m_i.

As this work is based on a generic framework that separates the solver method from the problem itself, we chose to instantiate our benchmark with the most simple and intuitive cost function satisfying the previously mentioned list of requirements, easy to understand and reproduce, so as to encourage the comparison of different solvers by interested readers, which might be unfamiliar with all the quirks of operational ATC costs. In a real environment, it should be modified to comply with aircraft performance models on one side and controllers' preferences on the other side. We could easily replace this cost by a more accurate one, taking into account the fuel consumption, or pilots and air traffic controllers preferences. This would however not change the model but only the values given to each maneuver of each aircraft. The same resolution methods would apply but provide different results.

Given an instance with n aircraft, we define the cost of a solution as the sum of the costs of the maneuvers for all aircraft:

Definition 7 (Cost of Conflict Resolution)**.** *The cost of a solution to a conflict resolution problem with n aircraft is:*

$$\mathrm{cost}(M) = \sum_{i=1}^{n} c(m_i)$$

3.2.4 Overall Mathematical Model

Eventually, we model conflict resolution as the following combinatorial optimization problem which summarizes the preceding sections.

Decision variables

For a problem with n aircraft, the set of decision variables is:

$$\mathcal{M} = \{m_i \in \{1, \ldots, n_{\text{man}}\},\ \forall i \in \{1, \ldots, n\}\}$$

where the maneuver of aircraft i is represented by decision variable m_i and all the maneuver options associated with allowed tuples $\langle t_0, t_1, \alpha, \delta_{\text{FL}}, \sigma \rangle$ are numbered from 1 to n_{man} as described in Section 3.2.1. Hence, the size of the search space is $n^{n_{\text{man}}}$. Note that different sets of possible maneuvers could also be specified for each aircraft without loss of generality.

Constraints

According to the 4D conflict matrix C defined in Section 3.2.2, for each element $C_{i,j,k,l} = 1$, maneuvers k of aircraft i and l of aircraft j cannot be chosen at the same time. The constraints of our problem are therefore defined by:

$$m_i \neq k \vee m_j \neq l,$$
$$\forall i \in \{1, \ldots, n\}, \forall j \in \{i+1, \ldots, n\}, \forall k, l \in \{1, \ldots, n_{\text{man}}\} \text{ s.t. } C_{i,j,k,l} = 1$$

Cost

The cost of an optimal solution to a conflict resolution problem is equal to:

$$\min_{\forall i \in \{1,\ldots,n\}, m_i \in \{1,\ldots,n_{\text{man}}\}} \sum_{i=1}^{n} c(m_i)$$

with

$$c(m_i) = \begin{cases} (n_0 - k_0)^2 + k_1^2 + k_\alpha^2 & \text{if } \alpha \neq 0 \\ (n_0 - k_0)^2 + k_1^2 + (1 + k_{\text{FL}})^2 & \text{if } \delta_{\text{FL}} \neq 0 \\ (n_0 - k_0)^2 + k_1^2 + k_\sigma^2 & \text{if } \sigma \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where maneuver m_i corresponds to allowed tuple $\langle t_0, t_1, \alpha, \delta_{\text{FL}}, \sigma \rangle$ and $k_\star \in \{1, \ldots, n_\star\}$ indexes the possible values associated with $\star \in \{0, 1, \alpha, \text{FL}, \sigma\}$ (a generic parameter subscript) by increasing absolute value (as described in Section 3.2.3), such that later, shorter, and least-deviating maneuvers are favored. Note that any cost function, possibly customized for each aircraft, could also be used in our model.

3.3 Resolution Algorithms

To solve en-route conflict resolution problem, we first use two methods, Memetic Algorithm and Integer Linear Programming, mentioned in Sections 2.2.1 and 2.2.2, then the cooperation of both algorithms to improve the performances.

The first one, a *Memetic Algorithm* (MA), is an extension of the traditional Genetic Algorithm (GA). It uses a local search technique to reduce the likelihood of a premature convergence. We can clearly see in Section 3.4, we show that for small instances, a MA is capable to find optimal solutions in a very limited amount of time. However, for larger instances, a MA may remain stuck in local optima and fail to discover an optimal solution. Moreover, metaheuristics are not exact algorithms, hence they can neither prove the optimality of a solution nor the infeasibility of an instance.

The second one, *Integer Linear Programming* (ILP), is based on the *Branch and Cut* algorithm, as described in Section 2.2.2 and able to quickly prove optimal solutions (or infeasibility) for small instances. Though for large instances (more than 50 aircraft), the resolution is too time-consuming for a real-time system.

To benefit from the advantages of both combinatorial solvers and find better solutions to large instances, our third approach consists in their *cooperation*. In the following sections, we first describe MA and ILP models for en-route conflict resolution, then provide an analysis about their cooperation.

3.3.1 Memetic Algorithm

The general principle of Memetic Algorithm (MA) has been presented in Section 2.2.1. In this section, we present how to instantiate the objective function and operators of an MA to solve the conflict resolution problem.

Objective Function

The objective function of our MA represents the function to minimize. Here, we first focus on finding a conflict-free set of maneuvers, with the smallest cost as a secondary objective. Therefore, we define the objective function as the linear combination of two terms, the number of remaining conflicts and the cost of a solution (cf. Definition 7):

$$f(s) = P \times \sum_{\forall m_i, m_j \in M \text{ s.t. } i<j} C_{i,j,m_i,m_j} + \text{cost}(s)$$

where P is a big (enough) integer to guarantee that the cost of a solution is always higher than the one of another solution if it has more conflicts.

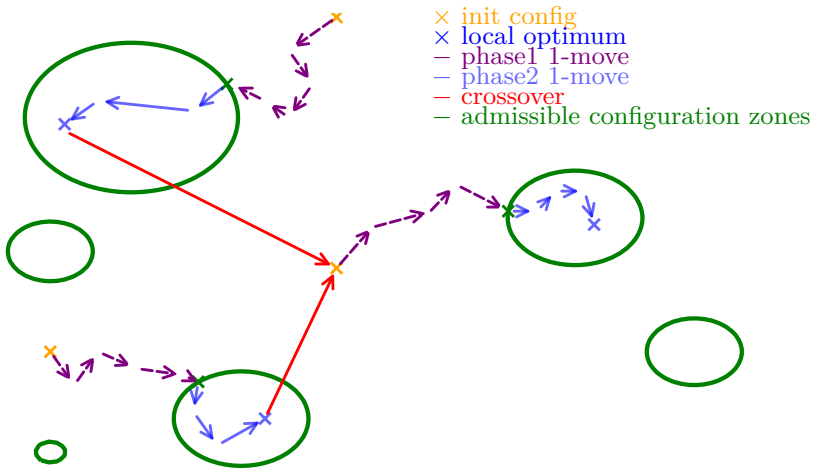


Figure 3.12 – Typical behaviour of the Memetic Algorithm in the landscape of solutions. The inside of green circles corresponds to a subset of admissible solutions.

Tabu Search

Tabu Search (TS), as mentioned in Section 2.2.1, is a local search algorithm that is used as a component of an MA to repair and improve a candidate solution. For the conflict resolution problem, the tabu list simply represents the list of forbidden maneuvers for each aircraft during a given number of iterations and the neighborhood used in line 5 of Algorithm 2.4 consists in the modification of the maneuver assigned to one of the aircraft (i.e. a "1-move" neighborhood, cf. Definition 8).

Definition 8 ("1-move" neighborhood)**.** *For a given candidate solution* $s = (m_1, \ldots, m_n)$*, we define the "1-move" neighborhood of* s*, noted as* $\mathcal{N}_s$*, as the set of solutions that differ from* s *on only one variable, and have a cost at most equal to the cost of* s*:*

$$\mathcal{N}_s = \{s' = (m'_1, \ldots, m'_n) \mid \exists!\, i \in \{1, \ldots, n\}, m'_i \neq m_i \wedge \text{cost}(s') \leqslant \text{cost}(s)\}$$

With the objective function previously described, there are two different successive phases in our TS process:

Phase 1 When the candidate solution s still involves some remaining conflicts, therefore $\sum_{\forall m_i, m_j \in M \text{ s.t. } i<j} C_{i,j,m_i,m_j} > 0$, the TS only minimizes the

number of remaining conflicts. The purple track in Figure 3.12 represents this first phase of our TS inside the solutions landscape.

Phase 2 When the candidate solution s is a conflict-free set of maneuvers, therefore $\sum_{\forall m_i,m_j \in M \text{ s.t. } i<j} C_{i,j,m_i,m_j} = 0$, then the TS minimizes the cost of the solution cost(M) accepting only neighborhood in each iteration without introducing new conflicts. Actually, if a "1-move" neighborhood, i.e. modification of one aircraft maneuver, creates conflict with other aircraft, this neighborhood will be abandoned directly. Termination criterion in line 3 of Algorithm 2.4 is when the cost has no amelioration within a certain iteration. The blue track in Figure 3.12 represents this second phase. Notice that during this phase, the candidate solution s cannot step outside its admissible configuration area (represented by green circles in Figure 3.12). Indeed, the admissible configuration set is not a connected space if we consider the 1-moves neighborhood.

If only the TS was used, each of its run would find a solution within a given admissible configuration area. As the number of admissible configuration areas might be exponential in the problem size, even a multi-start strategy on top of the TS would be inefficient. However, the crossover operator in MA is able to help the search to escape the admissible configuration areas while maintaining relatively structured candidate solutions.

Crossover

We have used a standard crossover operator, *uniform crossover*, that generates a single new candidate solution from two elements of the population. For each aircraft, this crossover operator randomly selects one of the two maneuvers of the parents and assigns it to their child. In Figure 3.12, the crossover of two admissible solutions, presented in red, will create a conflict solution. However, with applying afterwards TS, a new admissible configuration area can be explored, which prevent the research from being stuck in local minima. Other crossover operators, such as *one-point crossover* or *k-point crossover*, have been tested on our benchmark but with no better results.

3.3.2 Integer Linear Programming

State of the art ILP solvers like Gurobi [Gurobi Optimization, 2018], which was used to obtain the results presented in Section 3.4, significantly improve the efficiency of the basic Branch and Cut algorithm (described in Algorithm 2.6) with preliminary transformation techniques to reduce the size

of combinatorial problems, as well as heuristics to obtain better objective bounds during the search. These sophisticated refinements fall beyond the scope of our study and inquisitive readers may refer to [Achterberg et al., 2014] to obtain more information.

In the following sections, we first present a basic ILP model for conflict resolution, then we show how to obtain an equivalent but much more compact and efficient "aggregated" model.

Basic Model

In this section, we describe a straightforward ILP model of the en-route conflict resolution problem with binary assignment variables. First, the decision variables are defined by:

$$\forall i \in \{1,\dots,n\}, \forall k \in \{1,\dots,n_{\text{man}}\},$$
$$x_{i,k} = \begin{cases} 1 & \text{if maneuver } k \text{ is assigned to flight } i \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

As each flight must choose exactly one maneuver, the following constraint must be added for each flight:

$$\sum_{k=1}^{n_{\text{man}}} x_{i,k} = 1, \quad \forall i \in \{1,\dots,n\} \tag{3.3}$$

Moreover, conflicting trajectories cannot be chosen simultaneously:

$$x_{i,k} + x_{j,l} \leq 1,$$
$$\forall i \in \{1,\dots,n\}, \forall j \in \{i+1,\dots,n\}, \forall k,l \in \{1,\dots,n_{\text{man}}\}$$
$$\text{s.t. } C_{i,j,k,l} = 1 \tag{3.4}$$

As the cost of a maneuver does not depend on the aircraft in our model, the objective function can be expressed as the sum of the products of the cost $c(k)$ of each maneuver k by the sum of the corresponding binary decision variables $x_{i,k}$:

$$\min \sum_{k=1}^{n_{\text{man}}} c(k) \times \sum_{i=1}^{n} x_{i,k} \tag{3.5}$$

Note that specific maneuver costs $c(i,k)$ for each aircraft could easily be taken into account by generalizing Equation (3.5) with: $\sum_{i=1}^{n} \sum_{k=1}^{n_{\text{man}}} c(i,k) x_{i,k}$.

In this basic model, the conflict constraints of Equation (3.4) are straightforward but the resulting number of equations is in $O(n^2 \times n_{\text{man}}^2)$, which can be huge for large numbers of aircraft and maneuvers. For example, with an instance with 50 aircraft and 193 maneuvers, there will be more than 93×10^6 constraints.

Aggregated Model

To avoid the large number of constraints (quadratic with n and n_{man}) of the previous basic model, it is possible to aggregate, for a given trajectory k of a given aircraft i, all conflicting trajectories belonging to aircraft with indices $j > i$ into a single constraint:

$$\sum_{j=i+1}^{n} \sum_{l=1}^{n_{\text{man}}} C_{i,j,k,l} x_{j,l} \leq n_{i,k}(1 - x_{i,k}), \quad \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, n_{\text{man}}\} \tag{3.6}$$

with $n_{i,k} = |\{j \in \{i+1, \ldots, n\} \text{ s.t. } \sum_{l=1}^{n_{\text{man}}} C_{i,j,k,l} \geqslant 1\}|$, the number of aircraft (with indices greater than i) that have at least one maneuver in conflict with maneuver k of aircraft i.

If trajectory k of aircraft i is chosen, i.e. $x_{i,k} = 1$, the right-hand side of Equation (3.6) becomes 0, so all the $x_{j,l}$ conflicting with it (s.t. $C_{i,j,k,l} = 1$) on the left-hand side must also be assigned 0, as all the corresponding trajectories conflict with trajectory k of aircraft i. Otherwise, $x_{i,k} = 0$ and the constraint is relaxed such that any aircraft j can choose a trajectory conflicting with trajectory i of aircraft k. Conversely, if any conflicting $x_{j,l}$ of the left-hand side is assigned 1, then $x_{i,k}$ must be assigned 0 as trajectory k of aircraft i can no longer be chosen.

Combined with Equation (3.3) and objective 3.5, this aggregated model is equivalent to the basic model described in the previous section, but the number of constraints is reduced by orders of magnitude as there are only $O(n \times n_{\text{man}})$ constraints instead of $O(n^2 \times n_{\text{man}}^2)$. If we still take 50 aircraft and 193 maneuvers into account, there will be only 9,650 constraints (omitting the 50 constraints of Equation (3.3)). As this model consistently outperformed the basic one by orders of magnitude in our experiments, the results presented in Section 3.4 were all obtained using the aggregated model.

Contrarily to metaheuristics, BC provides a complete algorithm able to prove the optimality of a solution or the infeasibility of an instance. These properties also allow us to assess the efficiency of the MA, which succeeds in consistently finding the optimal solution for instances up to 40 aircraft, with comparable computation times (as described in Section 3.4). However, the amount of time of our ILP solver for finding the best solution becomes prohibitive for larger instances, as the BC algorithm has an exponential worst-case complexity w.r.t. the number of aircraft.

3.3.3 Cooperation Between the MA and the ILP

We presented a metaheuristic (Memetic Algorithm) in Section 3.3.1 and an exact ILP algorithm in Section 3.3.2, two classic techniques to solve the en-route conflict resolution problem. However, both approaches used separately were not efficient enough to solve our largest instances, therefore we use the cooperation approach described in Chapter 2 to run CMA (the collaborative version of MA presented in Algorithm 2.5) and CBC (its ILP counterpart described in Algorithm 2.8) simultaneously while exchanging information thanks to the distributed system described in Section 2.1.

Note that, the problem to solve is modeled differently in the various solvers: e.g. the n decision variables of the MA model directly represent maneuvers, whereas the $n \times n_{\text{man}}$ ILP ones correspond to possible assignments. However, they must comply to the common interface proposed by the server to exchange information such as the upper and lower bounds or feasible solutions. As a result, before sending feasible ILP solutions, we first translate its binary assignment variables to a more standard solution representing directly the maneuver of each aircraft.

3.4 Results

The methods described in the previous section have been implemented and thoroughly tested on a set of realistic en-route traffic instances. This section details the construction of these instances, which have been made publicly available, and presents the results obtained for conflict resolution.

3.4.1 Benchmark

To assess the performance of our resolution methods, we tested them on custom traffic scenarios within a fixed airspace volume. Figure 3.13 describes the construction of the scenarios which aims at mimicking typical converging traffic situations within en-route control sectors: aircraft are first evenly distributed on a 100 NM-radius circle, then their initial positions (i.e. the points marked O_k) are randomly disturbed within a 20 NM-wide square to avoid perfectly symmetrical instances (which may oversimplify the problem). The route for each aircraft, i.e. its initial heading, is also randomly chosen in a $\pm 60°$ interval around the direction that would lead the aircraft towards the center of the circle.

In the vertical dimension, aircraft are also randomly (and evenly) dispatched among five FLs (from FL280 to FL320) and initially leveled, which is generally the case in en-route sectors. Finally, the nominal speed for each

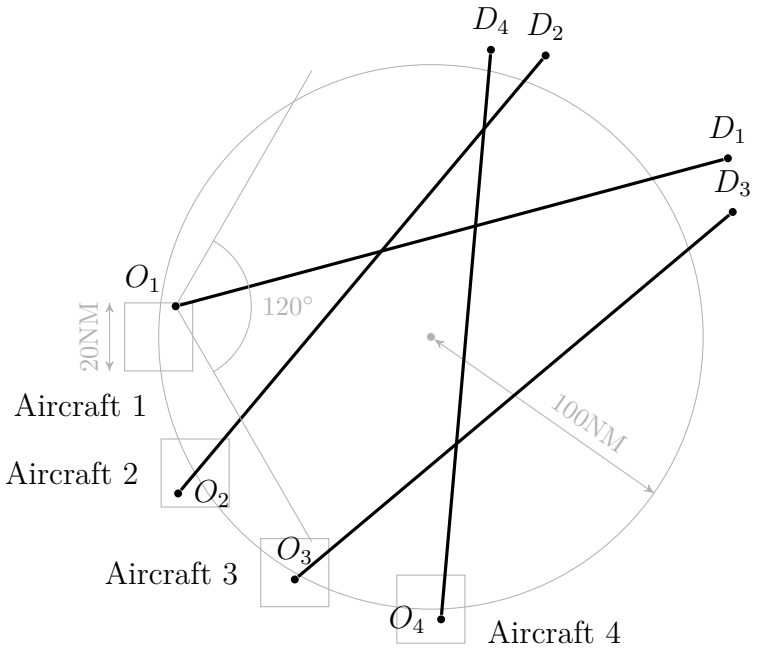


Figure 3.13 – Geometry of conflict scenario generation.

aircraft is randomly chosen in a $\pm 20\,\%$ interval around 480 kn, a typical speed for airliners. The nominal vertical speed for maneuvers is set at $600\,\mathrm{ft\,min^{-1}}$ for all aircraft. The density of scenarios is controlled by the number of aircraft in the airspace volume. For the experiments reported in the current study, this number varies from 20 to 60.

Based on this initial state, all possible trajectories are computed for every aircraft, using the maneuver parameters previously described in Table 3.1, then conflicts are detected to compute the conflict matrix (see Definition 5), with a medium uncertainty level corresponding to $E_{t_0} = 20\,\mathrm{s}$, $E_{t_1} = 20\,\mathrm{s}$, $E_\alpha = 2°$, $E_{v_h} = 4\,\%$ and $E_{v_v} = 10\,\%$ (cf. Table 3.2), using the method described in Section 3.2.2. As each scenario is randomly generated, we produced 10 different instances for each density to avoid the bias of some instances being particularly easy or difficult to solve. All the generated instances are available to download at `clusters.recherche.enac.fr`.

Real upper airspace sectors are generally smaller than the airspace volume we consider (less than 70 NM wide in France), but can be merged together when the traffic is not dense. Vertically, five FLs are reasonable as most current aircraft tend to fly at the same altitudes (which optimize their fuel consumption). When they are levelled they end up on a few Flight Levels. With the tools currently available, an air traffic controller can hardly handle

more than 30 aircraft at a time in a 70 NM sector. Also, aircraft flying in the upper airspace prefer flying a direct routes at optimal FL, while the similarity of current airliners performances tend to concentrate the demand on a small set of FLs. Thus, automatic solvers should target large airspace and may have to deal with high demand on a small amount of FLs, hence the design of our scenarios.

3.4.2 Experimental Setup

We report in this section the performances of the various techniques described in Section 3.3 to solve the instances of the benchmark described in Section 3.4.1. All experiments were carried out on a standard workstation consisting of a 3.4 GHz Intel® Xeon® octo-core processor with 16 GB of memory running Debian GNU/Linux 9.4. We used the Gurobi 8.1 Commercial Optimizer [Gurobi Optimization, 2018] for the ILP model, as well as the ZeroMQ 4.x messaging library [Akgul, 2013] to implement our cooperation framework.

As the MA is a stochastic algorithm using a pseudo-random number generator, 20 runs with different seeds were attempted for each instance to benefit from the diversification of the algorithm, and the best one is reported. All tests were done with a population of 50 individuals with 1000 iterations for the Tabu Search phase.

In the following sections, we first show that the Memetic Algorithm and the ILP solver both easily reach optimal solution to small instances of the problem. We discuss their strengths and weaknesses on larger instances, then detail how their cooperation described in Section 3.3.3 largely outperforms any single algorithm and scales well with larger and more complex instances in a limited amount of time (300 s).

3.4.3 Single Algorithms

As mentioned in Section 3.4.1, the limited number of FLs and high density of the generated instances make the conflict resolution quite challenging. Moreover, whereas ATC usually tries to find a feasible maneuver for aircraft involved in a conflict, our approach also focuses on the cost of the solution in terms of fuel consumption, aiming at an optimum of the cost function described in Section 3.2.

For small 3D instances up to 40 aircraft, both the MA and the ILP obtain an optimal solution, and the optimality can systematically be proved by Gurobi in very limited time. Figure 3.14 shows the mean time needed to compute the optimal solution for both MA and ILP for all instances ranging

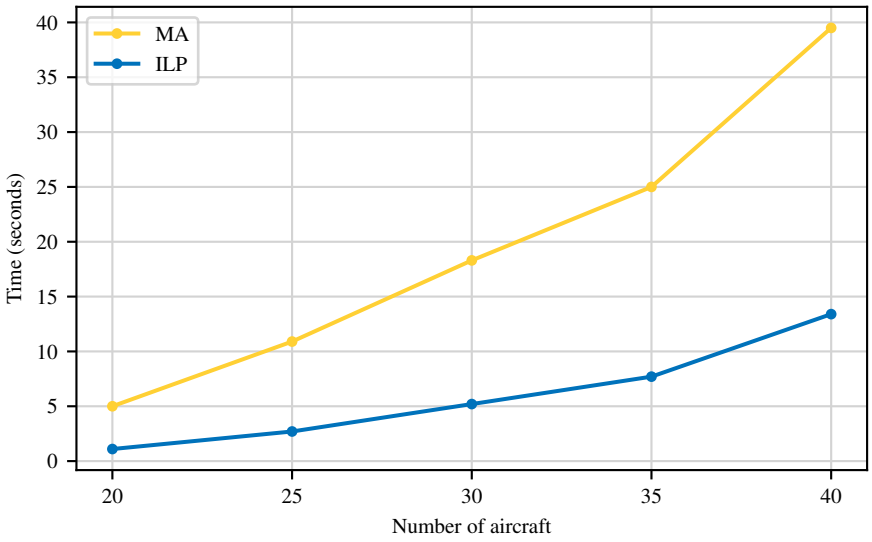


Figure 3.14 – Comparison of computation times to find an optimal solution for small instances with the MA and ILP w.r.t. the number of aircraft.

from 20 to 40 aircraft. Though the MA is efficient enough to reach an optimal solution under 40 s, the refinements of our new aggregated ILP model enable Gurobi to largely outperform the metaheuristic, even if the time spent to prove the optimality is included.

For larger instances however, the problem is more challenging, and finding the optimal solution might be out of reach in a reasonable amount of time. In order to stay within the limits of a potential real-time application, we decided to restrict the computation time to 300 s. Figure 3.15 shows the percentages of large instances (ranging from 40 to 60 aircraft) for which an optimal solution was found. This ratio quickly decreases when the number of aircraft (and thus their density) increases, with the ILP still being able to optimally solve about 20 % of our largest instances.

