# Check-In Desk Scheduling Optimisation at CDG International Airport

**Thibault Falque[1,2], Gilles Audemard[2], Christophe Lecoutre[2], Bertrand Mazure[2]**

[1]Exakis Nelite
[2]CRIL, Univ Artois & CNRS
falque@cril.univ-artois.fr, audemard@cril.univ-artois.fr, lecoutre@cril.univ-artois.fr, mazure@cril.univ-artois.fr

## Abstract

More than ever, air transport players (i.e., airline and airport companies) in an intensely competitive climate need to benefit from a carefully optimized management of airport resources to improve the quality of service and control the induced costs. In this paper, we investigate the Airport Check-in Desk Assignment Problem. We propose a Constraint Programming (CP) model for this problem, and present some promising experimental results from data coming from ADP (Aéroport de Paris). Our works are deployed in a preprod environment since 1 year.

## Introduction

Before the COVID-19 health crisis, the International Air Transport Association (IATA) forecasts showed that passengers would double by 2036, reaching 7.8 billion. The COVID-19 pandemic has slowed air traffic considerably, especially in 2020 and early 2021, but since then, the economic pressure is back again. Air traffic picked up in 2022 and is similar to 2019. Some airlines have even announced the return to service of Airbus 380 to manage demand. In such a context, optimizing airport resources management remains essential to control induced costs while keeping a good quality of services. For many planning and scheduling air transport problems, techniques and tools developed from mathematical and constraint programming remain essential. Specifically, when airline companies have access to the resources delivered at the airport, the consumption of these resources (e.g., check-in banks, aircraft stand) must be carefully planned while optimizing an objective function determined by some business rules; see, for example, (Mangoubi and Mathaisel 1985; Dincbas and Simonis 1991; Lim, Rodrigues, and Zhu 2005; Diepen et al. 2007; Simonis 2007). A classical air transport problem is the Airport Gate Assignment Problem (AGAP), which involves assigning each flight (aircraft) to an available gate while maximizing both passenger conveniences and the airport's operational efficiency; see surveys in (Bouras et al. 2014; Daş, Gzara, and Stützle 2020) and models in (Li 2008, 2009; L'Ortye, Mitici, and Visser 2021). Another classical problem is the Check-in Assignment Problem, which involves assigning each flight to one or more check-in desks depending on the airline's requirements. Different approaches in MILP (mixed-integer linear programming) have been proposed (Yan, Tang, and Chen 2004; Araujo and Repolho 2015). A recent survey (Lalita and Murthy 2022) presents different methods for solving this problem using integer programming or dynamic programming. Because significant improvements have been made during the last decade in Constraint Programming (CP), such as, e.g., efficient filtering (and compression) algorithms for table constraints (Le Charlier et al. 2017; Demeulenaere et al. 2016), or lazy clause generation (G. Chu et al. 2011), tackling optimization of airport tasks with CP remains an interesting issue. In this paper, we are interested in the Airport Check-In Desk Assignment Problem as defined at CDG International Airport. We propose a Constraint Programming (CP) approach and show its potential interest by presenting promising experimental results. The rest of this paper is organized as follows. In Section , we present Airport Check-In Desk Assignment Problem. In Section , we propose a Constraint Optimization model for this problem and some possible variants of this model. In Section , we discuss the architecture of the `XCSP` toolchain in the Paris airport system. Next, in Section , we present some experiments carried out in an in-situ experimental context with the Paris airport system. Before concluding, we discuss the deployment challenges of our approach. Finally, in Section , we conclude and give some perspectives for future works.



## Airport Check-In Desk Assignment Problem at Paris Airport

CDG Airport is the ninth-largest airport in the world regarding passenger traffic. There are approximately $1,400$ flight movements (takeoff or landing) per day. At the airport, one of the combinatorial problems to address is to set each flight (or group of flights) to one or more available check-in desks. This Section provides some information about the Airport Check-In Desk Assignment Problem. A *registration* corresponds to a flight or a set of flights of the same airline. For each registration, a task must be carried out: associating a set of check-in desks with it. Each *task* of registration (or check-in for a flight) starts at the same time and ends simultaneously. Note that the number of check-in desks depends on the number of passengers and is fixed in advance by the

airline and the airport. Figure 1 presents some registration tasks at Orly Airport with 1, 4, and 5 tasks.

Planning registrations can be achieved for one or more days. For the moment, the planning horizon we manage is one week (sometimes less). In the rest of the paper, a check-in desk will be called a bank, the set of all registrations (tasks) is denoted by $\mathcal{R}$, the set of all zones (groups of banks) is denoted by $\mathcal{Z}$, the set of banks by $\mathcal{C}$ and the maximal number of banks required by registration by $\nu$.

### Imposing Consecutive Desks

When attempting to model this problem, a first arising constraint is that the banks (check-in desks) used for a specific registration must be **consecutive** (as we can observe in Figure 1). Importantly, as banks are grouped by zones, we must pay attention to assigning only banks from the same zone to a registration. For example, in Figure 2, there are two zones (colored in blue and pink); so for registration, we cannot use both a blue and a pink bank.

