An efficient solution methodology for the airport slot allocation problem with preprocessing and column generation

Paula Fermín Cueto[1], **Sergio García**[1], **Miguel F. Anjos**[1]

[1] School of Mathematics, The University of Edinburgh, United Kingdom

e-mail: paula.fermin@ed.ac.uk

Airport coordination is a demand control mechanism that maximizes the use of existing infrastructure at congested airports. Aircraft operators submit a list of regular flights that they wish to operate over a five to seven-month period and a designated coordinator is responsible for allocating the available airport slots, which represent the permission to operate a flight at a specific date and time. From an optimization perspective, this problem is a special class of the Resource Constrained Project Scheduling Problem where the objective is to minimize the difference between the allocated and requested slots subject to airport capacity constraints and other operational restrictions. Most studies on the topic focus on developing complex models and fast heuristics. Little attention has been paid to exact methods despite their potential to obtain higher quality solutions with better airline acceptability and fewer slot rejections. In this paper, we present *Caracal*, an efficient column-and-row generation algorithm to solve the single airport slot allocation problem. We also present a problem-specific preprocessing scheme that can identify more redundant constraints and fixed variables than a commercial solver in a fraction of the time. We solve instances originating from some of the most congested airports coordinated by Airport Coordination Limited in the United Kingdom significantly faster than the best exact method in the literature to date. We also conduct experiments on a set of synthetic, realistic instances that we include in this paper, along with the code to generate them, to facilitate benchmarking of slot allocation software.

Key words: Airport slot allocation, integer linear programming, column generation, preprocessing, scheduling.

1 Introduction

Demand often exceeds capacity at congested airports. For this reason, airlines and other aircraft operators need to be granted permission to operate a flight at a specific date and time. This permission is known as a slot and each airport that is congested enough has a designated slot coordinator who is responsible for allocating slots in an transparent, unbiased manner, considering the interests of all the relevant stakeholders. Outside the United States, coordinators follow the World Airport Slot Guidelines (WASG), set out by the International Air Transport Association (IATA (2020)), as well as local regulations, such as the Council regulation (EEC) no. 95/93 (European Commission (1993)) in Europe.

The allocation process works as follows. The year is divided into a summer and a winter season, which span seven and five months, respectively. Several months before the start of each season, airports declare their capacity limits, the most common of which are maximum runway and terminal throughputs, expressed in numbers of flights and passengers, respectively. Shortly

after, airlines submit a list of regular flights that they wish to operate during the season. The coordinator must then make adjustments to the requested services to ensure the capacity is not exceeded. These adjustments may include offering alternative slots, which must be as close as possible to the requested times, or even denying permission to operate in certain cases. The result of this initial coordination is then disclosed to the airlines, which have some flexibility to request changes, return unwanted slots, or even swap slots with other airlines.

Airports require different levels of coordination depending on how congested they are. Level 1 airports are those where the capacity can accommodate all the demand without intervention, and are therefore not relevant for this work. Level 2 airports are airports with potential for congestion during peak periods, and Level 3 airports are those where the demand exceeds the available capacity significantly. The process described above applies to these last two categories. In addition, in Level 3 airports, IATA mandates that coordinators consider priority levels within the requested operations, and distribute the available slots sequentially: historic requests are allocated first, followed by changes to historic requests, new entrants and, lastly, other requests.

Coordinators may need to consider additional constraints such as minimum times on the ground between arriving and departing flights operated by the same aircraft (known as turnaround constraints), or secondary objectives, such as balancing different types of markets or encouraging airline competition. For a more detailed description of the slot allocation process we refer the reader to the WASG.

This problem is receiving increasing attention in the optimization literature in recent years. Slot coordinators use specialized software to perform the initial coordination, but these support tools have limited optimization capabilities and coordinators need to make adjustments manually, resulting in a time-consuming process. It has been shown that optimization-based algorithms have great potential for improving current practice. Ribeiro et al. (2018) analyzed several Portuguese airports and reported a potential improvement in solution quality with respect to the slot coordinator's solution between 4.7% to 27%. Zografos et al. (2012) reported even larger improvements of up to 95% in a study that analyzed the case of three regional Greek airports.

The focus in slot allocation papers is usually placed on modeling the problem. Industry guidelines are ambiguous in the way they define the allocation criteria and different slot coordinators, as well as different researchers, interpret these rules differently. This has resulted in a wide variety of complex optimization models in which the authors propose their own interpretation of the industry guidelines (see Zografos et al. (2016) for a comprehensive review of different slot allocation models in the literature). These models have offered valuable insights: they have identified inefficiencies in the process and provided interesting directions for future changes in policy. As an example, Ribeiro et al. (2019b) and Fairbrother and Zografos (2020) relaxed the rule that stipulates that all flights in the same request should be allocated to the same time, which reduced the total displacements (differences between requested and allocated slots) by 17% to 32% in the airports they tested.

A smaller body of work looks at the slot allocation problem from an algorithmic perspective. In those studies, heuristics are clearly favored over exact algorithms. Zografos et al. (2016) stated that the computational results of exact methods are not encouraging and that trade-offs between complexity and solution quality must be investigated. Subsequent studies have not challenged this assumption and various heuristic methods have been proposed to solve real-world instances. Some of these heuristics are based on the idea of local search (Ribeiro et al. (2019b); Androutsopoulos and Madas (2019); Pellegrini et al. (2011)), variable fixing (Zografos et al. (2012)), or ant colony optimization Castelli et al. (2011).

There exists a strong formulation for the slot allocation problem taken from the scheduling

literature. Androutsopoulos and Madas (2019) showed that this problem can be seen as a class of the minimum weighted earliness-tardiness Resource Constrained Project Scheduling Problem (RCPSP). Such problems are typically modeled using time-indexed formulations, which yield tight linear relaxations. The downside is that they contain a very large number of variables. In slot allocation, this issue is exacerbated by the sheer number of capacity and turnaround constraints. As an example, Edinburgh Airport, a medium-size airport, declared 12 types of capacity constraints in the runway and terminal and received 2,377 slot requests in the 2020 summer season. A minimal time-indexed formulation applied to this airport results in at least 686,953 decision variables and 277,273 constraints. Instances representing large airports and more complex models are considered intractable.

Despite this discouraging property of slot allocation problems, in our view, the search for optimal solutions merits further investigation: better solution quality translates into fewer rejected and/or displaced slots, which in turn leads to a maximized use of airport infrastructure. Moreover, heuristic methods may introduce a bias in the process that goes against the principle of fairness laid out in the WASG. As an example, some constructive heuristics developed for this problem (e.g. Ribeiro et al. (2019a); Fairbrother and Zografos (2020), allocate requests with greater frequency (more weekly operations) in first place, which could put airlines with less frequent services at a disadvantage.

Authors employing exact methods commonly solve the problem using a general-purpose Mixed Integer Programming (MIP) solver and only a few exploit properties of the slot allocation problem in the solution process. Zografos et al. (2012) perform some preprocessing by aggregating days with identical requests and they strengthen their lower bounds with reduce-and-split cuts. Fairbrother et al. (2019) avoid dividing the days into the standard 5-minute intervals when the declared capacity limits do not require this level of detail, and instead choose a greater interval that allows them to eliminate an important amount of decision variables. However, it is not clear how they define displacements between the requested and allocated slots or how they handle turnaround constraints in these cases. The same authors noted that most of the capacity constraints are inactive at the optimal solution and they added these constraints as lazy cuts. Although this approach seems sensible, it cannot solve the problem size issue completely: identifying violated constraints in each iteration remains an expensive operation.

There are a number of techniques with the potential to increase the efficiency of exact algorithms in the area of slot allocation. Column generation and more effective preprocessing are promising options that may alleviate the difficulties associated with problem size in time-indexed formulations (van den Akker et al. (2000)). Not only could they be effective to solve the models currently used in practice, but they could be the only viable strategies for obtaining optimal solutions to problems with a larger number of variables, such as the model with season segmentation of Fairbrother and Zografos (2020) or problems considering a network of airports simultenously (Benlic (2018); Corolli et al. (2014)). Lagrangian-based methods have also been suggested as a promising option (Zografos et al. (2016)). However, to the best of our knowledge, other than some basic preprocessing, none of these techniques have been explored in the literature.

In this paper we show that it is possible to find optimal solutions for real-world problems quickly and with modest memory requirements. We present a column-and-row generation algorithm for solving the general airport slot allocation problem that starts with a reduced set of variables and constraints and increases the problem size dynamically until a solution is obtained that is feasible for the complete problem. Throughout this process, a problem-specific preprocessing scheme is used to identify which variables and constraints are redundant and can be excluded from the model. We also show that this preprocessing pipeline can alternatively be used as an initial step in other

solution methods, including heuristics.

One important aspect hindering the research on this topic is the absence of publicly available data. Slot request data sets contain confidential information involving various stakeholders and they are difficult to access. Research groups usually establish collaborations with a slot coordinator and report computational results from two or three data sets provided by this coordinator. However, this data is not published, and this makes identifying the state-of-the art a difficult task, as two different models or algorithms are hardly ever tested on the same data set. There is a need to provide researchers with a large collection of test instances to facilitate benchmarking of slot allocation software. A secondary objective of this work is to develop a procedure to generate realistic synthetic data sets and to make this procedure and the resulting data publicly available.

The remainder of this paper is structured as follows: basic definitions, notation and the mathematical formulation are introduced in Section 2. Our preprocessing scheme is presented in Section 3, followed by a description of the column-and-row generation algorithm in Section 4. Details of the synthetic data generator are given in Section 5. Computational results using synthetic data as well as real-world data provided by Airport Coordination Limited (ACL) are presented in Section 6, followed by conclusions in Section 7.

2 The Slot Allocation Model

The complexity of the slot allocation problem is exacerbated by the lack of consensus in the literature as to how the IATA guidelines and other regulatory frameworks should be interpreted. This has led to the development of a number of integer programming models exploring different directions or providing new interpretations of an existing element. From equity and fairness considerations, to calendar segmentation, airline preferences or different displacement functions, every author—and possibly every slot coordinator—has their own point of view regarding how the problem should be modeled. Because our desire is to develop a slot allocation algorithm that is flexible and makes minimal assumptions, in this work we only include those elements that are present in virtually every slot allocation model: a linear displacement function, terminal and runway capacity constraints, and minimum turnaround constraints.

We choose a type of time-indexed formulation that has been widely used to model a variety of scheduling problems and has found its way into the slot allocation literature as well. It was first introduced by Christofides et al. (1987) in the context of the RCPSP. Zografos et al. (2012) adapted it to the slot allocation problem, proposing a "minimal" integer program that has served as a starting point for more complex models since then.

