IT 10 020

Examensarbete 30 hp
Juni 2010

Contingency Plans for Air Traffic Management

Karl Sundequist Blomdahl

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Contingency Plans for Air Traffic Management

Karl Sundequist Blomdahl

We present two heuristics based on constraint technology that solve the problem of generating air traffic management contingency plans, which are used in the case of a catastrophic infrastructure failure within EUROCONTROL, the European Organisation for the Safety of Air Navigation. Of the heuristics presented, one is based on constraint-based local search and tabu search, and the other one is a constraint programming and large neighbourhood search hybrid algorithm. The heuristics show that it is feasible to automate the development of contingency plans, which is currently done by human experts; this is desirable for several reasons, for example it would allow the contingency plans to be generated with an increased frequency. The generated plans were evaluated, by EUROCONTROL, to be at least as good as the human-made ones.

Handledare: Justin Pearson
Ämnesgranskare: Pierre Flener
Examinator: Anders Jansson
IT 10 020
Sponsor: EUROCONTROL

Tryckt av: Reprocentralen ITC

Acknowledgements

This work has been financed by the European Organisation for the Safety or Air Navigation (EUROCONTROL) under its Care INO III programme (grant 08-121447-C). The content of the paper does not necessarily reflect the official position of EUROCONTROL on the matter. We especially thank S. Manchon, E. Petit, and B. Kerstenne at the EUROCONTROL Experimental Centre.

Contents

1 Air Traffic Management and Contingency Planning

Air traffic management (ATM) at EUROCONTROL, the *European Organisation for the Safety of Air Navigation*, is about managing and ensuring a safe, efficient, and fair flow of air traffic, assuming a negligible amount of side-effects, such as adverse weather conditions. During normal operation, the *Central Flow Management Unit* (CFMU) of EUROCONTROL uses several stages, each in increasing detail, to satisfy its operational goals:

1. A strategic stage, taking place several months before the day of operation.
2. A pre-tactical stage that starts six days before the day of operation.
3. An online tactical stage during the day of operation. This stage is called the *air traffic flow and capacity management* (ATFCM) stage [2], and has two main functions:
   (a) Calculate the demand of each airspace volume using live flight plan information.
   (b) Adjust the number of allocated departure slots of the involved aerodromes, such that they optimise the objectives defined in the pre-tactical stage. These objectives typically include, but are not limited to, minimising the total flight delay and air volume overload.

During an average day, the ATFCM unit handles approximately 30 000 flights spread over about 1 500 aerodromes.

This study will focus on the special case of an ATFCM failure due to any reason, such as downtime of the computer-assisted slot allocation (CASA) system. In such a situation, where no timely updates from ATFCM are available and the air controllers of each aerodrome have no idea whether it is proper to release a flight or not, a safe alternative is necessary. EUROCONTROL addresses this by a *contingency plan*, which contains a pre-defined number of allocated departure slots for each major aerodrome in such a way that certain safety and efficiency objectives are satisfied, for a maximum duration of one day. During the last twelve years, such a situation has occurred once, for a few hours.

An excerpt from such a contingency plan can be seen in Figure 1. It defines the number of departure slots that the aerodrome with the *International Civil Aviation Organization* (ICAO) identifier `EBBR` (Brussels national airport, Belgium) is allowed to release for each hour to each destination aerodrome. For example, from 09:00 to 12:00, a maximum of 7 flights in the flow `EBBR1`, which is defined by the departure aerodrome `EBBR` and a destination aerodrome whose ICAO identifier starts with `C` (Canada), `EG` (Great Britain), `EI` (Ireland), `K` (United States), or `M` (Central America and Mexico) are allowed to take off. Similarly, only 4 flights whose departure and destination aerodrome match the description of the flow `EBBR2` are allowed to

Flow identifier	**Flow description**	**Time span**	**Hourly rate**
EBBR1	From EBBR	00:00 – 06:00	2
	To C EG EI K M	06:00 – 09:00	3
		09:00 – 12:00	7
		12:00 – 14:00	4
		14:00 – 22:00	8
		22:00 – 24:00	2
EBBR2	From EBBR	00:00 – 06:00	1
	To B EDDH EDDW EE EF EH	06:00 – 17:00	4
	EK EN ES	17:00 – 21:00	6
		21:00 – 24:00	2

Figure 1: A contingency plan excerpt, which describes the hourly take-off rates of two flows originating from the aerodrome `EBBR` (Brussels national airport).

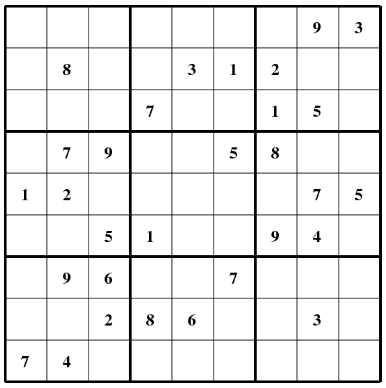
take off per hour from 06:00 to 17:00. The current contingency plan can always be downloaded from the CFMU homepage, in the upper-left corner of `https://www.cfmu.eurocontrol.int/PUBPORTAL/gateway/spec/`.

The generation of ATM contingency plans within the *EUROCONTROL Experimental Centre* (EEC) and the CFMU is currently done by two human experts (using a process described in Section 3.2 below), who biannually develop a three-fold plan, namely one for weekdays, one for Saturdays, and one for Sundays, with a total development time of two person-months per year. Therefore, automated contingency planning is desirable. This paper presents two heuristics that solve the subproblem of finding the optimal hourly numbers of departure slots for pre-defined flows and time spans (which typically do not change much between plans anyway), and is intended as a feasibility study about whether it is possible to replace the human experts with constraint technology. Other benefits with automating the process are that it could be done at the tactical level instead of the strategic level, which would increase the quality of the generated contingency plans.

The rest of this paper is split into six parts, each of which deals with the problem in increasingly concrete terms. In order of appearance: an introduction to *constraint programming* (CP) and *constraint-based local search* (CBLS) (Section 2), a formal definition of the problem (Section 3), a constraint model that implements the formal definition (Section 4), heuristics that operate on the constraint model (Section 5), experimental results (Section 6), and a conclusion (Section 7).