### Sharing Desks under Conditions

By default, a registration cannot share its assigned banks with another registration if the two registration tasks overlap. So at any time, no bank can be shared by two different registrations. However, for some reason of logistics (space) and under certain general conditions (called overlapping rules), some overlapping between flights from the same airline company may be tolerated for a limited period and/or for a limited number of tasks. In the latter case, if, for example, the number of banks required by registration is set to $4$ and the maximum number of overlapping situations is 2, then only two banks from the four banks associated with the registration can be shared with another registration that shares the same overlapping rule. We will note $\mathcal{O}$ the set of pairs of registrations $(\rho_1, \rho_2)$ that cannot strictly share banks (they may be time overlapping, but no rule exists permitting to have shared banks between them). We also note $\mathcal{OR}$ where each element $or$ is a pair $(\mathcal{R}_{or}, t)$ or triplet $(\mathcal{R}_{or}, t, m)$ where $\mathcal{R}_{or} \subset \mathcal{R}$ is the set of registration covered by the rule, $t$ is the overlapping duration tolerated by the rules, and $m$ is the maximum number of tasks that can overlap. Finally, we note for each rule $or$ and for each registration $\rho \in \mathcal{R}_{or}$, $\mathcal{N}_{\rho,or} \subset \mathcal{R}_{or}$ the set of neighbors (i.e., registrations that have a temporal overlap with $\rho$) of $\rho$ considering the rule $or$.

Figure 2 presents an example of planning that allows overlapping for $100\%$ of the time and without a limited number of tasks.

### Excluding Some Banks

Some banks are frequently unavailable for several hours to several days (for example, for maintenance reasons). The *unavailable constraints* ensures that certain banks are not available for a period of time (which may be periodic). In other words, we must remove from the domain the check-in desk for each task that overlaps with the period of exclusion. Another type of exclusion is to exclude the check-in desk for a given registration regardless of the time. The *exclusion constraints* ensures that certain banks are excluded for specific registration under some conditions.

### Pre-assigning Banks

Sometimes, users (from ADP) may want to force a specific set of banks to be associated with some registrations. We will note $(\rho, j, c)$ the triplet that represents the pre-assignment of bank $c$ as the jth bank used by registration $\rho$; all such triplets will be denoted by $\mathcal{P}$.

### Specifying the Objective

Of course, assigning a bank to a registration is subject to some placement preferences by airline companies. For each assigned bank, a reward is given: the reward of assigning the bank $c$ as the jth bank used by registration $\rho$ is denoted by $r^c_{\rho,j}$. Assuming that we have a series[1] of 0/1 variables $x^c_{\rho,j}$ associated with each registration task (indicating which check-in desk will be used), we can then define the overall objective function as follows:

$$\text{maximize} \sum_{\substack{\rho \in \mathcal{R} \\ j \in 1..\texttt{ntasks}(\rho)}} x^c_{\rho,j} \times r^c_{\rho,j} \qquad (1)$$

## Constraint Optimization Model

Now that the problem has been introduced in general terms, we need to describe it more formally using a constraint network. A *Constraint Network* (CN) consists of a finite set of variables subject to a finite set of constraints. Each variable $x$ can take a value from a finite set called the *domain* of $x$. Each constraint $c$ is specified by a relation that is defined over (the Cartesian product of the domains of) a set of variables. A *solution* of a CN is the assignment of a value to every variable such that all constraints are satisfied. A *Constraint Network under Optimization* (CNO) is a constraint network that additionally includes an objective function `obj` that maps any solution to a value in $\mathbb{R}$. For modeling CNOs, also called Constraint Optimization Problems (COPs), several modeling languages or libraries exist, such as, e.g., OPL (P. van Hentenryck 1999), MiniZinc (Nethercote et al. 2007; Stuckey, Becket, and Fischer 2010), Essence (Frisch et al. 2007) and PyCSP$^3$ (Lecoutre and Szczepanski 2020). Our choice is the recently developed Python library PyCSP$^3$ that permits to generate specific instances (after providing ad hoc data) in XCSP3 format (Boussemart et al. 2016, 2020), which is recognized by some well-known CP solvers such as `ACE` (AbsCon Essence) (Lecoutre 2023), OscaR (OscaR Team 2012), Choco (Prud'homme and Fages 2022), and PicatSAT (Zhou, Kjellerstrand, and Fruhman 2017). For simplicity, however, we formally describe below the model developed for the Airport Check-in desk problem in a higher "mathematical" form. Subsequently, we employ the data itself (for example, $\rho$) to serve as both the data and their corresponding indexes. Additionally, we adopt the notation $a[i, b]$ or $a[i][b]$ interchangeably to access the cells within a matrix

[1] Note that we shall not use 0/1 variables in the model proposed in Section .


Figure 1: An example of planning at Orly Airport

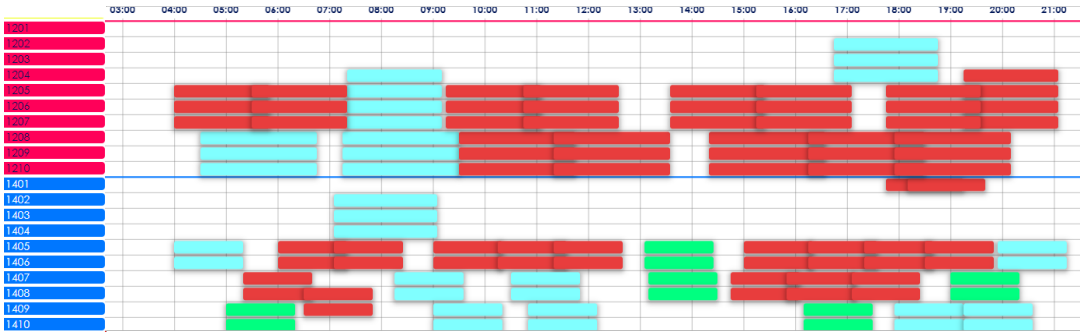


Figure 2: Example of a planning that allows overlapping.

denoted as $a$. Lastly, we employ the notation $\texttt{ntask}(\rho)$ to retrieve the count of tasks associated with the registration $\rho$. Firstly, we need to introduce the variables of our model. A registration must use check-in desks in coherence with its strategy. Therefore, rather than making domains containing all possible banks, the domains are initially reduced to those compatible with the registration strategy. For each registration $\rho$ we note this domain $\mathcal{D}_{x,\rho}$. Similarly, the domains for the variables representing airlines' rewards contain only the values corresponding to the allowed check-in desks. For each registration $\rho$ we note this domain $\mathcal{D}_{r,\rho}$. We also introduce a fictive bank $f$ with a reward of 0.