Let D denote the set of calendar days in the schedule under consideration, which typically corresponds to an aviation season. Each calendar day is divided into 288 5-minute intervals, denoted by the set $T = \{1, \ldots, 288\}$. A series of slots is a set of at least five flights that request to operate at the same time with the same characteristics (flight number, aircraft type, origin/destination, etc.), differing only in the date in which they operate. Let N denote the set of requested series of slots in the season under consideration. Parameter $\tau_i \in T$ represents the flight time requested for all flights in series $i \in N$ and we use a boolean parameter δ_{id} to indicate whether a series $i \in N$ operates on day $d \in D$.

E is the set of pairs of requests that represent linked arriving and departing flights operated by the same aircraft. For each pair $(i, j) \in E$ we must ensure that a minimum time on the ground t_{ij} is observed to allow the airline to perform handling activities such as embarking and disembarking passengers, cleaning or refuelling.

As for the airport parameters, we use C to denote the set of declared capacity restrictions. Each restriction $c \in C$ represents a maximum demand throughput allowed in small periods of time (or capacity windows) of a given duration w_c and with start times $s \in T_c$. These capacity windows have a periodic nature, meaning that times $s \in T_c$ are multiples of a given frequency, represented by parameter λ_c. For example, if $\lambda_c = w_c = 6$, then a capacity limit must be observed in each 30-minute period of each day, starting on the half hour ($T_c = \{1,\ 7,\ \ldots,\ 277,\ 283\}$). Sometimes $w_c\ > \lambda_c$, resulting in overlapping capacity windows. This is the case when an airport declares, for instance, a maximum runway throughput in 15-minute windows ($w_c = 3$), rolling every five minutes ($\lambda_c = 1$). Lastly, parameter B_c^{ds} represents the maximum number of passengers or flights of a capacity restriction $c \in C$ at start time $s \in T_c$ on day $d \in D$. In practice, these limits are usually the same on different days and may even be constant throughout the day.

Lastly, parameter a_{ic} represents the units of resource consumed by flights in series $i \in N$ with respect to all capacity constraints $c \in C$. If the series i is not relevant for those capacity constraints (for example, an arriving flight and a constraint on the number of departures or a mismatch in the terminal buildings), then $a_{ic} = 0$. Otherwise, a_{ic} is the number of passengers in the flight if c is a terminal constraint and $a_{ic} = 1$ in the case of runway constraints.

We define binary decision variables x_{it} that take value 1 if series $i \in N$ is allocated to time interval $t \in T$ and 0 otherwise. These variables do not need to be indexed by the calendar day, as the WASG recommend that flights in the same series should not be given different times unless the airline indicated that this is acceptable. The main difference between the slot allocation model that we use in this paper and the one introduced by Zografos et al. (2012) is the addition of binary variables y_i that take value 1 if and only if the request to operate the series $i \in N$ is rejected. Without this consideration, one must assume that the airport capacity is sufficient to accommodate all requested flights, which may not be true in congested airports, especially in Level 3 airports.

The model that we propose is the following:

$$\min_{x,y} \quad \sum_{i\in N}\sum_{t\in T} f_{it}\, x_{it} + \omega \sum_{i\in N}\sum_{d\in D} \delta_{id}\ y_i \tag{1}$$

$$\text{s.t.} \quad \sum_{t\in T} x_{it} + y_i = 1 \qquad \forall\ i \in N, \tag{2}$$

$$\sum_{i\in N}\sum_{t=s}^{s+w_c-1} \delta_{id}\, a_{ic}\, x_{it} \le B_c^{ds} \qquad \forall\ c \in C,\ d \in D,\ s \in T_c, \tag{3}$$

$$\sum_{t=s-t_{ij}+1}^{|T|} x_{it} + \sum_{t=1}^{s} x_{jt} \le 1 \qquad \forall\ (i,j) \in E,\ s \in T, \tag{4}$$

$$x_{it} \in \{0,1\} \qquad \forall\ i \in N,\ t \in T, \tag{5}$$

$$y_i \in \{0,1\} \qquad \forall\ i \in N. \tag{6}$$

The objective function (1) minimizes two terms: the number of rejected flights and the total displacement across all flights. f_{it} represents the cost of allocating all flights in series $i \in N$ to slot $t \in T$ and it is calculated as $\sum_{d\in D} \delta_{id}\ |t - \tau_i|$. Ideally, all series would operate at their requested

time τ_i. However, due to capacity restrictions—and unless we are solving a trivial problem—some requests will have a nonzero displacement in the optimal solution. We consider a weight ω much larger than $\max_{i,t}\{f_{it}\}$ (e.g. 10,000) to prioritize avoiding rejected requests. Constraints (2) are the assignment constraints and they ensure that exactly one time interval is selected for each series, unless the series is rejected. Capacity constraints are defined in (3) and constraints (4) impose a minimum turnaround time t_{ij} between linked flights.

Precedence relations, such as the turnaround constraints in this problem, are common in scheduling problems and they have a strong and a weak version in terms of the strength of their linear relaxation (Artigues (2017)). Constraints (4) use the strong version and they have the downside that they add $|T| \times |E|$ rows to the constraint matrix. The weak version of these constraints was proposed by Pritsker et al. (1969): $\sum_{t\in T} t\, x_{jt} - \sum_{t\in T} t\, x_{it} \geq t_{ij}$. This formulation only requires one constraint per pair of connected activities $(i,j) \in E$. However, this is an aggregation of constraints (4) and it yields weaker lower bounds. These two options have been used indistinctively in slot allocation models in recent years. We favor obtaining tighter relaxations because problem size is not an issue when we use the preprocessing techniques presented in Section 3 or the solution algorithm presented in Section 4.

Other authors have preferred time-indexed formulations with step variables z_{it} that take value 1 if and only if the series i is allocated *no earlier* than time t (see examples in Ribeiro et al. (2019b) and Jacquillat and Odoni (2015)). A polyhedral study carried out by Sousa and Wolsey (1992) revealed that these two formulations are equivalent, all other elements being equal.

3 Preprocessing

As discussed in Section 2, the main limitation of time-indexed formulations is their size. Zografos et al. (2012) reduced the number of capacity constraints by identifying calendar days with the same set of series and removing constraints of all but one of those days. Although this simple technique can eliminate a significant number of constraints, it assumes that the capacity limits are constant throughout the season, which may not be the case for some airports, and the remaining problems are still of a considerable size.

In this section we describe several preprocessing rules that can remove a large proportion of redundant constraints and fixed variables from the formulation described in (1)-(5) with minimal computational effort. These rules are based on well-known preprocessing principles, but they exploit the structure of the underlying problem to find dominance relations in a more efficient manner.

This preprocessing pipeline is particularly effective when requests can only be displaced within certain limits. This is the case when an assumption is made about the maximum displacement that is acceptable to an airline or when the column generation algorithm described in Section 4 is used. More generally, this preprocessing can bring substantial improvements in run times and memory usage even when variables x_{it} are defined for all $t \in T$.

3.1 Fixed variables

Preprocessing Rule 1. Dominated variables

This preprocessing step exploits dominance relations between variables and is a more formal description of the change of granularity proposed by Fairbrother et al. (2019). Given a MIP of the form $\min\{c^T x \;/\; Ax \leq b,\ \ell \leq x \leq u,\ x_j \in \mathbb{Z}\}$, it is known that variable x_i dominates x_j ($x_i \succ x_j$) if $c_i \leq c_j$ and $a_{ki} \leq a_{kj}$ for all constraints $a_k x \leq b_k$ in the problem (see Gamrath et al. (2015)).

In slot allocation, when the frequency λ_c is greater than five minutes for all capacity restrictions $c \in C$ declared at the airport—in other words, when no capacity constraints of the same type c start at consecutive 5-minute intervals—the constraint matrix contains sets of duplicate variables in the sense that they have the same coefficients in the same rows. Eliminating all but one of the variables in each of these sets can reduce the problem size substantially.

Example 1. *Consider an airport that only declared one capacity limit c imposing a maximum runway throughput in non-overlapping 30-minute windows ($w_c = \lambda_c = 6$, $T_c = \{1, 7, \ldots, 277, 283\}$), and consider also a series requesting to operate at 05:10 ($\tau_i = 62$). If turnaround constraints do not apply to this series, it is clear that there is at most one variable in each 30-minute window that can take value 1 in an optimal solution. Each of these variables, highlighted in black in Figure 1, dominate the other variables in the same window because they have the same coefficient in the corresponding capacity constraint but the dominated variables would cause a higher displacement, as they are further away from τ_i.*

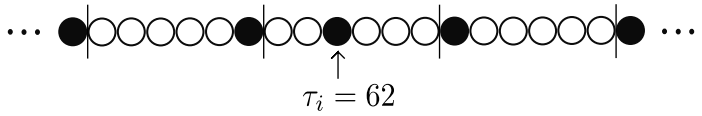

Figure 1: Dominated variables for a series i with τ_i=62 and capacity constraints c with $\lambda_c = w_c = 6$.

Proposition 1. *Given a slot allocation problem with one set of capacity constraints c with frequency λ_c and duration w_c multiple of λ_c, and no turnaround constraints, then variables x_{it} for indices $t \in T$ not listed below can be fixed to 0:*

- $t = \tau_i$,
- $t = k\lambda_c, \;\; k \in \mathbb{Z}^+, \;\; t > \tau_i$,
- $t = k\lambda_c - 1, \;\; k \in \mathbb{Z}^+, \;\; t < \tau_i$.

Proof. See Appendix A.1. □

To extend this to the more general case with multiple capacity restrictions, we simply need to identify which variables are dominated for each family of constraints $c \in C$ and take the intersection of all these sets of variables. If a variable is dominated for all $c \in C$, it can be fixed to 0.

Lastly, we explore the case where a series $i \in N$ is *linked*, namely, when turnaround constraints (4) must be added to impose a minimum ground time between i and some other series $j \in N$. In some cases, variables x_{it} that were identified as dominated in the previous step can still be fixed to 0.

Proposition 2. *Let $(i, j) \in E$ represent a pair of linked arriving and departing series. Variables x_{it} for all $t > \tau_i$ and variables x_{jt} for all $t < \tau_j$ are dominated with respect to turnaround constraints.*

Proof. See Appendix A.2. □

Special care needs to be taken when $i \in N$ is an arrival series and $t < \tau_i$, as well as the symmetric case: departures $j \in N$ and $t > \tau_j$. In such cases, some variables that were identified as dominated based on the analysis of the capacity constraints, may no longer be fixed.