2 Constraint Technology

This section will give a short introduction to *constraint technology* (CT), a declarative modelling paradigm for solving (hard) combinatorial problems, that evolved from logic programming in the late 1980s, e.g. [9], [7]. CT is based on



							9	3
	8			3	1	2		
			7			1	5	
	7	9			5	8		
1	2						7	5
		5	1			9	4	
	9	6			7			
		2	8	6			3	
7	4							

Figure 2: An example Sudoku puzzle.

the idea of modelling a problem by its *constraints*, i.e., the properties that the values of the decision variables of a solution to the problem must satisfy, rather than how to implement these properties. A constraint can range from something simple, like a must be less than two, to more complex relationships like a, b, and c must all have different values; where the latter is the very popular and well-studied *alldifferent* (or *distinct*) constraint.

A popular example of a combinatorial problem (a problem where a solution must satisfy some constraints) is a Sudoku puzzle (see Figure 2). In a solution to a Sudoku puzzle each square must be assigned a value in $\{1,\dots,9\}$ such that in each row, each column, and each highlighted 3×3 box each digit appears exactly once. Formulating a Sudoku puzzle as a combinatorial problem is very straightforward due to the declarative nature of CT. A typical model represents the grid using a 9×9 matrix of decision variables, where each decision variable can take a value in $\{1,\dots,9\}$ (their domain). The model has 27 *alldifferent* constraints, one for each row, each column, and each 3×3 box, which require that no digit appear more than once in each of the aforementioned rows, columns, and 3×3 boxes. In addition each hint (fixed digit) can be loaded by adding a constraint requiring that its decision variable be equal to the hint. Once the model has been formulated, some method is necessary to solve it. We will present two methods, *constraint programming* (CP) (Section 2.1) and *constraint-based local search* (CBLS) (Section 2.2). Note that the model does not specify how to implement the constraints, that is part of the solving method, and almost all common constraints (such as *alldifferent* and arithmetical expressions) have already been implemented using very efficient algorithms. However, for the sake of clarity, a method to solve each of the used constraints will be presented in this section.

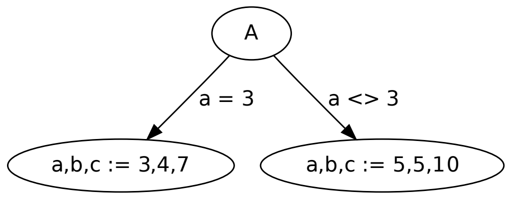


Figure 3: An illustration of the branching in the problem $a + b = c$, where $a \in \{1, 3, 4, 5\}$, $b \in \{4, 5\}$, and $c \in \{7, 10, 11\}$.

2.1 Constraint Programming

Constraint programming (CP) is a method to solve combinatorial problems by *propagation* and *branching*, where propagation eliminates infeasible solutions, i.e., solutions that violate one or more constraints, and branching explores the search space once no more solutions can be eliminated through propagation. Note that the two concepts are not separate phases; they are interleaved such that CP first propagates, then branches, then propagates, then branches, then propagates, etc. Note that propagation operates on the domains of the decision variables (denoted by $\text{dom}(x)$, for the decision variable x) and does not just check whether a solution is feasible once it has been found. For example, consider the problem $a+b = c$, where $a \in \{1, 3, 4, 5\}$, $b \in \{4, 5\}$, and $c \in \{7, 10, 11\}$. Propagation would eliminate the 1 and 4 from the domain of a, and 11 from the domain of c, then the two solutions $a, b, c := 3, 4, 7$ and $a, b, c := 5, 5, 10$ would be found by branching and propagating again. In this example branching was necessary (because there were more than one solution), but for many problems (such as the Sudoku puzzle in Figure 2) propagation is sometimes enough to completely solve the problem and branching is unnecessary. However, if branching is necessary, it works by making a decision that partitions the search space into smaller subsets (usually two), each of the subsets is then recursively explored using CP. When illustrated, branching produces a tree-like structure, where each node represents a subset of its parent's search space, and each leaf node is a solution. An example of such an illustration can be seen in Figure 3, which is an illustration of the aforementioned problem $a + b = c$, where $a \in \{1, 3, 4, 5\}$, $b \in \{4, 5\}$, and $c \in \{7, 10, 11\}$. A step-by-step guide to how this tree was produced follows:

1. The root of the problem is initialised, such that $\text{dom}(a) = \{1, 3, 4, 5\}$, $\text{dom}(b) = \{4, 5\}$, $\text{dom}(c) = \{7, 10, 11\}$, and the following *propagators* (which implement $a + b = c$) are added:

    - $\text{dom}(a) = \{a \in \text{dom}(a) \mid \exists b.(b \in \text{dom}(b) \wedge \exists c.(c \in \text{dom}(c) \wedge a+b = c))\}$

- $\text{dom}(b) = \{b \in \text{dom}(b) \mid \exists c.(c \in \text{dom}(c) \wedge \exists a.(a \in \text{dom}(a) \wedge a+b = c))\}$
- $\text{dom}(c) = \{c \in \text{dom}(c) \mid \exists a.(a \in \text{dom}(a) \wedge \exists b.(b \in \text{dom}(b) \wedge a+b = c))\}$

2. We create the root node (A) and run the propagators until they reach a fixpoint at $\text{dom}(a) = \{3, 5\}$, $\text{dom}(b) = \{4, 5\}$, and $\text{dom}(c) = \{7, 10, 11\}$; since they cannot eliminate anything more we branch. The chosen decision, say $a = 3$, is tried and splits the search space into two (one for which it is true, and one for which it is not):

   - For the case when $a = 3$ we create a node and run the propagators, since $\text{dom}(a) = \{3\}$ (due to the branching decision) the second and third propagator wake up and eliminate 5 from $\text{dom}(b)$ and 11 from $\text{dom}(c)$. The domains of the decision variables are now $\text{dom}(a) = \{3\}$, $\text{dom}(b) = \{4\}$, and $\text{dom}(c) = \{11\}$, which is the solution $a, b, c := 3, 4, 7$.
   - For the case when $a \neq 3$ we create a node and run the propagators, since $\text{dom}(a) = \{5\}$ (due to the branching decision) the second and third propagator wake up and eliminate 4 from $\text{dom}(b)$ and 7 from $\text{dom}(c)$. The domains of the decision variables are now $\text{dom}(a) = \{5\}$, $\text{dom}(b) = \{5\}$, and $\text{dom}(c) = \{10\}$, which is the solution $a, b, c := 5, 5, 10$.

Note how we did not even need to create leaf nodes for all possible solutions; the feasible solutions were found after just one branching even with these very simple propagation rules. Also note how CP explores the whole search space, not just a subset; this is usually a strength, but usually requires good branching decisions (made by a branching heuristic) in order to be sufficiently efficient.