We need two (2-dimensional) arrays of variables to represent assigned registration and associated rewards:

- $x$ is a matrix of $|\mathcal{R}| \times \nu$ variables having the set of values $\mathcal{D}_{x,\rho}$; $x[\rho][j]$ represents the index (code) of the check-in desk assigned to the jth task of the registration $\rho$.
- $w$ is a matrix of $|\mathcal{R}| \times \nu$ variables having the set of values $\mathcal{D}_{w,\rho}$; $w[\rho][j]$ represents the satisfaction of the airline for the jth registration task $\rho$.

Secondly, we need to introduce the constraints in our model. Because of the nature of the problem (and data), it is natural to post so-called table constraints, which explicitly enumerate either the allowed tuples (positive table) or the disallowed tuples (negative table) for a sequence of variables (representing the scope of a constraint). Efficient algorithms for such table constraints have been developed over the last decade (Lecoutre 2011; Lecoutre, Likitvivatanavong, and Yap 2015; Demeulenaere et al. 2016; Verhaeghe, Lecoutre, and Schaus 2017).

## A First COP Formulation

Let us consider the variables previously introduced, the problem can be formulated as follows:

$$x[\rho][j] = c, \forall (\rho, j, c) \in \mathcal{P} \tag{2}$$

$$\begin{aligned}(x[\rho][j] = x[\rho][j+1] - 1) \ \vee \\ (x[\rho][j] = f \ \wedge \ x[\rho][j+1] = f), \\ \forall \rho \in \mathcal{R}, \\ \forall j \in \texttt{ntask}(\rho)\end{aligned} \tag{3}$$

$$\begin{aligned}x[\rho_1][i] \neq x[\rho_2][j], \forall \rho_1, \rho_2 \in \mathcal{O}, \\ \forall i \in \texttt{ntask}(\rho_1), \\ \forall j \in \texttt{ntask}(\rho_2)\end{aligned} \tag{4}$$

$$x[\rho][i], r[\rho][i] \in \{(c, r^c_{\rho,i}), \forall c \in \mathcal{C} \cup \{f, 0\}\} \tag{5}$$

Constraints (2) ensure that each pre-assignment of $\mathcal{P}$ is respected. Constraints (3) ensure that the chosen check-in desks for registration are consecutive or used the the fictive check-in desk for each registration task (see Section ). The introduction of holes in the domains (e.g., useless check-in desks) makes it possible to manage this by imposing that a task must be equal to the following task minus one and by not including useless check-in desks in the domain. In this way, we insert a hole representing the zone's separation. Constraints (4) prevent two overlapping registration from being assigned to the same check-in desk (as presented in Section ). Constraints (5) use table constraint to map the check-in desk with this weight. We use the weight (reward) defined in Section . As detailed earlier, overlapping rules can be set up to tolerate registrations using the same check-in desk. There are three methods of implementing these rules for two registrations:

- if no rule exists between these registrations, or if a rule exists but is incompatible with the overlap period, then overlap is prohibited using a non-overlap constraint (see Constraint 4).
- if a rule exists as a pair (i.e., without specifying the $m$ in the rules), overlapping is tolerated, and no constraint is added.
- if a rule exists as a triplet, specific constraints are added to represent this particular case.

We add a new matrix of $|\mathcal{OR}| \times |\mathcal{R}| \times |\nu| \times |n|$ variables called od. $n$ is the maximum number of possible neighbors (i.e., $\max(\{|\mathcal{N}_{\rho,or}|, \forall or \in \mathcal{OR}, \forall \rho \in \mathcal{R}_{or}\})$). The domain of a variable $od[or, \rho, b, \rho_1]$ is a binary domain composed of the value 0 and the overlapping duration between the registration $\rho$ and $\rho_1$ considering the rule $or$. Next, we can add the constraints over this previous matrix.

$$od[or, \rho, b, \rho_1] = 0 \Leftrightarrow \bigwedge_{bb=0}^{\texttt{ntasks}(\rho_1)} (x[\rho, b] \neq x[n, bb]) \vee x[\rho, b] = f, \quad \forall or \in \mathcal{OR}, \forall \rho \in \mathcal{R}, \forall b \in \texttt{ntask}(\rho), \forall \rho_1 \in \mathcal{N}_{\rho,or} \tag{6}$$

Constraints 6 ensure that the overlap time between a registration $\rho$ and and one of its neighboring registration $n$ is equal to 0 if and only if the two tasks use different check-in desks or one of them uses the dummy bank. Based on the assignment in the $x$ matrix, this constraint is used to determine whether two registrations overlap or not. We now must introduce constraints considering the maximum number of overlaps possible on a registration.

$$\texttt{Sum}(\{od[or, \rho, b, \rho_1] > 0, \forall \rho_1 \in \mathcal{N}_{\rho,or}, \forall b \in \texttt{ntasks}(\rho)\}) \leq m, \quad \forall or \in \mathcal{OR}, \forall \rho \in \mathcal{R} \tag{7}$$

In Constraint 7, $od[or, \rho, b, \rho_1] > 0$ is true when the registrations $\rho$ and $\rho_1$ overlap on task $b$ of $\rho$. We ensure that the sum of these booleans for each task $b$ of a registration $\rho$ and the set of tasks of these neighbors is less than or equal to $m$ from the $or$ rule.