Proposition 3. *Let $(i, j) \in E$ represent a pair of linked arriving and departing series. Assume that some variables have been identified as dominated when only capacity constraints were considered. Given a time period $t' > \tau_j$, if there exists a variable $x_{it''}$ that was not fixed in previous preprocessing steps and for which the following conditions are met:*

1. *$\tau_j \leq t'' + t_{ij} < t'$,*
2. *variables x_{it} with $t'' < t \leq t' - t_{ij}$ have been fixed to 0,*

then, variable $x_{jt'}$ is dominated by variable $x_{j,t''+t_{ij}}$ and therefore it can be fixed to 0 with respect to minimum turnaround constraints.

The same is true for the arriving series i: a previously fixed variable $x_{it'}$, with $t' < \tau_i$, is fixed to 0 if there exists a variable $x_{jt''}$ that has not been fixed such that t'' satisfies (i) $t' < t'' - t_{ij} \leq \tau_i$ and (ii) variables x_{it} with $t' + t_{ij} \leq t < t''$ are fixed to 0.

Proof. See Appendix A.3. □

Figure 2 illustrates the application of Preprocessing Rule 1 to an instance with a pair of linked series $(i, j) \in E$ with requested times τ_i, τ_j, minimum ground time $t_{ij} = 4$, and one family of capacity constraints c with $\lambda_c = w_c = 6$. The solid circles denote variables that cannot be fixed due to capacity constraints c (Proprosition 2), and the circles with a striped pattern are variables that cannot be fixed due to turnaround constraints (Proposition 3).

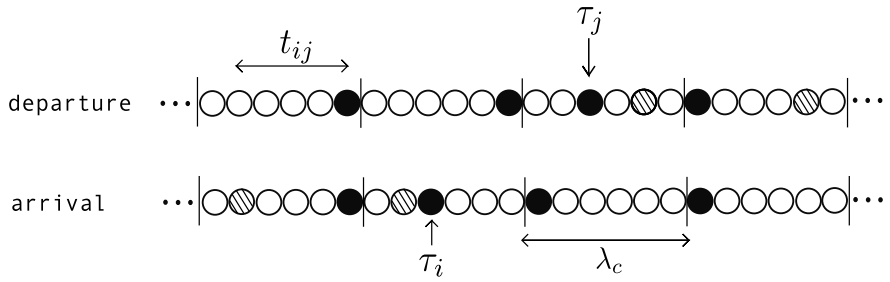

Figure 2: Illustration of process to identify fixed variables in small instance with two linked series.

Preprocessing Rule 2. Infeasible allocations

Consider a capacity constraint $c \in C$ on day $d \in D$ at time $s \in T$ (from now on we will use the notation (c, d, s) for simplicity). If variable x_{it} is included in the left-hand side of constraint (c, d, s) and $a_{ic} > B_c^{ds}$, then it can be fixed to 0, as choosing slot t for series i is not feasible.

3.2 Redundant capacity constraints

Due to the periodic nature of series of slots, which consist of regular flights, slot allocation problems contain many redundant capacity constraints. The following rules aim to identify most of these redundancies. Preprocessing Rules 4 and 5 are relevant only when variables x_{it} are defined within a time window $[e_i, \ell_i]$, $e_i, \ell_i \in T$, $e_i \leq \tau_i \leq \ell_i$.

Preprocessing Rule 3. Redundant scheduling days

This extends the preprocessing rule proposed by Zografos et al. (2012), which removes identical scheduling days. Consider the sets of all capacity constraints in two days $d, d' \in D$. Assume that the following two conditions are met:

- capacity limits on day d never exceed those in day d', namely, $B_c^{ds} \leq B_c^{d's} \quad \forall\, c \in C,\ s \in T_c$,
- all the series requested on day d' are also requested on day d, namely, $\delta_{id} \geq \delta_{id'} \quad \forall\, i \in N$.

Then, constraints (c, d, s) dominate (c, d', s):

$$\sum_{i \in N} \sum_{t=s}^{s+w_c-1} \delta_{id'}\, a_i\, x_{it} \leq \sum_{i \in N} \sum_{t=s}^{s+w_c-1} \delta_{id}\, a_i\, x_{it} \leq B_c^{ds} \leq B_c^{d's}. \tag{7}$$

Preprocessing Rule 4. Maximal activity does not exceed capacity limit

Given a capacity constraint $c \in C$, if allocating all possible flights at time $s \in T_c$ on day $d \in D$ does not break the capacity restriction B_c^{ds}, then this constraint is redundant. Let U_{cds} denote the set of series that have at least one variable x_{it} included in capacity constraint (c, d, s):

$$U_{cds} = \{i \in N \,/\, \delta_{id} = 1,\, \{t \in T \ /\ s \leq t < s + w_c\} \ \cap\ [e_i,\, \ell_i] \ \neq\ \varnothing\}\,. \tag{8}$$

The following inequality is true for all capacity constraints:

$$\sup \left\{ \sum_{i \in N} \sum_{t=s}^{s+w_c-1} \delta_{id}\ a_{ic}\ x_{it} \right\} \leq \sum_{i \in U_{cds}} a_{ic} \qquad \forall\ c \in C,\ d \in D,\ s \in T_c. \tag{9}$$

We can safely remove capacity constraints (c, d, s) that meet the condition

$$\sum_{i \in U_{cds}} a_{ic} \leq B_c^{ds}. \tag{10}$$

Preprocessing Rule 5. Dominated time windows

A natural extension of Preprocessing Rule 3 is to compare sets of series in smaller time periods, not only whole days.

Using U_{cds}, a constraint (c, d, s) of the form (3) can be rewritten as

$$\sum_{i \in U_{cds}} \sum_{t=s}^{s+w_c-1} a_{ic}\, x_{it} \leq B_c^{ds}. \tag{11}$$

By the same logic applied in Preprocessing Rule 3, if $U_{cd's} \subseteq U_{cds}$ and $B_c^{ds} \leq B_c^{d's}$ then constraint (c, d', s) is dominated by (c, d, s) and therefore it can be removed from the model.

Preprocessing Rule 6. Redundant arriving or departing limits

When declaring capacity limits in the runway, airports often define restrictions for arriving, departing and total movements with the same frequency λ_c and duration w_c. Consider three capacity constraints (c_A, d, s), (c_D, d, s) and (c_T, d, s), where c_A, c_D and c_T denote capacity limits for arrivals, departures and total movements, respectively. All other characteristics of these constraints

are identical. It is clear that $U_{c_A ds} \subseteq U_{c_T ds}$ and $U_{c_D ds} \subseteq U_{c_T ds}$. If $B^{ds}_{c_A} \geq B^{ds}_{c_T}$, then, following the same reasoning used in Preprocessing Rule 3, constraint (c_A, d, s) is dominated by (c_T, d, s) and can be removed. Similarly, if $B^{ds}_{c_D} \geq B^{ds}_{c_T}$, then constraint (c_D, d, s) is redundant.

It may appear that this step is unnecessary as long as the capacity limits have been defined properly by the airport: limits on total movements should never be lower than limits on arrivals or departures. However, in Level 3 airports, the limits B^{ds}_c must be updated between consecutive sub-problems for different priority groups, and this may give rise to situations where $B^{ds}_{c_D} \geq B^{ds}_{c_T}$ or $B^{ds}_{c_A} \geq B^{ds}_{c_T}$.

3.3 Redundant turnaround constraints

Preprocessing Rule 7. Dominance relations between consecutive constraints

Proposition 4. *Let $i \in N$ be an arrival series, $j \in N$ its linked departure series and t_{ij} the minimum turnaround time between them. If variable x_{js} for some $s \in T$ is fixed to 0, then the turnaround constraint with index s is dominated by the constraint with index $s-1$.*

Proof. See Appendix A.4. □

Preprocessing Rule 8. Minimum and maximum $s \in T$

A minimum turnaround constraint can be removed from the model if it does not include at least one assignment variable of each of the two linked requests $(i,\, j) \in E$ or if it is dominated by another turnaround constraint.

Proposition 5. *Let e_i and ℓ_i denote the earliest and latest available slots for series $i \in N$, respectively. Let $(i,j) \in E$ be a pair of linked series with minimum ground time t_{ij}. Any turnaround constraints with indices $s \in T$ not included in* (12) *are redundant.*

$$\sum_{t=s-t_{ij}+1}^{\ell_i} x_{it} + \sum_{t=e_j}^{s} x_{jt} \leq 1, \quad \forall\, (i,j) \in E,\ s \in T\ /\ \max\{e_j,\ e_i + t_{ij} - 1\} \leq\ s\ \leq \min\{\ell_j,\ \ell_i + t_{ij} - 1\}. \tag{12}$$

As with some of the preprocessing rules for capacity constraints, this rule is only useful if variables x_{it} are defined within some time periods $e_i,\, \ell_i \in T$.

Proof. See Appendix A.5. □

4 The *Caracal* Algorithm

In this section we present the *Caracal* (*Column-And-Row generation branch-And-Cut ALgorihtm*) algorithm for single airport slot allocation problems. It is inspired by *Zebra*, an algorithm developed by García et al. (2011) to find exact solutions of very large instances of the p-median problem. In essence, the authors take advantage of the problem structure introduced by a radius formulation to solve the linear relaxation in an iterative manner, starting with a reduced model that only contains a small subset of columns and rows and increasing these subsets in every iteration until an optimal solution is obtained that is feasible for the complete problem.

Despite the evident differences between these two problems, the ideas that gave rise to *Zebra* are also valid for slot allocation. In both problems we can anticipate that a large proportion of the assignment variables are unpromising due to their high cost. It is known from conversations with slot coordinators that most slot requests are usually granted permission to operate at the requested time. Another property of the p-median formulation exploited by García et al. (2011) is the natural ordering of variables in the sense that the cost coefficients increase monotonically with the variable index. This property is also found in the slot allocation model in (1)-(5), only in this case, the cost coefficients of variables x_{it} increase linearly with $|\tau_i - t|$, regardless of the direction of the displacement.

The remainder of this section describes the steps followed in *Caracal* to solve the linear relaxation (LP) of the slot allocation model (1)-(5), as well as the branch-and-cut framework that this process is embedded in to find integer solutions. The flow of the algorithm is depicted in Figure 3.