2.2 Constraint-based Local Search

Constraint-based local search (CBLS) [10] is another contraint-based way to solve combinatorial problems but is quite different from CP; it shares a lot of methodology with local search. Instead of constructing an assignment of values to the decision variables by starting from the empty assignment and logically eliminating infeasible values, as CP does, CBLS starts from some initial assignment and evolves it until its *violation count*, measuring how much each constraint is violated by the current assignment, reaches some threshold (typically zero). This iterative evolution is implemented by a heuristic, which during each iteration modifies the current assignment, based on the violation count and how it changes upon changing the value of some decision variable, in order to decrease the violation count as low as possible (hopefully to zero). The heuristic used is problem specific, but heuristics based on meta-heuristics, such as tabu search [5], are very popular.

An example of a problem to solve using CBLS is the problem $a + b = c$, where $a \in \{1, 3, 4, 5\}$, $b \in \{4, 5\}$, $c \in \{7, 10, 11\}$, and the initial assignment $a, b, c := 1, 4, 10$. We define the violation count of this problem as $|a + b - c|$, since it is simple and describes how far from being satisfied this constraint is for the current assignment. The heuristic and neighbourhood (the area explored during each iteration) used will be a hill-climbing that during each iteration assigns one decision variable a new value:

1. The root is initialised, such that $a, b, c := 1, 4, 10$, and the violation count is 5.

2. The neighbourhood during the first iteration contains six possible moves, $a := 3$, $a := 4$, $a := 5$, $b := 5$, $c := 7$, and $c := 11$; for each move an assignΔ is calculated, that is how the violation count would change if the move was tried. For each possible move the assignΔ would be: $a := 3$ has assign$\Delta = -2$, $a := 4$ has assign$\Delta = -3$, $a := 5$ has assign$\Delta = -4$, $b := 5$ has assign$\Delta = -1$, $c := 7$ has assign$\Delta = -3$, and $c := 11$ has assign$\Delta = 1$. Since $a := 5$ has the smallest assignΔ (which decreases the violation count by 4) it is chosen. The assignment is now $a, b, c := 5, 4, 10$ and the violation count is 1.

3. The neighbourhood during the second iteration contains six possible moves, $a := 1$, $a := 3$, $a := 4$, $b := 5$, $c := 7$, and $c := 11$; however, since we just moved from $a := 1$ we consider that move taboo and remove it from the neighbourhood. Each of the possible moves have the following assignΔ: $a := 3$ has assign$\Delta = 2$, $a := 4$ has assign$\Delta = 1$, $b := 5$ has assign$\Delta = -1$, $c := 7$ has assign$\Delta := 1$, and $c := 11$ has assign$\Delta = 1$. Since $b := 5$ has the smallest assignΔ (which decreases the violation count by 1) it is chosen. The assignment is now $a, b, c := 5, 5, 10$ and the violation count is 0.

4. Since the violation count is zero, the search terminates successfully.

Note how the success of CBLS is highly dependent on its heuristics; however, this is not as bad as it seems, since very good results can usually be found even with a simple hill-climbing. Also note that CBLS does not systematically explore the whole search space (as CP does); for example in the $a + b = c$ example it only found one (of two feasible) solutions. This is both a strength and a weakness, as it can miss good solutions because of a decision made long ago, but on the other hand, with a good heuristic, it can quickly find good solutions. Another common problem with local search heuristics is that they get stuck in local minima (i.e., assignments to which no immediate improvements can be found); in order to escape from such an assignment a meta-heuristic can be used, for example, tabu search. Most meta-heuristics work by performing a series of non-improving iterations, whenever a local minima has been reached, such that the heuristic can continue evolving the current assignment in an unexplored part of the search space and hopefully find a new solution better than any solution previously found.

3 The Contingency Planning Problem

We give a detailed description of the *contingency planning problem* (CPP), the current state of the art algorithm, and a comparison with other problems.

3.1 Formal Definition

Each instance of the CPP is defined by the following input and output data, where identifiers starting with capital letters denote sets, subscripted identifiers denote constants, identifiers with indices within square brackets denote decision variables, identifiers that are Greek letters denote parameters, and all time moments are measured in seconds since some fixed origin:

- A set of flights $F = \{f_1, \ldots, f_m\}$, where each flight f_ℓ has a departure aerodrome $adep_\ell$, a destination aerodrome $ades_\ell$, an expected take-off time $etot_\ell$, an expected landing time $eldt_\ell$, and a take-off delay $delay[\ell]$. All later specified sets of flights are subsets of F.
- A set of flows $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$, where each flow $\mathcal{F}_f$ consists of a set of flights F_f and a set of span-rate pairs $\mathcal{R}_f = \{r_1, \ldots, r_{o_f}\}$, where each span-rate pair r_i consists of a time span $span_i$ for when it is active, and an hourly number of allocated departure slots $rate[i]$. Further, for any two span-rate pairs r_i and r_j, where $i \neq j$, their spans must not overlap; however, the union of all spans does not need to be 00:00–24:00. There is also a set $F_f \subseteq F$ for each $\mathcal{F}_f$ that contains all flights matching the flow description. For example, Figure 1 defines two flows `EBBR1` and `EBBR2`, where the flights are defined by a subset of F that matches the flow description, and the spans and rates are defined by the two right-most columns.
- A set of air volumes $AV = \{av_1, \ldots, av_p\}$, where each air volume $av_a \in AV$ has a capacity cap_a that limits the hourly number of flights that can enter it for the duration dur_a. There is also a set $F_a \subseteq F$ for each av_a that contains all flights that pass through the air volume, where each flight $f_\ell \in F_a$ has an expected entering time $enter_{a,\ell}$. In the real world, an air volume can represent either a part of the airspace, or an aerodrome.

Recall that ATM has three operational goals: minimise the cost of the total flight delay, ensure a safe flow of air traffic, and ensure a fair flow of air traffic. During a crisis situation, safety is especially important.