Finally, we can use the matrix $w$ for posting the objective function (see Section ):

$$\text{maximize} \sum_{\substack{\rho \in \mathcal{R} \\ j \in 1..\texttt{ntasks}(\rho)}} w^c_{\rho,j} \tag{8}$$

## Gathering Binary Difference Constraints

We will now strengthen this natural formulation by reformulating the set of constraints (4) using the *AllDifferentExcept* constraint. This latter enforces all variables to take distinct values, except those assigned to a special (joker) value (here it is our fictive bank $f$).

$$\texttt{AllDifferentExcept}(\{x[\rho_1], x[\rho_2]\}, f) \tag{9}$$

For each pair $\rho_1, \rho_2$ in the set of forbidden overlaps $\mathcal{O}$. Note that we used the notation $x[\rho_1]$ and $x[\rho_2]$ for a shortcut that integrates the entire second dimension of the matrix into the constraint (i.e., each task of $\rho_1$ or $\rho_2$). This formulation allows us to reduce the number of constraints about no-overlapping tasks considerably, as the previous formulation needs a quadratic number of *not-equal* constraints.

## Gathering *AllDifferentExcept* Constraints

Even though the formulation above notably reduces the number of posted constraints, the solver remains too slow to find acceptable results (bounds) in a reasonable amount of time. We have thus gathered all AllDifferentExcept constraints into a unique, pragmatic constraint called GatherAllDifferentExcept. For this particular constraint, we use a specific fast propagator that performs a limited form of filtering (i.e., does not enforce generalized arc consistency). This is a very pragmatic approach, which is equivalent to the initial set of binary constraints but faster (only one constraint being posted).

## Refinement of the Overlapping Constraint

To reduce the dimensions of the $od$ matrix, an alternative approach involves utilizing not the cardinality of $\mathcal{R}$ for the second dimension, but solely the size of the set of registrations governed by rules. This size is represented by the union of all registrations involved in the rules, denoted as $\mathcal{R}_{\mathcal{OR}} = \bigcup_{\forall \mathcal{R}_{or} \in \mathcal{OR}} \mathcal{R}_{or}$. We can also use a true binary domain (i.e., {0,1}) to avoid the reification constraint introduced by $od[or, \rho, b, \rho_1] > 0$. Consequently, Constraint 7 can be reformulated in two possible ways: First, in Constraint 10, we exploit the binary domain to sum the variables directly. This eliminates the need for reification and results in the following form:

$$\texttt{Sum}(\{od[or, \rho, b, \rho_1], \forall \rho_1 \in \mathcal{N}_{\rho,or}, \forall b \in \texttt{ntasks}(\rho)\}) \leq m, \quad \forall or \in \mathcal{OR}, \forall \rho \in \mathcal{R} \tag{10}$$

Alternatively, in Constraint 11, we make use of a Count constraint along with the binary domain, taking advantage of the associated propagator:

$$\texttt{Count}(\{od[or, \rho, b, \rho_1], \forall \rho_1 \in \mathcal{N}_{\rho,or}, \forall b \in \texttt{nt}(\rho)\}, v = 1) \leq m, \quad \forall or \in \mathcal{OR}, \forall \rho \in \mathcal{R} \tag{11}$$

# Architecture

The sequence diagram, depicted in Figure 3, elucidates the intricate integration process of the XCSP toolchain into the newly developed software, referred to as DCB (Demand Capacity Balancing), at Paris Airport. The architecture of DCB is structured into two distinct components.

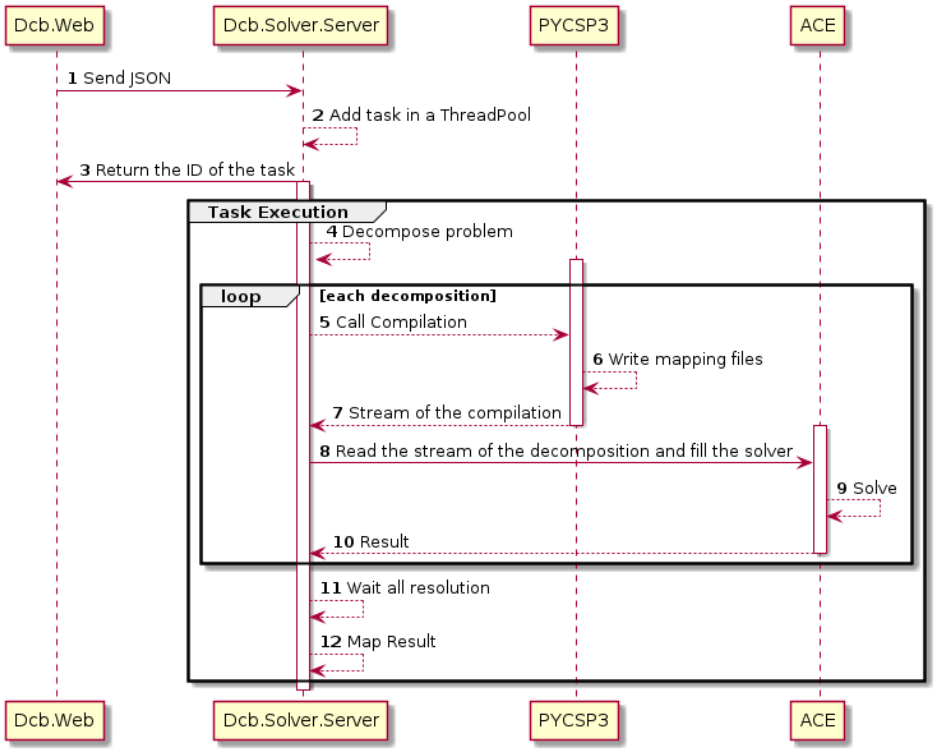


Figure 3: Sequence diagram of the XCSP toolchain.