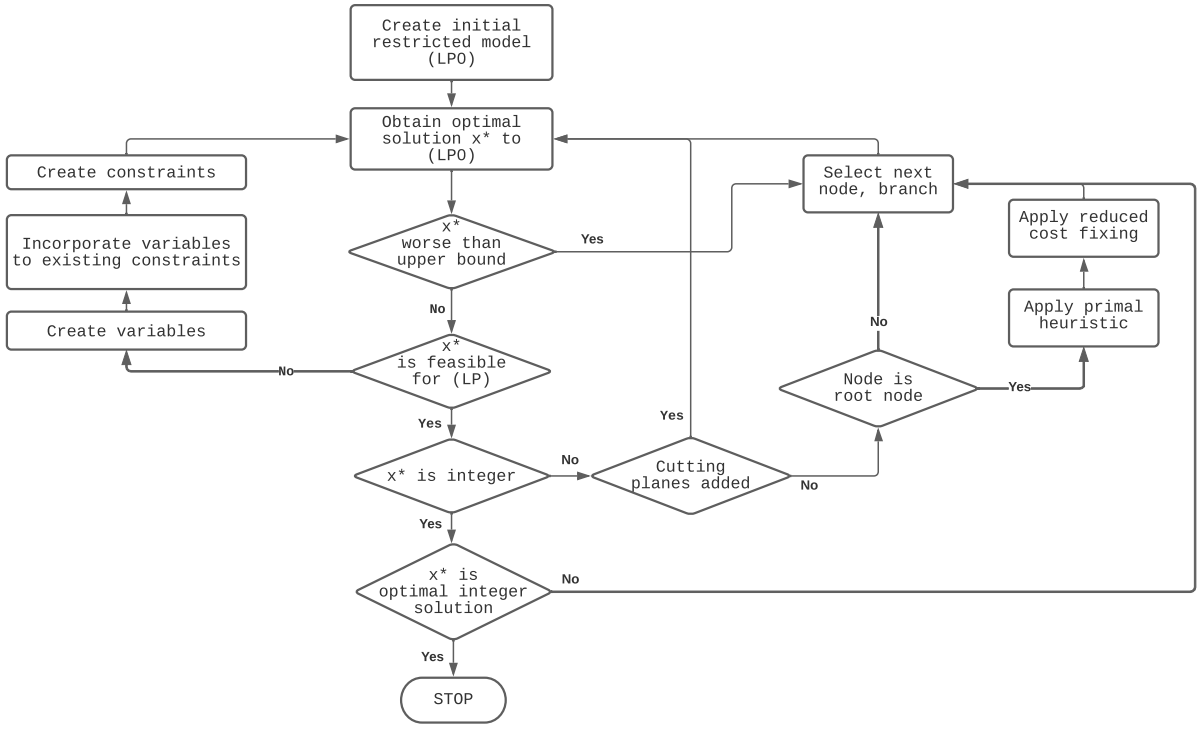

Figure 3: Flow chart of the *Caracal* algorithm, from an initial relaxed model (LP_0) to the optimal integer solution.

4.1 Initial reduced model

The starting point of *Caracal* is a reduced model (LP_0) with a small subset of variables and constraints from the linear relaxation of the original formulation in (1)-(5), denoted by (LP). The first step to obtain (LP_0) is to fix variables in (LP) using the preprocessing rules in Section 3.1. Next, variables x_{it} such that $t \neq \tau_i$ are removed from all capacity (3) and turnaround constraints (4). Note that because constraints (3) and (4) take the form $a\,x \leq b$, $a \geq 0$ and this is a minimization problem, this last step gives rise to a relaxation of (LP).

Let $e_i \in T$ denote the latest slot such that $e_i < \tau_i$ and variable x_{ie_i} was not eliminated in

preprocessing, and let ℓ_i be the earliest slot such that $\ell_i > \tau_i$ with $x_{t\ell_i}$ still in the model after preprocessing. Variables x_{it} with $t < e_i$ or $t > \ell_i$ are now dominated by x_{ie_i} and $x_{i\ell_i}$ in (LP_0), respectively, since they have the same coefficients in the constraint matrix (they only appear in assignment constraints) but variables x_{ie_i} and $x_{i\ell_i}$ represent a lower displacement (they are closer to the requested time τ_i). Therefore all variables x_{it} with $t < e_i$ or $t > \ell_i$ can be fixed to 0.

Removing all variables except $x_{i\tau_i}$ from all turnaround and capacity constraints makes a great proportion of these constraints redundant. The final step to obtain (LP_0) consists in applying the preprocessing rules outlined in Sections 3.2 and 3.3 to remove most of these redundant constraints.

4.2 Column-and-row generation

Once the initial reduced model (LP_0) has been created and preprocessed, the next step in *Caracal* is to apply the dual simplex method to obtain an optimal solution to (LP_0), denoted by x^*. If $x^*_{ie_i} = x^*_{i\ell_i} = 0 \ \forall \ i \in N$, then this solution is feasible for (LP), since x_{ie_i} and $x_{i\ell_i}$ are the only variables that are excluded from capacity and turnaround constraints in (LP_0). If x^* is feasible for (LP) then it is also optimal, since (LP_0) is a relaxation of (LP). Otherwise, if $x^*_{ie_i} \neq 0$ or $x^*_{i\ell_i} \neq 0$ for some $i \in N$, we need to add some variables x_{it} to (LP_0) following the steps in Algorithm 1 and repeat this process until the optimal solution of (LP) is found.

Step 1.	For each $i \in N$, initialize e_i and ℓ_i as highest cost slots such that x_{ie_i} and $x_{i\ell_i}$ are included in (LP_0) and $e_i < \tau_i < \ell_i$;
Step 2.	Construct the set $\Gamma = \{i \in N : x^*_{ie_i} > 0\} \cup \{i \in N : x^*_{i\ell_i} > 0\}$;
Step 3.	Construct the set $\mathcal{A}$ of *original* capacity and turnaround constraints from (LP) such that the reduced version of them is included in (LP_0) and is active in solution x^*;
Step 4.	For each $i \in \Gamma$:
	Step 4.1. Compute e'_i as the latest $t \in T$ such that $e'_i < e_i$ and variable $x_{ie'_i}$ does not appear in any constraint in $\mathcal{A}$. *Step 4.2.* Compute ℓ'_i as the earliest $t \in T$ such that $\ell'_i > \ell_i$ and variable $x_{i\ell'_i}$ does not appear in any constraint in $\mathcal{A}$;
Step 5.	For each $i \in \Gamma$:
	Step 5.1. Create variables x_{it} for each $t \in [e'_i, e_i) \cup (\ell_i, \ell'_i]$, excluding fixed variables. *Step 5.2.* Incorporate variables x_{it} created in Step 5.1. in assignment constraints and objective function. *Step 5.3.* Incorporate variables x_{it} for each $t \in (e'_i, e_i] \cup [\ell_i, \ell'_i)$ in capacity and turnaround constraints.

Algorithm 1: Algorithm to generate variables given a reduced model (LP_0) and optimal solution x^*.

The row generation part of the algorithm arises when variables x_{it} are incorporated into capacity and turnaround constraints in Step 5.3. of Algorithm 1. After the column generation step, some constraints that were removed from (LP_0) when the initial model was created may no longer be redundant. The preprocessing rules described in Sections 3.2 and 3.3 are used to determine which previously removed constraints must be added back to (LP_0).

In *Zebra*, at most one variable per facility is generated in a single iteration of the algorithm. In *Caracal*, multiple x_{it} for one $i \in N$ can be generated at the same time, as described in Steps 5.1.-5.3. in Algorithm 1. This is done to avoid unnecessary iterations in situations where it can be anticipated that the next available variable is not promising. This is best explained with an example.

Example 2. *Consider a scenario with two series $i_1, i_2 \in N$ requesting to operate at 12:25 and 12:30, respectively, in an airport that declared two runway limits: a maximum of one flight per hour and a maximum of one flight in each 10-minute period. Preprocessing Rule 1 renders half of the variables redundant (see Figure 4). Also in Figure 4, the diagram on the top left shows the variables included in the initial model (LP_0) and its optimal solution x^*. We cannot conclude that this solution is optimal for (LP), since $x^*_{i_2,e_{i_2}} > 0$. If we use a naive approach and generate a single variable x_{it} for each $i \in \Gamma$, it will take 12 iterations to reach optimality, one for each new variable in the bottom right diagram in Figure 4 that was not present in the initial (LP_0).*

However, we know that at most one of i_1 or i_2 can be scheduled between 12:00 and 12:55 due to the hourly runway limit, so we can save a few steps if we generate enough variables in each iteration to allow a series to "escape" the time window of this active constraint. As a result, we reach optimality in just three iterations: two to generate variables outside the interval 12:00-12:55 for each of the series, and one to verify optimality. This is the rationale behind Algorithm 1.

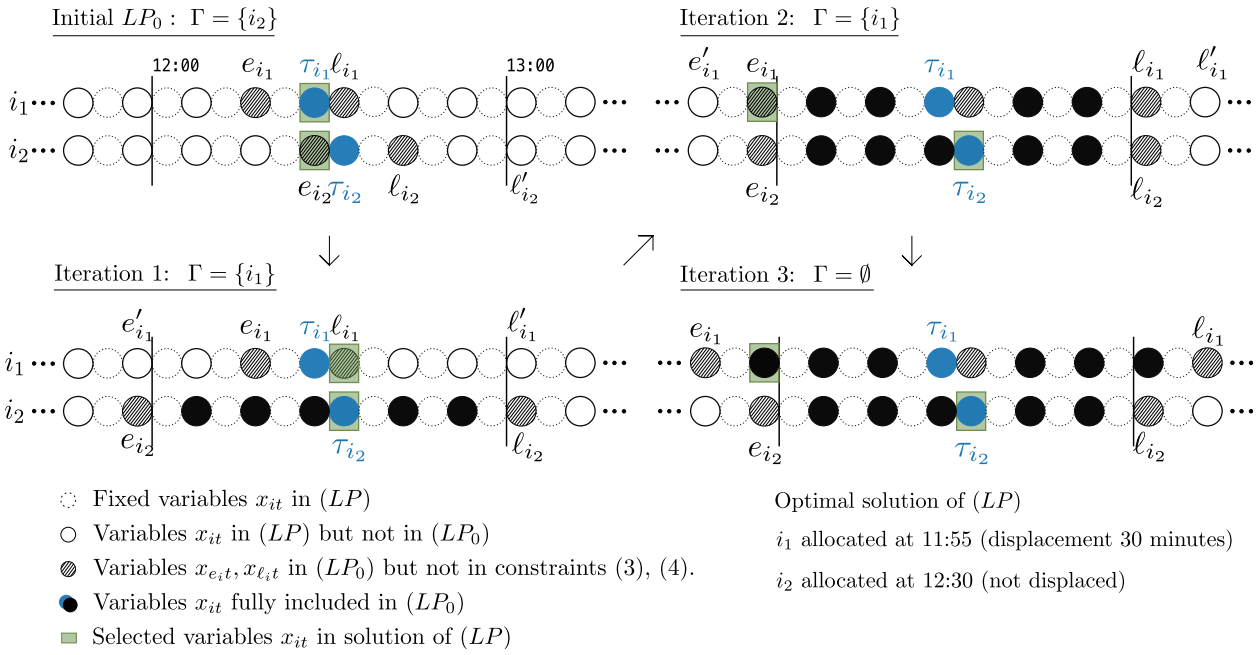

Figure 4: Column generation process in a small instance from initial (LP_0) to convergence to solution of (LP).