Next section shows that the cooperation between MA and ILP significantly increases the success rate of finding an optimum, and thus enhances the cost of the resulting solution.

3.4.4 Cooperation

We present in this section the results obtained with the cooperative version of the MA (cf. Algorithm 2.5) and the Branch and Cut (cf. Algorithm 2.8) integrated in the distributed framework described in Section 3.3.3. The behavior of the resulting solver exhibits a much better behavior when the density of

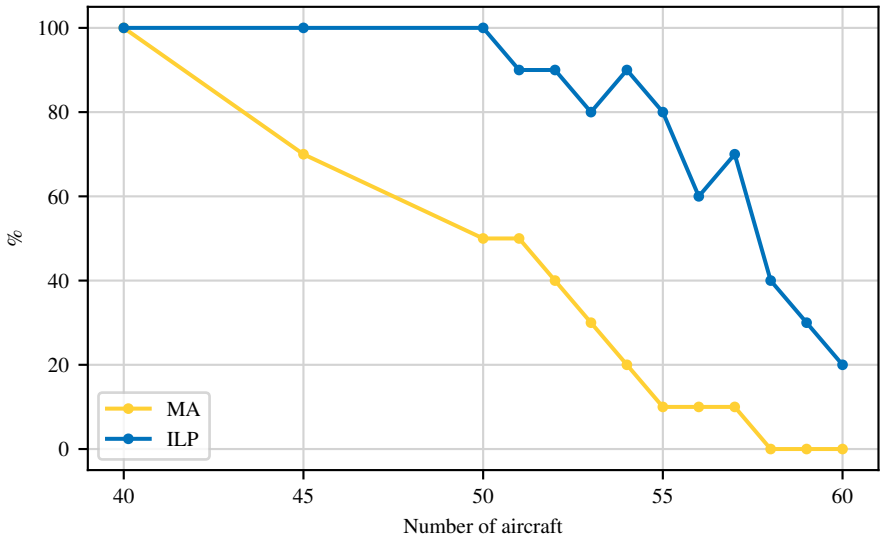


Figure 3.15 – Percentage of success for finding an optimal solution within a 300 s time limit with the MA and ILP w.r.t. the number of aircraft.

instances increases. It is able to give optimal solutions for almost all feasible instances within the 300 s time limit, while MA or ILP alone could not reach the optimum, as depicted in Figure 3.16.

In Figure 3.17, we compare the cost (averaged over 10 runs for each density) of the best solutions found by the MA, ILP and their cooperation, with respect to the optimal value. Here, 0 % represents the optimum cost, while a larger value represents the cost of a non-optimal solution. As expected from our previous observations, the cooperation systematically reaches an optimal solution. For even larger instances (with 70 to 100 aircraft), MA or ILP alone can be much farther from the optimal value: up to 3.5 % on average, and up to 10 % on some particular instances.

Figure 3.18 shows the evolution of the cost of the best solution during the search for one of the most difficult instances in our set, involving 59 aircraft. First, we observe that, at the end of the 300 s limit, the cost of the solutions found by MA and ILP are similar, but are about 10 % higher than the cost of the cooperation solution. Second, we can see from the graph that the cooperation proved the optimality of its solution, as the process stopped at about 280 s. This optimal solution was found after about 90 s. In the meantime, MA and ILP seemed to be stuck, probably on a locally optimal solution, after 100 s to 150 s. In the first part of the search, the cooperation follows exactly the same convergence profile than the MA, while the ILP algorithm does not provide any solution. This is due to the fact that the

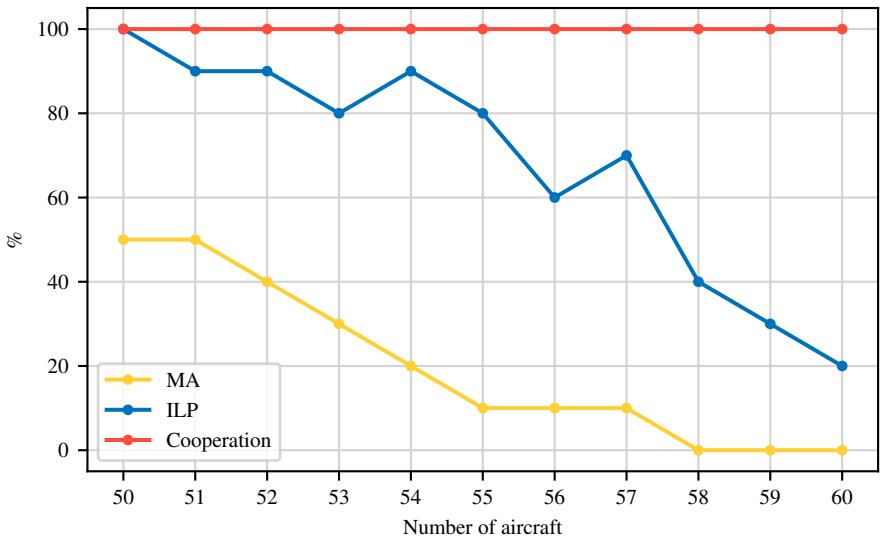


Figure 3.16 – Percentage of success for finding an optimal solution with a 300 s time limit with MA, ILP and their cooperation w.r.t. the number of aircraft.

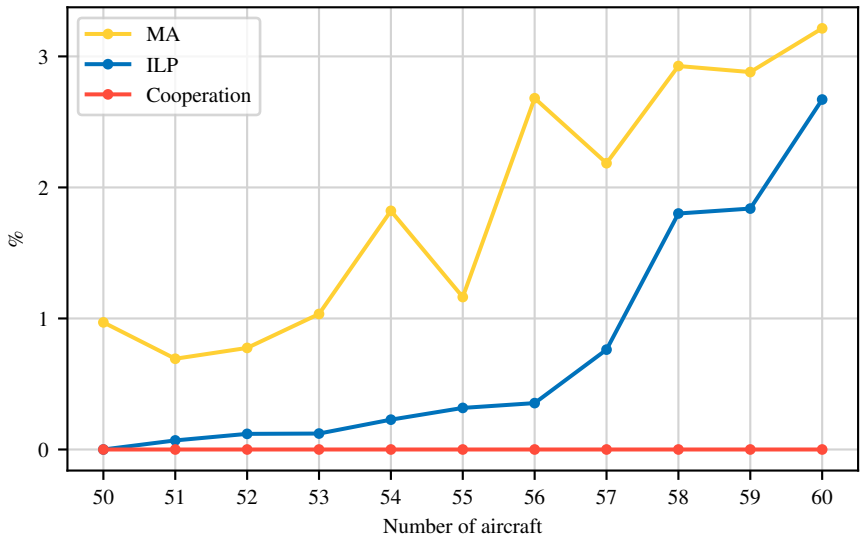


Figure 3.17 – Average cost of the best solution found within 300 s, expressed as the ratio of the difference to the optimum w.r.t. the number of aircraft.

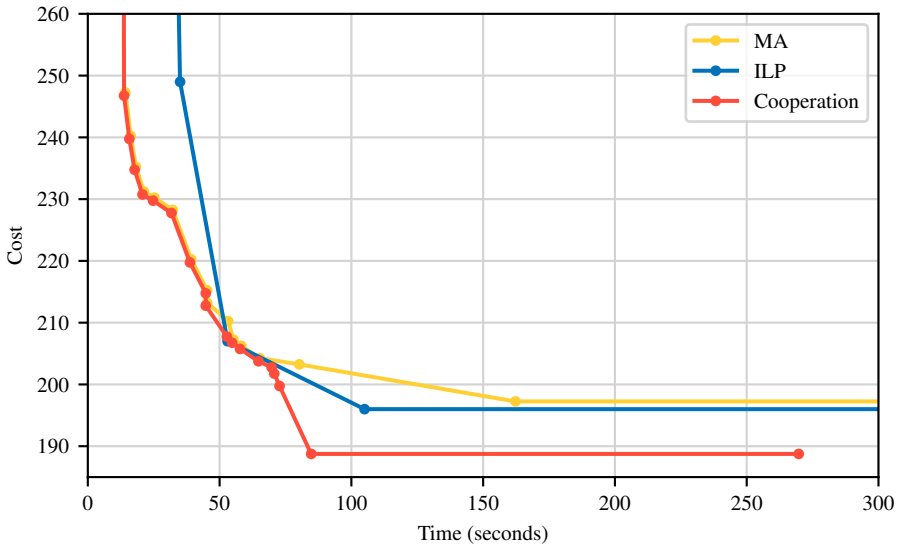


Figure 3.18 – Convergence of the cost with a 300 s time limit w.r.t. the execution time.

solver used for ILP performs some internal transformations of the problem to provide a more efficient search afterwards. During this process, no solution can be proposed. The cooperation highly benefits from this mechanism, as in the meantime the MA can quickly converge to solutions with a good cost, thus saving time for the ILP solver when it is ready to execute the BC algorithm.

In order to assess how the cooperation impacts the optimality of the solutions, we performed a new series of tests without the 300 s time limit, and measured the time needed for the ILP solver alone and for the cooperation to find and prove the optimal solution on large instances (50 to 60 aircraft). The results of this experiment are shown in Figures 3.19 and 3.20 respectively. Note that the y-axis is in log scale on both figures.

Figure 3.19 shows that the advantage of the cooperation over the ILP solver alone increases with the density of the problem: for instances around 50 aircraft, it is only 1.2 to twice as fast to find the optimum, while for larger instances, it can be up to 10 times as fast.

For the proof of optimality, we see in Figure 3.20 that the difference is more consistent, with cooperation being approximately twice as good as ILP alone. This is explained by the fact that the MA does not contribute to the proof of optimality due to its lack of completeness properties. Thus, once an optimal solution has been found, the cooperation has no further advantage over the ILP solver alone. Also, we observe that proving optimality is signifi-

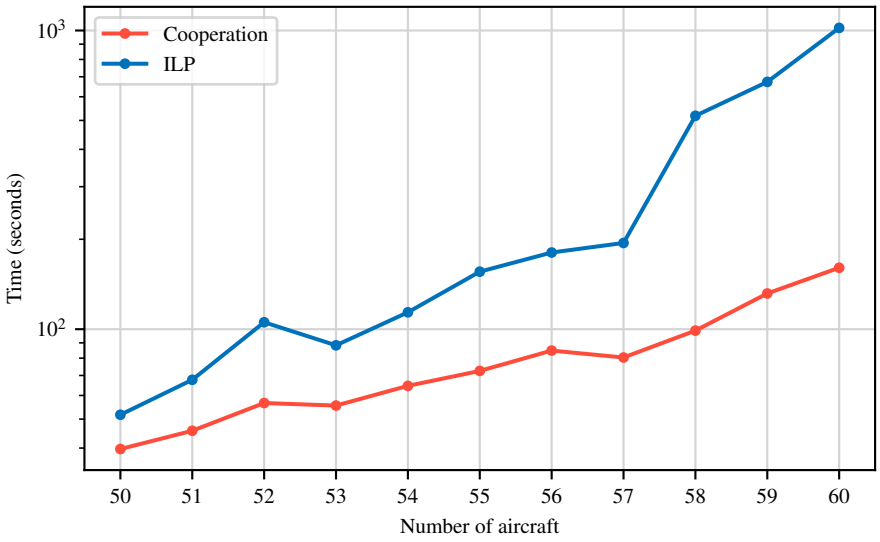


Figure 3.19 – Comparison of computation times to find an optimal solution for the cooperation and the ILP solver alone w.r.t. the number of aircraft.

cantly costlier than finding an optimal solution (by a 10 to 100 factor). In the reported experiment, it was even out of reach (in a reasonable computation time) for most 60-aircraft instances.

3.4.5 Infeasible Instances

In an operational context, any real-time software that manages a non-interruptible critical system should provide a fallback scheme, were an instance infeasible, i.e. without valid conflict-free solutions. In such cases, the solver should provide a set of maneuvers "as good as possible", trying to minimize the number and severity of conflicts, and report perilous situations.

Table 3.3 – Value of the uncertainty bounds w.r.t. the error level l.

uncertainty bound	E_{t_0}	E_{t_1}	E_α	E_{v_h}	E_{v_v}
value for level l	$l \times 10\,\mathrm{s}$	$l \times 10\,\mathrm{s}$	$l \times 1°$	$l \times 2\,\%$	$l \times 5\,\%$

One of the advantages of combining a complete algorithm (like Branch and Cut) able to prove optimality or infeasibility with a metaheuristic (like an MA) is that the latter directly processes candidate solutions in the maneuver space, possibly with remaining conflicts. Moreover, the minimization of the number of conflicts is the main criterion used by its objective function,

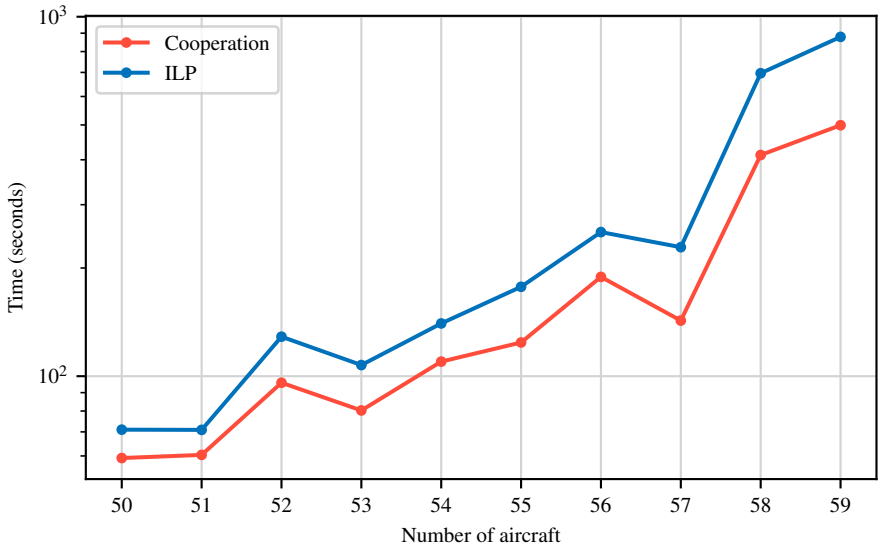


Figure 3.20 – Comparison of computation times to prove optimality for the cooperation and the ILP solver w.r.t. the number of aircraft.

before attempting to minimize its maneuver cost as described in Section 3.3.1. Consequently, when no legal solution is found, the best current solution of the MA tends to minimize the number of conflict violations.

Figure 3.21 shows the mean of the number of remaining conflicts over five scenarios with 60 aircraft, w.r.t. the level of uncertainty $l \in \{5, \ldots, 9\}$ of the detection phase, which was artificially raised until the problem becomes unfeasible (e.g. for $l = 9$, the aircraft may deviate up to 9° relatively to the requested heading change, which is much more than operational error levels). Indeed, envelopes representing the position of aircraft will expand with the level of uncertainty, as well as the number of conflicts. Table 3.3 gives the value of the various uncertainty bounds defined in Section 3.2.2 w.r.t. to the uncertainty level l. Note that the ILP solver is consistently able to report the infeasibility of the tested instances.

In such a case, our cooperation framework can be adapted in several ways:

- whenever a complete solver proves that the instance is infeasible, the cooperation mode can be switched to report the best solution obtained by a metaheuristic within the time limit;
- similarly, if no valid solution nor proof of infeasibility were obtained within the time limit, the best solution of the MA can be reported as well;

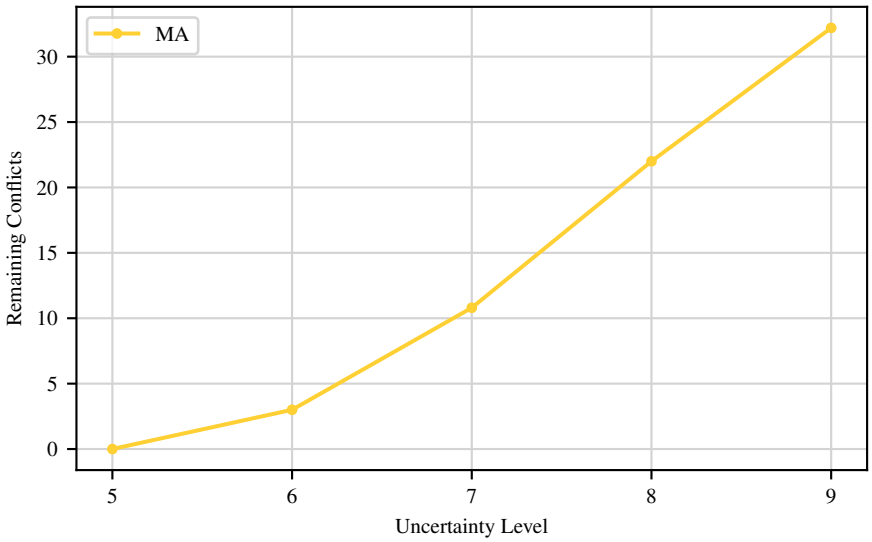


Figure 3.21 – Average of the number of remaining conflicts over 5 instances with 60 aircraft w.r.t. the level of uncertainty l, resolved by the MA with a 300 s time limit.

- eventually, provided that there is enough time to restart the search, the separation norm could be reduced until the problem becomes solvable.

Moreover, in an operational context, all conflicts are not equivalent because violations can be almost inconsequential (e.g. the closest point of approach is 4.5 NM instead of the required 5 NM in the horizontal plane) or severe (e.g. less than 0.5 NM), only last a fraction of a second or continue during one minute. To handle infeasibility more precisely, we could compute the severity of each potential conflict during the detection phase described in Section 3.2.2 and store the information in the conflict matrix. The objective function of the MA can then be modified to minimize the sum of conflict violations weighted by their severity to obtain the safest maneuver set.

Furthermore, in an operational setting, conflict resolution would be iterated over a Rolling Horizon taking into account a limited time window (e.g. 20 min), which is then shifted by a given time step (e.g. 5 min). Therefore, the volume of a given aircraft position envelope (cf. Section 3.2.2) at a given time in the current window will shrink in the next one, as the envelope will be closer to the initial position of the aircraft. Consequently, conflicts that cannot be solved in the current time window might be easier to solve during a later iteration, or even disappear. As explained in Section 3.3.1 concerning the cost of valid solutions, the MA could also be tuned to favor solutions with remaining conflicts appearing as late as possible, in the hope

that the reduction of uncertainty will make them easier to solve during the next iterations.

Eventually, the range of maneuvers could also be extended to widen the solution space with, for example, heading changes of 5°, 15° and 25°, vertical changes of ±3000 ft or speed adjustments greater than a −6 % slow down or greater than a +3 % speed up.

3.5 Conclusion and Further Work

We have presented an innovative framework for the modeling and resolution of en-route conflicts in three dimensions. The model is clearly separated from the resolution, thus giving the opportunity to compare various resolution methods on the same instances. Horizontal, vertical and speed maneuvers are taken into account, and a comprehensive uncertainty model is described as well. The proposed modeling is also totally generic, and gives users the possibility to integrate their own maneuvers. Based on our model, a large set of realistic instances of the problem have been generated, with various densities and difficulties. These instances have been made freely available to the research community at `clusters.recherche.enac.fr`.

Building on previous work (see [Allignol et al., 2013]), we have proposed two algorithms for the resolution of the en-route conflict problem: a Memetic Algorithm (MA) and a Branch and Cut (BC) algorithm to solve Integer Linear Programs (ILPs). The MA has the advantage of finding feasible solutions in a very short time, even for dense instances, while the ILP is able to find and prove optimal solutions for low to medium density instances. Based on these observations, we used a generic framework for the cooperation of algorithms for the resolution of optimization problems, in which any algorithm can share information such as partial solutions, lower or upper bounds for the cost, etc. This framework was successfully tested with the MA and ILP solver.

Instances of low density were optimally solved by all three methods within less than one minute. For larger instances, we have restricted the computation time by 300 s, in order to make the approach compatible with a real-time context. With this limit, both the MA and ILP solver were able to provide good solutions to high density instances, without reaching optimality though. The cooperation between the MA and the ILP solver outperformed both approaches on all instances, and made it possible to reach and prove optimality in most cases, even for very dense instances. Moreover, should a particular instance be infeasible, our solver is able to provide a set of maneuvers that minimizes the number of remaining conflicts, thanks to the incremental

properties of the MA.

The proposed framework is completely generic, from the model to the cooperation method, which provides many opportunities for future research. On the optimization side, we could plug other algorithms in the cooperation tool to further enhance the performance of the resolution. We could also weigh the conflicts according to their severity during the detection phase to handle infeasibility with more accuracy and provide "as-good-as-possible solution". Regarding the Technology Readiness Level of our solver, our next step is to integrate this framework into a more realistic air traffic simulator, with a finer model of aircraft performances, to validate the approach before assessing its real-life abilities in an operational context.

Chapter 4

Application to Gate Allocation

Contents

4.1 Related Works 82

4.2 Fixed Job Scheduling 84

4.2.1 Instance 84

4.2.2 Fictive Tasks and Renumbering 85

4.2.3 Decision Variables 86

4.2.4 Non-Overlapping Constraints 87

4.2.5 Transition Cost 87

4.2.6 Global Compatibility Graph 88

4.3 Basic CP Model 89

4.3.1 Constraints on Maximal Cliques 90

4.3.2 Symmetries 90

4.3.3 The IdleCost Constraint 91

4.3.4 Per-Resource Propagation of Transition Costs . . . 92

4.4 Global Propagation of Transition Cost 97

4.4.1 Relaxation of FJS to Path Covering 97

4.4.2 Successor Variables and Reduction to the Linear Assignment Problem 99

4.4.3 The MinWeightAllDiff Constraint 100

4.4.4 Channelling Constraints 101

4.4.5 Search Strategies 103

4.5 Basic ILP Model 107

4.5.1 Tasks and Resources Compatibility 107
4.5.2 Decision Variables 108
4.5.3 Constraints 109
4.5.4 Objective 110
4.6 Flow Model 111
4.6.1 Graph Model 111
4.6.2 ILP Model 115
4.7 Results 116
4.7.1 Data 117
4.7.2 Per-Resource vs. Global CP Models 118
4.7.3 Search Strategies 118
4.7.4 CP vs. ILP Models 121
4.7.5 Robustness of the Schedule 123
4.8 Conclusion 125

The Gate Allocation Problem (GAP) is one of the numerous operational problems that all busy airports have to handle and to optimize every day. It is usually tackled with specific usages and preferences that are difficult to model or list rigorously — some airlines may even manage their own stands area with their own undisclosed strategy. This can lead to a very complex organization, in which the parking stands can have various configurations during each period of the day.

In the scope of this chapter, we consider the GAP at Paris-CDG, one of the busiest European international airports: terminals and their gates are well described, and the types of aircraft that can operationally use a given gate can be easily reconstructed from the available actual traffic records. Figures 4.1 and 4.2 are extracted from the official airport charts and show the location of the gates in the main terminals at Paris-CDG.

Paris-CDG has implemented the Airport Collaborative Decision Making (A-CDM) program during the last decade as indicated in [EUROCONTROL, 2017]: this program defines which and how accurate information (arrival and departure times, aircraft delay…) can be shared between the different airport stakeholders, in order to help them make more efficient decisions in real time. The A-CDM program also includes a major improvement in the departures management: the delay due to the runways capacities is anticipated and aircraft are preferably delayed at gate, engines off, rather than near the runway after start-up, burning fuel and emitting CO_2. The benefits are twofold: it results in less taxiing traffic, which decreases congestion and nuisance, and also provides a more accurate departure schedule. However, it has a negative effect on gates occupancy, which can significantly increase, especially during peak hours. For these reasons, the robustness of the gate allocation towards additional gate occupancy due to departures delay becomes more and more important for such airports: significant disruptions can be caused in case of gate conflicts between arriving aircraft and delayed departures.