First, there is `DCB.WEB`, which operates as an `API`, facilitating communication with the front-end developed in `REACT`. This component is interconnected with various vital systems of Paris Airport, enabling seamless access to resources, traffic data, and other pertinent information. Furthermore, it is connected with the second component, `DCB.SOLVER.SERVER` (also called `SOLVER`). `SOLVER` functions as another `API`, responsible for receiving a `JSON` representation of the problem. Subsequently, the problem is added to a `thread-pool`, and the associated `ID` is provided to `DCB.WEB` to facilitate tracking the resolution status. Following this initial step, the problem undergoes a decomposition process, with various strategies available for this decomposition, which will be discussed in Section . For each distinct decomposition, the PyCSP$^3$ model is invoked, generating an `XML` file that encapsulates the problem's details. To facilitate solution retrieval and reconstruction, a corresponding mapping file is also generated. Each of these generated `XML` files is then submitted to the solver `ACE` to derive a solution. Each Java solver is connected to `DCB.SOLVER.SERVER` through a `UNIX` socket. The process continues as we await the completion of each solver instance. Once all solvers have finished their operations, the solutions are merged into a global solution. When `DCB.WEB` queried `SOLVER` for a specific problem's solution with an associated `ID`, the corresponding solution is sent if the resolution process is complete.

# Experiments Results

## Instances

Table 1 presents some factual aspects concerning our instances based on real data from Paris Airport and representing realistic scenarios. The first column is the instance's name, while the second, third, fourth, and fifth columns indicate the number of check-in desks, tasks, overlapping rules, and strategy rules, respectively. We consider 3 kinds of instances. The first is the instance without overlapping rules (denoted by $\Phi_1$), the second is the instance with overlapping rules but without a limited number of tasks (denoted by $\Phi_2$), and the last case is the instances with overlapping rules that specify a limited number of tasks (denoted by $\Phi_3$). For space reasons, we have limited the results to only ORLY instance from $\Phi_2$, but all the results are available and reproducible[2] (thanks to *Metrics*[3]).

## Decomposition

Because decomposing the problem is possible without degrading results, it was decided to break the problem into simpler sub-problems to solve them successfully. We have three kinds of decomposition. The first strategy (also called *complete decomposition*) consists of breaking down the problem into groups of terminals based on assignment strategies. If

[2]see supplementary materials or https://gitlab.com/productions-tfalque/articles/check-in-scheduling-optim-cdg-airport/experiments-iaai-2024

[3]https://github.com/crillab/metrics

| Instance | $\|\mathcal{C}\|$ | # Tasks | $\|\mathcal{OR}\|$ | # Strategy rules |
|---|---|---|---|---|
| ORLY 1234-2023-05-08/2023-05-14 | 326 | 2224 | 19 | 31 |
| CDG T1-2023-07-03/2023-07-09 | 137 | 630 | 6 | 38 |
| CDG T2-B & D-2023-07-03/2023-07-09 | 108 | 766 | 1 | 21 |

Table 1: Description of the main instances uses in this study.

a strategy for registration covers the check-in desks of several terminals, then we group the terminals; otherwise, we leave them separate. Finally, for each group of terminals, we can re-decompose them day by day. If there are night flights, these are pre-assigned before launching resolution. In the second strategy, we keep the terminals together, and we decompose only day by day. In the last strategy, we generate a global problem without decomposition.

## Environment

In our experimentation, the time limit for each execution (part of the decomposition) is limited to 30 seconds (compilation time in XCSP format is not included in this timeout). They are launched in a real environment in the Paris airport system equipped with 64 GB of RAM and two 10-core Intel Xeon Silver 4210R (2.4 GHZ) and running *Windows Server 2019*. We have limited the number of parallel jobs to 10 (i.e., no more than 10[4] "resolution[5]" jobs can be run simultaneously). Note that the solver is stopped when no more improvement has been made during a period of 5 seconds (since the last solution was found). Since the choice of stopping the solver after 5 seconds makes it non-deterministic, we run each configuration on each decomposition 5 times. For our study, we use `frba/dom` (Li, Yin, and Li 2021) as variable-ordering heuristic (this heuristic was observed as the best one on this problem), `solution-saving` (Vion and Piechowiak 2017; Demirovic, Chu, and Stuckey 2018) for simulating a form of large neighborhood search. Concerning the value-ordering heuristic, we have tested different configurations: `BIVS` (Fages and Prud'Homme 2017), until the first solution is found (after that, the smallest value in the domain is systematically selected if solution-saving cannot be applied), `Static`, a static order based on the rewards of check-in desks in the strategies, and `BIVS` + `Static`, an approach that mixes the two heuristics: `BIVS` until the first solution is found (after that, `Static` is used if solution-saving cannot be applied). We use the solver `ACE`[6] and in particular the `JUniverse`[7] adapter of `ACE`: `ACEURANCETOURIX`[8] which allows interaction with `ACE` via an *interface*.

## Comparison with the First Release ons $\Phi_2$

In the initial iteration of our approach, the preliminary processing of ADP data was executed directly in PyCSP$^3$ model. Additionally, `SOLVER` was limited to conducting a complete decomposition exclusively. The improvements in the second version can be summarized as follows:

- The preliminary processing was executed directly in `C#` before calling PyCSP$^3$ model.
- The variables representing tasks are now ordered based on their start dates.
- Certain computations related to the integration of the GatherAllDifferent constraint within the solver have undergone optimization.

For this Section, we consider instances with overlapping rules that do not have a limited number of tasks and the best solver's configuration of the first version reproduced identically in the second version. Note that as there is no overlap rule with a limited number of tasks, the constraints 6 and 7 are not present.