4.3 Cutting planes

Even though the time-indexed formulation (1)-(5) has a tight linear relaxation, cutting planes are required to obtain a satisfactory performance in harder instances. We implemented two types of cutting planes that proved to be effective, simple to implement and easy to combine with column generation: lifted knapsack cover cuts and lifted Generalized Upper Bound (GUB) knapsack cover cuts.

To derive lifted knapsack inequalities that cut off a fractional solution x^* we follow the separation procedure of Gu et al. (1998) with the enhancements proposed by Kaparis and Letchford (2008).

This procedure consists of a coefficient-independent, greedy heuristic for cover generation followed by sequential down-lifting and up-lifting.

A GUB cover for a knapsack constraint is a cover such that no two elements in the cover belong to the same GUB constraint, given a set of non-overlapping GUB constraints. This additional requirement usually leads to stronger inequalities, but it makes the lifting problem much harder (Gu et al. (1998)). We take advantage of the structure of the slot allocation problem to overcome this issue. For the sake of readability, let T_{cs} denote the set of slots included in the time window of a constraint $c \in C$, namely, $T_{cs} = \{t \in T \ / \ s \leq t \leq s + w_c - 1\}$. The following integer program represents the separation problem to find the most violated GUB cover—or to prove that none exists—introduced by Wolsey (1990), adapted to a terminal capacity constraint (c, d, s) with the set of non-overlapping GUBs derived from assignment constraints (2):

$$\alpha = \min_{z} \quad \sum_{i \in N} \sum_{t \in T_{cs}} (1 - \sum_{t \in T_{cs}} x^*_{it})\ z_{it}, \tag{13}$$

$$\text{s.t.} \quad \sum_{i \in N} \sum_{t \in T_{cs}} \delta_{id}\, a_{ic}\, z_{it} > B^{ds}_c, \tag{14}$$

$$\sum_{t \in T_{cs}} z_{it} \leq 1 \quad \forall\, i \in N, \tag{15}$$

$$z_{it} \in \{0, 1\} \quad \forall\, i \in N,\, t \in T_{cs}. \tag{16}$$

Variables z_{it} with the same index i have the same coefficient in every constraint and the same costs in the objective function. This detail simplifies the separation problem substantially: we need to know whether a variable from a request i is selected in the optimal solution of the separation problem, but we do not need to know which one. This allows us to apply the transformation $\zeta_i = \sum_{t \in T_{cs}} z_{it}$. The separation problem in (13)-(16) can now be rewritten as

$$\alpha = \min_{\zeta} \quad \sum_{i \in N} (1 - \sum_{t \in T_{cs}} x^*_{it})\ \zeta_i, \tag{17}$$

$$\text{s.t.} \quad \sum_{i \in N} \delta_{id}\, a_{ic}\, \zeta_i > B^{ds}_c, \tag{18}$$

$$\zeta_i \in \{0, 1\} \quad \forall\, i \in N. \tag{19}$$

The above problem is a standard knapsack cover separation problem that can be solved using the procedure employed for lifted knapsack covers, adding one step at the end: once a lifted cover based on variables ζ is found, it is converted into a GUB knapsack cover inequality in the original space by simply including all variables x_{it} for each request i in the cover (requests such that $\zeta^*_i = 1$) and each time period $t \in T_{cs}$.

Combining these knapsack inequalities with column generation is simple. Adding variables to an existing capacity constraint does not invalidate any previously generated cuts, but these new variables will give rise to stronger cuts if we can lift the new variables into existing cover inequalities. In the case of knapsack cover cuts, we apply the same up-lifting procedure used when deriving the

initial cut to the new variable. In the case of GUB knapsack cover cuts, we first check if the series i associated with the new variable x_{it} is in the GUB cover. If it is, x_{it} is added to the inequality with the same lifting coefficient as the other variables of series i, otherwise, the up-lifting problem is solved to determine the largest possible lifting coefficient.

These cuts are applied in every node of the branching tree. We generate variables until we obtain a solution for (LP_0) that is feasible for the complete problem, then we add cutting planes, reoptimize the linear relaxation, and repeat this process until no more cuts can be generated.

4.4 Additional implementation details

Early stopping: As pointed out by García et al. (2011), the intermediate solutions of (LP_0) yield lower bounds on the optimal solution of (LP). When solving the linear relaxation of a node of the branching tree with column-and-row generation, if the objective value of some intermediate solution is worse than the current upper bound on the optimal integer solution, this node can be discarded immediately, before the linear relaxation is solved to optimality.

Primal heuristic: We apply a heuristic similar to RENS (Berthold (2014)) in the root node. Starting from the solution to the linear relaxation, x^*, we fix the variables of all series that were allocated to the requested time, namely, series $i \in N$ with $x^*_{i\tau_i} = 1$, and solve the resulting sub-MIP.

Reduced cost fixing: Once the root node is solved to optimality and after applying the primal heuristic described above, we eliminate all variables with a reduced cost RC such that $LB + RC > UB$, where LB and UB are the lower and upper bound on the optimal solution, respectively. Then, the preprocessing rules outlined in Section 3.2 are used to eliminate redundant capacity constraints after reduced cost fixing.

Branching and node selection: We use reliability branching with the procedure and default parameters proposed by Achterberg et al. (2005). The node selection strategy is depth-first search exploring the up branch first, to encourage integer solutions early in the search. When depth-first search gets stuck because it encounters either an infeasible node or an integer solution, then best-bound search is used to select the next node to explore.

LP reoptimization: A dual problem remains feasible when new rows are added to the primal. The dual simplex algorithm can be warm-started from the previous basis by simply treating the slack variables of any new rows as basic variables (Hillier and Lieberman (2001)). However, in *Caracal*, we also add new columns and modify variable coefficients of existing rows, which can invalidate the basis and cause the dual simplex to restart the optimization after each pricing iteration. This can be avoided following these steps in the column-and-row generation process:

1. In Step 5.1 of Algorithm 1, add new variables x_{it} to (LP_0) as nonbasic at the lower bound, leaving the basic status of all other variables unchanged. This new basic solution is still feasible and optimal: the new variables are only incorporated to assignment constraints and their cost coefficient is higher than that of other variables of the same slot request already in the model.

2. For each constraint where a variable x_{it} needs to be incorporated (Steps 5.2 and 5.3 of Algorithm 1), we look at the basic status of its slack variable. Modifying the left-hand side of a constraint may force the slack variable to take on a different value. If the slack was

nonbasic, it may be forced to become basic, thus invalidating the previous basis and forcing the solver to either discard it or perform many operations to recover an optimal basis. For this reason, we only add variables to an existing constraint if the slack variable of the constraint is basic. Otherwise, we leave the constraint as is, we add a copy of it with the incorporated variable and reoptimize with the dual simplex. We leave the old constraint in the model until it becomes basic after some dual simplex reoptimization. At that point it can be safely removed (Thompson et al. (1966)).

3. Nothing special needs to be done when adding new constraints to (LP_0) in Step 5.3. Most MIP solvers automatically reoptimize from the previous basis when new constraints are added.

5 Data

We generate synthetic data sets for single airport slot allocation problems (Fermín Cueto (2022)). We use a procedure similar to the one used by Androutsopoulos et al. (2020). Inspired by slot request data sets published by Agência Nacional de Aviação Civil (ANAC) (2021) and capacity declaration reports published by Airport Coordination Limited (ACL) (2021), we create distributions for a set of parameters that characterize airport demand and capacity, such as requested arrival / departure times, turnaround times, series length or aircraft sizes. We then sample from these distributions to generate realistic slot requests and capacity limits. The code to generate these synthetic data sets can be found in `https://github.com/paulafernalia/slot-allocation-data-gen`, and the data sets used in our computational experiments can be found in `https://datashare.ed.ac.uk/handle/10283/4374`.

For the sake of simplicity, this synthetic data does not include priority groups: generated instances can be seen as slot requests and capacity declarations for Level 2 airports, which can be harder to solve, as there are no historic requests that get automatically allocated to their requested time or other priority groups that allow to break down the problem into smaller sub-problems that are solved sequentially.

To assess the degree to which these synthetic data sets can approximate real-world slot allocation problems, in Section 6.2 we test *Caracal* on problem instances generated from seven UK airports of varied sizes coordinated by ACL, and compare the solution quality to that of our synthetic instances.

6 Computational Study

Our computational experiments were conducted on an Intel® Xeon® 2.2GHz, 256GB RAM, setting a maximum of 4 cores. Our algorithms were coded in C++. The LP relaxations in *Caracal* and the MIPs in other methods are solved using Gurobi 9.1 (Gurobi Optimizer Reference Manual (2021)) using its C++ API with default settings unless otherwise stated. A time limit of 10,000 seconds was set for each instance and algorithm.

The experiments presented here were designed to analyze the performance of *Caracal* and the preprocessing methodology presented in Section 3. To this end, each instance is solved using three different algorithms:

- **Baseline**: Gurobi's MIP solver with the preprocessing step proposed by Zografos et al. (2012) that aggregates identical calendar days, the initial constructive heuristic proposed

by Fairbrother and Zografos (2020) and the weak version of minimum turnaround constraints. Capacity constraints were not added as lazy cuts, as proposed by Fairbrother et al. (2019), because doing so led to a worse computational performance in our experiments.
- **Preprocessing**: same set up as the Baseline algorithm, adding the preprocessing pipeline described in Section 3 and deactivating Gurobi's preprocessing.
- ***Caracal***: strong turnaround constraints, the algorithm described in Section 4 with Gurobi's LP solver, Gurobi's preprocessing deactivated and our implementation of branch-and-cut.