To optimize the robustness of a schedule, [Bolat, 2001] suggests to minimize the variance of the durations of idle times of gates in order to keep a sufficient amount of buffer between successive flights. In this chapter, we first model gate allocation as Fixed Job Scheduling (FJS), which is a well-known resource allocation problem [Eliiyi and Azizoğlu, 2011] with many applications where jobs (or *tasks*, i.e. flights for the GAP) with fixed start and end times are to be processed on different machines (or *resources*, i.e. gates for the GAP). Then we present a new global constraint for Constraint Programming (CP) solvers to propagate the *transition costs* for FJS, which are a generalization of idle times. To minimize the variance of idle times, transition costs are then defined as the squares of idle times between the successive tasks allocated on the same resource.

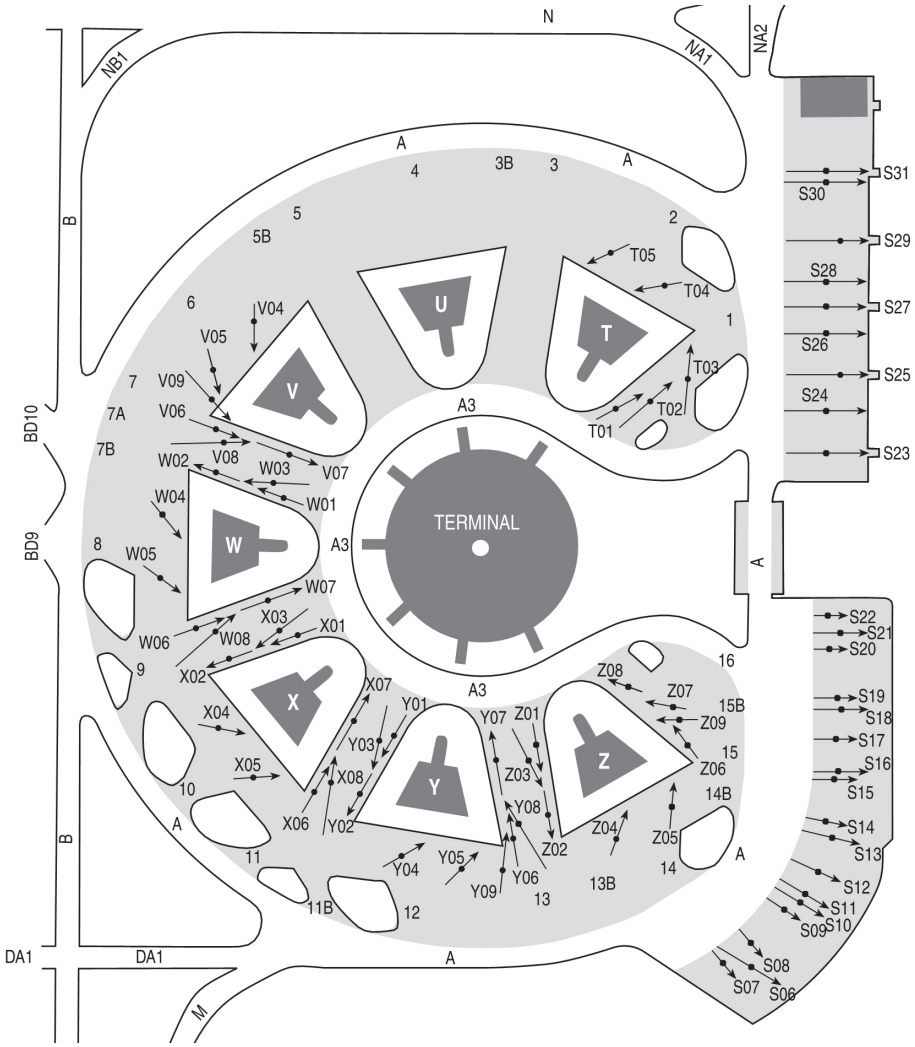


Figure 4.1 – Paris-CDG 1: terminals T, U, V, W, X, Y, Z.

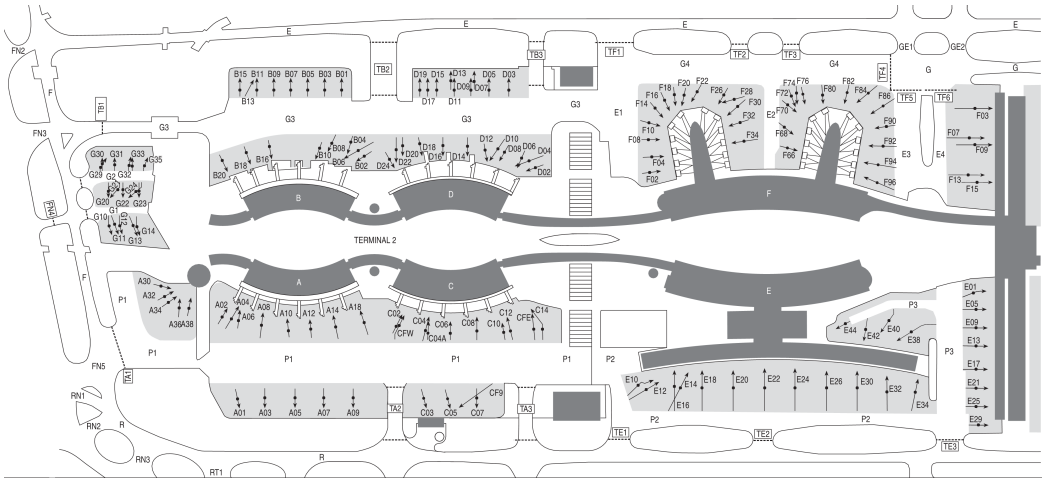


Figure 4.2 – Paris-CDG 2: terminals A, B, C, D, E, F

The propagation of the constraint is based on the computation of the *shortest path* in the *compatibility* directed acyclic graph of each resource (see Section 4.2.6) to obtain an exact lower bound. It ensures Bound Consistency on the cost of a single resource in polynomial time, as well as the filtering of the resource variables associated with compatible tasks. However, the corresponding relaxation is not of good quality w.r.t. the global lower bound, as a task may be simultaneously scheduled on all its compatible resources.

Hence, we also present a new CP model based on the *MinWeightAllDiff* optimization constraint [Caseau and Laburthe, 1997] to compute the lower bound of a *Path Cover* (PC) of the *Global Compatibility Graph* (cf. Section 4.2.6) of the whole problem. This relaxation is much tighter as the constraint directly propagates on the total cost, considering all resources and all tasks simultaneously. We also describe how the optimal PC computed by the constraint can efficiently guide the search strategy. The resulting CP solver, implemented with FaCiLe [Barnier and Brisset, 2001], using parallel cooperation between the strategies, not only outperforms the previous approach by orders of magnitude, but also consistently outperforms a basic Integer Linear Programming (ILP) model resolved with a state-of-the-art MIP solver on real instances of the GAP at Paris-Charles-de-Gaulle international airport.

At the same time, we also point out the problems of the basic ILP model, then propose a new Minimum Cost Flow Problem (MCFP) model for the GAP. This MCFP model outperforms the basic one by one order of magnitude and competes well against our CP solver using parallel cooperation between the strategies. For most instances, the parallel CP solver outperforms the ILP one, except for the densest instances of the busiest terminal

for which the CP solver always manages to give solutions of very good quality in less than 60 s, but their optimality can hardly be proved in a reasonable time.

This chapter is organized as follows: Section 4.1 presents the literature on the GAP, then we provide a precise formulation of the problem in Section 4.2. Section 4.3 details our CP approach, which includes the description of a new global optimization constraint which propagates the transition cost of a single resource, then of the propagation of the global cost thanks to an incremental implementation of the *MinWeightAllDiff* constraint which relaxes FJS as *Path Covering*, and eventually of several search strategies and their parallel cooperation, which is able to outperform the MIP solvers presented in Section 4.5 on real instances of the GAP as shown in Section 4.7. Afterwards, we discuss the problems of the ILP model for the GAP introduced in [Bolat, 2001] and propose a new improved ILP model in Section 4.6. The comparison with the parallel CP solver is detailed in Section 4.7 and we conclude in the last section.

The work carried out on the GAP problem has been published in [Wang and Barnier, 2018, Wang et al., 2019, Wang and Barnier, 2020].

4.1 Related Works

The Gate Allocation Problem[1] (GAP) consists in assigning arriving flights with fixed occupancy periods to available compatible gates while maximizing both conveniences to passengers and operational efficiency of airports [Bouras et al., 2014]. [Steuart, 1974] was one of the first to introduce the problem in 1974, but the literature is scarce until the 2000s. However, many variants of the GAP have been studied since, as mentioned in [Bouras et al., 2014].

Were there no compatibility restriction between gates and aircraft, the corresponding decision problem could be modeled as the *coloring* of an *interval graph*, which is polynomial [Gupta et al., 1982] for the minimization of the number of colors (i.e. gates). Airport gates are generally not equivalent resources though, as they are dimensioned to accommodate specific types of aircraft. Therefore, the set of compatible gates for an aircraft usually is a strict subset of all the available gates and the decision version of the allocation problem is rather a *list-coloring* problem, which is NP-Complete even for interval graphs [Biró et al., 1992].

Moreover, gates may also be endowed with other secondary features (e.g. compatible airlines, domestic or international, terminal or apron, etc.) which

[1] This problem is sometimes called Airport Gate Allocation Problem (AGAP) or Stand Allocation Problem (SAP) in the literature.

should match the characteristics of the flight and the preferences of airlines as much as possible. These preferences can be modeled as costs associated with each possible assignment, and standard GAP objectives often include the minimization of their sum, which is NP-Hard [Kroon et al., 1997]. From the airport operations perspective, the objective could be to maximize the utilization of the available gates and terminal [Steuart, 1974, Yan and Huo, 2001, Li and Xu, 2012], minimize the flight delay [Yan and Tang, 2007] or maximize the preferences [Dorndorf et al., 2007] (i.e. certain aircraft should go for particular gates). Other classic objectives can be the passengers walking distance [Kim, 2013] (or other connection means like buses), which is similar to the *Quadratic Assignment Problem*, or the number of towing movements [Guépet et al., 2015].

Even if flights occupancy is fixed, many real-life factors of uncertainty (e.g. traffic delays, severe weather conditions, equipment failure, etc.), can lead to deviations from the original schedule. If there is not enough buffer time between successive flight occupancies at a given gate, a delay may propagate to other flights, then to other gates, and hinder the operational efficiency of the whole terminal. In order to absorb potential delays and avoid costly disruptions, our study rather focuses on optimizing the *robustness* of the allocation as proposed by [Bolat, 2000], which minimizes the variance of idle times to balance and spread them over time and resources. Despite its practical importance, research on the robustness of solutions to the GAP is quite limited.

To solve these very diverse variants of the GAP, many classic combinatorial optimization methods were experimented, depending on the linearity of the model, the size of the instances and the requirements on the execution time of the solver. One of the most used tools is Mixed Integer Programming or Integer Linear Programming (ILP) solvers, like Gurobi or CPLEX, to obtain proved optimal solution like in [Guépet et al., 2015]. Several studies present their ILP model as a Multi-commodity Flow Problem (MFP), like [Zhang and Klabjan, 2017] which models re-assignment while minimizing passengers cost, or [Maharjan and Matis, 2012] which minimizes taxiing and connection costs (though the model description is, unless we are mistaken, technically unsound). Another complete combinatorial optimization technique, Constraint Programming, was also experimented in [Simonis, 2007] to solve the GAP as a *scheduling* problem similar to Fixed Job Scheduling (FJS).

As previously mentioned, all considered variants of the GAP are NP-Complete or NP-Hard, so many publications have suggested various heuristic approaches for solving large instances or non-linear models in a reasonable time. [Yan and Chang, 1998] developed an algorithm based on the

Lagrangian relaxation of the GAP, with subgradient methods, accompanied by a shortest path algorithm and a Lagrangian heuristic to solve an MFP model. Additionally to the ILP model previously mentioned, [Benlic et al., 2017] proposed a multi-objective heuristic approach base on Breakout Local Search, with a particular focus on the perturbation strategy. [Deng et al., 2017] proposed an improved Particle Swarm Optimization algorithm to solve a multi-objective (walking distance of passengers, number of flights at parking apron, etc.) optimization model. [Zhang and Klabjan, 2017] used two heuristic algorithms to minimize the weighted sum of the total flight delays. More research work can be found in a survey about the GAP [Bouras et al., 2014].

More generally, there are already many studies on the GAP focusing on different criteria. However, few of them except [Bolat, 2000, 2001] really consider the robustness of the assignment, which appears to be one of the most important criteria for major international airports like Paris-CDG. Moreover, the resolution process should be efficient enough to be computed in a few minutes, as the GAP does not occur only once (one day before the date of the traffic), but has also to be solved almost in real time when a severe disruption leads to a necessary re-assignment of gates.

4.2 Fixed Job Scheduling

The scheduling of tasks with fixed start and end times on non-identical[2] resources is a versatile NP-complete problem [Arkin and Silverberg, 1987] which occurs in many applications beside the GAP, like processors scheduling or staff rostering. Though various objectives can be associated with this problem, our approach is dedicated to optimize the transition cost between tasks, particularly to obtain robust solutions w.r.t. delays.

We present in the following sections the integer model used in our study (whereas classic ILP approaches consider boolean variables only, as in [Bolat, 2000]), with the introduction of *fictive tasks* to model the opening and closing of resources, and of the *compatibility graph* used to define our new CP model in Section 4.3.

4.2.1 Instance

An instance of the FJS problem is defined by:

[2]Note that with identical resources (i.e. all tasks can be assigned to any resource), this problem becomes equivalent to the coloring of an interval graph, which is polynomial [Gupta et al., 1982].

- $\mathcal{T} = \{t_1, \ldots, t_n\}$ a set of n tasks, with $\forall t_i \in \mathcal{T}$:
  - t_i^s and t_i^e the start and end times of task t_i;
  - $\mathcal{R}_i \subseteq \mathcal{R}$ a set of compatible resources on which the task can be executed.
- $\mathcal{R} = \{r_1, \ldots, r_m\}$ the set of m resources, with $\forall r_j \in \mathcal{R}$:
  - r_j^s and r_j^e the opening and closing times of r_j. However, all resources are considered available during the same period[3] in the following, i.e. $\forall j, r_j^s = r^s$ and $r_j^e = r^e$.
  - $\mathcal{T}_j = \{t_i \in \mathcal{T} \text{ s.t. } r_j \in \mathcal{R}_i\}$ the set[4] of compatible tasks that can be executed on resource r_j.

Without loss of generality, the tasks of $\mathcal{T}$ are supposed to be numbered by increasing start time, i.e. $\forall i < i', t_i^s \leq t_{i'}^s$.

For conciseness, we also define the *duration* function d, "overloaded" on the following sets:

- $\mathcal{T} \mapsto \mathbb{N}$ the duration of a task: $d(t_i) = t_i^e - t_i^s$;
- $2^{\mathcal{T}} \mapsto \mathbb{N}$ the total sum of the durations of a subset of (possibly overlapping) tasks: $d(\mathcal{T}') = \sum_{t_i \in \mathcal{T}'} d(t_i)$;
- $\mathcal{R} \mapsto \mathbb{N}$ the availability of a resource: $d(r_j) = r_j^e - r_j^s$;
- $2^{\mathcal{R}} \mapsto \mathbb{N}$ the total sum of the availability of a set of resources: $d(\mathcal{R}') = \sum_{r_j \in \mathcal{R}'} d(r_j)$.

4.2.2 Fictive Tasks and Renumbering

As already mentioned, our model is designed to minimize the variance of idle times. To obtain a uniform formulation of our model, even when the idle time considered occurs between the opening of a resource and its first task or between its last task and its closing, we introduce $2m$ additional fictive tasks with a singleton compatible resource set and null duration corresponding to the openings and closings of the m resources.

We accordingly renumber the tasks of $\mathcal{T}$ and the resource sets $\mathcal{R}_i$, while preserving the ordering by increasing start time:

[3] Without loss of generality, as the unavailability of a resource can be modelled by an additional task with a singleton resource set.

[4] Redundantly defined from $\mathcal{R}_i$ to simplify notations afterwards.

- openings: $t_1, \ldots, t_m$ with $t_j^s = t_j^e = r^s$ and $\mathcal{R}_j = \{r_j\}$;
- actual tasks: $t_{m+1}, \ldots, t_{m+n}$ with $\mathcal{R}_{m+i}$ equal to $\mathcal{R}_i$ of Section 4.2.1 (before the renumbering);
- closings: $t_{m+n+1}, \ldots, t_{2m+n}$ with $t_{m+n+j}^s = t_{m+n+j}^e = r^e$ and $\mathcal{R}_{m+n+j} = \{r_j\}$.

The task set is therefore redefined by:

$$\mathcal{T} = \{\underbrace{t_1, \ldots, t_m}_{\text{openings}}, \underbrace{t_{m+1}, \ldots, t_{m+n}}_{\text{actual tasks}}, \underbrace{t_{m+n+1}, \ldots, t_{2m+n}}_{\text{closings}}\}$$

We also define the subsets of predecessors $\mathcal{T}^s$, successors $\mathcal{T}^e$ and actual tasks $\mathcal{T}^a$ (i.e. $\mathcal{T}$ in the previous section):

$$\begin{array}{rcl} \mathcal{T}^s & = & \{t_1, \ldots, t_{m+n}\} \\ \mathcal{T}^e & = & \{t_{m+1}, \ldots, t_{2m+n}\} \\ \mathcal{T}^a & = & \{t_{m+1}, \ldots, t_{m+n}\} \end{array}$$

We also extend each set $\mathcal{T}_j$ of compatible tasks to include its fictive opening t_j and closing t_{m+n+j}.

4.2.3 Decision Variables

A solution to an FJS problem consists in assigning a resource to each task while satisfying the non-overlapping constraints of Equation (4.1) described in the next section. As the fictive tasks are already assigned to their associated resource, we define the set of decision variables over actual tasks only:

Definition 9 (Resource Variables)**.** *A solution to an FJS problem is represented by a set of* resource variables*:*

$$\mathcal{X} = \{x_i, \forall t_i \in \mathcal{T}^a\}$$

where $\forall x_i \in \mathcal{X}, d_{x_i} = \{j \text{ s.t. } r_j \in \mathcal{R}_i\}$ *is the domain of* x_i*, such that* $x_i = j$ *iff task* t_i *is assigned to resource* r_j*.*

Note that tasks and possible resource sets have been renumbered in the previous section (compared to Section 4.2.1), so that actual task t_i becomes t_{i+m} and resource set $\mathcal{R}_i$ becomes $\mathcal{R}_{i+m}$, $\forall i \in \{1, \ldots, n\}$.

4.2.4 Non-Overlapping Constraints

The only type of constraints of this essential version of the problem is the non-overlapping of the tasks scheduled on the same resource. As tasks execution times are fixed, we require that overlapping tasks are assigned to different resources:

$$x_i \neq x_{i'}, \quad \forall t_i \neq t_{i'} \in \mathcal{T}^a \text{ s.t. } [t_i^s, t_i^e[\cap [t_{i'}^s, t_{i'}^e[\neq \varnothing \tag{4.1}$$

However, specific applications of FJS like the GAP are often described with many additional hard and soft constraints to account for operational requirements (e.g. a large aircraft might occupy two adjacent stands) or user preferences (e.g. favor terminal gates over remote apron stands).

4.2.5 Transition Cost

Many different kinds of costs can be taken into account to optimize the allocation of fixed tasks on non-identical resources. For our target application, the GAP, one of the most crucial objectives is the robustness of the schedule, as air traffic operations can be burdened by many uncertainties such as late arrival or departure. To be able to absorb those possible delays, [Bolat, 2000] proposes to minimize the variance of *idle times*, which tends to balance them over resources and time while allowing necessary short or large pauses required by some instances.

Since the mean of the idle times is constant for our problem (as the overall duration of tasks and availability of resources are constant, and all tasks must be scheduled), minimizing their variance amounts to minimizing the sum of their squares:

$$\text{cost} = \sum_{\forall t_i \in \mathcal{T}^s} \left(\text{next}(t_i)^s - t_i^e\right)^2 \tag{4.2}$$

where function next : $\mathcal{T}^s \mapsto \mathcal{T}^e$ returns the successor of a task (closings having no successor and openings no predecessor). So the cost c_j for a single resource r_j can be expressed as: $c_j = \sum_{t_i \in \mathcal{T}_j \text{ s.t. } x_i = j} \left(\text{next}(t_i)^s - t_i^e\right)^2$.

More generally, our approach is generic and able to optimize the sum of the *transition* costs $c_{t_i,t_{i'}}$ between successive tasks $t_i \in \mathcal{T}^s$ and $t_{i'} \in \mathcal{T}^e$, with any positive cost matrix C:

$$\text{cost} = \sum_{\forall t_i \in \mathcal{T}^s} c_{t_i,\text{next}(t_i)} \tag{4.3}$$

4.2.6 Global Compatibility Graph

To model the FJS transition cost in our CP solver, as described in Section 4.3, we define the notion of *compatibility* on a pair of tasks, then of the *Global Compatibility directed acyclic Graph* (GCG) of the whole problem.

The *compatibility* predicate indicates whether two ordered tasks can both be scheduled on the same resource:

Definition 10 (Compatibility)**.** *The* compatibility *predicate* $\gamma : \mathcal{T}^s \times \mathcal{T}^e \mapsto \mathbb{B}$ *is defined over all pairs of ordered tasks* t_i *and* $t_{i'}$ *s.t.* $i < i'$ *by:*

$$\gamma(t_i, t_{i'}) = (\mathcal{R}_i \cap \mathcal{R}_{i'} \neq \varnothing) \ \wedge \ (t_i^e \leqslant t_{i'}^s)$$

If $\gamma(t_i, t_{i'})$ *holds, then* t_i *and* $t_{i'}$ *are said to be* compatible*.*

We can then define the GCG of the whole problem that represents each task (actual and fictive) as a node and each ordered pair of compatible tasks as an arc weighted by their transition cost:

Definition 11 (Global Compatibility Graph (GCG))**.** *The weighted* Global Compatibility Graph $G = (V, E)$ *of an FJS problem is defined by:*

- $V = \{v_i, \forall t_i \in \mathcal{T}\}$
- $E = \{(v_i, v_{i'}), \forall t_i \in \mathcal{T}^s, \forall t_{i'} \in \mathcal{T}^e, \ s.t. \ i < i' \wedge \gamma(t_i, t_{i'})\}$
- $w : E \mapsto \mathbb{R}_{\geq 0}$ *with* $w((v_i, v_{i'})) = c_{t_i, t_{i'}}$

We will take $c_{t_i,t_{i'}} = (t_{i'}^s - t_i^e)^2$ to optimize the robustness of GAP instances in Section 4.7. Once more, note that any positive function $w : E \mapsto \mathbb{R}_{\geq 0}$ could be used to weigh the transition between tasks instead of the square of idle times. We will also use the following notation:

- $V_j = \{v_i, \forall t_i \in \mathcal{T}_j\}$, $\forall r_j \in \mathcal{R}$, i.e. the set of nodes corresponding to tasks compatible with resource r_j;
- $G_j = G[V_j]$ the subgraph of the GCG induced by the nodes of resource r_j;
- V^s, V^e and V^a the restriction of V to $\mathcal{T}^s$, $\mathcal{T}^e$ and $\mathcal{T}^a$;
- $N^+(v_i) \subset V^e$ the set of successors of node v_i.

4.3 Basic CP Model

We present in this section how the previous formulation of the FJS problem translates in a CP context. We first describe how all maximal cliques of the conflict graph of the tasks can be easily computed to efficiently model the mutual exclusion of overlapping tasks, and how symmetries on resources and tasks can be broken, then we introduce a new global constraint named *IdleCost* to propagate the idle times cost (or any positive transition cost) on each resource independently. Particularly, an incremental shortest path algorithm is used on the restricted compatibility graph G_j (see Section 4.2.6) of the possible (and assigned) tasks of each resource r_j to compute the lower bound of its contribution to the global cost. However, the resulting CP solver was only able to solve small instances up to 40 tasks and could not compete with the ILP model of [Bolat, 2001], because the lower bound of the global cost can be $O(m)$ times worse than the actual bound when the uniqueness of task assignment is relaxed.