3.1.1 Cost of the Total Flight Delay

The take-off delay $delay[\ell]$ of flight f_ℓ is the difference between its calculated take-off time $ctot[\ell]$ and expected take-off time $etot_\ell$, where $ctot[\ell]$ is calculated using the allocated departure slots as defined by the rate-span pairs for each flow. These slots are assigned to flights using the first-scheduled, first-served principle [3]. For example, consider the flow `EBBR1` (defined in Figure 1), where there are two departure slots allocated for each hour between 00:00 and 06:00;

if three flights with an $etot_\ell$ of 03:00, 03:10, and 03:20 were available, then they would get a $ctot[\ell]$ of 03:00, 03:30, and 04:00, and a delay of 0, 1200, and 2400 seconds, respectively. Similarly, each flight f_ℓ has a calculated entering time $cnter[a,\ell]$ into air volume av_a, which is the sum of $enter_{a,\ell}$ and $delay[\ell]$. The cost of each flight delay is defined as a weight function, which was suggested to us by our partners at the EEC:

$$delayCost[\ell] = \begin{cases} 1 & \text{if } delay[\ell] < 3600 \text{ seconds} \\ 10 & \text{if } 3600 \leq delay[\ell] < 7200 \text{ seconds} \\ 20 & \text{if } 7200 \leq delay[\ell] < 10800 \text{ seconds} \\ 50 & \text{otherwise} \end{cases}$$

The weight function scales exponentially because the real-world consequences do; for example, a flight with a low delay will probably only cause a slight interruption in the schedule, while a high delay might cause many flights to be cancelled. The cost of the total flight delay is the sum of all flight delay costs.

3.1.2 Air Traffic Safety

The *safety* of air traffic is determined by how crowded the air volumes are; for example the air volume av_a is capable of handling up to cap_a flights entering per hour, so any flight above this capacity creates an additional risk. Hence, safety is here defined by the amount that each air volume's hourly capacity is exceeded. For each air volume av_a, a set T_a is defined that contains the beginning times of all one-hour-long time intervals that fit inside the air volume's capacity duration dur_a (with a five minute step):

$$T_a = \{t \in dur_a \mid t + 3600 \in dur_a \wedge t \bmod 300 = 0\}$$

A *demand overload* is calculated for each time $t \in T_a$ as the number of flights, beyond the air volume capacity, entering the air volume during the right-open interval $[t, t + 3600)$:

$$overload[a,t] = \max\left(0, |\{f_\ell \in F_a \mid cnter[a,\ell] \in [t, t+3600)\}| - cap_a\right)$$

The overload cost $overloadCost[a,t]$ of each air volume av_a and time $t \in T_a$ is a piecewise linear function of the overload percentage $\frac{overload[a,t]}{cap_a}$, where a weight is defined for the overload percentages 0%, 10%, 20%, 30%, and 40%. An illustration of this function can be seen in Figure 4. Again, the cost scales exponentially, because a small overload will likely only increase the workload of the affected ATM personnel slightly, while a large overload might result in a mistake by the ATM personnel. The cost of the total traffic demand overload is the sum of the cost for each air volume and time.

3.1.3 Air Traffic Fairness

The *fairness* of air traffic is here defined by how fairly the departure slots are allocated among the flows. No formal definition of fairness will be given at this point, as it is instead handled differently in each of our heuristics.

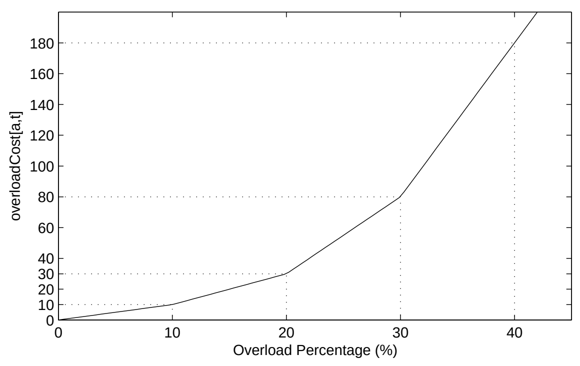


Figure 4: An illustration of the overload cost $overloadCost[a, t]$, for the overload percentages between 0% and 40%.

3.1.4 The Objective Function

The objective function is a linear combination of the total delay cost and the total overload cost, where α and β are parameters:

$$cost = \alpha \cdot \sum_{\ell \in F} delayCost[\ell] + \beta \cdot \sum_{a \in AV} \sum_{t \in T_a} overloadCost[a, t] \tag{1}$$

Experimental results and feedback from our partners at the EEC suggest that $\alpha = 6$ and $\beta = 1$ are good for our benchmark instances; however, they can be changed to reflect a desired balance between a low delay and a low traffic demand overload, for each problem instance.

3.2 Current State of the Art

The current state of the art, and the only known procedure, to solve the CPP is the unpublished process used by the CFMU and EEC human experts. It has been described to us in the following high-level steps:

1. A statistical analysis is performed in order to point out the airspace volumes with a high demand. The duration and capacity of each air volume are recorded (there may be several periods per volume).

2. An analysis of departing flows is made:

   - For the major European airports (i.e., with more than 2 arrivals or departures per hour on average), the traffic needs to be divided into main flows, where several destinations are grouped into each flow.
   - For the other airports, flows are mainly divided into two categories: domestic flights and international flights. If the number of domestic

flights is low, it seems better that a local flow manager handles this traffic.

Recall that it takes one person-month for two senior human experts to perform this procedure, and that all this is done twice a year, for weekdays, Saturdays, and Sundays.

3.3 Comparison with Other Problems

The CPP resembles several well-studied problems, especially scheduling problems. However, none of the studied problems can be used directly to solve the CPP. A case study of a few selected problems follows, in order to highlight the unique aspects of the CPP.

3.3.1 Cumulative Job Shop Scheduling

The *cumulative job shop scheduling problem* (CJSSP) is a well-studied multi-machine and multi-stage scheduling problem, and is proven $\mathcal{NP}$-hard. An instance is given by a set of activities, a set of resources AV, and a set of jobs F, where each job consists of a set of activities. Each activity $act_{a,\ell}$ has a processing time and a demand for each resource to be executed. Each resource $av_a \in AV$ has a capacity cap_a. A schedule is an assignment to each activity's starting time, and a schedule is feasible if it satisfies the capacity constraints for each resource, which requires that no more than cap_a units of each resource are required at once, as defined by the demand of each running activity.

Comparison with the CPP. With some extensions, such as soft capacities [8], the CJSSP is very closely related to the CPP. However, it cannot directly solve the CPP because of the relationship between the flow rates and the flight take-off delays, which causes the domain of any activity's starting time to be a function of all other activities' starting times (since two activities in the same flow must correspond to the same rate variable assignment). This complication means none of the established heuristics for solving the CJSSP can be directly applied to the CPP.