This computational study has two distinct parts: first, we compare these algorithms using the mathematical model presented in Section 2 and a problem set consisting of 100 synthetic instances generated using the methodology referenced in Section 5. Then we present an application on real-world data: seven Level 2 and Level 3 UK airports coordinated by ACL.

6.1 Synthetic data

Figure 5 shows the percentage of instances that are solved within the times marked on the x axis, for each of the three algorithms. Tables 1 and 2 compare the overall performance and preprocessing performance, respectively, of the three algorithms. To create these tables, the test set was divided into six groups, according to the "hardness" of the instances. A group denoted "$(k, m]$" includes the instances for which the fastest algorithm solved the problem in more than k seconds but at most m. Geometric means are used instead of arithmetic means to limit the effect of individual large values. Following the benchmarking methodology employed by Achterberg (2009), a shift $s = 1$ is used for computational times and $s = 10$ for nodes in the calculation of geometric means to decrease the strong influence of trivial instances. A solution time of 10,000 seconds is assumed for instances that could not be solved within the time limit, which introduces a bias against the algorithms that hit the time limit less often. These results must therefore be read together with the number of timeouts.

In Table 1, "Instances" shows the number of instances from the test set in each sub-group displayed in the "Group" column, "Timeouts" denotes number of instances that could not be solved to optimality within the time limit, "Time" and "Nodes" show the geometric mean of the solution times in CPU seconds and the number of nodes in the branching tree, respectively, and "Faster" shows the number of instances for which an algorithm was faster than the Baseline. The sub-group corresponding to instances that were not solved within 10,000 seconds by any method is not displayed, but is included in the last row, which aggregates all groups. In Table 2, columns "Group" and "Instances" follow the same definition used in Table 1, "Init NZ" is the number of nonzeros (in millions) in the constraint matrix of the initial model, using weak turnaround constraints; "-V" and "-C" represent the reduction in number of variables and constraints, respectively, during preprocessing; "NZ" is the number of nonzeros (in millions) in the constraint matrix after preprocessing, and "Time" is the geometric mean of the CPU times in seconds spent in preprocessing.

Tables 4-9 in Appendix B show more detailed information than Table 1, with results broken down by instance.

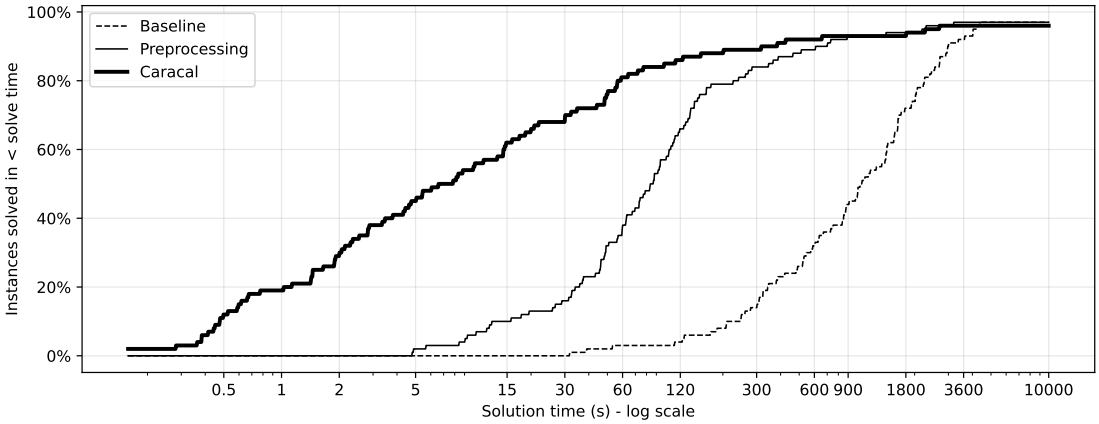

Figure 5: Performance profile comparing the Baseline, Preprocessing and *Caracal* algorithms on synthetic instances.

		Baseline			Preprocessing				*Caracal*			
Group	Instances	Timeouts	Time	Nodes	Timeouts	Time	Faster	Nodes	Timeouts	Time	Faster	Nodes
(0, 1]	19	0	403.1	1.0	0	27.0	19	1.0	0	0.5	19	1.0
(1, 10]	35	0	763.6	1.7	0	64.3	35	2.2	0	3.2	35	3.3
(10, 60]	27	0	1,248.0	2.7	0	145.9	27	5.2	0	25.6	27	7.3
(60, 300]	9	0	1,485.6	8.1	0	224.3	8	25.6	0	119.1	9	27.5
(300, 10k]	7	0	2,332.8	138.2	0	985.4	5	108.5	1	1,027.8	5	288.5
All	*100*	*3*	*890.8*	*8.2*	*3*	*93.6*	*94*	*10.2*	*4*	*10.1*	*95*	*11.2*

Table 1: Aggregated computational results of Baseline, Preprocessing and *Caracal* on synthetic instances.

			Baseline				Preprocessing				Caracal			
Group	Instances	Init NZ	-V	-C	NZ	Time	-V	-C	NZ	Time	-V	-C	NZ	Time
(0, 1]	19	109.4	32%	27%	47.9	183.6	35%	59%	26.0	17.1	99%	99%	0.006	0.170
(1, 10]	35	126.0	17%	13%	68.6	321.4	30%	56%	34.5	25.2	99%	98%	0.007	0.193
(10, 60]	27	174.2	17%	14%	100.6	525.8	27%	56%	59.0	37.8	99%	98%	0.014	0.264
(60, 300]	9	192.8	15%	10%	106.7	583.2	29%	55%	47.5	36.9	99%	98%	0.016	0.287
(300, 10k]	7	207.7	7%	5%	134.3	682.7	9%	48%	93.4	38.4	99%	98%	0.018	0.275
>10k	3	129.4	39%	8%	59.7	463.7	66%	46%	24.6	25.3	99%	98%	0.014	0.183
All	*100*	*147.7*	*20%*	*15%*	*81.0*	*371.2*	*30%*	*56%*	*44.5*	*27.9*	*99%*	*98%*	*0.011*	*0.221*

Table 2: Comparison of preprocessing in Baseline, Preprocessing and *Caracal* on synthetic instances.

It can be seen from Figure 5 that *Caracal* outperforms the other two methods, especially the Baseline, in most instances. It solved 79% of the test instances in less than one minute and 53% in less than 10 seconds. The Baseline could only solve 4% of the instances in less than two minutes. According to the results summarized in Table 1, *Caracal* was 83 times faster than the Baseline and 8.7 times faster than the Preprocessing algorithm on the entire test set.

The Preprocessing method is also very competitive. It is faster than the Baseline method in 94 of the 100 instances, being 9.5 times faster on the average. This improvement stems from the time saved in preprocessing. Our preprocessing rules exploit the underlying problem structure

to identify dominance relations more efficiently, resulting in 10% more variables and 41% more constraints eliminated 13.3 times faster, as shown in Table 2, although it should be noted that in some instances Gurobi adds a small number of variables and constraints to the model during preprocessing.

Figure 5 also shows that both the Preprocessing and the Baseline algorithms eventually catch up with *Caracal* at around 800 minutes and 2 hours, respectively. This behaviour is observed in instances with a less tight linear relaxation, which require more branching (see Tables 7 and 8). In those instances, *Caracal* spends more time solving each node of the tree than the other algorithms, despite the sub-problems being significantly smaller and despite *Caracal* being much faster at solving the root node. This is likely to be attributed to Gurobi's more efficient implementation of branch-and-cut, with a larger repertoire of cutting planes, primal heuristics or faster codes. Gurobi can reuse the LU factorization from the parent node when performing depth-first search, which speeds up the reoptimization of the child node. In *Caracal*, we perform warm start, that is, the optimal basis of the parent node is used as a starting point for the LP of the child node. However, these computational results do not suggest that Gurobi preserves the LU factorization in this case.

Tables 5-9 in Appendix B show that the primal heuristic described in Section 4.4 is very effective and can find good quality solutions from the LP solution of the root node, providing a "safety net" in hard instances. In problems that could not be solved in the root node, the heuristic found a solution that is at most 2.2% from the optimal solution on average. The largest gap was 26.2%, for an instance in the "> 10K" bracket, but this upper bound was the best found by any of the three algorithms.

From the results presented in Tables 4-9 in Appendix B it can be observed that the Baseline and Preprocessing algorithms become less effective with the size of the instance measured in number of nonzeros. *Caracal* has better scalability in terms of model size and seems to be affected mainly by the level of congestion at the airport and season under consideration. The harder instances for *Caracal* tend to have larger values of average displacements per flight. This was expected, as *Caracal* is effective at reducing the size of large models, but the linear relaxation is less tight (and the algorithm requires more inefficient branching) when more capacity constraints are active, leading to more displacements.

An important advantage of our two proposed methods, especially *Caracal*, is their reduced memory usage. To analyze this, we look at the number of nonzeros in the constraint matrix of the root node problem. Each nonzero consumes 12 bytes and, according to Gurobi Support Portal (2021), the model is represented in memory at least three times before attempting to solve the problem. Therefore the number of nonzeros times 36 represents a lower bound on the peak memory in bytes required to solve these problems. It can be inferred from the "Init NZ" column in Table 2 that the average Baseline model before preprocessing takes up 5.3 Gb. With the Preprocessing method, all the preprocessing is done before the model is created and therefore the memory required to represent the model comes from the "NZ" column: 1.6 Gb on average. With *Caracal*, the average initial root node model only take up 385 Kb. Even though the model will continue to grow as more variables and constraints are added, the average displacements per flight (see Tables 4-9) suggest that the memory usage will still be orders of magnitude lower than that of the other two algorithms.

6.2 Application to ACL airports

Lastly, we test the performance of the same three algorithms on seven UK airports of a wide range of sizes: Belfast International (BFS), Edinburgh (EDI), Glasgow (GLA), London City (LCY), Birmingham (BHX), London Gatwick (LGW) and London Heathrow (LHR). Several modifications

are made to the slot allocation problem described in Section 2 to account for some additional requirements from ACL:

- Coefficients f_{it} on the objective function are multiplied by 1.05 so that, if there are multiple solutions with the same or similar total displacements, then solutions with a lower maximum displacement are encouraged.
- For the Level 3 airports, nine priority groups are considered and their corresponding allocation problems are solved sequentially, updating the remaining capacity before the next group is allocated.
- Historic requests are all allocated to their requested time.
- For changes to historic requests we consider two categories: requests that only accept a slot at either the requested time or the historic time, and requests that are more flexible. For the latter, a displacement limit is introduced to ensure they are not given a displacement worse than the difference between the historic and the requested time.
- Even though ACL coordinators typically consider a limit on the largest displacement for all requests types, we assume that most requests in the remaining categories can accept any displacement (except two sub-groups in the new entrant and new incumbent categories, for which a displacement limit of ± 1 hour is introduced). This is done so that the results represent a worst case scenario in terms of computational performance.