Tables 2 and 3 present some results with different configurations for instance ORLY 1234 for the week from 2023-05-08 to 2023-05-14. After the decomposition step, this latter is decomposed in two groups of terminals (*ORLY 1,2,4* and *ORLY 3*). The first column presents the configuration of `ACE`, and the second column indicates the arity limit for which intension constraints are transformed into extension constraints by the solver. The third column indicates if we gather or not the `AllDifferentExcept` constraints, *False* corresponds to the second formulation (Section ) and *True* to the third formulation (Section ). The next 10 columns present results for the first version and the second version. For both versions, we found 5 columns:

- *Time* presents the best and worst case of resolution time (including compilation time) over the 5 executions if we have run the resolution sequentially.
- *Solver T* is similar to the column *Time* but only for the solver.
- *First* and *Last* contain the mean of the first (resp. last) bound computed over the 5 executions and the best and worst case runtime for obtaining the first (resp. last) bound.
- *# not* contains the number of registrations that are not assigned (i.e., the number of registrations that use the fictive check-in desk).

We can see that the updated version leads to a significant reduction in the computational time (column *Time*), with improvements of 17% for the best-case scenario and 21% for the worst-case scenario. The improvements made by this second version limit the impact of using Python and allow us to continue using the PyCSP$^3$ library to model problems.

[4] We use the value returned by `Environment.ProcessorCount` (20) divide by two.

[5] "resolution" means *compilation phase* and *solving phase*

[6] https://github.com/xcsp3team/ace

[7] https://github.com/crillab/juniverse

[8] https://github.com/crillab/aceurancetourix

| Configuration | ALE | GAT | Time | Solver T | First | Last | # not |
|---|---|---|---|---|---|---|---|
| `Bivs` + `Static` | 4 | True | 139-182 | 80-102 | 22,648,100 (5-10) | 23,694,100 (30-50) | 255 |
| `Bivs` | 0 | True | 118-159 | 77-110 | 22,648,100 (5-7) | 23,620,900 (26-44) | 258 |
| `Static` | 0 | True | 119-187 | 80-112 | 17,512,500 (4-7) | 23,940,100 (30-44) | 243 |

Table 2: Result on decomposition ORY124 (7 days) from planning ORLY 1234 of $\Phi_2$.

| Configuration | ALE | GAT | Time | Solver T | First | Last | # not |
|---|---|---|---|---|---|---|---|
| `Bivs` + `Static` | 4 | True | 109-135 | 85-108 | 22,692,100 (4-8) | **23,779,100** (34-51) | 254 |
| `Bivs` | 0 | True | 112-158 | 91-119 | 22,692,100 (4-9) | **23,819,100** (42-56) | 253 |
| `Static` | 0 | True | 87-116 | 67-96 | 22,624,900 (3-8) | **23,974,100** (21-39) | 243 |

Table 3: Result on decomposition ORY124 (7 days) from planning ORLY 1234 of $\Phi_2$.

### Results of the Second Version Over $\Phi_2$

Tables 4 and 5 present results corresponding to various decompositions of instance ORLY 1234, while Table 6 illustrates results for the same instance but without any decomposition. It's worth noting that in the latter table, configurations where GatherAllDifferent is set to `false` have been excluded due to resulting in a *TIMEOUT* situation. Moreover, it's notable that despite relatively short resolution times (refer to column *Solver T*), the global times still remain high (as indicated in column *Solver T*), which can be attributed to the increased time taken for compilation processes. Finally, we can see that decomposition seems to be a good option because it gives a better bound and fewer unassigned flights than when no decomposition is used ($9+253 = 262$). For example, for `Bivs` + `Static` the bound is $45255160$ ($23795100+21460060$) and the number of unassigned flight is $262$ ($253 + 9$).

## Deployment Challenges

Our current approach represents a comprehensive reorganization of the existing planning tool, originally built on technologies dating back approximately two decades. The cornerstone of our strategy is to enable future maintainers to focus primarily on PyCSP$^3$ models. Through specialized training in this library, these individuals can adeptly adjust and customize the models to meet the Paris Airport group's dynamic needs, all while bypassing the need to alter the solver's intricate components. However, it's noteworthy that employing Python may introduce certain limitations in speed, particularly during the compilation stages, which could potentially become a bottleneck. A significant technical challenge we encountered was integrating the primary components of DCB, which are developed in C#, with both the Python-based PyCSP$^3$ models and the Java-based solver, `ACE`. This integration posed a multifaceted engineering challenge, albeit one that falls outside the purview of this paper. The system we describe is deployed in a preprod environment in a real-world context at Paris Airport for Demand Capacity Balancing (DCB). Its deployment runs in parallel with the legacy solution, enabling user-led visual comparative analyses. It's crucial to acknowledge that direct comparisons between the previous and new methodologies are constrained due to the differences in modeling approaches. Given the disparities in the scales of the objective functions, users rely on visual methods for comparing the two systems. One of the principal advantages of our approach is the significant cost savings associated with license fees for the previous commercial solution. By opting to develop its own solution, Paris Airport retains complete control over its data, avoiding the purchase of an expensive external system. Moreover, the user experience has been greatly enhanced by transitioning to a modern web interface, replacing the outdated, heavy client UI of the legacy system. This improvement is notable not just in the solver aspect but also in user interactions for configuration and manual corrections on the Gantt chart. The adoption of open-source tools, in contrast to the former commercial solution, not only reduces costs but also markedly enhances the tool's adaptability and accessibility, aligning well with the strategic objectives of Paris Airport.