3.3.2 Multi-Commodity Flow

The *multi-commodity flow problem* (MCF) is a network flow problem, which given a graph $G = (V, E)$, a set of sources $S \subset V$, a set of sinks $T \subset V$, and a set of commodities C, minimises some cost as the commodities C flow over the network from source vertices to sink vertices. Since the MCF is a network flow problem, many well-studied extensions such as limited commodity quantities, restricted commodity flow paths, edge capacities, and dynamic flows (for modelling a notion of time) can be applied.

Comparison with the CPP. All state of the art solutions to the MCF are based on linear programming. This poses a problem since the objectives of the CPP are non-linear, and while rewriting them to linear expressions might be possible, no further investigation of the MCF as a modelling tool has been done yet.

4 The Constraint Model

Our constraint model implements the formal definition of the CPP using constraint technology. The model consists of five main parts: the decision variables and their domains, the problem constraints, the channelling constraints between the contingency plan and the flight delays, the channelling constraints between the flight delays and the air volume overloads, and the objective function.

4.1 The Decision Variables and their Domains

Recall that the decision variables of the model are the identifiers that use square brackets rather than subscripts. In most cases, the domains of the decision variables can be derived from their definition; however, the following decision variables have explicitly defined domains:

- $\forall f_\ell \in F : \mathit{delay}[\ell] \in \mathbb{N}$ (a smaller domain is calculated from Section 4.3)
- $\forall \mathcal{F}_f \in \mathcal{F},\ r_i \in \mathcal{R}_f : \mathit{rate}[i] \in [1, \mathit{demand}_{f,i}]$, where $\mathit{demand}_{f,i}$ is the maximum number of flights that are planned to depart in flow $\mathcal{F}_f$ during the time span span_i, and T_i is defined like T_a, but for span_i instead of dur_a:
$$\mathit{demand}_{f,i} = \max_{t \in T_i} |\{f_\ell \in F_f \mid \mathit{ctot}[\ell] \in [t, t+3600)\}|$$

4.2 Problem Constraints

The following problem constraints for any flight f_ℓ and air volume av_a:

$$\mathit{ctot}[\ell] = \mathit{etot}_\ell + \mathit{delay}[\ell]$$
$$\mathit{cnter}[a,\ell] = \mathit{enter}_{a,\ell} + \mathit{delay}[\ell]$$

establish the relationships between computed and expected times.

4.3 Channelling between Contingency Plan and Flight Delays

The channelling between the contingency plan and the flight delays is defined by the mappings $\mathcal{T}_f \in A_f \to D_f$ for each flow $\mathcal{F}_f$, where $A_f \subseteq \mathbb{N}_0^{|\mathcal{R}_f|}$ is the Cartesian product of the flow rate decision variable domains, and $D_f \subseteq \mathbb{N}_0^{|F_f|}$ are the take-off delays according to each element in A_f. For example, consider the flow

A_f		$\mapsto$	D_f		
1	3	$\mapsto$	0	1800	0
2	3	$\mapsto$	0	600	0

Table 1: The table produced for the *table* constraint, where each row contains one map in some $\mathcal{T}_f$.

$\mathcal{F}_f$, where $\mathcal{R}_f = \{r_1, r_2\}$, $\mathrm{dom}(\mathit{rate}[1]) = \{1, 2\}$, and $\mathrm{dom}(\mathit{rate}[2]) = \{3\}$; therefore $A_f = \{\langle 1,3\rangle, \langle 2,3\rangle\}$ and $\mathcal{T}_f = \{\langle 1,3\rangle \mapsto \langle \ldots\rangle, \langle 2,3\rangle \mapsto \langle\ldots\rangle\}$, where the actual take-off delays as calculated by $\mathit{rate}[1], \mathit{rate}[2] := 1, 3$ or $\mathit{rate}[1], \mathit{rate}[2] := 2, 3$ have been omitted for space reasons.

In *constraint-based local search* (CBLS), $\mathcal{T}_f$ can instead be used as a look-up table for the take-off delays whenever a rate decision variable changes, since only one-way reasoning is desirable. In classical CP (by complete tree search interleaved with propagation at every node), each mapping $\mathcal{T}_f$ can be implemented by a *table* constraint, that takes as input the tuples formed by the maps $x \mapsto y$ in some $\mathcal{T}_f$ such that each tuple corresponds to one map $x \mapsto y$ in some $\mathcal{T}_f$, the rate decision variables, and the take-off delay decision variables; the *table* constraint then requires that the tuple formed by the decision variables matches one of the input tuples from the mapping. For example, consider the example from above (with some take-off delays added), where $\mathcal{T}_f = \{\langle 1,3\rangle \mapsto \langle 0, 1800, 0\rangle, \langle 2,3\rangle \mapsto \langle 0, 600, 0\rangle\}$, this example would produce Table 1. Propagation would then work by pruning values from the decision variables domains that does not match any tuple in the produced table. For example, if, through propagation of some other constraint, the first rate decision variables domain is reduced to $\{2\}$ (from its previous $\{1, 2\}$) then the first row in Table 1 would violate the constraint and therefore be removed from the table, which in turn would prune 1800 from the domain of the second take-off delay decision variable. Similarly, if 1800 had been pruned first, then 1 would have been pruned from the domain of the decision variable for the same reason.

4.4 Channelling between Flight Delays and Air Volume Overloads

The channelling between the flight delays and the air volume overloads is modelled as a *cumulative job shop scheduling problem* (CJSSP) with a time step of five minutes, where each air volume av_a is a resource, each flight f_ℓ a job, and the activities are defined by the air volumes each flight passes through. Each such activity $act_{a,\ell}$ has the following parameters:

- $\mathit{resource}[act_{a,\ell}] = a$
- $\mathit{start}[act_{a,\ell}] = \mathit{cnter}[a, \ell] - \mathit{cnter}[a, \ell] \bmod 300$ (this is the calculated entering time rounded down to the closest five minute tick)
- $\mathit{duration}[act_{a,\ell}] = 3600$ seconds (since capacity is defined hourly)

- $end[act_{a,\ell}] = start[act_{a,\ell}] + duration[act_{a,\ell}]$
- $demand[act_{a,\ell}] = 1$ unit

The capacity of each resource av_a is cap_a. Further, as the time set T_a of an air volume av_a might not cover the entire day, one must make sure any overload that occurs during a time not in T_a does not contribute to the air volume cost. There are multiple ways of doing this: the chosen method is to add at most two activities for each day, namely one starting at the beginning of the day and ending at $\min(dur_a)$ (provided it is not empty), and the other starting at $\max(dur_a)$ and ending at the end of the day (provided it is not empty), both with a demand of $-|F_a|$. Since the worst-case scenario is that all flights are in the one-hour interval starting at the same t, adding an activity with a demand of $-|F_a|$ ensures that $overload[a,t] = 0$.