To improve our CP approach, we then introduce a new model based on a much tighter relaxation: we compute the *Minimum Weight Path Cover* (MWPC) [Ntafos and Hakimi, 1979] of the GCG (cf. Definition 11) in polynomial time, thanks to an optimization constraint that directly propagates on the total cost, considering all resources and all tasks simultaneously. More precisely, MWPC in a *Directed Acyclic Graph* (DAG) can be reduced to the *Linear Assignment Problem* (LAP), solvable by the *Hungarian method* [Kuhn, 1955] in $O(|V^s||E|)$, with $|V^s|= |\mathcal{T}^s|= n+m$. [Sellmann, 2002] describes how this algorithm can be used to achieve Generalized Arc Consistency (GAC) for the *Minimum Weight All Different (MinWeightAllDiff)* optimization constraint which can be posted on *successor* variables of the tasks to model the corresponding LAP.

Even if much tighter than the former relaxation, an MWPC is not in general a solution to FJS, as consistency on resources along each path is not taken into account. So resource variables are still necessary, as well as channelling constraints to link them with the successor variables involved in the MWPC. So after the definition of the IdleCost constraint and its propagation rules, we present how the optimization of the transition cost of FJS can be relaxed to MWPC, which is modelled with successor variables and reduced to the LAP. Eventually we describe our incremental version of the MinWeightAllDiff constraint and the channelling constraints that link the successor and resource variables and how the minimum matching computed while propagating the constraint can efficiently guide the search strategy.

4.3.1 Constraints on Maximal Cliques

To specify the mutual exclusion constraints of Equation (4.1) on unitary resources, it would be sufficient to model each of them directly with a binary difference constraint. However, stronger propagations can be obtained with the well-known *all-different* global constraint on *cliques* of the associated *conflict graph* of the tasks, as noted in [Simonis, 2007]. Only constraints corresponding to all distinct *maximal* cliques need to be added, as any other clique would be subsumed by a maximal one.

In the general case, computing the *maximum* clique of an arbitrary graph is NP-Hard. However, [Gupta et al., 1982] mentions how all maximal cliques (including the maximum one) of an interval graph of n vertices can be generated by a sweep algorithm in $\Theta(n \log n)$, or even linear time $\Theta(n)$ if the endpoints of the intervals are already sorted.

Indeed, once the $2n$ endpoints of the tasks are sorted by ascending order, all maximal cliques of the corresponding intersection graph can be easily detected by maintaining the list of overlapping tasks at each endpoints. While scanning the sorted list of endpoints, the corresponding task is added whenever its left endpoint is encountered and removed upon reaching its right endpoint. For each minimal size of the maintained list (except when it is empty), a new clique can be started with all currently overlapping intervals, collecting all subsequently opened new intervals until a local maximum is reached, which corresponds to a new maximal clique. All-different constraints can then be posted in linear time on all maximal cliques to improve propagation and allow global reasoning over multiple resources.

However, as mentioned in [Kutz et al., 2008], a collection of all-different constraints that share subsets of variables but propagate independently may keep inconsistent values w.r.t. their conjunction. [Simonis, 2007] mentions the use of the non-overlap global constraint *diffn* introduced by [Beldiceanu and Contejean, 1994] to solve placement problems. However, its propagation rules would be useless with the FJS because tasks are fixed and have a unitary demand. Moreover, it has been proved in [Kutz et al., 2008] that the corresponding decision problem is NP-complete. As the instances of our target application, the GAP, are relatively easy to solve w.r.t. the allocation decision problem, it was deemed efficient enough to use a simple collection of all-different constraints in our model.

4.3.2 Symmetries

Depending on the instance at hand, allocation problems may exhibit symmetries on equivalent resources and equivalent tasks. For the GAP, the former

is much more frequent on real instances: many adjacent stands in a terminal share the same set of characteristics, whereas equivalent flights with the same aircraft type, company and dates are seldom.

Resources

Resources that have exactly the same set of possible tasks can be exchanged while preserving the admissibility and optimality of solutions. Therefore, whenever a (yet) unused resource is assigned to a task, all other unused equivalent resources should be removed from its domain upon backtracking.

To this end, all the equivalence classes $\mathcal{C}_k$ of resources (with at least two elements) are computed before search. Before each assignment of a task i to an unused resource r_j, we check if other unused resources remain in the corresponding class $\mathcal{C}$ to add the following goal:

$$(x_i = r_j) \vee (x_i \notin \{j' \text{ s.t. } r_{j'} \in \mathcal{C} \wedge \text{unused}(r_{j'})\})$$

with predicate unused : $\mathcal{T} \mapsto \mathbb{B}$ indicating whether a resource is still free of any task or not.

Tasks

Two tasks i and i' that have the same compatible resource set and arrival and departure dates can also be exchanged while preserving solutions. So they can be arbitrarily ordered with the following constraint:

$$\forall i < i' \text{ s.t. } \mathcal{R}_i = \mathcal{R}_{i'}, t_i^s = t_{i'}^s, t_i^e = t_{i'}^e, \quad x_i < x_{i'} \tag{4.4}$$

4.3.3 The IdleCost Constraint

Modeling transition cost with standard CP reification constraints would be cumbersome and inefficient. Therefore, we introduce a new global optimization constraint *IdleCost* to tighten the bounds of the cost of a single resource and the domains of its possible tasks. In the following sections, we define the semantic of this constraint as well as static bounding schemes for the overall sum of transition costs. Propagation rules for the *IdleCost* constraint are then discussed in Section 4.3.4.

Optimization Constraint for a Single Resource

We introduce the *IdleCost* optimization constraint on a single resource $r_j \in \mathcal{R}$ to tighten its idle times cost and filter the resource variables of its set of possible tasks $\mathcal{T}_j$:

Definition 12 (The *IdleCost* Constraint)**.** *Let $\mathcal{X}_{\mathcal{T}_j} \subseteq \mathcal{X}$ be the set of resource variables associated to $\mathcal{T}_j$ and $f : \mathbb{N} \mapsto \mathbb{N}$ a positive elementary cost function that represents the contribution of a single idle time interval. The optimization constraint* $\text{IdleCost}(r_j, \mathcal{T}_j, \mathcal{X}_{\mathcal{T}_j}, f, c_j)$ *is satisfied iff:*

- $c_j = \sum_{t_i \in \mathcal{T}_j \text{ s.t. } x_i = j} f(\text{next}(t_i)^s - t_i^e)$ *with the* next *function defined as in Section 4.2.5;*
- *the set of mutual exclusion constraints Equation* (4.1) *of Section 4.2.4 restricted to variables of $\mathcal{X}_{\mathcal{T}_j}$ is satisfied.*

Static Lower Bound

A static lower bound for the overall sum of the squares of idle times can be easily computed as it is minimal when all idle times have the same size and are evenly distributed among all resources. For n tasks to be executed on m resources, there are exactly $n+m$ idle time periods (taking the first and last idle times of every resource into account). Therefore, we can compute the following global lower bound for the objective:

$$\text{cost} \geq (n+m) \left\lfloor \frac{d(\mathcal{R}) - d(\mathcal{T})}{n+m} \right\rfloor^2 \tag{4.5}$$

Static Upper Bound

For instances with identical availability for all resources, a simple upper bound could also be obtained by saturating the first resources except the last non-empty one (where tasks are stacked at the beginning). The corresponding bound would then be:

$$\text{cost} \leq m' d(r)^2 + (d(r) - k)^2 \tag{4.6}$$

with $d(r) = r^e - r^s$, $m' = m - \left\lceil \frac{d(\mathcal{R}) - d(\mathcal{T})}{d(r)} \right\rceil$ the number of empty resources and $k = (d(\mathcal{R}) - d(\mathcal{T})) \mod d(r)$ the time taken by the tasks scheduled on the last non-empty resource. But this bound is very loose and not really significant to help close the optimality gap.

4.3.4 Per-Resource Propagation of Transition Costs

We present in this section a new polynomial algorithm to achieve *Bound Consistency* on the cost of a single resource for the *IdleCost* constraint: after its execution, a partial assignment can be extended to the possible tasks of

the resource such that the cost can be assigned either its lower or upper bound.

After describing a tight upper bound and a naive relaxed lower bound in Section 4.3.4, we introduce in more details our algorithm to achieve Bound Consistency on the lower bound in Section 4.3.4. We define the following notations, used in these sections, w.r.t. a resource r_j:

- $\overline{\mathcal{T}_j} = \{t_i \in \mathcal{T}_j \text{ s.t. } j \in \text{dom}(x_i)\}$ its possible tasks;
- $\underline{\mathcal{T}_j} = \{t_i \in \mathcal{T}_j \text{ s.t. } x_i = j\}$ its necessary tasks;
- ub_j the upper bound induced by $\underline{\mathcal{T}_j}$;
- lb_j the lower bound induced by $\underline{\mathcal{T}_j}$ and $\overline{\mathcal{T}_j}$;

with $\text{dom}(x_i)$ the set of currently possible values for x_i.

Bounds Based on the Union of Tasks

In the next paragraphs, we present simple algorithms to compute the upper bound and an approximation of the lower one.

Assignment and Upper Bound The upper bound of a single resource can be computed thanks to the set of necessary tasks $\underline{\mathcal{T}_j}$ already assigned to r_j. If the cost was linear, it would be enough to maintain the sum of the durations of the tasks of $\underline{\mathcal{T}_j}$ and subtract it from the availability of the resource to obtain a tight upper bound on the cost: $d(r_j) - \sum_{t_i \in \underline{\mathcal{T}_j}} d(t_i)$. Upon assignment of a task t_i, we would just have to subtract its duration to incrementally maintain the bound in constant time: $\text{ub}_j \leftarrow \text{ub}_j - d(t_i)$.

Though for a positive *non-linear* cost like the sum of the *squares* of idle times, we have to thoroughly maintain the set of idle intervals to be able to determine which one will be impacted by the new assigned task t_i, and compute the new upper bound incrementally. Once the idle interval $[a, b[$ is identified, the upper bound is updated accordingly:

$$\text{ub}_j \leftarrow \text{ub}_j - (b-a)^2 + (t_i^s - a)^2 + (b - t_i^e)^2 \tag{4.7}$$

As $\underline{\mathcal{T}_j}$ only contains disjoint tasks, a simple *binary search tree* could be used to maintain the set of intervals and update the upper bound in logarithmic time upon assignment, provided that the data structure be easily "backtrackable". However, considering the low number of tasks associated with each resource in the instances of our target application (the GAP), we only implemented a simple linear algorithm in the solver to avoid the likely overhead costs of a more sophisticated data structure.

Removal and Lower Bound Approximation The removal is not as straightforward as the assignment because tasks are not disjoint within $\overline{\mathcal{T}_j}$, so the removal of one task from a resource r_j does not entail that a new necessary idle time appears on r_j. However, by maintaining the union of intervals corresponding to the tasks of $\overline{\mathcal{T}_j}$, necessary idle times can be detected whenever $\overline{\mathcal{T}_j}$ does not span the entire availability. The lower bound can then be updated as follows upon removal:

$$\mathrm{lb_j} \leftarrow \sum_{k=1}^{l} (h_k^e - h_k^s)^2 \tag{4.8}$$

with $\overline{\mathcal{H}_j} = [r_j^s, r_j^e[\setminus \bigcup_{\forall t_i \in \overline{\mathcal{T}_j}} [t_i^s, t_i^e[= \bigcup_{k=1}^{l} [h_k^s, h_k^e[$ the necessary "holes" (possibly $\varnothing$, in which case $\mathrm{lb}_j \leftarrow 0$).

A *segment tree* data structure [de Berg et al., 2000] can be used to maintain the tasks intervals, and augmented to also aggregate the upper bound of each subtree, including the root node which holds the bound for the whole resource, in $\Theta(k \log n)$ with n the number of intervals and k the number of newly discovered necessary idle times. However, for reasons already mentioned in the previous section, our solver only use a naive (but simple) quadratic algorithm.

Tight Lower Bound

Among the bounds presented in the previous section, only the upper one is tight, because the assignment of a new possible task on a resource necessarily decreases the cost. In contrast, the lower bound is not, as the union of tasks doesn't take into account temporal conflicts. Therefore, the actual lower bound could be arbitrarily larger as, for example, with the following set of n possible tasks on a given resource depicted in Figure 4.3:

Example 1 (Overlapping tasks with unbounded lower bound approximation ratio)**.** $\forall i \in [0..\frac{n}{2} - 1]$*:*

- $t_{2i+1}^s = 2i(g+1)$ *and* $t_{2i+1}^e = 2i(g+1) + g + 2$
- $t_{2i+2}^s = (2i+1)(g+1)$ *and* $t_{2i+2}^e = (2i+1)(g+1) + g + 2$

with $r^s = 0$*,* $r^e = n(g+1) + 1$ *and some constant* $g \in \mathbb{N}_{>0}$*.*

The lower bound of Section 4.3.4 applied to Example 1 would be 0 whereas the actual lower bound is $\frac{n}{2}g^2$. Therefore, the ratio of the actual lower bound over the one defined by Equation (4.8) can be arbitrarily large.

Figure 4.3 – The resource and tasks of Example 1 with unbounded lower bound approximation ratio.

To achieve Bound Consistency on the cost lower bound, the best admissible solution for the single resource should be taken into account. This best solution must be a maximal independent set of the conflict graph, as adding another task can only decrease the cost of a resource, but it is not necessarily either the maximum independent set (as the number of tasks is not relevant on instances with different durations) or even the largest (duration wise) maximal one as it would be the case with a linear cost.

More precisely, a solution corresponding to the lower bound of a resource r_j must be a *shortest path* between vertices v_j and v_{m+n+j} corresponding to the fictive opening task t_j and fictive closing task t_{m+n+j} of the resource in the Global Compatibility Graph (GCG) $G = (V, E)$, introduced in Section 4.2.6.

In the GCG, the shortest path between two vertices can be computed in linear time $\Theta(|V|+|E|)$ as mentioned in [Cormen et al., 2009], provided a topological ordering of the vertices (corresponding to the tasks sorted by increasing start time, i.e. for each edge $(v_i, v_j) \in E$, the start time of t_i is smaller than the start time of t_j). Therefore, the lower bound of an *IdleCost* optimization constraint can be computed in linear time with respect to G (i.e. possibly quadratic time w.r.t. the number of tasks n):

$$\mathrm{lb_j} \leftarrow \mathrm{dist}(v_j, v_{m+n+j}) \tag{4.9}$$

with $\mathrm{dist} : V^2 \mapsto \mathbb{N}$ the length of the shortest path from v_j to v_{m+n+j} in G.

After the initialization steps where $\overline{\mathcal{T}_j}$ and G are built and the bounds computed, the *IdleCost* constraint must propagate whenever:

- A task t_i is assigned to r_j:
  - the upper bound must be updated (cf. Section 4.3.4);
  - the arcs of the predecessors of v_i pointing to vertices $v_{i'}$ s.t. $i' > i$ (i.e. that "skip" v_i) must be deleted;
  - if v_i does not belong to the previous shortest path, a new one must be computed and the lower bound updated accordingly.
- A task t_i is removed from r_j:

  - the arcs of the predecessors of v_i pointing to v_i must be deleted;
  - if v_i belongs to the previous shortest path, a new one must be computed and the lower bound updated accordingly.

Moreover, thanks to the previous tightening algorithms, we are also able to filter the domains of the decision variables corresponding to tasks of $\overline{\mathcal{T}_j}$ whenever the bounds of the resource cost variable c_j are updated:

- If the upper bound $\overline{c_j}$ of c_j is modified:
  - if $\overline{c_j} < \text{lb}_j$, a failure is triggered;
  - if $\overline{c_j} < \text{ub}_j$, we try to determine if some tasks of $\overline{\mathcal{T}_j} \setminus \underline{\mathcal{T}_j}$ (i.e. the set of possible tasks not yet assigned) must be added: for each task t_i of this set, if its removal entails a new lower bound lb'_j s.t. $\text{lb}'_j > \overline{c_j}$, then the task must be assigned to r_j, i.e. $x_i = j$.
- Conversely, if the lower bound $\underline{c_j}$ of c_j is modified:
  - if $\underline{c_j} > \text{ub}_j$, a failure is triggered;
  - if $\underline{c_j} > \text{lb}_j$, we try do determine if some unassigned tasks must be excluded from r_j: if the addition of a task entails a new upper bound ub'_j s.t. $\text{ub}'_j < \underline{c_j}$, then the task must be excluded from r_j, i.e. $x_i \neq j$.

Note that all the modifications of the maintained data structures induced by these tests must be undone whenever the triggering condition is not met.

To sum up, the *IdleCost* optimization constraint achieves Bound Consistency on the cost of a single resource and the filtering of the resource variables of its possible tasks in:

- $O(n^2)$ when a task is assigned: $O(\log n)$ to update the upper bound, $O(n^2)$ to delete the bypassing edges of G and $O(n^2)$ to find a shortest path;
- $O(n^2)$ when a task is removed: $O(n)$ to delete edges and $O(n^2)$ to compute the shortest path;
- $O(n^3)$ when the upper bound of c_j is modified: $O(n)$ shortest paths to compute;
- $O(n^2 \log n)$ when the lower bound of c_j is modified: there might be $O(n)$ uncovered idle times at most for the $O(n)$ tasks.

The overall propagation algorithm has then a worst-case time complexity in $O(n^3)$. However, most of the time, the propagation events and the structure of the GCG of the resource at a given node will not trigger the necessary conditions to meet this worst case.

Nonetheless, the computation of shortest paths, involved in the most costly rules, can also be implemented incrementally. Each time a task t_i is assigned on the resource, let $t_{i'}$ and $t_{i''}$ be the preceding and succeeding assigned tasks (possibly t_0 and t_{n+1}) w.r.t. t_i, then only the shortest path between $v_{i'}$ and $v_{i''}$ needs to be recomputed (therefore the lower bound can be updated incrementally), and the shortest path problem is divided into two independent subproblems between $v_{i'}$ and v_i, and v_i and $v_{i''}$.

The results presented in Section 4.7 were obtained with this incremental algorithm, not detailed here for the sake of brevity. However, this new constraint *IdleCost* and its propagation is only local to a single resource, so the lower bound of the global cost can be $O(m)$ times worse than the actual bound as the uniqueness of task is relaxed. So in the next sections, we present a new CP model based on a much tighter relaxation, considering all resources and all tasks simultaneously.

4.4 Global Propagation of Transition Cost

As mentioned in the previous section, the lower bound of the transition cost obtained with per-resource constraints can be far from the optimal value and we show in this section how to globally propagate the cost, taking all resources into account simultaneously, thanks to the MinWeightAllDiff optimization constraint. We also present several search strategies which take advantage of the MinWeightAllDiff constraint to efficiently guide the search.

4.4.1 Relaxation of FJS to Path Covering

In the GCG, a solution to the FJS problem (as can be seen in Figure 4.4 for an instance with 5 tasks and 3 resources) corresponds to m vertex-disjoint simple paths $(v_j, \ldots, v_{m+n+j})$, $\forall j \in \{1, \ldots, m\}$, joining the opening v_j to the closing v_{m+n+j} of each resource r_j while covering all vertices exactly once. Therefore, an optimal solution to the FJS w.r.t. the cost defined by Equation (4.3) corresponds to a set of m such paths with minimal total length.

More generally, a set of vertex-disjoint simple paths that covers all vertices of an unweighted directed graph is called a *vertex-disjoint Path Cover* (PC), whose objective is to minimize the number of paths [Ntafos and Hakimi,

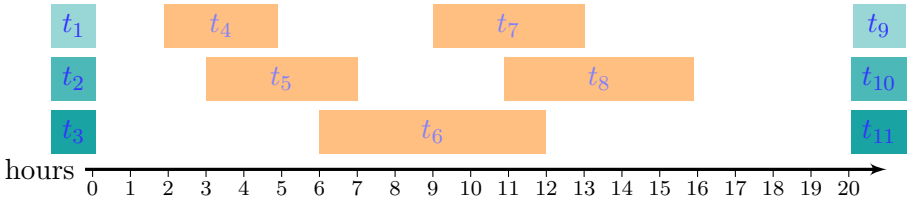


Figure 4.4 – A solution to an instance of FJS.

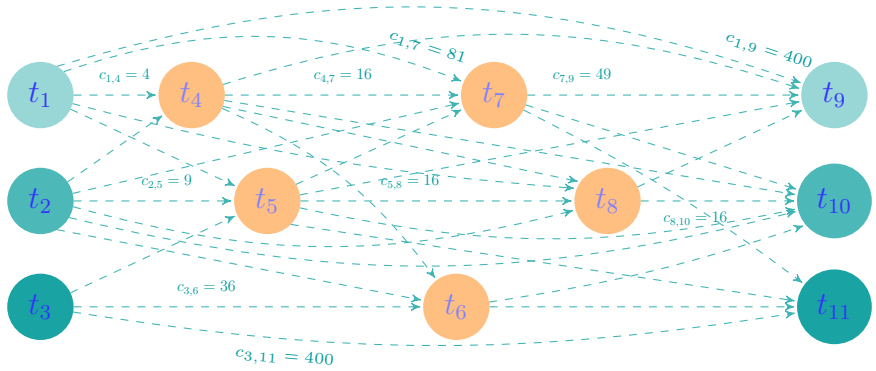


Figure 4.5 – Relaxation of FJS to Minimum Weight Path Cover (MWPC).

1979]. For the GCG of an FJS problem, there must be exactly m paths in any minimal PC, as there are exactly m sources v_j, $\forall j \in \{1, \ldots, m\}$, and m sinks v_{m+n+j} (corresponding to the opening and closing of all resources). So we can use a variant of the PC problem called the *Minimum Weight Path Cover* (MWPC) whose objective is to produce a cover that minimizes the total weight of its edges, as depicted in Figure 4.5.

We then obtain a relaxation of the FJS problem which is much tighter than the one presented in Sections 4.3.3 and 4.3.4 w.r.t. the lower bound of the total cost, as all tasks are scheduled exactly once, instead of possibly $|\mathcal{R}_i| \leq m$ times for each task t_i. For example, an instance with a single task, spanning the entire availability of resources and compatible with all of them, would lead to a 0 lower bound using IdleCost constraints, whereas the exact lower bound $(m-1)(r^e - r^s)^2$ is provided by the MWAD constraint.

However, an MWPC is not in general a solution to FJS, as the consistency on resources along a path is not entailed by the model:

- A path starting at v_j, corresponding to the opening of resource r_j, may end at $v_{m+n+j'}$, with $j' \neq j$, the closing of another resource. E.g. with the instance depicted in Figure 4.6, the path starting at t_3, corresponding to the opening of resource r_3, should end at t_{11}, the closing

of the same resource; however it may end at t_{10}, the closing of r_2, because immediate successors are all compatible along this path, even if the corresponding subset of tasks cannot be assigned on the same resource.

- More generally, if $\gamma(t_i, t_{i'}) \wedge \gamma(t_{i'}, t_{i''})$, nodes v_i, $v_{i'}$ and $v_{i''}$ can succeed each other along the same path, even though it does not imply that $\mathcal{R}_i \cap \mathcal{R}_{i'} \cap \mathcal{R}_{i''} \neq \varnothing$, which is necessary to assign t_i and $t_{i''}$ on the same resource.

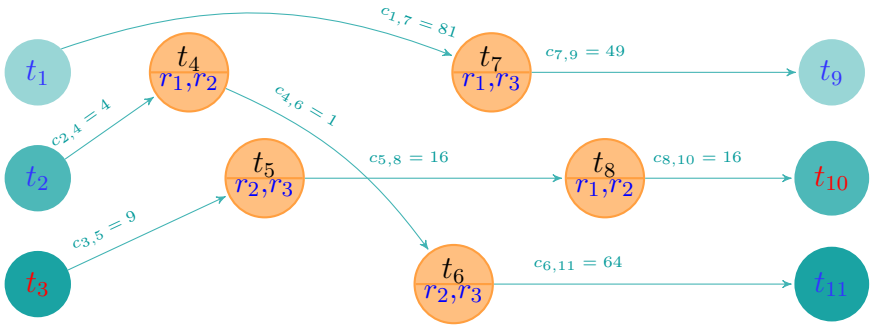


Figure 4.6 – A solution to MWPC is not generally a solution to FJS.