Other constraints considered by ACL coordinators such as apron constraints, fairness considerations or season segmentation are out of scope for this computational study. Studying the application of *Caracal* to more complex slot allocation models is left for future work.

The results of these experiments are presented in Table 3. "Level" is the coordination level of the airport, "Init NZ" is the number of nonzeros (in millions) in the constraint matrix of the problem using weak turnaround constraints. "Rejected ops" is the number of flights that could not be accommodated at all, "Total disp" and "Largest disp" are the sum of displacements and the maximum displacement across all flights in the optimal solution, respectively. "Avg displacement" is the average displacement per displaced flight. "Time" is the solution time in CPU seconds. "NZ" is the number of nonzeros (in millions) in the constraint matrix after preprocessing. In Level 3 airports, "Init NZ" and "NZ" are based on the largest sub-problem across all priority groups.

These results are consistent with the results discussed in the previous section, especially for the Level 2 airports. *Caracal* is the fastest algorithm in six of the seven instances, requiring less than 5 seconds and 1 minute to solve the Level 2 and Level 3 airports, respectively. The Preprocessing algorithm is also competitive; it required 155.2 seconds to solve the hardest instance in the set. The Baseline had a good performance on the Level 3 airports, but it required about 20 and 30 minutes to solve GLA and EDI, respectively.

								Baseline		Preprocessing		Caracal	
Airport	No. flights	Level	Init NZ	Rejected ops.	Total disp	Largest disp	Avg disp	Time	NZ	Time	NZ	Time	NZ
BFS	27,021	2	42.8	0	4,835	25	6.1	150	3.6	14.3	8.6	**0.2**	0.006
GLA	51,952	2	161.5	0	8,205	40	10.9	1,048	50.8	34.5	40.7	**3.4**	0.012
EDI	89,501	2	241.1	0	14,445	40	6.5	1,788	88.8	155.2	69.6	**1.5**	0.008
LCY	54,398	3	168.2	728	635,610	545	44.1	247	0.3	85.5	2.2	**17.8**	0.003
BHX	79,791	3	12.3	0	319,580	605	64.6	22.4	1.5	12.9	1.6	**0.5**	0.003
LGW	232,485	3	49.0	10,409	9,435,810	1,130	181.6	92.8	2.6	58.0	5.8	**44.7**	0.007
LHR	315,446	3	74.7	1,600	12,811,025	1,160	185.9	**12.3**	0.5	21.1	0.7	68.2	0.005

Table 3: Comparison of three algorithms on instances from Level 2 and Level 3 UK airports.

It is interesting to see that these Level 3 airports are easier to solve with branch-and-cut than the Level 2 airports. The former are considerably more congested—the total displacements are up to three orders of magnitude larger and in the case of LCY, LGW and LHR many requests had to be rejected. However, being Level 3 means introducing request priorities, which means breaking down the problem into multiple smaller sub-problems and automatically allocating historic requests, which can represent a significant proportion of all the series (e.g. in LGW, 49% of the series belonged to this category). The pre-allocation of historic requests also simplifies the subsequent optimization problems, as multiple periods throughout the season become "full" and this allows to eliminate many decision variables and capacity constraints and with computationally cheaper preprocessing operations.

While the reduction in computational times achieved by *Caracal* is not as dramatic in Level 3 airports, this algorithm still has the advantage of being light in memory usage, requiring only 432 Kb to represent the initial root node model of the largest instance, which is significantly lower than the 8.7 Gb required by the root node model of the Baseline algorithm.

7 Conclusions

In this paper we presented *Caracal*, a column-and-row generation algorithm that can solve real-world instances of the slot allocation problem significantly faster than the best exact methods known in the literature with an efficient memory management. The effectiveness of this algorithm can be attributed to the fact that, in practice, most of the series are allocated to their requested time, making the great majority of variables in the model unnecessary, which in turn allows to eliminate a large proportion of capacity and turnaround constraints. The simple LP-based heuristic implemented in the root node obtains good quality integer solutions and serves as a safety net in the hardest instances. These results show that seemingly hard instances of the slot allocation problem are in fact easy MIPs in disguise: when the problem size is managed effectively, they are relatively easy to solve.

Moreover, *Caracal* scales well with the problem size and the results obtained in this paper suggest that it is well equipped to cope with harder variants of the slot allocation problem, such as network problems or single airport problems with an element of scheduling flexibility. The application of *Caracal* to such problems constitutes an interesting direction for future work.

Our problem-specific preprocessing pipeline achieved a substantial reduction of solution times by identifying a large number of unnecessary variables and constraints in the model in a short period of time. It has the additional advantage of being an algorithm-agnostic procedure. Since these preprocessing rules constitute a set of simple, easy to implement steps to identify redundant

capacity and turnaround constraints and infeasible or uninteresting time slots for certain series, they can be applied to reduce the size of integer programs used in exact methods, but also to speed up heuristic approaches, as they have the potential to simplify feasibility checks substantially.

Lastly, we provide a test set of synthetic instances with the aim of enabling benchmarking of slot allocation algorithms. These test instances cover different levels of complexity and yield computational results consistent with real-world Level 2 airports.

Acknowledgements

The authors thank Airport Coordination Limited for providing valuable insights and real-world slot coordination data.

References

Achterberg, T. 2009. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, **1**(1):1–41.

Achterberg, T., Koch, T., and Martin, A. 2005. Branching rules revisited. *Operations Research Letters*, **33** (1):42–54.

Agência Nacional de Aviação Civil (ANAC). Slot Coordination, 2021. `www.anac.gov.br/en/air-services/slot-coordination`.

Airport Coordination Limited (ACL). Latest Airport Info, 2021. `www.acl-uk.org/latest-airport-info/`.

Androutsopoulos, K.N. and Madas, M.A. 2019. Being fair or efficient? A fairness-driven modeling extension to the strategic airport slot scheduling problem. *Transportation Research Part E: Logistics and Transportation Review*, **130**:37 – 60.

Androutsopoulos, K.N., Manousakis, E.G., and Madas, M.A. 2020. Modeling and solving a bi-objective airport slot scheduling problem. *European Journal of Operational Research*, **284**(1):135–151.

Artigues, C. 2017. On the strength of time-indexed formulations for the resource-constrained project scheduling problem. *Operations Research Letters*, **45**(2):154–159.

Benlic, U. 2018. Heuristic search for allocation of slots at network level. *Transportation Research Part C: Emerging Technologies*, **86**:488 – 509.

Berthold, T. 2014. RENS. *Mathematical Programming Computation*, **6**(1):33–54.

Castelli, L., Pellegrini, P., and Pesenti, R. Ant colony optimization for allocating airport slots. In *2nd International Conference on Models and Technologies for ITS*. Katolieke Universiteit Leuven, 2011.

Christofides, N., Alvarez-Valdes, R., and Tamarit, J. 1987. Project scheduling with resource constraints: A branch and bound approach. *European Journal of Operational Research*, **29**(3):262 – 273.

Corolli, L., Lulli, G., and Ntaimo, L. 2014. The time slot allocation problem under uncertain capacity. *Transportation Research Part C: Emerging Technologies*, **46**:16 – 29.

European Commission. Council regulation (EEC) no. 95/93 of 18 January 1993 on common rules for the allocation of slots at community airports, 1993.

Fairbrother, J. and Zografos, K.G. 2020. Optimal scheduling of slots with season segmentation. *European Journal of Operational Research*, **291**(3):961 – 982.

Fairbrother, J., Zografos, K.G., and Glazebrook, K.D. 2019. A slot-scheduling mechanism at congested airports that incorporates efficiency, fairness, and airline preferences. *Transportation Science*, **54**(1):115–138.

Fermín Cueto, P. Slot allocation synthetic data. University of Edinburgh. School of Mathematics, 2022.

Gamrath, G., Koch, T., Martin, A., Miltenberger, M., and Weninger, D. 2015. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, **7**(4):367–398.

García, S., Labbé, M., and Marín, A. 2011. Solving large p-median problems with a radius formulation. *INFORMS Journal on Computing*, **23**(4):546–556.

Gu, Z., Nemhauser, G.L., and Savelsbergh, M.W.P. 1998. Lifted cover inequalities for 0-1 integer programs: Computation. *INFORMS Journal on Computing*, **10**(4):427–437.

Gurobi Optimizer Reference Manual. Gurobi Optimization, LLC, 2021. `www.gurobi.com`.

Gurobi Support Portal. Gurobi Optimization, LLC. `https://tinyurl.com/gurobisupportportal`, 2021.

Hillier, F.S. and Lieberman, G.J. In *Introduction to Operations Research*, chapter 6. McGraw-Hill, USA, 2001.

IATA. Worldwide Airport Slot Guidelines - Edition 1. `www.iata.org/en/policy/slots/slot-guidelines/`, 2020.

Jacquillat, A. and Odoni, A.R. 2015. An integrated scheduling and operations approach to airport congestion mitigation. *Operations Research*, **63**(6):1390–1410.

Kaparis, K. and Letchford, A.N. 2008. Local and global lifted cover inequalities for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, **186**(1):91–103.

Pellegrini, P., Castelli, L., and Pesenti, R. 2011. Metaheuristic algorithms for the simultaneous slot allocation problem. *Intelligent Transport Systems, IET*, **6**(4):453–462.

Pritsker, A.A.B., Waiters, L.J., and Wolfe, P.M. 1969. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, **16**(1):93–108.

Ribeiro, N.A., Jacquillat, A., Antunes, A.P., Odoni, A.R., and Pita, J.P. 2018. An optimization approach for airport slot allocation under IATA guidelines. *Transportation Research Part B: Methodological*, **112**: 132–156.

Ribeiro, N.A., Jacquillat, A., and Antunes, A.P. 2019a. A large-scale neighborhood search approach to airport slot allocation. *Transportation Science*, **53**(6):1772–1797.

Ribeiro, N.A., Jacquillat, A., Antunes, A.P., and Odoni, A. 2019b. Improving slot allocation at level 3 airports. *Transportation Research Part A: Policy and Practice*, **127**:32–54.