Unfortunately, practice has shown that it is impossible, and sometimes undesirable, to find a solution that satisfies such a *cumulatives* constraint, i.e., the problem is often over-constrained. The chosen method is to use a *soft cumulative* constraint (inspired by [8]), which calculates the cost of each air volume av_a and time t either by using a sweep-line algorithm [1], or by explicitly defining a decision variable for each air volume and time. Which of the two approaches is better depends on the circumstances: an explicit definition allows constant-time updates of the cost when all values are fixed (and is therefore used in the CBLS heuristic described in Section 5.1), but the sweep line provides reduced memory use (and is therefore used in the large neighbourhood search heuristic described in Section 5.2).

4.5 The Objective Function

The objective function of our model, to be minimised, is (1).

5 Local Search Heuristics

Our local search heuristics operate on the model, and based on their current state try to modify their rate decision variables in such a way that the objective function is minimised. Two such heuristics have been devised, namely: (i) a tabu search heuristic, and (ii) a *large neighbourhood search* (LNS) heuristic that uses classical *constraint programming* (CP) to search its neighbourhood. Both heuristics are described in detail in their respective sub-section, and their *generalised local search machines* (GLSM) can be seen in Figure 5. A GLSM [6] is a state machine that describes a local search heuristic by breaking it down into smaller algorithms, such that each state represents an individual algorithm and the edges represent the conditions for switching between these algorithms.

5.1 Tabu Search

Our first heuristic uses a tabu search as the core. It uses a slightly modified objective function, which adds a penalty term to (1) in order to guide the

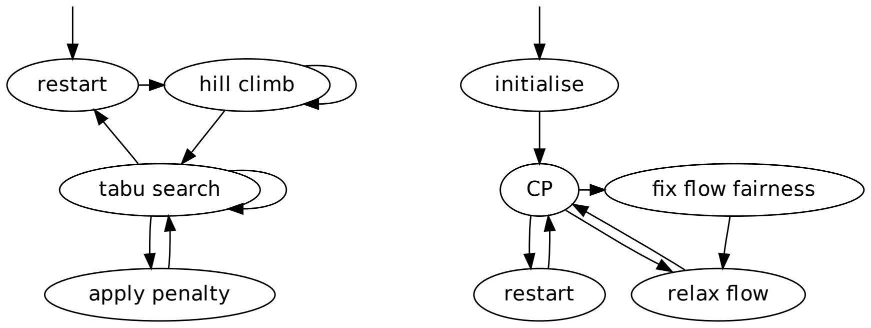


Figure 5: To the left, the *generalised local search machine* (GLSM) [6] of our tabu search heuristic (nodes are described in Section 5.1). To the right, the GLSM of our LNS heuristic (nodes are described in Section 5.2).

heuristic toward a fair traffic flow, where *Penalty* is a set of integer invariants:

$$cost = \alpha \cdot \sum_{\ell \in F} delayCost[\ell] + \beta \cdot \sum_{a \in AV} \sum_{t \in T_a} overloadCost[a,t] + \sum_{p \in Penalty} p$$

The heuristic can be summarised in the following steps, where each step and new terminology will be described in further details later:

1. Restart the search by assigning each $rate[i]$ a random value in its domain.

2. Hill-climb the current solution, until a local minimum has been reached.

3. Do a single iteration of tabu search, and then:

   (a) Pick a random real number $u \in [0,1]$; if $u < 0.05$, then pick a $rate[i]$ decision variable with an unfair value, add its penalty to *Penalty*, and go to Step 3; otherwise, do nothing and go to Step 3b.

   (b) If more than 200 iterations have gone by since the last improvement, then go to Step 1. Otherwise, repeat Step 3.

The main source of diversity is Step 1, the main source of intensification is Step 2, and Step 3 performs mix of both.

5.1.1 The Restart Mechanism

The restart mechanism is the main source of diversity in the heuristic. It completely restarts the search by assigning each $rate[i]$ decision variable a random value in its domain. It also clears the tabu list.

5.1.2 Hill-climbing

The hill climbing algorithm is a non-greedy algorithm. During each iteration, it picks the first $\langle rate[i], v\rangle$ move such that the objective function is decreased, until no such move can be found, i.e., a local minimum has been reached. The method used to find this assignment is through the use of a meta-neighbourhood, which is a circular list of neighbourhoods $\{N_1, \ldots, N_q\}$, where q is the number of rate decision variables, that are searched in successive order until an improving assignment is found, where each neighbourhood N_i corresponds to all variable-value pairs in $\{rate[i]\} \times \mathrm{dom}(rate[i])$. The algorithm terminates once a cycle has been completed with no improving assignment found.

5.1.3 Tabu Search

The tabu search is the core of the heuristic. While it is the main contributor of neither intensity nor diversity, it ensures that the tabu search neighbourhood of a local minimum has been properly explored and no improvements have been missed. During each iteration, it searches a neighbourhood (to be defined later) for a best non-taboo move $\langle rate[i], v\rangle$ and, after making the inverse move taboo for the number of iterations defined by the tabu tenure, it does the assignment $rate[i] := v$. The only exception to this process is the aspiration criterion, which kicks in if the candidate solution is better than any solution found so far. If this is the case, then the move is performed even if it is in the tabu list. The current implementation uses a tabu tenure of $\tau = 8$.

The tabu search uses an asymmetrical stochastic neighbourhood that is designed to reduce the most severe overloads. It does so by finding the peak of each air volume demand overload, and then picks one of these peaks to reduce at random, where the probability of each peak being picked is proportional to its overload, hence higher peaks have a higher probability to be reduced. Once a peak has been determined, all flows $\mathcal{F}_f$ that contain a flight contributing to this peak (flights that cannot be anywhere else can be ignored) have all $\{rate[i]\} \times \mathrm{dom}(rate[i])$, where $r_i \in \mathcal{R}_f$, added to the current neighbourhood.

5.1.4 Penalty Invariant

The apply penalty state is the part of the heuristic that tries to ensure a high level of air traffic fairness. It does so by modifying the cost function at random points in time, such that the rate variable $rate[i]$ with the minimum $\frac{rate[i]}{demand_{f,i}}$ quotient is deemed unfair and an expression that tries to guide $rate[i]$ toward a fairer value is added to *Penalty*. It is an exponential expression that decreases the higher the value of $rate[i]$:

$$\gamma \cdot e^{-8 \cdot \frac{rate[i]}{demand_{f,i}}}$$

where γ is a parameter that controls how aggressively the heuristic should be guided toward fairness; the current implementation uses $\gamma = 200$, which is only slightly aggressive.