To obtain an exact FJS model, resource variables are kept in our new model and channelling constraints must be added to link them with the *successor* variables introduced in Section 4.4.2, which model the MWPC on the GCG. To implement the minimization constraint between the successor variables and the cost, we first reduce MWPC to the Linear Assignment Problem in the next section.

4.4.2 Successor Variables and Reduction to the Linear Assignment Problem

A PC in the GCG can be modelled by $m + n$ variables y_i that represent the *successor* of each node v_i in V^s, with a domain equal to the indices of its possible successors:

Definition 13 (Successor Variables)**.** *A PC in the GCG of an FJS problem is represented by the following set of* successor variables*:*

$$\mathcal{Y} = \{y_i, \forall v_i \in V^s\}$$

where $\forall y_i \in \mathcal{Y}, d_{y_i} = \{i', \forall v_{i'} \in N^+(v_i)\}$ *is the domain of* y_i*, such that* $y_i = i'$ *iff* $\text{next}(t_i) = t_{i'}$*.*

In the following, we will denote by $\text{dom}(y_i)$ the current domain of successor variable y_i.

Note that the union of the domains of $\mathcal{Y}$ must be equal to the indices of V^e, i.e. $\bigcup_{y_i \in \mathcal{Y}} \text{dom}(y_i) = \{m+1, \ldots, 2m+n\}$, as all actual tasks (with indices in $\{m+1, \ldots, m+n\}$) have at least one predecessor (the openings of its compatible resources) and all closings (with indices in $\{m+n+1, \ldots, 2m+n\}$) as well (its corresponding opening and compatible tasks). To obtain a valid PC, the variables of $\mathcal{Y}$ must have distinct values, such that the assignment is a perfect matching between the $m+n$ variables and the $m+n$ values. So finding an MWPC on G reduces to the *Linear Assignment Problem* (LAP) between $\mathcal{Y}$ and the union of its domains, with cost $\sum_{\forall v_i \in V^s} w((v_i, v_{y_i})) = \sum_{\forall t_i \in \mathcal{T}^s} c_{t_i,\text{next}(t_i)}$, i.e. the cost of the FJS in equation (4.3).

An optimal matching can be obtained in polynomial time, i.e. $O(n(d + m \log m))$ with an efficient enough implementation of the Hungarian algorithm (with n the size of the left set of vertices, m the size of the right one and d the number of edges). We can then model an MWPC with the *MinWeightAllDiff* global optimization constraint introduced by [Caseau and Laburthe, 1997], for which [Sellmann, 2002] describes how the Hungarian algorithm can be used to propagate the *MinWeightAllDiff* constraint, which is able to enforce generalized arc-consistency over the constraint variables representing the end of each task, i.e. the successor of each node of the GCG in our case.

4.4.3 The MinWeightAllDiff Constraint

To constrain the transition cost of FJS, we add to our model a MinWeightAllDiff constraint over the successor variables and the global cost, where the contribution of an assignment $y_i = i'$ is defined by $w'(i, i') = w((v_i, v_{i'})) = c_{t_i,t_{i'}}$ for all ordered pairs of compatible tasks t_i and $t_{i'}$ with $i < i'$:

$$\text{MinWeightAllDiff}(\mathcal{Y}, w', \text{cost}) \tag{4.10}$$

which is satisfied iff $\text{cost} = \sum_{\forall y_i \in \mathcal{Y}} w'(i, y_i)$ and variables of $\mathcal{Y}$ have distinct values. We will denote by $\{lb, \ldots, ub\}$ the current domain of the cost.

[Sellmann, 2002] describes how the Hungarian algorithm can be used to compute lb, then how the variables of $\mathcal{Y}$ can be pruned to withdraw values that only belong to assignments exceeding ub. To our knowledge, there is no implementation of the MinWeightAllDiff constraint in any of the main publicly available CP solvers. Therefore, we implemented an incremental version of the Hungarian algorithm for the FaCiLe OCaml constraint library [Barnier and Brisset, 2001], based on the C++ source code of [Payor, 2017], to compute lb in $O(|\mathcal{Y}|d)$, with $|\mathcal{Y}| = n + m$ and $d = \sum_{\forall y_i \in \mathcal{Y}} |\text{dom}(y_i)| = |E|$.

Our constraint propagates only when edges belonging to the previous minimal matching are removed: the remaining assignments are kept and augmented until a new perfect matching is obtained, then the optimal matching is computed and a failure is triggered if its weight is strictly greater than ub.

As the results obtained with the computation of lb alone outperformed by orders of magnitude our previous approach, we postponed to a later study the implementation of the pruning of $\mathcal{Y}$ described in [Sellmann, 2002] and the assessment of potential additional speed-ups for FJS. Note that the MinWeightAllDiff constraint is intended for constraint programs that minimizes the assignment cost, so the constraint does not compute ub (nor prunes successor values w.r.t. lb). However, the same algorithm could be used to implement a maximization version and achieve Bound Consistency for the cost and GAC for $\mathcal{Y}$ with the same complexity.

However, as mentioned in Section 4.4.1, a solution to the LAP (therefore an MWPC), is generally not a solution to the FJS problem and variables of $\mathcal{Y}$ must also be constrained with the resource variables $\mathcal{X}$ as described in the next section.

4.4.4 Channelling Constraints

To obtain valid FJS solutions, we have to keep the AllDifferent constraints on the cliques of overlapping tasks (see Section 4.3.1) to prevent incompatible tasks to be assigned on the same resource. Moreover, implication constraints that ensure *resource consistency* along each path of an MWPC must be added to link the successor variables and the resource ones. Furthermore, a task succeeds to another one iff the *hole* (see Definition 14) between them is empty, so additional propagation rules can be associated with each resource to improve our model.

Resource Consistency

As mentioned in Section 4.4.1, each path of a PC must connect the opening t_j to the closing t_{m+n+j} of a resource r_j, and all tasks in-between must be assigned to r_j to obtain a valid FJS solution. Therefore, the following channelling constraints must be added to our model to ensure that successive tasks are assigned to the same resource:

$$y_i = j \;\Rightarrow\; x_i = x_j, \quad \forall y_i \in \mathcal{Y}, \forall j \in \mathrm{dom}(y_i) \tag{4.11}$$

where resource variables x_i of $\mathcal{X} = \{x_{m+1}, \ldots, x_{m+n}\}$ are extended to openings, i.e. $\forall j \leq m$, $x_j = j$, and closings, i.e. $x_{m+n+j} = j$, with bound variables. Note that we use *arc-consistent* equality constraints (instead of bound-

consistent ones) on the RHS of the implication to improve the filtering of successor variables whenever the contraposition holds, i.e. $x_i \neq x_j \Rightarrow y_i \neq j$.

Tasks Exclusion

When a successor variable is assigned, i.e. $y_i = i'$, on a known resource, all tasks that could be scheduled between t_i and $t_{i'}$ should be removed from the resource. Conversely, when no task can be scheduled between t_i and $t_{i'}$, the successor variable must be assigned $y_i = i'$.

First, we define the notion of *hole* between assigned tasks to help specify the tasks exclusion constraints:

Definition 14 (Hole)**.** *For any pair of tasks t_i and $t_{i'}$ assigned on resource r_j (i.e. $x_i = x_{i'} = j$) with no assigned task in-between (i.e. $\nexists t_{i''} \in \mathcal{T}_j$ s.t. $x_{i''} = j \wedge t_i^e \leq t_{i''}^s \wedge t_{i''}^e \leq t_{i'}^s$),* hole $\mathcal{H}_j^{i,i'}$ *is defined as the set of all unassigned tasks that fit between t_i and $t_{i'}$:*

$$\mathcal{H}_j^{i,i'} = \{t_{i''}, \forall t_{i''} \in \mathcal{T}_j \text{ s.t. } j \in \mathrm{dom}(x_{i''}) \wedge t_i^e \leq t_{i''}^s \wedge t_{i''}^e \leq t_{i'}^s\}$$

We denote by $\mathcal{H}_j$ the set of all holes of resource r_j.

For each resource r_j, we introduce a new *TaskExclusion* constraint on $\mathcal{X}_j$ and $\mathcal{Y}_j$, the restrictions of $\mathcal{X}$ and $\mathcal{Y}$ to $\mathcal{T}_j$, to remove j from the resource variables of the hole between connected tasks and, conversely, to chain the sides of holes that become empty:

$$\mathrm{TaskExclusion}(r_j, \mathcal{X}_j, \mathcal{Y}_j), \quad \forall r_j \in \mathcal{R} \tag{4.12}$$

which is satisfied when $y_i = i' \Leftrightarrow \mathcal{H}_j^{i,i'} = \varnothing$, $\forall \mathcal{H}_j^{i,i'} \in \mathcal{H}_j$.

$\forall y_i \in \mathcal{Y}_j$ s.t. $y_i = i'$, $x_i = j$ and $x_{i'} = j$, it holds that $\forall t_{i''} \in \mathcal{H}_j^{i,i'}$, $x_{i''} \neq j$ and, conversely, $\forall \mathcal{H}_j^{i,i'} \in \mathcal{H}_j$, it holds that $\mathcal{H}_j^{i,i'} = \varnothing \Rightarrow y_i = i'$. Therefore, to propagate constraint (4.12), the following rules are triggered upon successor assignment or resource modification (with $t_{i''} \in \mathcal{H}_j^{i,i'}$ initially), which are depicted in Figures 4.7 and 4.8:

- $y_i = i' \Rightarrow x_{i''} \neq j, \forall t_{i''} \in \mathcal{H}_j^{i,i'}$: to propagate successor assignment $y_i = i'$, we remove j from the resource variables of all tasks $t_{i''} \in \mathcal{H}_j^{i,i'}$, i.e. $x_{i''} \neq j$;
- $x_{i''} = j \Rightarrow (\mathcal{H}_j^{i,i''} = \varnothing \Rightarrow y_i = i'') \wedge (\mathcal{H}_j^{i'',i'} = \varnothing \Rightarrow y_{i''} = i')$: if a hole becomes empty on either side of a task upon resource assignment, i.e. $x_{i''} = j$, then both sides of the holes are connected, i.e. $y_i = i''$ if the left hole is empty, $y_{i''} = i'$ if the right hole becomes empty;

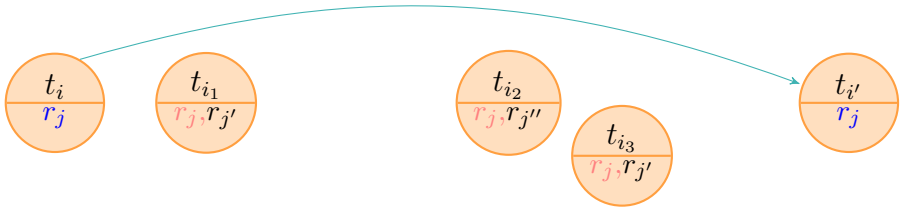


Figure 4.7 – If successor variable is assigned on a known resource r_j, then possible in-between tasks t_{i_1}, t_{i_2} and t_{i_3} are removed from the resource r_j.

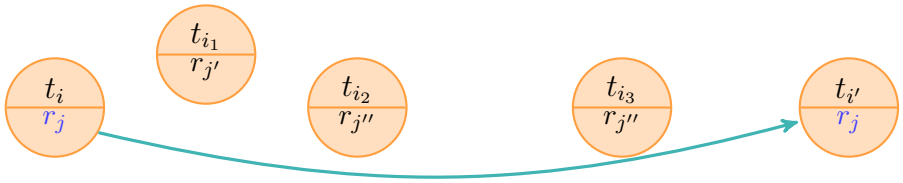


Figure 4.8 – If no task can be scheduled on resource r_j between two already assigned tasks t_i and $t_{i'}$, then successor variable must be assigned, i.e. $y_i = i'$.

- $x_{i''} \neq j \Rightarrow (\mathcal{H}_j^{i,i'} = \varnothing \Rightarrow y_i = i')$: if the last task of a hole is removed from the resource, i.e. $x_{i''} \neq j$, then both sides of the hole are connected.

For each resource, the non-empty holes of $\mathcal{H}_j$ (initialized to $\{\mathcal{H}_j^{j,m+n+j}\}$, with $\mathcal{H}_j^{j,m+n+j} = \mathcal{T}_j \backslash \{t_j, t_{m+n+j}\}$) can be maintained in logarithmic time with a Binary Search Tree[5] (BST) upon a successor assignment (removal of a hole) and a task assignment (a hole must be divided in two) or exclusion (the task must be removed from its hole). This set can be maintained and queried in $O(\log|\mathcal{T}_j|)$ to remove a task $t_{i''}$ from a hole $\mathcal{H}_j^{i,i'}$ (implemented as a hash table) when j is filtered from $x_{i''}$ and to exclude j from the resource variables of a hole upon assignment. When a task of a hole $\mathcal{H}_j^{i,i'}$ is assigned on r_j, $\mathcal{H}_j^{i,i'}$ must be split in two new nodes, which can be done in $O(\log|\mathcal{T}_j|+|\mathcal{H}_j^{i,i'}|)$. However, we only implemented a simple linear algorithm to query holes, letting the implementation of the BST for a later study, as a similar optimization of our previous model only improved resolution times by less than 10 %.

4.4.5 Search Strategies

This section presents several search strategies to optimize the robustness of the GAP instances solved in Section 4.7, i.e. to balance the idle times over time and resources. As an MWPC is a tight relaxation of FJS, all our

[5]Lexicographically ordered by pair $(t_i^e, t_{i'}^s)$ for each hole $\mathcal{H}_j^{i,i'}$.

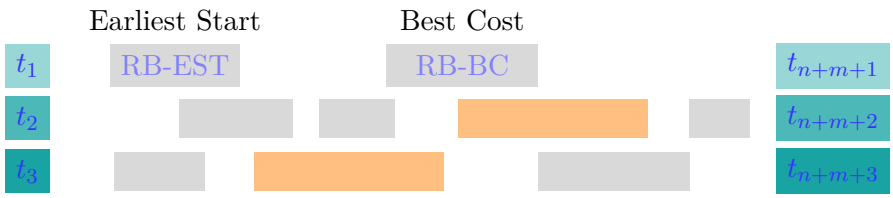


Figure 4.9 – On this partial solution (where assigned tasks are in orange and unassigned ones are in grey), the first resource is currently the least loaded, so the Resource Balancing strategy would select either task RB-EST or RB-BC.

strategies follow the optimal assignment computed by the MinWeightAllDiff constraint when trying to assign a successor variable.

The simplest strategy aims at selecting the least loaded resource, while the second one generalizes the notion of resources to non-empty holes (see Definition 14). Then both first assign a task on the resource or hole, before assigning its successor according to the optimal matching. The third strategy simply follows the optimal matching on one of the side of the best selected hole. As no single strategy was robust enough to efficiently solve our various instances, we eventually obtained the best results with their parallel cooperation.

Resource Balancing

For many simple instances of FJS, we observed that the resource loads were balanced in optimal schedules. So the *Resource Balancing* (RB) strategy (illustrated in Figure 4.9) selects the least loaded resource r_j, i.e. with the greatest amount of idle time, among the ones where unassigned tasks can still be scheduled. Then it assigns to r_j either the task with the *Earliest Start Time* (EST) or the one that improves the cost the most, call *Best Cost* (BC), before assigning the successor variable of the selected task according to the optimal matching computed by the MinWeightAllDiff constraint. Note that this second assignment contributes to the reduction of the search space as one of the channelling constraints (4.11) will be able to propagate. We will denote these variants by:

- **RB-EST**: choose least loaded resource, assign earliest task;
- **RB-BC**: choose least loaded resource, assign task with best cost.

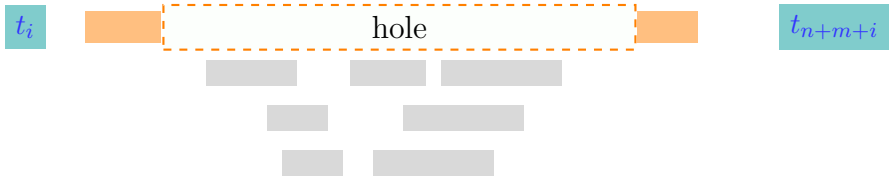


Figure 4.10 – A partial solution (where assigned tasks are in orange and unassigned ones are in grey) with a non-empty hole on resource r_i.

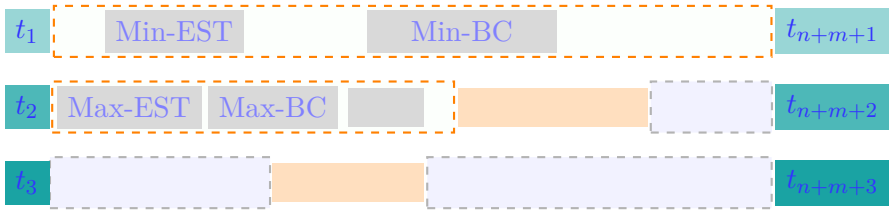


Figure 4.11 – On this partial solution (where assigned tasks are in orange and unassigned ones are in grey), the Critical Hole strategy would either select the hole of resource r_1 (CH-Min) or the one of resource r_2 (CH-Max), then choose among its possible tasks the one with earliest start time (EST) or best cost (BC).

Critical Hole

In order to better take into account the structure of partial solutions where blocks of successive tasks can be discarded, we may consider each non-empty hole (see Definition 14 and Figure 4.10) as an independent resource. So the *Critical Hole* (CH) strategy (illustrated in Figure 4.11) selects the one (among all resources) with the fewest (called *Min*) or the most (called *Max*) tasks, using the largest span to break ties, rather than the least loaded resource of strategy RB. Then it unfolds identically. This leads to four variants:

- **CH-Min-EST**: choose hole with fewest tasks, assign earliest task;
- **CH-Min-BC**: choose hole with fewest tasks, assign task with best cost;
- **CH-Max-EST**: choose hole with most tasks, assign earliest task;
- **CH-Max-BC**: choose hole with most tasks, assign task with best cost.

Task Chaining

Instead of selecting a new task to insert into the critical hole based on its start time or its impact on the cost as the previous strategy, the *Task Chaining*

(TC) strategy simply follows the optimal matching provided by the MinWeightAllDiff constraint from the beginning of the hole, or backward from its end, by assigning the corresponding successor variable. We then choose to extend the side which reduces the most the span of the hole, called *Best Extension* (BE), or the one corresponding to the successor variable with the smallest domain, called *Next Size* (NS), which gives four variants:

- **TC-Min-BE**: choose hole with fewest tasks, assign task with best hole reduction;
- **TC-Min-NS**: choose hole with fewest tasks, assign task with smallest domain;
- **TC-Max-BE**: choose hole with most tasks, assign task with best hole reduction;
- **TC-Max-NS**: choose hole with most tasks, assign task with smallest domain.

Furthermore, the reduced costs computed by the Hungarian algorithm when propagating the MinWeightAllDiff constraint could be used to guide the search as well, as mentioned by [van Hoeve, 2005].

Parallel Cooperation

Even if the TC-Min-BE strategy gives excellent results with many instances of the GAP (as shown on Figure 4.16), solving them in a few dozens of backtracks, other instances were more time-consuming and better solved by one of the other variants. Therefore, we can build a new strategy that benefits from all of the previous variants by exploring the search space in parallel while exchanging upper bounds between processes, provided there are enough available cores on the computer.

We have developed a cooperative version of FaCiLe, which benefits from the distributed system presented in Section 2.1, provides a parallel search goal that forks its process before applying each strategy and solves the same model while communicating bounds to all the children through their parent whenever a better solution is found. So the parent process corresponds to the *server* of Section 2.1.1 and the children to the *clients*. Note that the parallel cooperation of strategies may be strictly better than any single strategy on some instances as the addition of new upper bounds might shorten its resolution time. This strategy will be denoted by **CP_COOP** in Section 4.7.

4.5 Basic ILP Model

In [Bolat, 2001], the author proposes several mathematical models for the GAP, with different kinds of optimization criteria, in order to capture the robustness of gate assignment solutions towards flight schedule disruptions. In particular, he proposes to minimize the variance of the *idle times* of the gates to evenly distribute the global leeway over resources and time. This criterion allows for short or long pauses that might be necessary in some instances without hindering the optimization of the rest of the schedule.

An idle time is defined as a time period, possibly of null duration, between two successive flights assigned to the same gate and during which the gate is unoccupied. The underlying idea of this minimization criterion is that the more the flights are regularly distributed to gates, the less the corresponding gate allocation solution should be sensitive to schedule disruptions.

Indeed, even if the number and the sum of the idle times is considered to be constant in [Bolat, 2001], because the arriving and departing times of flights are fixed, their respective durations directly depend on the gate assignment. Hence, the more the idle times are balanced w.r.t. their durations, the smaller the sum of their squares (or any other convex function). The author proposes an Integer Linear Programming (ILP) model to solve the GAP with the sum of the idle times squares as minimization criterion, with $O(mn^2)$ binary variables $z_{i,j,k}$ stating that flight t_j is the successor of flight t_i on gate r_k.

4.5.1 Tasks and Resources Compatibility

In this section, we define a *ternary* compatibility predicate, adapted from the one of Definition 10, to help define compact ILP models in Sections 4.5 and 4.7.

For a given instance, all the transition costs c_{t_i,t_j}, $\forall i, j \in \{0, \ldots, n+1\}$, $i < j$, need not be defined, but only for the ordered pairs of indices of *compatible* tasks, i.e. that have at least a possible resource in common and do not overlap. In Section 4.2.6, we have defined a *binary compatibility* predicate $\gamma(t_i, t_{i'})$, which indicates when t_i and $t_{i'}$ are *compatible*. However, as decision variables of ILP models are defined by a triplet of indices (of two possibly successive tasks and a given resource), we define a new *ternary compatibility* predicate to specify the corresponding compatible triplets of indices and allow the ILP models described in the following sections to discard unnecessary decision variables:

Definition 15 (Ternary Compatibility Predicate)**.** *The* ternary compatibility *predicate* $\gamma' : \{0, \ldots, n\} \times \{1, \ldots, n+1\} \times \{1, \ldots, m\} \mapsto \mathbb{B}$ *on two ordered*

task indices i and j and a resource index k is defined by:

$$\forall i \in \{0,\ldots,n\},\ \forall j \in \{i+1,\ldots,n+1\},\ \forall k \in \{1,\ldots,m\},$$
$$\gamma'(i,j,k) = \begin{cases} r_k \in \mathcal{R}_j & \textit{if } i = 0 \wedge j \leqslant n \textit{ (first task)} \\ r_k \in \mathcal{R}_i & \textit{if } i > 0 \wedge j = n+1 \textit{ (last task)} \\ \textit{true} & \textit{if } i = 0 \wedge j = n+1 \textit{ (no task)} \\ r_k \in \mathcal{R}_i \cap \mathcal{R}_j \wedge t_i^e \leqslant t_j^s & \textit{otherwise (pairs of tasks)} \end{cases}$$

t_i, t_j and r_k are then said to be compatible.

In the following sections, we reformulate the ILP model of [Bolat, 2001], taking into account the heterogeneity of the gates and discarding variables that can statically be set to 0 because of an incompatibility, which was not explicitly done in the article.

4.5.2 Decision Variables

A binary variable is defined to denote the successive assignment of a pair of compatible tasks to the same compatible resource:

$\forall i \in \{1,\ldots,n\}, \forall j \in \{i+1,\ldots,n\}, \forall k \in \{1,\ldots,m\}$, s.t. $\gamma'(i,j,k)$,

$$z_{i,j,k} = \begin{cases} 1 & \text{if tasks } t_i \text{ and } t_j \text{ are successively assigned to } r_k \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

$\forall j \in \{1,\ldots,n\}, \forall k \in \{1,\ldots,m\}$, s.t. $\gamma'(0,j,k)$,

$$z_{0,j,k} = \begin{cases} 1 & \text{if } t_j \text{ is the first task assigned to resource } r_k \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

$\forall i \in \{1,\ldots,n\}, \forall k \in \{1,\ldots,m\}$, s.t. $\gamma'(i,n+1,m)$,

$$z_{i,n+1,k} = \begin{cases} 1 & \text{if } t_i \text{ is the last task assigned to resource } r_k \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

$\forall k \in \{1,\ldots,m\}$,

$$z_{0,n+1,k} = \begin{cases} 1 & \text{if there is no task assigned to resource } r_k \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

If the instance is feasible, there is exactly one $z_{i,j,k}$ variable equal to 1 for each task t_i, so the allocation can trivially be deduced from the decision variables:

$$\forall t_i \in \mathcal{T},\ \exists!\, j \in \{i+1, \ldots, n\},\ \exists!\, k \in \{1, \ldots, m\},\ \text{s.t. } z_{i,j,k} = 1$$

Note that variables $z_{i,j,k}$ are similar to the ones of typical ILP models of the *Multi-commodity Flow Problem* [Wang, 2018], where they are interpreted as the amount of a commodity k flowing through an arc from node i to node j in a network. Nevertheless, as shown hereafter, the model is not formulated as a flow problem model.