## Conclusion

In this paper, we have been interested in the Airport Check-in Desk Problem as defined at Paris Airport. We have formulated a COP model for this problem, mainly exploiting table constraints and developing an ad-hoc constraint (for reducing the number of constraints and accelerating the resolution consequently). We have presented an empirical evaluation of our approach. Our results look quite promising as the ADP group starts replacing their current proprietary solution with ours, based on generic open-source tools (modeling library and constraint solver). Our work is an integral part of redesigning Paris Airport planning tools, leading to the creation of the new DCB tool. However, despite the encouraging performance metrics and the seamless integration that empowers users to focus solely on PyCSP$^3$-written models, the reliance on Python in the current execution workflow has unveiled a potential bottleneck. Consequently, we are currently exploring avenues to avoid the Python dependency and, if feasible, to communicate with the solver API with the same user-friendliness akin to the PyCSP$^3$ library.

| Configuration | ALE | GAT | Time | Solver T | First | Last | # not |
|---|---|---|---|---|---|---|---|
| `Bivs + Static` | 0 | False | 118-150 | 98-128 | 22,692,100 (35-60) | 22,890,100 (50-79) | 281 |
| `Bivs + Static` | 0 | True | 109-141 | 89-120 | 22,692,100 (4-12) | **23,795,100** (36-60) | 253 |
| `Bivs + Static` | 4 | False | 120-155 | 99-131 | 22,692,100 (36-59) | 22,890,100 (51-80) | 281 |
| `Bivs + Static` | 4 | True | 109-135 | 85-108 | 22,692,100 (4-8) | 23,779,100 (34-51) | 254 |
| `Bivs` | 0 | False | 130-164 | 108-143 | 22,692,100 (40-68) | 22,896,100 (60-90) | 281 |
| `Bivs` | 0 | True | 112-158 | 91-119 | 22,692,100 (4-9) | **23,819,100** (42-56) | 253 |
| `Bivs` | 4 | False | 124-148 | 102-126 | 22,692,100 (38-53) | 22,89,0100 (53-75) | 281 |
| `Bivs` | 4 | True | 109-132 | 89-107 | 22,692,100 (4-9) | **23,819,100** (41-49) | 253 |
| `Static` | 0 | False | 115-470 | 78-134 | 22,624900 (18-31) | 23,073,760 (37-79) | 251 |
| `Static` | 0 | True | 87-116 | 67-96 | 22,624,900 (3-8) | 23,974,100 (21-39) | 243 |
| `Static` | 4 | False | 127-293 | 85-168 | 22,624,900 (16-28) | 23,109,440 (36-75) | 251 |
| `Static` | 4 | True | 93-120 | 71-85 | 22,624,900 (3-6) | **24,000,100** (25-33) | 242 |

Table 4: Result on decomposition ORLY 124.

| Configuration | ALE | GAT | Time | Solver T | First | Last | # not |
|---|---|---|---|---|---|---|---|
| `Bivs + Static` | 0 | False | 217-288 | 177-248 | 21,232,800 (107-166) | 21,237,620 (124-184) | 12 |
| `Bivs + Static` | 0 | True | 135-176 | 94-139 | 21,232,800 (7-18) | **21,460,060** (39-54) | 9 |
| `Bivs + Static` | 4 | False | 206-275 | 167-235 | 21,232,800 (96-158) | 21,237,900 (114-170) | 12 |
| `Bivs + Static` | 4 | True | 123-219 | 84-170 | 21,232,800 (5-10) | 21,428,900 (14-69) | 10 |
| `Bivs` | 0 | False | 201-251 | 164-214 | 21,232,800 (93-139) | 21,238,220 (110-158) | 12 |
| `Bivs` | 0 | True | 125-203 | 85-151 | 21,232,800 (4-15) | **21,421,400** (26-54) | 9 |
| `Bivs` | 4 | False | 200-253 | 164-216 | 21,232,800 (92-141) | 21,238,540 (110-158) | 12 |
| `Bivs` | 4 | True | 116-176 | 77-137 | 21,232,800 (4-10) | **21,442,480** (23-54) | 9 |
| `Static` | 0 | False | 179-579 | 137-280 | 21,049,200 (15-52) | 21,190,220 (72-165) | 16 |
| `Static` | 0 | True | 150-219 | 108-179 | 21,049,200 (4-11) | 21,721,160 (51-122) | 3 |
| `Static` | 4 | False | 182-854 | 130-244 | 21,049,200 (14-30) | 21,194,800 (60-167) | 16 |
| `Static` | 4 | True | 154-214 | 112-173 | 21,049,200 (3-10) | **21,758,460** (45-116) | 2 |

Table 5: Result on decomposition ORLY 3.

| Configuration | ALE | GAT | Time | Solver T | First | Last | # not |
|---|---|---|---|---|---|---|---|
| `Bivs + Static` | 0 | True | 112-114 | 18-19 | 43,924,300 (9-10) | 43,927,600 (9-10) | 298 |
| `Bivs + Static` | 4 | True | 112-121 | 18-26 | 43,924,300 (9-14) | 43,927,600 (10-16) | 298 |
| `Bivs` | 0 | True | 112-117 | 18-23 | 43,924,300 (9-13) | 43,927,600 (9-14) | 298 |
| `Bivs` | 4 | True | 112-120 | 18-25 | 43,924,300 (9-14) | 43,927,600 (10-16) | 298 |
| `Static` | 0 | True | 107-122 | 12-27 | 43,674,100 (4-5) | 43,677,380 (4-18) | 274 |
| `Static` | 4 | True | 107-108 | 12-14 | 43,674,100 (3-5) | 43,674,100 (3-5) | 274 |

Table 6: Result on instance ORLY 1234 without decomposition.

## References


Araujo, G. E.; and Repolho, H. M. 2015. Optimizing the Airport Check-In Counter Allocation Problem. *Journal of Transport Literature*, 9(4): 15–19.