5.2 Large Neighbourhood Search

Our second heuristic is a hybrid heuristic based on classical *constraint programming* (CP) and *large neighbourhood search* (LNS). Given a feasible solution, LNS works by *relaxing* part of the solution, that is, it picks some decision variables and restores their domains to their initial values, and uses constraint programming to search the resulting search space for a better solution. Our LNS heuristic can be summarised in the following steps, where each step and new terminology will be described in further details later:

1. Set each $rate[i]$ decision variable to the maximum of its domain, and go to Step 3 in *solve* mode.
2. If in *solve* mode, use CP to find a feasible solution; else (in *optimise* mode) use CP to find a feasible solution with the minimum cost.
3. Select a $rate[i]$ decision variable from a fixed circular list that contains all rate decision variables, in an arbitrary order:
   (a) If a full circle in the list has gone by with no improvement, then restore the domains of all rate variables, post a constraint that any next solution must be better than the current best, and go to Step 2 in *solve* mode.
   (b) If $rate[i]$ is unfair, then post a constraint that $rate[i]$ must be fair, relax the neighbourhood of $rate[i]$ according to Step 3c, and then go to Step 2 in *optimise* mode.
   (c) Relax the neighbourhood of $rate[i]$ using an algorithm based on maximum set coverage, post a constraint that a solution must be better than the current best, and go to Step 2 in *optimise* mode.

5.2.1 Constraint Propagation & Search

The CP state uses constraint propagation and search to find feasible solutions. It can do this in two available modes of operation: (i) the *solve* mode, in which it returns the first feasible solution, and (ii) the *optimise* mode, in which it exhaustively searches for the best solution using a depth-first search tree traversal. Which mode it uses depends on which was requested by the incoming call (edge in Figure 5); it does not use any internal heuristics to determine which is better. The branching heuristic used is to pick a variable $rate[i]$ at random and the value $\max(\mathrm{dom}(rate[i]))$. The reason for this is that when searching a relaxed neighbourhood most of the search is done using propagation rather than branching, hence even if a more complicated heuristic were used not much improvement could be found.

5.2.2 Restart Strategy

The restart strategy, which is triggered when all neighbourhoods have been searched and no improving move has been found, restores the domains of all decision variables of all flows. It also removes any constraints added by the heuristic, except for the constraint that any next solution must be better than the current best.

5.2.3 Relaxation

Relaxation is the most important part of the heuristic, as it defines the neighbourhood searched during each iteration. This neighbourhood is a cyclic list of neighbourhoods $\{N_1, \ldots, N_n\}$, where each neighbourhood N_f is designed to relax the decision variables closely interconnected with flow $\mathcal{F}_f$. This interconnectivity is defined by the number of air volumes that two flows have in common. In more detail, for each flow $\mathcal{F}_f$, a set S_f is defined that contains all air volumes that some flight in F_f passes through:

$$S_f = \{av_a \in AV \mid F_a \cap F_f \neq \emptyset\} \tag{2}$$

The interconnectivity of the flows $\mathcal{F}_f$ and $\mathcal{F}_h$ is then defined as $|S_f \cap S_h|$, the number of common air volumes that flights in $\mathcal{F}_f$ and $\mathcal{F}_h$ pass through. However, more than one flow with high interconnectivity is necessary for a good neighbourhood: what is desired is to give $\mathcal{F}_f$ a certain degree of freedom such that it can actually change in a meaningful way when relaxed; hence the neighbourhood of a flow $\mathcal{F}_f$ is defined as the *maximum set coverage* (MSC) of the set S_f and the set collection $S \setminus \{S_f\}$, where $S = \{S_1, \ldots, S_n\}$, with the slight modification that rather than limiting the number of sets that can be chosen, as is typically done in the MSC problem, the *size* of the resulting search space is instead limited, i.e., $\prod_{V \in S'} \prod_{i \in \mathcal{R}_V} |\operatorname{dom}(\mathit{rate}[i])|$, where $S' \subseteq S$ contains S_f and the selected sets from S, and $\mathcal{R}_V$ is the set $\mathcal{R}_f$ such that S_f is V. Luckily, in practice the interconnectivity between the sets in S seems to be high, hence this is not a very hard problem to solve. Using a greedy algorithm for solving MSC problems is sufficient to produce on average over 90% coverage when using a search space limit of $\delta = 100\,000$ candidate solutions.

During each iteration, the greedy algorithm maintains two auxiliary sets: (i) $U \subseteq S_f$, which are the still uncovered elements of S_f, and (ii) $S' \subseteq \{S_1, \ldots, S_n\}$, which are the sets picked as neighbours of S_f. Then, as long as S is not empty, it picks a set $V \in S \setminus S'$ with the largest intersection with U (i.e., $|V \cap U|$), where ties are broken by the largest intersection with S_f. Then, the auxiliary sets are updated, such that all elements in V are removed from U, and V is added to S', unless doing so would make the solution space larger than the limit δ, in which case V is instead discarded. Note that the U set can be empty during an iteration; this is the reason for the lexicographic comparison when selecting a $V \in S \setminus S'$. This algorithm can be seen in Algorithm 1.

Returning to the relaxation state, once S' has been determined, the neighbourhood N_f is defined for all decision variables $\mathit{rate}[i]$, where $r_i \in \mathcal{R}_f$ and $S_f \in S'$. Then this neighbourhood is relaxed by restoring the domain of each of the variables in N_f to its initial value, followed by adding two constraints: (i) the cost of any next solution must be smaller than the current best, and (ii) for each air volume, its maximum overload must not be larger than the maximum

Algorithm 1 The greedy maximum set coverage algorithm used to determine the *maximum set coverage* (MSC) of flow $\mathcal{F}_f$, with a maximum solution space size δ.

```
1: Calculate the set collection $S = \{S_1, \ldots, S_n\}$ according to (2) for each flow
   in $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$.
2: $S' \leftarrow \{S_f\}$
3: $S \leftarrow S \setminus \{S_f\}$
4: $U \leftarrow S_f$
5: while $S \neq \emptyset$ do {Invariant: $U \subseteq S_f \wedge S \cap S' = \emptyset$}
6:    Select the set $V$ among $S$ with the maximum $\langle |U \cap V|, |S_f \cap V| \rangle$.
7:    $S'' \leftarrow S' \cup \{V\}$
8:    if search space size of $S'' < \delta$ then
9:       $S \leftarrow S \setminus \{V\}$
10:      $S' \leftarrow S''$
11:      $U \leftarrow U \setminus V$
12:   else
13:      $S \leftarrow S \setminus \{V\}$
14: return $S'$
```

overload of the same air volume in the current best solution. The first constraint is a standard optimisation technique, whereas the second is there to improve the propagation and to allow proper energy feasibility calculations in the *soft cumulative* constraint [8].