4.5.3 Constraints

A solution to the GAP must cover all tasks, ensure that the same resource is used by successive tasks and that tasks do not overlap, and be integral.

Covering As each task has to be assigned to exactly one compatible resource, the following constraints are introduced to ensure that exactly one task is assigned immediately before a given task t_j (Equation (4.17)), and only one task is assigned immediately after a given task t_i (Equation (4.18)):

$$\forall j \in \{1, \ldots, n\}, \sum_{\substack{i \in \{0,\ldots,j-1\}, k \in \{1,\ldots,m\} \\ \gamma'(i,j,k)}} z_{i,j,k} = 1 \tag{4.17}$$

$$\forall i \in \{1, \ldots, n\}, \sum_{\substack{j \in \{i+1,\ldots,n+1\}, k \in \{1,\ldots,m\} \\ \gamma'(i,j,k)}} z_{i,j,k} = 1 \tag{4.18}$$

Moreover, Equation (4.19) is added to specify that there are exactly $n + m$ binary variables that should be equal to 1 in a feasible solution:

$$\sum_{\substack{i \in \{0,\ldots,n\}, j \in \{i+1,\ldots,n+1\}, k \in \{1,\ldots,m\} \\ \gamma'(i,j,k)}} z_{i,j,k} = n + m \tag{4.19}$$

Resource Consistency The resource used by several successive tasks must be the same. So if a pair of tasks (t_i, t_j) with $j \leqslant n$ (i.e. t_i is not the last task on its resource) are successively assigned to a resource r_k, there must be a task $t_{j'}$ with $j' \in \{j+1, \ldots, n+1\}$ such that the pair of tasks $(t_j, t_{j'})$ are also successively assigned to r_k, but there cannot be any task $t_{j'}$ such that the pair of tasks $(t_j, t_{j'})$ are successively assigned to a resource $r_{k'} \neq r_k$. This consistency constraint is modeled with Equation (4.20),

which states that, for any pair of tasks (t_i, t_j) and any resource r_k, either $z_{i,j,k} = 1$, and then no $z_{j,j',k'}$ with $j' > j$ and $k' \neq k$ can be equal to 1 ($\sum_{k' \in \{1,\ldots,m\} | k' \neq k, j' \in \{j+1,\ldots,n+1\}} z_{j,j',k'} = 0$); or $z_{i,j,k} = 0$, and then at most one $z_{j,j',k'}$ with $j' > j$ and $k' \neq k$ equals to 1 (which is always true due to Equations (4.17) and (4.18)):

$$\forall i \in \{0,\ldots,n-1\}, \forall j \in \{i+1,\ldots,n\}, \forall k \in \{1,\ldots,m\}, \gamma'(i,j,k), \quad z_{i,j,k} + \sum_{\substack{j' \in \{j+1,\ldots,n+1\}, k' \in \{1,\ldots,m\} \setminus \{k\} \\ \gamma'(j,j',k')}} z_{j,j',k'} \leqslant 1 \quad (4.20)$$

Note that these consistency constraints are not formulated as a flow balance constraint but as the linearization of the logical constraint:

$$\forall i \in \{0,\ldots,n-1\}, \forall j \in \{i+1,\ldots,n\}, \forall k \in \{1,\ldots,m\}, \gamma'(i,j,k),$$
$$\forall j' \in \{j+1,\ldots,n+1\}, \forall k' \in \{1,\ldots,m\} \setminus \{k\}, \gamma'(j,j',k'),$$
$$z_{i,j,k} = 1 \Rightarrow z_{j,j',k'} = 0$$

These $\mathcal{O}(mn^2)$ constraints dominate the number of constraints of this model.

Capacity Each resource can execute at most one task at a time. Due to the previous consistency constraints, it is sufficient to check that each resource should have at most one assigned first task:

$$\forall k \in \{1,\ldots,m\}, \sum_{\substack{j \in \{1,\ldots,n+1\} \\ \gamma'(0,j,k)}} z_{0,j,k} \leqslant 1 \quad (4.21)$$

Domain As mentioned above, the decision variables are binary:

$$\forall i \in \{0,\ldots,n\}, \forall j \in \{i+1,\ldots,n+1\}, \forall k \in \{1,\ldots,m\}, \gamma'(i,j,k), \quad z_{i,j,k} \in \{0,1\} \quad (4.22)$$

4.5.4 Objective

As previously mentioned, the objective function proposed in [Bolat, 2001] consists in minimizing the sum of the squares of the idle times, in order to find robust solutions towards operational arrival and departure delays. Therefore, the following minimization criterion is proposed:

$$\min \sum_{\substack{i \in \{0,\ldots,n\}, j \in \{i+1,\ldots,n+1\}, k \in \{1,\ldots,m\} \\ \gamma'(i,j,k)}} I_{i,j}^2 z_{i,j,k} \quad (4.23)$$

where $I_{i,j}$ is the idle time duration between two successive tasks t_i and t_j assigned to the same resource. Note that any positive transition cost, possibly non-uniform (i.e. which depends on the resource), could be used instead.

4.6 Flow Model

As mentioned in the previous section, the FJS is similar to a *Minimum-Cost Flow Problem* (MCFP) [Wang, 2018], more precisely an *Unsplittable Multi-commodity Flow Problem* (UMFP) [Alvelos and De Carvalho, 2003] with additional constraints, which translates into a tighter and smaller ILP model. We first introduce the corresponding graph flow model and then describe its translation to ILP as an improvement of the basic model constraints.

4.6.1 Graph Model

The FJS can be seen as a kind of Unsplittable Multi-commodity Flow Problem (UMFP) [Alvelos and De Carvalho, 2003] with unit capacity for each arc and unit demand for each commodity. But our problem is defined on a directed *multigraph* $G = (V, E)$ with *parallel arcs* between nodes v_i and v_j (corresponding to tasks t_i and t_j) whenever $t_i^e \leqslant t_j^s \ \wedge \ |\mathcal{R}_i \cap \mathcal{R}_j| > 1$ that restrict the compatible commodities rather than their amount. Arcs are *labeled* with the corresponding compatible resource index k (represented by a distinct color in the example of Figure 4.12) s.t. $\gamma'(i, j, k)$, and are specified by triplets of $V \times V \times \{1, \dots, m\}$. More classically, the opening of resources are the *sources* and their closing the *sinks*.

Definition 16 (Compatibility (Multi)Graph)**.** *Given an FJS instance* $G = (V, E, w, d)$ *with* weight $w : E \mapsto \mathbb{R}_{\geqslant 0}$ *and* supply $d : V \mapsto \mathbb{R}$*, its* compatibility (multi)graph *is defined by:*

- $V = V_{\mathcal{T}} \cup V_{\mathcal{R}}^s \cup V_{\mathcal{R}}^e$*:*

$$V_{\mathcal{T}} = \{v_i, \forall i \in \{1, \dots, n\}\} \tag{4.24a}$$
$$V_{\mathcal{R}}^s = \{v_0^k, \forall k \in \{1, \dots, m\}\} \tag{4.24b}$$
$$V_{\mathcal{R}}^e = \{v_{n+1}^k, \forall k \in \{1, \dots, m\}\} \tag{4.24c}$$

  $V_{\mathcal{T}}$ corresponds to regular tasks, $V_{\mathcal{R}}^s$ to fictive openings of the resources and $V_{\mathcal{R}}^e$ to their endings, and $|V| = 2m + n$.

- $E = E_{\mathcal{T}} \cup E^s_{\mathcal{R}} \cup E^e_{\mathcal{R}} \cup E_{\mathcal{R}}$:

$$E_{\mathcal{T}} = \{(v_i, v_j, k),\ \forall i \in \{1, \ldots, n-1\},\ \forall j \in \{i+1, \ldots, n\},\ \forall k \in \{1, \ldots, m\}\ s.t.\ \gamma'(i,j,k)\} \quad (4.25a)$$

$$E^s_{\mathcal{R}} = \{(v^k_0, v_j, k),\ \forall j \in \{1, \ldots, n\},\ \forall k \in \{1, \ldots, m\},\ s.t.\ r_k \in \mathcal{R}_j\} \quad (4.25b)$$

$$E^e_{\mathcal{R}} = \{(v_i, v^k_{n+1}, k),\ \forall i \in \{1, \ldots, n\},\ \forall k \in \{1, \ldots, m\},\ s.t.\ r_k \in \mathcal{R}_i\} \quad (4.25c)$$

$$E_{\mathcal{R}} = \{(v^k_0, v^k_{n+1}, k),\ \forall k \in \{1, \ldots, m\}\} \quad (4.25d)$$

$E_{\mathcal{T}}$ corresponds to idle times between two regular successive tasks, $E^s_{\mathcal{R}}$ to idle times between the opening of the resources and their first tasks, $E^e_{\mathcal{R}}$ to the ones between their last tasks and the closing of the resources, and $E_{\mathcal{R}}$ to empty resources. $|E|$ is equal to the number of compatible pairs of (real) tasks plus $2\sum_{k\in\{1,\ldots,m\}} |\mathcal{T}_k|$, the number of arcs from the opening and closing of the resources to compatible tasks, i.e. in $O(2mn + n^2)$ in the worst case.

- *Each arc has a* unit capacity *and a* weight *(or* cost*) w defined by:*

$$w((v_i, v_j, k)) = c_{i,j} = I^2_{i,j}$$

(also valid for start and end vertices v^k_0 and v^k_{n+1}). Note that our model is generic enough to take into account non uniform resources (gates) for which the cost may depend on k.

- *Each node has a* supply *value d defined by:*

$$d(v) = \begin{cases} 1 & \text{if } v \in V^s_{\mathcal{G}} \text{ (openings)} \\ -1 & \text{if } v \in V^e_{\mathcal{G}} \text{ (closings)} \\ 0 & \text{otherwise (i.e. } v \in V_{\mathcal{F}}\text{, regular flights)} \end{cases}$$

As an example, Figure 4.12 represents the graph model of the following FJS instance:

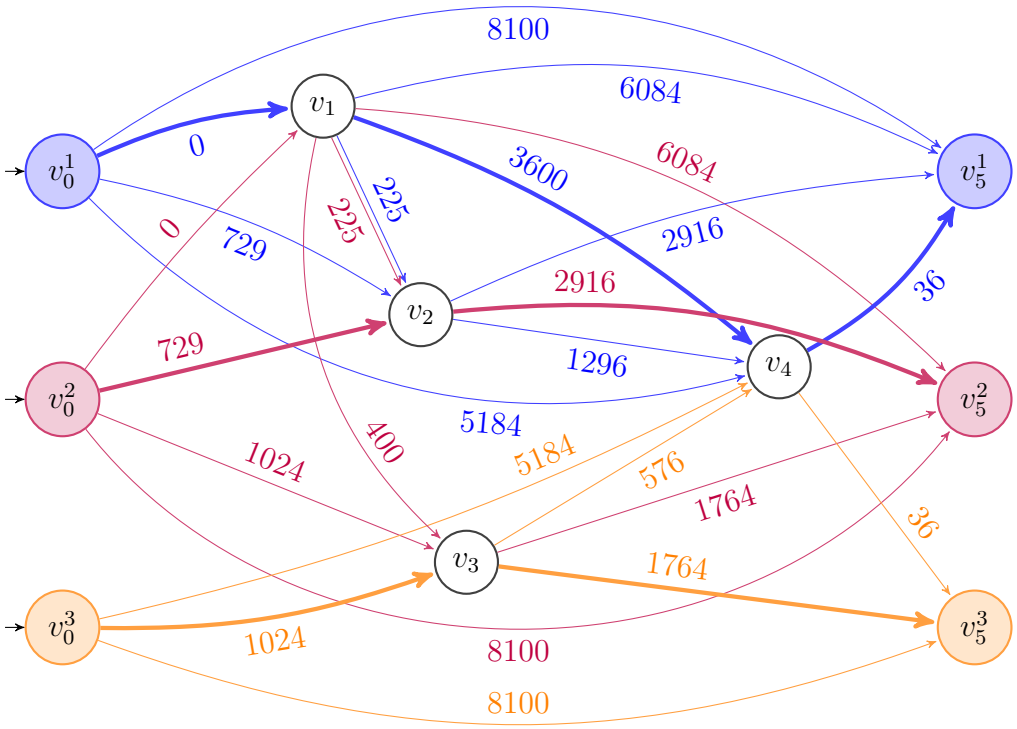


Figure 4.12 – Colored representation of the *compatibility graph* of Example 2 where commodities (or gates) are represented by different colors. Weights (i.e. costs of idle times) are indicated in hundreds of min^2. The flow corresponding to the optimal solution is shown with bold arcs.

Example 2 (FJS Instance with 4 Tasks and 3 Resources)**.** $\mathcal{T} = \{t_1, t_2, t_3, t_4\}$, $\mathcal{R} = \{r_1, r_2, r_3\}$ *with* $r^s = 06:00$, $r^e = 21:00$ *(in hh : mm format):*

i	t_i^s	t_i^e	$\mathcal{R}_i$
1	06 : 00	08 : 00	$\{r_1, r_2\}$
2	10 : 30	12 : 00	$\{r_1, r_2\}$
3	11 : 20	14 : 00	$\{r_2, r_3\}$
4	18 : 00	20 : 00	$\{r_1, r_3\}$

Feasibility Like a standard flow problem, an instance of the FJS is *feasible* iff there exists a *binary flow* $\phi : E \mapsto \{0, 1\}$ such that the imbalance between

outgoing and incoming flows is equal to the supply:

$$\forall v \in V, \quad \sum_{e \in E_v^+} \phi(e) - \sum_{e \in E_v^-} \phi(e) = d(v) \tag{4.26}$$

with $E_v^+ = \{e, \forall e = (u, v, k) \in E\}$ the *incoming* arcs of node v and $E_v^- = \{e, \forall e = (v, w, k) \in E\}$ its *outgoing* ones. However, for the GAP, the flow must also satisfy the following additional constraints.

Additional constraints In general, a solution to a UMFP is not a solution to the FJS, because all internal nodes (corresponding to real flights) must be covered by the flow (Equation (4.27)) and paths must be "monochromatic" (Equation (4.28)), i.e. consistently use the same resource along a path:

- A solution to the FJS must cover all tasks of $\mathcal{T}$, therefore all nodes of $V_\mathcal{T}$ must have a unit inflow and outflow:

$$\forall v \in V_\mathcal{T}, \sum_{e \in E_v^+} \phi(e) = \sum_{e \in E_v^-} \phi(e) = 1 \tag{4.27}$$

- The same resource must be used to enter and leave a node, therefore the flow imbalance of Equation (4.26) (null for internal nodes) must be expressed individually for each resource (or "color"):

$$\forall v_i \in V_\mathcal{T}, \forall r_k \in \mathcal{R}_k, \sum_{(v_{i'}, v_i, k) \in E_v^+} \phi((v_{i'}, v_i, k)) = \sum_{(v_i, v_{i''}, k) \in E_v^-} \phi((v_i, v_{i''}, k)) \tag{4.28}$$

The flow of each edge (v_i, v_j, k) (or (v_0^k, v_j, k), (v_i, v_{n+1}^k, k), (v_0^k, v_{n+1}^k, k)) corresponds to the binary decision variable $z_{i,j,k}$ (or $z_{0,j,k}$, $z_{i,n+1,k}$, $z_{0,n+1,k}$) of the basic ILP model described in Section 4.5.2, thus the assignment of flights to gate can be deduced accordingly.

Objective The cost of a flow is defined by:

$$\sum_{e \in E} w(e)\phi(e) \tag{4.29}$$

As mentioned in the previous paragraph, this cost corresponds to the one of the basic ILP model described in Section 4.5.4, and is therefore also equal to the transition cost. Figure 4.13 depicts the Gantt diagram of an optimal solution to Example 2 with cost: cost $= 10069 \times 100\text{min}^2$.

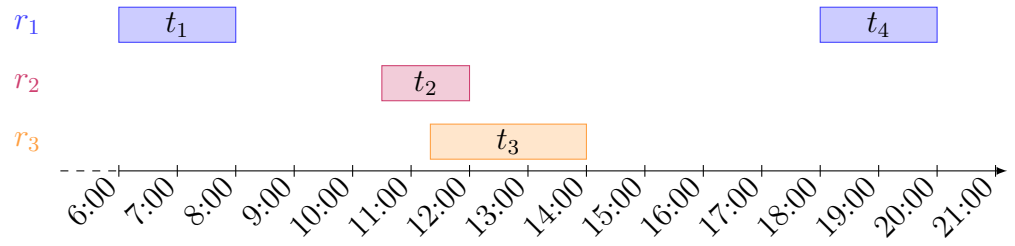


Figure 4.13 – Gantt diagram of the optimal solution to Example 2.

4.6.2 ILP Model

The previously described FJS model as a UMFP with additional constraints can be translated in an ILP model equivalent to the basic model of Section 4.5, based on the same $z_{i,j,k}$ binary decision variables and cost expression but with tighter and fewer constraints. The UFMP flow imbalance constraints of Equation (4.26) ensures that all sources (or resource openings) have a unit outflow as their supply value is 1:

$$\forall k \in \{1, \ldots, m\}, \quad \sum_{\substack{j \in \{1,\ldots,n+1\} \\ \gamma'(0,j,k)}} z_{0,j,k} = 1 \tag{4.30}$$

and that all sinks (resource closings) have a unit inflow (supply value of -1):

$$\forall k \in \{1, \ldots, m\}, \quad \sum_{\substack{i \in \{0,\ldots,n\} \\ \gamma'(i,n+1,k)}} z_{i,n+1,k} = 1 \tag{4.31}$$

which amounts to $O(m)$ constraints.

The covering constraints of Equation (4.27) for nodes of $V_{\mathcal{T}}$ (i.e. regular tasks) lead to Equation (4.18) of the basic model, i.e. all internal nodes have a unit outflow:

$$\forall i \in \{1, \ldots, n\}, \quad \sum_{\substack{j \in \{i+1,\ldots,n+1\}, k \in \{1,\ldots,m\} \\ \gamma'(i,j,k)}} z_{i,j,k} = 1$$

which amounts to $O(n)$ constraints.

More specifically, the constraints of Equation (4.28) ensuring that the path of a commodity is monochromatic, can be directly translated to:

$$\forall i \in \{1, \ldots, n\}, \ \forall k \in \{1, \ldots, m\}, \quad \sum_{\substack{i' \in \{0,\ldots,i-1\} \\ \gamma'(i',i,k)}} z_{i',i,k} = \sum_{\substack{i'' \in \{i+1,\ldots,n+1\} \\ \gamma'(i,i'',k)}} z_{i,i'',k}$$

(4.32)

which amounts to $O(mn)$ constraints.

Other constraints of the basic model that ensure that nodes of $V_{\mathcal{F}}$ have a unit inflow (cf. Equation (4.17)), the continuity of path (cf. Equation (4.20)) or the number of variables equal to 1 (cf. Equation (4.19)) would be redundant in our flow model and are therefore omitted. Overall, this model has only $O(nm)$ constraints in the worst case instead of $O(mn^2)$ for the basic one.

4.7 Results

All experiments were carried out on a standard Debian GNU/Linux 9.6 workstation with a 2.0 GHz 16-core processor and 48 GB of RAM, with the OCaml 4.05.0 compiler [Leroy et al., 2017] and Gurobi 8.1 [Gurobi Optimization, 2018]. Note that all execution times and backtrack amounts graphs are plotted with a base-10 logarithmic scale.

In this section, we first describe how we have extracted the traffic demand and the available gates from actual traffic records at Paris-CDG airport in Section 4.7.1. We focus here on optimizing the robustness of the overall schedule, in order to absorb possible deviations from the original schedule due to traffic delays, severe weather conditions, equipment failures, etc. Hence, our version of the GAP is a FJS problem where the sum of the transition costs, defined as the square of idle times (cf. Section 4.2.5), should be minimized. Nevertheless, many of the aforementioned side-constraints or secondary objectives could be easily added to our CP model.

We show in Section 4.7.2 that our new model based on the MinWeightAllDiff constraint outperforms by orders of magnitude our previous model using the *IdleCost* constraint, and compare in Section 4.7.3 the various strategies described in Section 4.4.5. Then we show in Section 4.7.4 that our CP approach not only outperforms the basic ILP model,but also our improved Flow model (cf. Section 4.6) solved by the state-of-the-art solver Gurobi, on instances recorded from real data at Paris-CDG international airport. However, we will also show in the same section the limit of the CP approach with large instances of the densest terminal in Paris-CDG.

As mentioned in Section 4.7.1, the limited number of gates and high density of several terminals make the robustness of gate allocation quite sensible to gate occupancy modification as a result of DMAN processing, traffic delays, etc. Hence, we discuss in Section 4.7.5 the benefits of a robust gate allocation in terms of idle times distribution between the initial traffic and optimal robust solutions.

Table 4.1 – Number of flights and gates (per day on average) w.r.t. the terminal. The density is the ratio of the average number of flights by the number of gates.

Terminal	Flights	Gates	Density
B	15	10	1.5
F	185	27	6.8
J	72	20	3.6
K	53	19	2.8
Q	50	17	3.0

4.7.1 Data

The data sample used for this study is available at:

`http://recherche.enac.fr/~wangrx/ecai_gap`

It consists in:

- An actual traffic demand during a whole heavy month of traffic (July, 2017) at 5 terminals in Paris-CDG airport;
- A set of allowed aircraft types at each gate.

The traffic demand is extracted from actual traffic records: for each flight, the data sample provides the arrival time at the gate, the departure time from the gate, the aircraft type, the runway and the gate used. Table 4.1 gives the number of flights (on average per day) and the number of gates by terminal in this data sample.

In some cases (probably due to some gate-to-gate movements that were not recorded), one of the two times was missing. The data were completed as follows:

- When the arrival time is missing, the flight is considered at the gate from the beginning of the day if there is no other flight previously occupying the same gate. Otherwise, the flight is considered at the gate 30 min before its departure time (which did not cause any gate conflict in the data set).
- In the same way, when the departure time is missing, The flight is considered at the gate until the end of the day if there is no other flight occupying the same gate afterwards. Otherwise, the flight is considered at the gate 30 min after its arrival time (which did not cause any gate conflict in the data set).

Table 4.2 – Number of backtracks and execution time w.r.t. the number of aircraft to prove optimality for ICTAI and MWAD with 7 gates.

# Flights	# backtracks		execution time		speed-up
	ICTAI	MWAD	ICTAI	MWAD	
33	17,925	**68**	1.41 s	**0.06 s**	23
34	373,502	**22**	19.2 s	**0.03 s**	640
35	516,989	**2**	39.6 s	**0.02 s**	1,980
36	6,768,752	**24**	458.1 s	**0.04 s**	11,453

The sets of allowed aircraft types at each gate is also deduced from the actual traffic sample: we consider that the only aircraft types allowed at a given gate are the ones that actually used this gate during the month.

4.7.2 Per-Resource vs. Global CP Models

Table 4.2 and Figure 4.14 compare the performance of the per-resource CP model, named *ICTAI*, and the improved model based on the MinWeightAllDiff optimization constraint, named *MWAD*, with the TC-Min-BE strategy (cf. Section 4.4.5). They show the number of backtracks and execution time (in seconds) to prove optimality w.r.t. the number of aircraft on instances of the GAP with 7 gates. Note that both y-axes are in logarithmic scale.

MWAD systematically outperforms ICTAI by orders of magnitude in terms of backtracks as well as execution time, with speed-ups exceeding 10,000 for the largest instance.