Bouras, A.; Ghaleb, M. A.; Suryahatmaja, U. S.; and Salem, A. M. 2014. The Airport Gate Assignment Problem: A Survey. *The Scientific World Journal*, 2014: e923859.

Boussemart, F.; Lecoutre, C.; Audemard, G.; and Piette, C. 2016. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems. *CoRR*, abs/1611.03398.

Boussemart, F.; Lecoutre, C.; Audemard, G.; and Piette, C. 2020. XCSP3-core: A Format for Representing Constraint Satisfaction/Optimization Problems. *CoRR*, abs/2009.00514.

Daş, G. S.; Gzara, F.; and Stützle, T. 2020. A Review on Airport Gate Assignment Problems: Single versus Multi Objective Approaches. *Omega*, 92: 102146.

Demeulenaere, J.; Hartert, R.; Lecoutre, C.; Perez, G.; Perron, L.; Régin, J.-C.; and Schaus, P. 2016. Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets. In *Proceedings of CP'16*, 207–223.

Demirovic, E.; Chu, G.; and Stuckey, P. 2018. Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers. In *Proceedings of CP'18*, 99–108.

Diepen, G.; Akker, J.; Hoogeveen, J.; and Smeltink, J. 2007. Using Column Generation for Gate Planning at Amsterdam Airport Schiphol.

Dincbas, M.; and Simonis, H. 1991. APACHE - A Constraint Based, Automated Stand Allocation System. Automated Stand Allocation System Proc. Of Advanced Software Technology in Air Transport (ASTAIR'91) Royal Aeronautical Society, 267–282.

Fages, J.-G.; and Prud'Homme, C. 2017. Making the First Solution Good! In *ICTAI 2017*, 1073–1077. Boston, MA: IEEE. ISBN 978-1-5386-3876-7.

Frisch, A.; Grum, M.; Jefferson, C.; Hernandez, B. M.; and Miguel, I. 2007. The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems. In *Proceedings of IJCAI'07*, 80–87.

G. Chu; P. Stuckey; M. Garcia de la Banda; and C. Mears. 2011. Symmetries and Lazy Clause Generation. In *Proceedings of IJCAI'11*, 516–521.

Lalita, T. R.; and Murthy, G. S. R. 2022. The Airport Check-in Counter Allocation Problem: A Survey. arxiv:2208.13544.

Le Charlier, B.; Khong, M. T.; Lecoutre, C.; and Deville, Y. 2017. Automatic Synthesis of Smart Table Constraints by Abstraction of Table Constraints. In *Proceedings of IJCAI'17*, 681–687.

Lecoutre, C. 2011. STR2: Optimized Simple Tabular Reduction for Table Constraints. *Constraints : an international journal*, 16(4): 341–371.

Lecoutre, C. 2023. ACE, a Generic Constraint Solver. *CoRR*, abs/2302.05405.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. 2015. STR3: A Path-Optimal Filtering Algorithm for Table Constraints. *Artificial Intelligence*, 220: 1–27.

Lecoutre, C.; and Szczepanski, N. 2020. PyCSP3: Modeling Combinatorial Constrained Problems in Python. *CoRR*, abs/2009.00326.

Li, C. 2008. Airport Gate Assignment: New Model and Implementation. *arXiv:0811.1618 [cs]*.

Li, C. 2009. Airport Gate Assignment A Hybrid Model and Implementation. *arXiv:0903.2528 [cs]*.

Li, H.; Yin, M.; and Li, Z. 2021. Failure Based Variable Ordering Heuristics for Solving CSPs. In Michel, L. D., ed., *CP 21*, volume 210, 9:1–9:10. ISBN 978-3-95977-211-2.

Lim, A.; Rodrigues, B.; and Zhu, Y. 2005. Airport Gate Scheduling with Time Windows. *Artificial intelligence review*, 24(1): 5–31.

L'Ortye, J.; Mitici, M.; and Visser, H. 2021. Robust Flight-to-Gate Assignment with Landside Capacity Constraints. *Transportation Planning and Technology*, 44(4): 356–377.

Mangoubi, R. S.; and Mathaisel, D. F. X. 1985. Optimizing Gate Assignments at Airport Terminals. *Transportation Science*, 19(2): 173–188.

Nethercote, N.; Stuckey, P.; Becket, R.; Brand, S.; Duck, G.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Proceedings of CP'07*, 529–543.

OscaR Team. 2012. OscaR: Scala in OR. https://github.com/pschaus/oscar. Accessed: 2024-02-09.

P. van Hentenryck. 1999. *The OPL Optimization Programming Language*. The MIT Press.

Prud'homme, C.; and Fages, J.-G. 2022. Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*, 7(78): 4708.

Simonis, H. 2007. Models for Global Constraint Applications. *Constraints : an international journal*, 12(1): 63–92.

Stuckey, P.; Becket, R.; and Fischer, J. 2010. Philosophy of the MiniZinc Challenge. *Constraints*, 15(3): 307–316.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2017. Extending Compact-Table to Negative and Short Tables. In *Proceedings of AAAI'17*, 3951–3957.

Vion, J.; and Piechowiak, S. 2017. Une Simple Heuristique Pour Rapprocher DFS et LNS pour les COP. In *Proceedings of JFPC'17*, 39–45.

Yan, S.; Tang, C.-H.; and Chen, M. 2004. A Model and a Solution Algorithm for Airport Common Use Check-in Counter Assignments. *Transportation Research Part A: Policy and Practice*, 38(2): 101–125.

Zhou, N. F.; Kjellerstrand, H.; and Fruhman, J. 2017. *Constraint Solving and Planning with Picat*. Springer.