5.2.4 Flow fairness

The fix flow fairness state addresses any unfair values assigned to rate variables. It does this by adding a couple of constraints when a flow that has a rate variable with an unfair value compared to all other flows is selected for relaxation. A value is *unfair* if it has a statistically outlying $q_i = \frac{rate[i]}{demand_{f,i}}$ quotient compared to all other flows. A value q_i is a statistical outlier if:

$$q_i \notin [E(q) - \text{std}(q), E(q) + \text{std}(q)]$$

where $E(x)$ is the expected value of the set x and $\text{std}(x)$ its standard deviation. If q_i is a statistical outlier, then a constraint requiring that $E(q) - \text{std}(q) \leq q_i \leq E(q) + \text{std}(q)$ is added; the neighbourhood of $\mathcal{F}_f$, where $r_i \in \mathcal{R}_f$, is relaxed as previously described, and a solution is sought in *optimise* mode. Note that no constraint requiring the solution to be better than the current best is added, because fairness is more important than a low cost.

6 Experimental Results

EUROCONTROL maintains two yearly timetables, one for the summer and one for the winter. Further, in each timetable weekdays, Saturdays, and Sundays

Contingency Plan	$E(delay[f])$	$p_{95}(delay[f])$	$E(overload)$	$p_{95}(overload)$
EEC 2008-06-27	645.6 sec	2340.0 sec	29%	100%
EEC 2008-08-30	528.1 sec	1800.0 sec	23%	61%
EEC 2008-08-31	407.0 sec	1500.0 sec	29%	68%
tabu 2008-06-27	310.2 sec	1200.0 sec	27%	72%
tabu 2008-08-30	316.1 sec	1200.0 sec	22%	56%
tabu 2008-08-31	345.9 sec	1264.5 sec	24%	57%
LNS 2008-06-27	535.5 sec	2185.0 sec	29%	100%
LNS 2008-08-30	512.1 sec	1800.0 sec	23%	60%
LNS 2008-08-31	504.1 sec	1628.0 sec	34%	100%

Table 2: The experimental results of the different algorithms.

have distinct traffic patterns. We have been provided, by the EEC, three real-life problem instances from the summer 2008 timetable that represent worst case scenarios for each distinct traffic pattern and are comparable to those used by EUROCONTROL when generating the official contingency plans:

- A Friday (June): 261 flows (320 rates), 36 161 flights, 348 air volumes.
- A Saturday (August): 256 flows (387 rates), 29 842 flights, 348 air volumes.
- A Sunday (August): 259 flows (397 rates), 31 024 flights, 348 air volumes.

When translated into a constrained optimisation problem, each instance yields approximately 150 000 constraints and 50 000 decision variables. All experimental results were done on a Linux x86-64 dual-core laptop with 4GB of primary memory, 2MB of L2 cache, and a CPU frequency of 2.2GHz. The tabu search heuristic have been implemented in Comet [10] version 2.0.1, and the LNS heuristic using Gecode [4] version 3.3.1. The tabu search usually terminated after approximately three CPU hours, while the LNS heuristic was interrupted after one CPU week (details below).

A comparison between our heuristics and a few contingency plans generated by the EUROCONTROL human experts (denoted by EEC) can be seen in Table 2, where the cost is presented as the expected take-off delay, the 95th percentile of the take-off delay, the expected air volume overload percentage (where overloads equal to zero have been omitted), and the 95th percentile of the air volume overload percentages (where overloads equal to zero have been omitted).

The first observation that can be made is that our heuristics decrease both the take-off delay and the air volume overload of the contingency plans generated by the EUROCONTROL human experts; this was expected, due to the similarities between the CPP and scheduling problems, which have been solved successfully using constraint technology for decades. However, the observation that our tabu search heuristic performs better than our LNS heuristic was unexpected, because the neighbourhood of the tabu search is a subset of the LNS neighbourhood, and should therefore perform at least as well as the tabu search heuristic. This performance difference has been attributed to the lack of runtime for the LNS heuristic, which was interrupted before reaching a local minimum, even after one week of runtime; further, this lack of runtime can probably be attributed to an inefficient implementation rather than a fault in our heuristic. However, regardless of the difference in performance between our heuristics, they show the feasibility of solving the CPP using constraint technology. The relative performance of our heuristics has been reproduced by the EEC, using their internal validation tool COSAAC and one of the current human planners. They compared our and their contingency plans on realistic test flight plans (not given to us), though *not* according to the objective function we used during the optimisation, but more realistically according to a CASA-style slot allocation, as if CASA was actually *not* down.

7 Conclusion

This work is intended as a feasibility study about whether it is possible to automate the development of contingency plans for EUROCONTROL, the *European Organisation for the Safety of Air Nagivation*. Based on the experimental results, it seems to be possible efficiently with constraint technology. Recall that this paper addresses the subproblem of finding the optimal number of allocated departure slots for predefined flows and time spans. The later have been produced by human experts, and do not change much from one year to another. However, the dependency on predefined flows and time spans must be eliminated. Currently, this is our most important issue; ideally the search for the optimal set of flows and time spans could be integrated into our heuristics.

References

[1] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000.

[2] EUROCONTROL. *Air Traffic Flow & Capacity Management Operations ATFCM Users Manual*. 14.0 edition, 2010. Available at `http://www.cfmu.eurocontrol.int/j_nip/cfmu/public/standard_page/library_handbook_supplements.html`.

[3] EUROCONTROL. *General & CFMU Systems*. 14.0 edition, 2010. Available at `http://www.cfmu.eurocontrol.int/j_nip/cfmu/public/standard_page/library_handbook_supplements.html`.

[4] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[5] F. Glover and M. Laguna. Tabu search. In *Modern heuristic techniques for combinatorial problems*, pages 70–150. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[6] H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[7] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM Press.

[8] T. Petit and E. Poder. Global propagation of practicability constraints. In *Proceedings of CPAIOR'08*, volume 5015 of *LNCS*, pages 361–366. Springer, 2008.

[9] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[10] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, 2005.