4.7.3 Search Strategies

In this section, we present our results on 150 real instances at Paris-CDG airport and discuss the performance of the search strategies mentioned in Section 4.4.5. Figure 4.15 presents the Gantt diagram of an optimal solution to one of these instances with 20 gates and 78 aircraft at terminal J.

Figure 4.16 gives the percentage of instances at terminal B, J, K and Q, that MWAD is able to optimally solve within 60 s w.r.t. the search strategies described in Section 4.4.5, during the busiest month of 2017 at Paris-CDG airport. In all our tests, Gurobi and FaCiLe were allowed to exploit all 16 cores of our workstation, though the CP solver used only 11, one for the server and ten for the parallel cooperation of the various strategies described in Section 4.4.5.

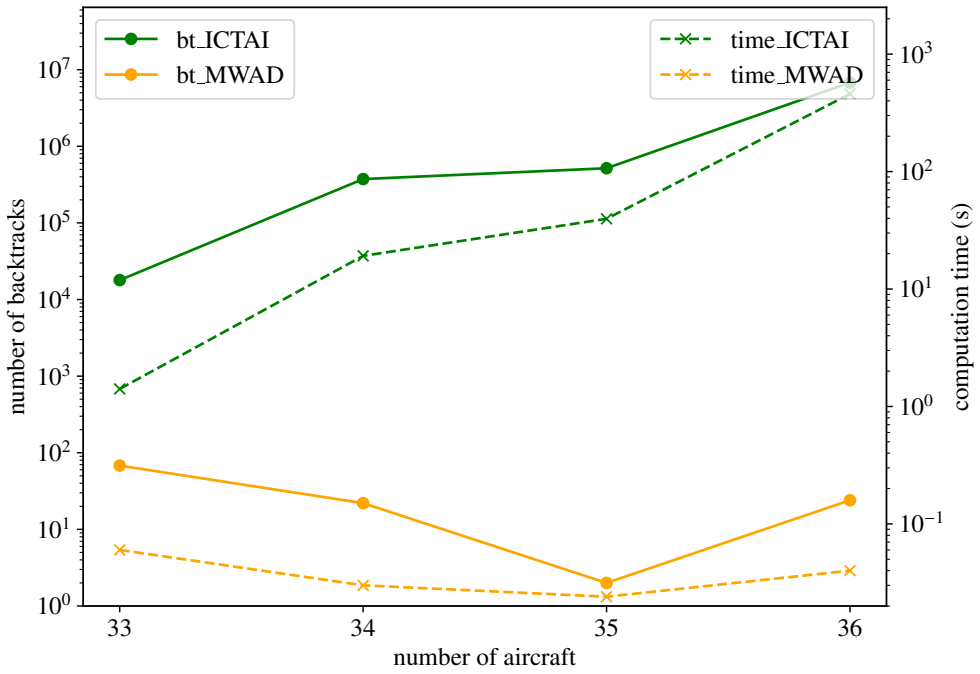


Figure 4.14 – Number of backtracks (solid lines) and execution time in seconds (dashed lines) w.r.t. the number of aircraft to prove optimality for ICTAI and MWAD with 7 gates.

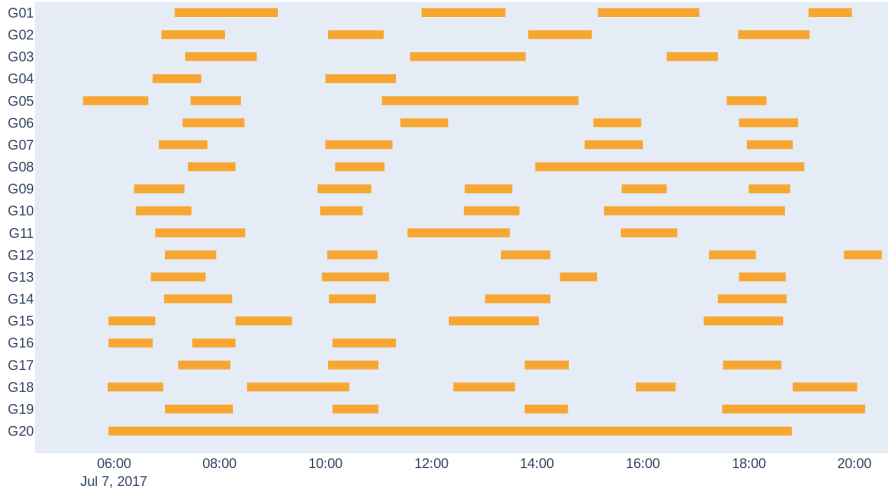


Figure 4.15 – Gantt diagram of an optimal solution to an instance with 20 gates and 78 aircraft at terminal J of Paris-CDG airport.

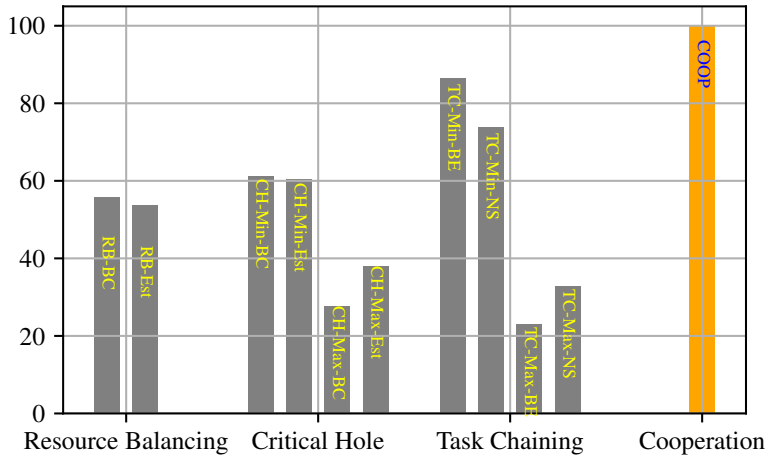


Figure 4.16 – Percentage of instances solved optimally within 60 s by MWAD for all instances at Paris-CDG airport, w.r.t. the strategy.

As expected, the CH and TC strategies are more robust than the simpler RB one, with a success rate of more than 80 % for TC-Min-BE, which is the best variant for our instances. We can also observe that it is more efficient to concentrate the search efforts on holes with the fewest number of compatible flights (or tasks), i.e. the "Min" variants of CH and TC, following the "first-fail" principle, than to schedule more requested ones, as the gate restrictions are generally not too strict and still leave many assignment opportunities. However, TC-Min-BE is not systematically the best strategy, and none of the variants could easily be discarded as each one occasionally obtained the best results on some instances.

In Table 4.3, we compare the cost of the best solutions found by different strategies to the optimal value. Here, 0.00 % represents the optimum, while a larger value corresponds to a suboptimal solution. Similarly to our previous observations, the cooperation systematically reaches an optimal solution, whereas single strategies all fail to do so in some occurrences. For larger instances of terminal J (with 20 gates and more than 70 aircraft), they can even be farther from the optimal value: up to 6.65 % on average with strategy CH-Min-BC, and up to 11.9 % on some particular instances. Note that even though we use 60 s as the time limit, the average time spent by the CP_COOP strategy to solve an instance is much smaller than 60 s, only slightly less than 10 s for the most difficult instances as shown in Figure 4.17.

Table 4.3 – Average gap between the optimal cost and the one of the best solution found within 60 s w.r.t. the strategy.

Strategy	B	K	Q	J
RB-BC	**0.00 %**	+0.12 %	+0.30 %	+1.14 %
RB-Est	**0.00 %**	+0.05 %	+0.26 %	+0.11 %
CH-Min-BC	**0.00 %**	+0.12 %	+0.47 %	+6.65 %
CH-Min-Est	**0.00 %**	+0.62 %	+0.51 %	+4.27 %
CH-Max-BC	**0.00 %**	+4.26 %	+3.10 %	+5.00 %
CH-Max-BC	**0.00 %**	+0.25 %	+0.20 %	+1.29 %
TC-Min-BE	**0.00 %**	**+0.01 %**	+0.07 %	**+0.05 %**
TC-Min-NS	**0.00 %**	+0.05 %	**0.00 %**	+0.55 %
TC-Max-BE	**0.00 %**	+0.11 %	+0.32 %	+0.38 %
TC-Max-NS	**0.00 %**	+0.01 %	+0.30 %	+0.36 %
CP_COOP	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**

4.7.4 CP vs. ILP Models

In this part, we compare our MWAD model, using the CP_COOP strategy and solved with the development version of FaCiLe, to the ILP models described in Sections 4.5 and 4.6 and solved by the Gurobi Commercial Optimizer 8.1.0 [Gurobi Optimization, 2018], with the same real traffic at the Paris-CDG airport than the one used in the previous section to compare search strategies.

Figure 4.17 displays the execution time to prove an optimal solution, averaged over the entire month of July, 2017, w.r.t. the terminals and the models (distinguished by their color). All models are able to reach optimal solutions, and the optimality is always proved in reasonable time, more precisely in less than 10 s with the Flow and MWAD models. The Flow model systematically outperforms its Basic counterpart in terms of computation times, as does the cooperative CP solver, up to 6.2 times faster for terminal J. Moreover the latter also beats our best ILP model consistently. We think that the combination of a tight relaxation (thanks to the MinWeightAllDiff constraint) to compute the lower bound of the global cost, efficient heuristics that follows the corresponding optimal cover, and their parallel cooperation (to be able to solve instances with distinct features), is what enable our CP solver to compete with and outperform a state-of-the-art MIP solver like Gurobi.

Finally, we have also tested our different methods on instances of the densest terminal (F), with more than 180 aircraft and 27 gates. Table 4.4 compares the average gap between the cost of the optimal solution found

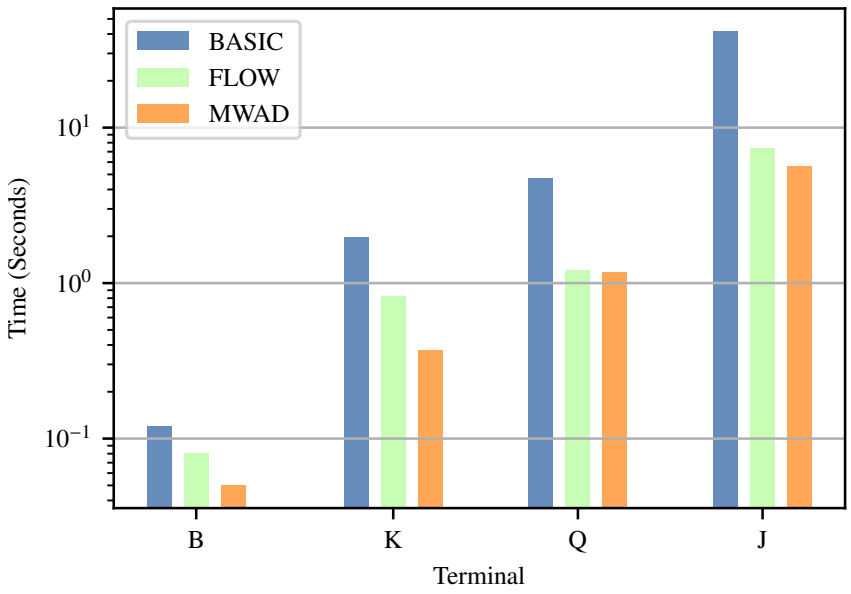


Figure 4.17 – Mean of execution times (in seconds) to prove optimality with the basic ILP model and the Flow ILP model solved by Gurobi, and our new MWAD CP model solved with FaCiLe for instances at Paris-CDG, w.r.t. the terminal.

Table 4.4 – Average gap between the cost of optimal solution and the best solution found with various ILP models and MWAD within different time limits.

Time limit	Basic	Flow	MWAD
30 s	$+\infty$	$+\infty$	**+0.8 %**
60 s	$+\infty$	$+\infty$	**+0.1 %**
120 s	$+\infty$	**+0.0 %**	+0.1 %
300 s	$+\infty$	**+0.0 %**	+0.05 %

by the ILP and CP models within different time limits. We observe that the MWAD solver is still capable to provide a very good solution at less than 0.1 % from the optimal on average within a 60 s time limit, whereas the ILP ones are unable to provide any solution in this time frame. This is due to the fact that Gurobi (like most state-of-the-art ILP solvers) initially performs some internal transformations of the problem (the "presolve" phase) to reduce the search effort afterwards. The Basic ILP model is not even able to find any solution with an extended time limit of 300 s.

However, our CP solver is still unable to consistently reach or prove optimal solutions to such large instances within 300 s, while the Flow ILP solver generally succeeds to do so in less than 120 s, thanks to the rewriting rules and sophisticated heuristics used by Gurobi. Nevertheless, the MWAD solver seems to be adequate in an operational context, especially when quick reassignment of near-optimal quality are needed upon major disturbances that invalidate the current schedule.

4.7.5 Robustness of the Schedule

To assess the robustness of a gate allocation, we have measured the idle times corresponding to the computed solution, as small idle times are likely to cause issues on the day of operations. Table 4.5 shows the average idle time between two successive flights assigned to the same gate (i.e. not counting the idle times at gates openings and closings). The average for the initial allocation recorded on the day of the traffic (named "Initial") is 120 min, which is improved by 37 min with the schedule obtained by our algorithms (named "Robust").

Table 4.5 – Average idle time between two consecutive aircraft at the same gate.

Allocation	Initial	Robust
Average idle time	120 min	157 min

More interestingly, Figure 4.18 shows the distribution of idle times for the entire airport. Those distributions are plotted for the Initial (in blue) and Robust (in orange) allocations. As we can see, the Initial allocation leads to a high number of short idle times: more than 1000 idle times (about 6 %) between 0 and 10 min and more than 1300 idle times (almost 8 %) between 10 and 20 min. The Robust allocation drastically reduces the number of idle times of less than an hour, thus increasing the robustness of the allocation as

expected. With this model, only 150 idle times (less than 1 %) are less than 10 min. The presence of long idle times in all distributions (more than 2 h) comes from the fact that all terminals are aggregated in this figure, even the smallest ones with a very low density.

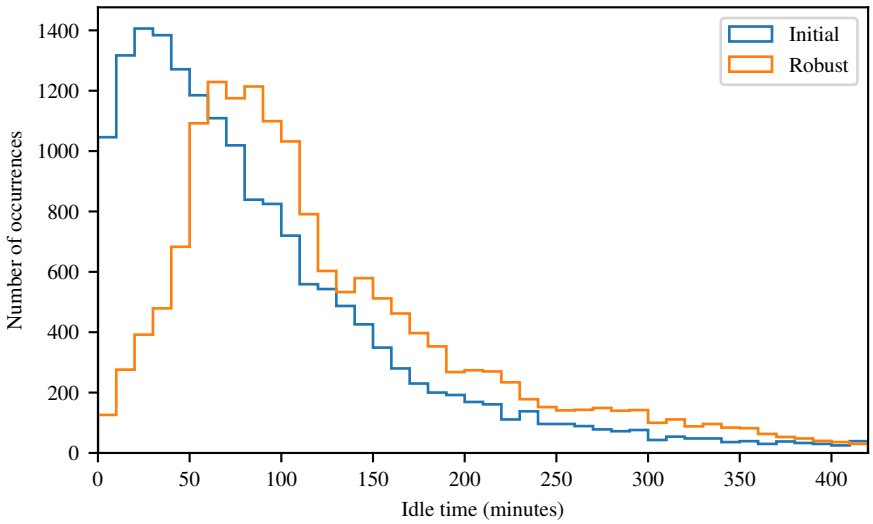


Figure 4.18 – Distribution of idle times for all terminals over all tested instances.

Table 4.6 and Figure 4.19 present similar results, focusing on terminal F only. Even for this particularly busy terminal, the Robust allocation manages to provide solutions that significantly reduce the occurrences of small idle times. Note also that there are very few idle times of more than 2 h, as the density (the ratio between the number of flights and the number of gates) is very high at this terminal.

Table 4.6 – Average idle time between two consecutive aircraft at the same gate for terminal F only.

Allocation	Initial	Flow
Average idle time	65 min	73 min

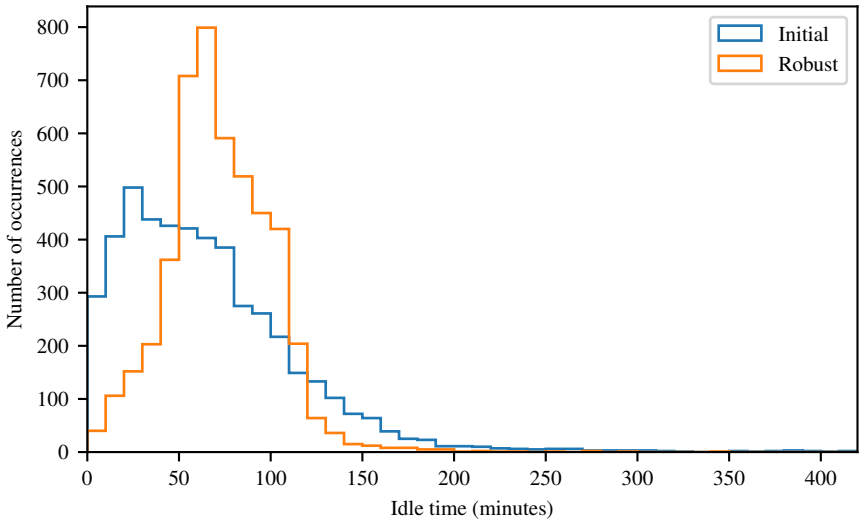


Figure 4.19 – Distribution of idle times at terminal F over all tested instances.

4.8 Conclusion

In this section, we have first presented a new global constraint which improves the efficiency of CP solvers to find and prove robust solutions to the FJS problem that minimize the variance of idle times (or any positive transition cost) as proposed by [Bolat, 2001]. This new optimization constraint, named *IdleCost*, ensures the Bound Consistency of the transition cost associated with each resource and the filtering of the resource variables associated with each possible task.

Then, we propose an improved CP model and search strategies which considerably improves the previous approach to optimize the transition cost of FJS. The main contribution of this novel approach consists in a much better relaxation to constrain the global cost, which simultaneously takes all resources into account instead of independent constraints that propagate the cost for a single resource and may underestimate the lower bound by far. We show that FJS can be relaxed to a Path Covering problem in a DAG and that minimizing the sum of transition costs corresponds to Minimum Weight Path Covering, which can itself be reduced to the Linear Assignment Problem. Therefore, our model is able to compute a much better lower bound of the global cost thanks to the MinWeightAllDiff optimization constraint [Sellmann, 2002] on successor variables. However, a path cover is generally not a solution to FJS and the resource variables of our previous model must be linked by channelling constraints to the successor variables in order to obtain

a valid schedule. Afterwards, we also point out the disadvantages of the ILP model proposed in [Bolat, 2001] and propose a new efficient Minimum Cost Flow model for the FJS problem.

Our latter CP model, implemented with an incremental version of the MinWeightAllDiff constraint for the FaCiLe CP library and solved with a parallel cooperation of various strategies guided by the optimal covering computed by the constraint, outperforms our previous approach by orders of magnitude, as well as the basic ILP model (solved by a state-of-the-art MIP solver) on real instances at Paris-CDG airport of the GAP. When compared with our improved Flow formulation of the ILP model, our parallel CP solver still manages to perform better on most instances of the GAP. However, when confronted to the densest instances, it is much harder for our cooperation strategy to find optimal solution in a reasonable time, whereas the Flow ILP model scales well.

Conclusion and Perspectives

Combinatorial optimization problems in Air Traffic Control and Management become more and more difficult to solve because of the increasing scale of their instances combined with the multiple uncertainties inherent to aviation. On the other hand, the rapid development of multi-core processors and fast networks has made the use of cooperative and parallel computing easier, a technology which can increase the search speed, improve the quality of solutions and allow to solve larger instances.

Contribution

In this thesis, we have studied the generic cooperation of different combinatorial solvers by sharing solutions and optimization bounds in order to speed up the overall resolution process. We have specified a distributed system based on a *client-server* scheme, consisting of a central process (i.e. the server), which acts as a data manager, and any number of solvers (i.e. the clients). To post and retrieve new information, solvers communicate asynchronously through the server which holds the current state of the "solver" (i.e. best solution and bounds). Solvers can execute in parallel on a set of processors, either on a single multi-core computer or over a network. We have also produced high-level client bindings for the main three programming languages used in our code base (C, C++ and OCaml) in order to ease the setup of new cooperative algorithms for our framework. Then, we have focused on three classic optimization methods: Integer Linear Programming (ILP), Constraint Programming (CP) and a metaheuristic, and showed how to adapt them to our framework. Finally, we have applied the resulting parallel solver to the resolution of two large-scale combinatorial problems in the field of air traffic control and management: the en-route conflict resolution problem and the Gate Allocation Problem (GAP) at airports.

To solve the former, we have presented a novel scheme for the modelling and resolution of aircraft conflicts in three dimensions. The model is strictly separated from the resolution, which allows to easily compare optimization

methods on the same instances and make them cooperate. Uncertainties are handled in a realistic manner, and the model is fully generic with respect to the type of maneuvers allowed, which provides ground for further experimentation with real traffic data to validate the whole approach. Realistic instances were generated with up to 60 aircraft and various difficulties and densities. To solve this problem, we have combined a Memetic Algorithm (MA) with an ILP solver as the clients of our cooperation framework, and compared its results with both algorithms alone. For instances of low density, optimality was reached by all three methods in a very short time. For larger instances, a 5 min computation time limit was imposed in order to comply with a real-time context. In this time frame, both the MA and ILP solver were able to provide good solutions, though the latter could no longer prove optimality. As expected, their cooperation outperformed both approaches on all instances, making it possible to reach and prove an optimal solution in most cases, even on very dense instances.

To solve the GAP, a problem which can be seen as a kind of Fixed Job Scheduling (FJS), though with specific requirements or objectives, we have introduced the new *IdleCost* global optimization constraint in the FaCiLe CP library to efficiently find and prove robust solutions by minimizing the variance of idle times between successive flights. This new optimization constraint enforces bound consistency on the transition cost of each resource independently and filters the resource variables associated with each possible task. However, our solver could not compete with a state-of-the-art MIP solver like Gurobi, as per-resource constraints do not offer a good enough relaxation of the problem on the lower bound of the global cost, which could be very far from the optimal. To overcome this issue, we have also proposed a novel approach which simultaneously takes all resources into account, with a much tighter relaxation on the global cost as a Path Covering problem in a directed acyclic graph. We propagate this new model thanks to an incremental version of the MinWeightAllDiff constraint implemented with FaCiLe and solve the GAP/FJS with a parallel cooperation of various strategies that outperforms the previous approach as well as a basic ILP model (solved with Gurobi) on real instances of the GAP at Paris-CDG airport. Finally, we have also pointed out performance issues of the basic ILP model and proposed a new efficient Minimum Cost Flow model for the GAP/FJS problem. When compared with the new proposed ILP model, our parallel CP solver can also perform better on most instances of the GAP. However, when resolving the densest instances, it is much harder for our cooperation strategy to find optimal solution in a reasonable time, whereas the Flow ILP model scales well.

Perspectives

In this section, we present some leads to enhance the performance and usability of the framework, as well as the performance and operational accuracy of our solvers dedicated to ATM problems.

Cooperation Framework

Even though it only carries best solutions, bounds and optimality proof at the moment, our framework is fully generic and could potentially handle any type of information. Future investigations include the sharing of learnt clauses or nogoods (i.e. reasons for failure within tree search) from CP or SAT solvers, as in [Gebser et al., 2011, Chu and Stuckey, 2012], to avoid the pitfalls that were previously identified. Cutting planes from MIP solvers could also be exchanged to forbid the exploration of regions were no solution can be found. To help metaheuristics within this framework, many additional information might also be exchanged, such as partial solutions or subsets of populations (for population-based metaheuristics).

One of the main drawbacks of our cooperation scheme is that the user has to implement as many models as there are types of algorithms. In order to facilitate the setup of the framework on a new problem, it would be much more convenient to write the problem once in an agnostic high-level modeling language and then automatically derive each algorithm-specific model (e.g. CP, MIP, LS, etc.), as described in the Comet programming language [Michel and Van Hentenryck, 2005], the IBM Concert Technology. Among the issues that have to be addressed are the linearization of high-level constraints to feed the MIP solver [Belov et al., 2016] or the definition of a neighborhood for LS algorithms. Moreover, we have seen in Chapter 4 that the best internal model for a solver depends on its resolution algorithm and the expressiveness of the associated language, and often demands a lot of expertise. Therefore the automatic rewriting of models would probably not produce a very efficient parallel solver compared to carefully hand-tailored ones, but it would constitute a simple and robust solver for non-experts.

ATC/ATM Applications

To enhance the technology readiness level of our applications, more detailed models and scenario simulations should be designed. The CATS [Granger and Durand, 2003] en-route simulator and ATOS [Gotteland and Durand, 2003] airport simulator, both developed at ENAC, could be used to integrate

our solvers into an environment that models aircraft performance more accurately and takes many operational constraints into account, such as airspace sectorization or ground traffic congestion. In such a context, the solvers need also to be adapted to a *rolling horizon* mechanism, where subproblems must be iteratively solved on a sliding time window, or more generally to partial reassignments upon a schedule disruption, where a part of the previous solution is kept while the rest must be recomputed with updated constraints.

For the en-route conflict resolution problem, one missing aspect towards an operational decision support tool is the ability to provide an air traffic controller with several alternative solutions. A distance must be defined between trajectories to be able to build a set of solutions that are not too similar to each other, and a suitable GUI should be designed to allow the controller to select the the most convenient one easily.

For the gate allocation problem, the integration of our solver into the ATOS simulator, which already implements runway sequencing of departures and arrivals as well as ground traffic management, would enable to have a global view of the interactions between all these operational tasks, and thus to analyze and refine their interactions. To enhance the performance of our solver further, we could also complete the implementation of the MinWeightAllDiff constraint, which only computes the lower bound of the assignment cost in the current version, but could also filter the domains of successor variables to remove assignments that would lead to exceed the upper bound of the cost, as described in [Sellmann, 2002].

Bibliography

Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Multi-row presolve reductions in mixed integer programming. In *Proceedings of the Twenty-Sixth RAMP Symposium*, Tokyo, October 2014. 63

Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013. 67

Sami Al-Maqtari, Habib Abdulrab, and Ali Nosary. Constraint programming and multi-agent system mixing approach for agricultural decision support system. In *Emergent Properties in Natural and Artificial Dynamical Systems*, pages 197–211. Springer, 2006. 9

Enrique Alba and Bernabé Dorronsoro. *Cellular genetic algorithms*, volume 42. Springer Science & Business Media, 2009. 14

Richard Alligier, Nicolas Durand, and Gregory Alligier. Efficient conflict detection for conflict resolution. In *ICRAT, 8th International Conference on Research in Air Transportation*, 2018. 47, 56

Cyril Allignol, Nicolas Barnier, Pierre Flener, and Justin Pearson. Constraint programming for air traffic management: a survey. *Knowledge Engineering Review*, 27(3):361–392, 2012. 17

Cyril Allignol, Nicolas Barnier, Nicolas Durand, and Jean-Marc Alliot. A new framework for solving en-routes conflicts. In *ATM Seminar, 10th USA/Europe Air Traffic Management R&D Seminar*, 2013. 45, 75

Cyril Allignol, Nicolas Barnier, Nicolas Durand, Alexandre Gondran, and Ruixin Wang. Large scale 3D en-route conflict resolution. In *ATM Seminar, 12th USA/Europe Air Traffic Management R&D Seminar*, 2017. 44, 45, 47

Jean-Marc Alliot, Jean-François Bosc, Nicolas Durand, and Lionnel Maugis. Cats: A complete air traffic simulator. In *16th DASC. AIAA/IEEE Dig-*

ital Avionics Systems Conference. Reflections to the Future. Proceedings, volume 2, pages 8–2. IEEE, 1997. 45

Antonio Alonso-Ayuso, Laureano F. Escudero, and F. Javier Martín-Campo. Collision avoidance in air traffic management: a mixed-integer linear optimization approach. *IEEE Transactions on Intelligent Transportation Systems*, 12(1):47–57, 2011. 45

Antonio Alonso-Ayuso, Laureano F. Escudero, and F. Javier Martín-Campo. An exact multi-objective mixed integer nonlinear optimization approach for aircraft conflict resolution. *Top*, 24(2):381–408, 2016a. 45

Antonio Alonso-Ayuso, Laureano F. Escudero, and F. Javier Martín-Campo. Multiobjective optimization for aircraft conflict resolution. a metaheuristic approach. *European Journal of Operational Research*, 248(2):691–702, 2016b. 45

Filipe Alvelos and JM Valério De Carvalho. Comparing branch-and-price algorithms for the unsplittable multicommodity flow problem. In *International Network Optimization Conference (INOC'2003)*, pages 7–12, Évry/-Paris, October 2003. 111

Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A multicore tool for constraint solving. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. 8

Alejandro Arbelaez and Youssef Hamadi. Improving parallel local search for SAT. In *International Conference on Learning and Intelligent Optimization*, pages 46–60. Springer, 2011. 8

Esther M. Arkin and Ellen B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987. 84

Philippe Averty, Björn Johansson, John Wise, and Corinne Capsie. Could ERASMUS speed adjustments be identifiable by air traffic controllers. In *ATM Seminar, 8th USA/Europe Air Traffic Management R&D Seminar*, volume 22, 2007. 48

Noor H. Awad, Mostafa Z. Ali, Ponnuthurai N. Suganthan, and Robert G. Reynolds. CADE: a hybridization of cultural algorithm and differential evolution for numerical optimization. *Information Sciences*, 378:215–241, 2017. 47

Nicolas Barnier and Cyril Allignol. Trajectory deconfliction with constraint programming. *The Knowledge Engineering Review*, 27(03):291–307, 2012. 54

Nicolas Barnier and Pascal Brisset. FaCiLe: a functional constraint library. In *CICLOPS – Colloquium on Implementation of Constraint and LOgic Programming Systems, CP'01 Workshop*, Paphos, Cyprus, December 2001. 3, 81, 100

Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. 90

Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In *Principles and Practice of Constraint Programming – CP 2016*, volume 9892 of *Lecture Notes in Computer Science*. Springer, 2016. 129

Una Benlic, Edmund K. Burke, and John R. Woodward. Breakout local search for the multi-objective gate allocation problem. *Computers & Operations Research*, 78:80–93, 2017. 84

Timo Berthold. *Heuristic algorithms in global MINLP solvers*. Verlag Dr. Hut Munich, 2014. 11

M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. i. interval graphs. *Discrete Mathematics*, 100(1):267–279, 1992. 82

Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016. 8

Christian Blum, Carlos Cotta, Antonio J. Fernández, José E. Gallardo, and Monaldo Mastrolilli. Hybridizations of metaheuristics with branch & bound derivates. In *Hybrid Metaheuristics*, pages 85–116. Springer, 2008. 46

Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011. 46

Ahmet Bolat. Procedures for providing robust gate assignments for arriving aircrafts. *European Journal of Operational Research*, 120(1):63–80, 2000. 83, 84, 87

Ahmet Bolat. Models and a genetic algorithm for static aircraft-gate assignment problem. *Journal of the Operational Research Society*, 52(10): 1107–1120, Oct 2001. 2, 79, 82, 84, 89, 107, 108, 110, 125, 126

Abdelghani Bouras, Mageed A. Ghaleb, Umar S. Suryahatmaja, and Ahmed M. Salem. The airport gate assignment problem: a survey. *The scientific world journal*, 2014, 2014. 82, 84

Tim Bray, Jean Paoli, CM Sperberg-McQueen, Eve Maler, and François Yergeau. eXtensible markup language (XML) 1.0. *World Wide Web Consortium*, 2006. 27

Yves Caseau and François Laburthe. Solving various weighted matching problems with constraints. In G. Smolka, editor, *Principles and Practice of Constraint Programming - CP'97*, volume 1330 of *Lecture Notes in Computer Science*, pages 17–31, Berlin, Heidelberg, 1997. Springer. 81, 100

Geoffrey Chu and Peter J. Stuckey. Inter-instance nogood learning in constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 238–247. Springer, 2012. 129

Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *International conference on principles and practice of constraint programming*, pages 226–241. Springer, 2009. 6

Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg. Cooperative solution of constraint satisfaction problems. *Science*, pages 1181–1183, 1991. 9

Jean-François Cordeau and Mirko Maischberger. A parallel iterated tabu search heuristic for vehicle routing problems. *Computers & Operations Research*, 39(9):2033–2050, 2012. 11

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 95

Teodor Gabriel Crainic. Parallel computation, co-operation, tabu search. In *Metaheuristic Optimization via Memory and Evolution*, pages 283–302. Springer, 2005. 13

Teodor Gabriel Crainic and Michel Toulouse. Parallel strategies for meta-heuristics. In *Handbook of metaheuristics*, pages 475–513. Springer, 2003. 7

Teodor Gabriel Crainic and Michel Toulouse. Parallel meta-heuristics. In *Handbook of metaheuristics*, pages 497–541. Springer, 2010. 13

Teodor Gabriel Crainic, Michel Toulouse, and Michel Gendreau. Parallel asynchronous tabu search for multicommodity location-allocation with balancing requirements. *Annals of Operations Research*, 63(2):277–299, 1996. 14

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000. 94

Wu Deng, Huimin Zhao, Xinhua Yang, Juxia Xiong, Meng Sun, and Bo Li. Study on an improved adaptive PSO algorithm for solving multi-objective gate assignment. *Applied Soft Computing*, 59:288–302, 2017. 84

Ralf Diekmann, Reinhard Lüling, Burkhard Monien, and Carsten Spräner. Combining helpful sets and parallel simulated annealing for the graph-partitioning problem. *International Journal of Parallel, Emergent and Distributed Systems*, 8(1):61–84, 1996. 11

Ulrich Dorndorf, Andreas Drexl, Yury Nikulin, and Erwin Pesch. Flight gate scheduling: State-of-the-art and recent developments. *Omega*, 35(3): 326–334, 2007. 83

Ali Dorri, Salil S Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *Ieee Access*, 6:28573–28593, 2018. 6

Nicolas Durand and Géraud Granger. A traffic complexity approach through cluster analysis. In *ATM Seminar, 5th USA/Europe Air Traffic Management R&D Seminar*, 2003. 45

Nicolas Durand, Jean-Marc Alliot, and Joseph Noailles. Automatic aircraft conflict resolution using Genetic Algorithms. In *Proceedings of the Symposium on Applied Computing, Philadelphia*. ACM, 1996. 45

Nicolas Durand, David Gianazza, Jean-Baptiste Gotteland, Charlie Vanaret, and Jean-Marc Alliot. *Metaheuristics*, chapter Applications to Air Traffic Management, pages 439–484. Springer, 2016. 17

Mohammed El-Abd and Mohamed Kamel. A taxonomy of cooperative search algorithms. In *International Workshop on Hybrid Metaheuristics*, pages 32–41. Springer, 2005. 47

Deniz Türsel Eliiyi and Meral Azizoğlu. Heuristics for operational fixed job scheduling problems with working and spread time constraints. *International Journal of Production Economics*, 132(1):107–121, 2011. 79

Heinz Erzberger, Russell A. Paielli, Douglas R. Isaacson, and Michelle M. Eshowl. Conflict probing and resolution in the presence of Errors. In *ATM Seminar, 1st USA/Europe Air Traffic Management R&D Seminar*, 1997. 45, 54

EUROCONTROL. *Airport CDM Implementation Manual*. EUROCONTROL Airport CDM Team, 2017. 79

Filippo Focacci, Andrea Lodi, and Michela Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002. 6

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, and Bettina Schnor. Cluster-based ASP solving with *claspar*. In *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*. Springer, 2011. 129

Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, Apr 1988. 56

Jean-Baptiste Gotteland and Nicolas Durand. Genetic algorithms applied to airport ground traffic optimization. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 1, pages 544–551. IEEE, 2003. 129

Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. In *Information Processing Letters*, 1992. 54

Géraud Granger and Nicolas Durand. A traffic complexity approach through cluster analysis. In *ATM Seminar, 5th USA/Europe Air Traffic Management R&D Seminar*, 2003. 129

Géraud Granger, Nicolas Durand, and Jean-Marc Alliot. Optimal resolution of en-route conflicts. In *ATM Seminar, 4th USA/Europe Air Traffic Management R&D Seminar*, 2001. 45, 47, 56

Julien Guépet, Rodrigo Acuna-Agost, Olivier Briant, and Jean-Philippe Gayon. Exact and heuristic approaches to the airport stand allocation problem. *European Journal of Operational Research*, 246(2):597–608, 2015. 83

Long Guo, Said Jabbour, Jerry Lonlac, and Lakhdar Sais. Diversification by clauses deletion strategies in portfolio parallel SAT solving. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 701–708. IEEE, 2014. 12

Udaiprakash I. Gupta, Der-Tsai Lee, and J. Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982. 82, 84, 90

LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018. URL `http://www.gurobi.com`. 62, 67, 116, 121

Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012. 12

Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6: 245–262, 2008. 8, 12

Jin-Kao Hao. *Memetic Algorithms in Discrete Optimization*, pages 73–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012a. 29

Jin-Kao Hao. Memetic algorithms in discrete optimization. In *Handbook of memetic algorithms*, pages 73–94. Springer, 2012b. 6

Ali Asghar Heidari, Ibrahim Aljarah, Hossam Faris, Huiling Chen, Jie Luo, and Seyedali Mirjalili. An enhanced associative learning-based exploratory whale optimizer for global optimization. *Neural Computing and Applications*, pages 1–27, 2019. 47

Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly, March 2013. 25

Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997. 7

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013. 8

Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey. *50 Years of integer programming 1958–2008: From the early years to the state-of-the-art*. Springer Science & Business Media, 2009. 32

Madjid Khichane, Patrick Albert, and Christine Solnon. Intégration de l'optimisation par colonies de fourmis dans CP Optimizer. In *Sixièmes Journées Francophones de Programmation par Contraintes*, 2010. 6

Sang Hyun Kim. *Airport Control Through Intelligent Gate Assignment*. PhD thesis, Georgia Institute of Technology, 2013. 83

Leo G. Kroon, Arunabha Sen, Haiyong Deng, and Asim Roy. The optimal cost chromatic partition problem for trees and interval graphs. In *Graph-Theoretic Concepts in Computer Science*, pages 279–292. Springer, 1997. 83

Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. 89

Martin Kutz, Khaled Elbassioni, Irit Katriel, and Meena Mahajan. Simultaneous matchings: Hardness and approximation. *Journal of Computer and System Sciences*, 74(5):884–897, 2008. 90

Thibault Lehouiller, Jérémy Omer, François Soumis, and Guy Desaulniers. A flexible framework for solving the air conflict detection and resolution problem using maximum clique in a graph. In *ATM Seminar, 11th USA/Europe Air Traffic Management R&D Seminar*, 2015. 46

Thibault Lehouillier, Jérémy Omer, François Soumis, and Guy Desaulniers. Two decomposition algorithms for solving a minimum weight maximum clique model for the air conflict resolution problem. *European Journal of Operational Research*, 256(3):696–712, 2017. 46

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.05): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2017. 116

Bin Li and Steven CH Hoi. Online portfolio selection: A survey. *ACM Computing Surveys (CSUR)*, 46(3):1–36, 2014. 6, 7

Ruizhi Li, Shuli Hu, Yiyuan Wang, and Minghao Yin. A local search algorithm with tabu strategy and perturbation mechanism for generalized vertex cover problem. *Neural Computing and Applications*, 28(7):1775–1785, 2017. 46

Wang Li and Xiaofang Xu. Optimized assignment of airport gate configuration based on immune genetic algorithm. In *Measuring Technology and Mechatronics Automation in Electrical Engineering*, pages 347–355. Springer, 2012. 83

Marius Lindauer, Holger Hoos, and Frank Hutter. From sequential algorithm selection to parallel portfolio selection. In *International Conference on Learning and Intelligent Optimization*, pages 1–16. Springer, 2015. 8

Binod Maharjan and Timothy I. Matis. Multi-commodity flow network model of the flight gate assignment problem. *Computers & Industrial Engineering*, 63(4):1135–1144, 2012. 83

Ciaran McCreesh and Patrick Prosser. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. *ACM Transactions on Parallel Computing*, 2(1):8, 2015. 6

Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*. Springer, 2005. 129

Seyedali Mirjalili and Andrew Lewis. The whale optimization algorithm. *Advances in engineering software*, 95:51–67, 2016. 46

John E Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, pages 65–77, 2002. 34

Simeon C. Ntafos and S. Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, SE-5(5):520–529, September 1979. 89, 97

Angela Nuic, Damir Poles, and Vincent Mouillet. BADA: An advanced aircraft performance model for present and future ATM systems. *International journal of adaptive control and signal processing*, 24(10):850–866, 2010. 45

Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. 12

Lucia Pallottino, Éric Féron, and Antonio Bicchi. Conflict resolution problems for air traffic management systems solved with mixed integer programming. *IEEE Transactions on Intelligent Transportation Systems*, 3 (1):3–11, 2002. 45

James Payor. Fast C++ implementation of the Hungarian algorithm. GitHub repository `https://github.com/jamespayor/weighted-bipartite-perfect-matching`, 2017. 100

Michael Polacek, Siegfried Benkner, Karl F. Doerner, and Richard F. Hartl. A cooperative and adaptive variable neighborhood search for the multi depot vehicle routing problem with time windows. *Business Research*, 1 (2):207–218, 2008. 13

Günther R. Raidl and Jakob Puchinger. Combining (integer) linear programming techniques and metaheuristics for combinatorial optimization. In *Hybrid metaheuristics*, pages 31–62. Springer, 2008. 46

Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In *Handbook of parallel constraint reasoning*, pages 283–336. Springer, 2018. 11

V. Nagesjwara Rao and Vipin Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on parallel and distributed systems*, 4(4):427–437, 1993. 6

Yaping Ren, Chaoyong Zhang, Fu Zhao, Huajun Xiao, and Guangdong Tian. An asynchronous parallel disassembly planning based on genetic algorithm. *European Journal of Operational Research*, 269(2):647–660, 2018. 13

David Rey and Hassan Hijazi. Complex number formulation and convex relaxations for aircraft conflict resolution. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 88–93. IEEE, 2017. 46

David Rey, Christophe Rapine, Rémy Fondacci, and Nour-Eddin El Faouzi. Minimization of potential air conflicts through speed regulation. *Transportation Research Record: Journal of the Transportation Research Board*, 2300:59–67, 2012. 45

John R. Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976. 7

Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. 38

Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008. 40

Olivier Roussel. Description of ppfolio (2011). *Proc. SAT Challenge*, page 46, 2012. 8

Meinolf Sellmann. An arc-consistency algorithm for the minimum weight all different constraint. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 744–749, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 89, 100, 101, 125, 130

Paul Shaw. Using constraint programming and local search Methods to Solve Vehicle Routing Problems. In Michael Maher and Jean-François Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 6

Jialong Shi and Qingfu Zhang. A new cooperative framework for parallel trajectory-based metaheuristics. *Applied Soft Computing*, 65:374–386, 2018. 13

Helmut Simonis. Models for global constraint applications. *Constraints*, 12 (1):63–92, March 2007. 83, 90

Mauricio Solar, Víctor Parada, and Rodrigo Urrutia. A parallel genetic algorithm to solve the set-covering problem. *Computers & Operations Research*, 29(9):1221–1235, 2002. 11

Gerald N. Steuart. Gate position requirements at metropolitan airports. *Transportation Science*, 8(2):169–189, 1974. 82, 83

El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009. 6, 13

Reiko Tanese. *Distributed genetic algorithms for function optimization*. PhD thesis, University of Michigan, 1989. 14

Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of graphics tools*, 4(2):7–25, 1999. 56

Willem-Jan van Hoeve. *Operations Research Techniques in Constraint Programming*. PhD thesis, Institute for Logic, Language and Computation (ILLC), University of Amsterdam, April 2005. 106

Charlie Vanaret, Jean-Baptiste Gotteland, Nicolas Durand, and Jean-Marc Alliot. Preventing premature convergence and proving the optimality in evolutionary algorithms. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 29–40. Springer, 2013. 12

Adan Vela, Senay Solak, William Singhose, and John-Paul Clarke. A mixed integer program for flight-level assignment and speed control for conflict resolution. In *Proceedings of the Joint 48th IEEE Conference on Decision and Control and 28th Chinese Control Conference*. IEEE, 2009. 45

I-Lin Wang. Multicommodity network flows: A survey, part i: Applications and formulations. *International Journal of Operations Research*, 15:145–153, 12 2018. 109, 111

Ruixin Wang and Nicolas Barnier. Propagation of idle times costs for fixed job scheduling. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 718–725, Nov 2018. 82

Ruixin Wang and Nicolas Barnier. Global propagation of transition cost for fixed job scheduling. In *24th European Conference on Artificial Intelligence. Santiago de Compostela, Spain*, 2020. 82

Ruixin Wang, Cyril Allignol, Nicolas Barnier, and Jean-Baptiste Gotteland. Departure management with robust gate allocation. In *ATM Seminar, 13th USA/Europe Air Traffic Management R&D Seminar*, 2019. 82

Ruixin Wang, Richard Alligier, Cyril Allignol, Nicolas Barnier, Nicolas Durand, and Alexandre Gondran. Cooperation of combinatorial solvers for en-route conflict resolution. *Transportation Research Part C: Emerging Technologies*, 114:36–58, 2020. 44

Shangyao Yan and Chia-Ming Chang. A network model for gate assignment. *Journal of Advanced Transportation*, 32(2):176–189, 1998. 83

Shangyao Yan and Cheun-Ming Huo. Optimization of multiple objective gate assignments. *Transportation Research Part A: Policy and Practice*, 35(5): 413–432, 2001. 83

Shangyao Yan and Ching-Hui Tang. A heuristic approach for airport gate assignments for stochastic flight delays. *European Journal of Operational Research*, 180(2):547–567, 2007. 83

Dong Zhang and Diego Klabjan. Optimization for gate re-assignment. *Transportation Research Part B: Methodological*, 95:260–284, 2017. 83, 84

Acronyms

A-CDM	Airport Collaborative Decision Making
AABB	Axis-Aligned Bounding Box
ACO	Ant Colony Optimization
ATC	Air Traffic Control
ATM	Air Traffic Management
BADA	Base of Aircraft Data
BB	Branch and Bound
BC	Branch and Cut (Chapter 2)
BC	Best Cost (Chapter 4)
BE	Best Extension
BP	Branch and Prune
BST	Binary Search Tree
CATS	Complete Air Traffic Simulator
CBC	Collaborative Branch and Cut
CBP	Collaborative Branch and Prune
CH	Critical Hole
CMA	Collaborative Memetic Algorithm
COP	Combinatorial Optimization Problem
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DE	Differential Evolution
DMAN	Departure Manager
ENAC	École Nationale de l'Aviation Civile
EST	Earliest Start Time
FJS	Fixed Job Scheduling
FL	Flight Level
FMS	Flight Management System
GA	Genetic Algorithm
GAC	Generalized Arc Consistency
GAP	Gate Allocation Problem

GCG	Global Compatibility directed acyclic Graph
GPL	GNU General Public License
GPU	Graphics Processing Unit
IB&C	Interval Branch and Contract
ILP	Integer Linear Programming
LAP	Linear Assignment Problem
LS	Local Search
MA	Memetic Algorithm
MCFP	Minimum Cost Flow Problem
MFP	Multi-commodity Flow Problem
MH	Metaheuristic
MILP	Mixed Integer Linear Programming
MIP	Mixed Integer Programming
ML	Machine Learning
MPI	Message Passing Interface
MWPC	Minimum Weight Path Cover
NM	Nautical Mile
NS	Next Size
P-MH	Population-based Metaheuristic
PC	Path Cover
PSO	Particle Swarm Optimization
RB	Resource Balancing
RH	Rolling Horizon
RHS	Right Hand Side
S-MH	Single-solution-based Metaheuristic
SA	Simulated Annealing
SAT	Boolean satisfiability problem
SESAR	Single European Sky ATM Research
TC	Task Chaining
TS	Tabu Search
UMFP	Unsplittable Multi-commodity Flow Problem
XML	Extensible Markup Language
∅MQ	ZeroMQ