Sousa, J.P. and Wolsey, L.A. 1992. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, **54**(1-3):353–367.

Thompson, G.L., Tonge, F.M., and Zionts, S. 1966. Techniques for removing nonbinding constraints and extraneous variables from linear programming problems. *Management Science*, **12**(7):588–608.

van den Akker, J., Hurkens, C., and Savelsbergh, M. 2000. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*, **12**(2):111–124.

Wolsey, L.A. 1990. Valid inequalities for 0–1 knapsacks and mips with generalised upper bound constraints. *Discrete Applied Mathematics*, **29**(2-3):251–261.

Zografos, K.G., Salouras, Y., and Madas, M.A. 2012. Dealing with the efficient allocation of scarce resources at congested airports. *Transportation Research Part C: Emerging Technologies*, **21**(1):244–256.

Zografos, K.G., Madas, M.A., and Androutsopoulos, K.N. 2016. Increasing airport capacity utilisation through optimum slot scheduling: review of current developments and identification of future needs. *Journal of Scheduling*, **20**(1):3–24.

Appendix A Omitted proofs

A.1 Proof of Proposition 1

Proposition 1. *Given a slot allocation problem with one set of capacity constraints c with frequency λ_c and duration w_c multiple of λ_c, and no turnaround constraints, then variables x_{it} for indices $t \in T$* not *listed below can be fixed to 0:*

- $t = \tau_i$,
- $t = k\lambda_c,\ \ k \in \mathbb{Z}^+,\ \ t > \tau_i$,
- $t = k\lambda_c - 1,\ \ k \in \mathbb{Z}^+,\ \ t < \tau_i$.

Proof. If w_c is a multiple of λ_c, then the capacity windows start times $s \in T_c$ of constraints (3) are also multiples of λ_c, and the periods $s + w_c - 1 \in T$ are multiples of $\lambda_c - 1$. In other words, no capacity constraints start or finish inside an interval $[s, s + \lambda_c],\ s \in T_c$. Then, given a series $i \in N$ and a time period $s = k\lambda_c \in T_c$, for some $k \in \mathbb{Z}^+$ such that $k\lambda_c > \tau_i$, any variable x_{it} with $k\lambda_c < t < (k+1)\,\lambda_c$ is dominated by x_{is}, since all these variables appear in the same capacity constraints with the same coefficients and variables x_{is} have a lower displacement. Using the same reasoning we can prove the symmetric case: when $t < \tau_i$, $x_{i,k\lambda_c-1} \succ x_{it}$ for t such that $(k-1)\lambda_c \le t < k\,\lambda_c - 1$. Lastly, given a starting period $k\,\lambda_c \in T$ such that $k\,\lambda_c < \tau_i < (k+1)\,\lambda_c$, a similar reasoning shows that variables x_{it} with $k\lambda_c < t < (k+1)\,\lambda_c$, $t \neq \tau_i$ are dominated by $x_{i\tau_i}$. □

A.2 Proof of Proposition 2

Proposition 2. *Let $(i, j) \in E$ represent a pair of linked arriving and departing series. Variables x_{it} for all $t > \tau_i$ and variables x_{jt} for all $t < \tau_j$ are dominated with respect to turnaround constraints.*

Proof. We consider the weak version of turnaround constraints: $\sum_{t\in T} t\,x_{jt} - \sum_{t\in T} t\,x_{it} \ge t_{ij}$. Since these constraints and constraints (4) result in the same set of integer solutions, any variables that are dominated when we consider the weak version are also dominated using the strong version. Rearranging terms we obtain $-\sum_{t\in T} t\,x_{jt} + \sum_{t\in T} t\,x_{it} \le -t_{ij}$. Since $\tau_i < t$ and $f_{i\tau_i} = 0 < f_{it}$, applying the principle outlined at the start of Section 3, it is clear that $x_{i\tau_i} \succ x_{it}$ and x_{it} can be fixed to 0. The same logic can be applied to prove the symmetric case: when j is a departure and $t < \tau_j$. □

A.3 Proof of Proposition 3

Proposition 3. *Let $(i, j) \in E$ represent a pair of linked arriving and departing series. Assume that some variables have been identified as dominated when only capacity constraints were considered. Given a time period $t' > \tau_j$, if there exists a variable $x_{it''}$ that was not fixed in previous preprocessing steps and for which the following conditions are met:*

1. *$\tau_j \le t'' + t_{ij} < t'$ and*
2. *variables x_{it} with $t'' < t \le t' - t_{ij}$ have been fixed to 0,*

then, variable $x_{jt'}$ is dominated by variable $x_{j,t''+t_{ij}}$ and it can be fixed to 0 with respect to minimum turnaround constraints.

The same is true for the arriving series i: a previously fixed variable $x_{it'}$, with $t' < \tau_i$, is still fixed to 0 if there exists a variable $x_{jt''}$ that has not been fixed such that t'' satisfies (i) $t' < t'' - t_{ij} \le \tau_i$ and (ii) variables x_{it} with $t' + t_{ij} \le t < t''$ are fixed to 0.

Proof. We prove the first part of Proposition 3, concerning variables x_{jt}. The same reasoning can be used to prove the symmetric case (variables x_{it}).

Consider an integer solution x^* to (1)-(5), excluding capacity constraints, with $x^*_{jt'} = 1$ for some $t' \in T \ / \ t' > \tau_j$. Since this solution is feasible, then there exists a slot $t'' \in T$ such that $x^*_{it''} = 1$ and $t' - t'' \geq t_{ij}$. If variables x_{it} with $t'' < t \leq t' - t_{ij}$ are fixed to 0 (Proposition 3) then this is the shortest feasible turnaround that can be achieved with $x^*_{jt'} = 1$. However, under these conditions, there exists an alternative solution with $x^*_{it''} = 1$ and $x^*_{j,t''+t_{ij}} = 1$ which is also feasible (the turnaround time is exactly t_{ij}) but results in a lower displacement (the displacement of i is the same and the displacement of j is now $t'' + t_{ij} - \tau_j$, which is lower than $t' - \tau_j$ by the first condition in Proposition 3). Therefore, variable $x_{jt'}$ is dominated by variable $x_{j,t''+t_{ij}}$ and can be fixed to 0. □

A.4 Proof of Proposition 4

Proposition 4. *Let $i \in N$ be an arrival series, $j \in N$ its linked departure series and t_{ij} the minimum turnaround time between them. If variable x_{js} for some $s \in T$ is fixed to 0, then the turnaround constraint with index s is dominated by the constraint with index $s-1$.*

Proof. Consider two consecutive turnaround constraints with indices $s-1,\ s \in T$ and the pair of linked requests $(i,j) \in E$:

$$s-1: \qquad \sum_{t=s-t_{ij}}^{|T|} x_{it} + \sum_{t=1}^{s-1} x_{jt} \leq 1, \tag{20}$$

$$s: \qquad \sum_{t=s-t_{ij}+1}^{|T|} x_{it} + \sum_{t=1}^{s} x_{jt} \leq 1. \tag{21}$$

These two constraints only differ in two terms: the constraint with index s has the additional term x_{js} in the left-hand side (LHS) and is missing the term $x_{i,s-t_{ij}}$ present in constraint with index $s-1$. It is clear that if x_{js} is fixed to 0, then the LHS of constraint (20) can only be greater or equal to the LHS of constraint (21). Therefore the latter constraint can be removed without altering the set of solutions. □

A.5 Proof of Proposition 5

Proposition 5. *Let e_i and ℓ_i denote the earliest and latest available slots for series $i \in N$, respectively. Let $(i,j) \in E$ be a pair of linked series with minimum ground time t_{ij}. Any turnaround constraints with indices $s \in T$ not included in* (22) *are dominated and can be excluded from the slot allocation model* (1)-(5).

$$\sum_{t=s-t_{ij}+1}^{\ell_i} x_{it} + \sum_{t=e_j}^{s} x_{jt} \leq 1, \quad \forall\,(i,j) \in E,\ s \in T \ / \ \max\{e_j,\ e_i + t_{ij} - 1\} \leq\ s\ \leq \min\{\ell_j,\ \ell_i + t_{ij} - 1\}. \tag{22}$$

Proof. Let us first analyze the smallest value of s included, $s_{\min} = \max\{e_j,\ e_i + t_{ij} - 1\}$. There are two possibilities:

1. $e_j \geq e_i + t_{ij} - 1$. In this case, $s_{\min} = e_j$ and the second sum in the constraint (12) with index $s = s_{\min}$ would only include one term: x_{je_j}. For any other $s < e_j$, the sum would be empty, the resulting turnaround constraint would not include any variables for series j and this constraint would be redundant.

2. $e_i + t_{ij} - 1 > e_j$. In this case, $s_{\min} = e_i + t_{ij} - 1$ and the turnaround constraint (12) with $s = s_{\min}$ would be $\sum_{t=e_i}^{\ell_i} x_{it} + \sum_{t=e_j}^{e_i+t_{ij}-1} x_{jt} \leq 1$. Any other $s = s_{\min} - k$ with $k \in \mathbb{Z}^+$ would yield a constraint $\sum_{t=e_i-k}^{\ell_i} x_{it} + \sum_{t=e_j}^{e_i+t_{ij}-1-k} x_{jt} \leq 1$. Considering that e_i is the first slot available for flights i, this inequality is equivalent to $\sum_{t=e_i}^{\ell_i} x_{it} + \sum_{t=e_j}^{e_i+t_{ij}-1-k} x_{jt} \leq 1$, and from this expression it can be seen that this constraint can be safely eliminated as it is dominated by the turnaround constraint with $s = s_{\min}$.

The same logic can be applied to show that any constraints with $s > s_{\max}$ are redundant. □

Appendix B Tables

The following tables present detailed computational results of all synthetic instances, divided in six groups as per the classification described in Section 6.1. The tables are organized as follows:

- **Instance** is the unique identifier of the instance.
- **NZ** is the number of nonzeros (in millions) in the constraint matrix of the original formulation, after applying the preprocessing proposed by Zografos et al. (2012).
- **UB** is the best upper bound on the optimal solution found by the algorithm.
- **LB** is the best lower bound on the optimal solution found by the algorithm.
- **Time** is the solution time in CPU seconds. "*" indicates that the limit of 10,000 seconds was reached before the optimal solution was found.
- **Nodes** is the size of the branching tree in number of nodes when the optimal solution was found or the time limit of 10,000 seconds was reached.
- **Heur** is the upper bound obtained via the primal heuristic described in Section 4.4 when using *Caracal*. "-" indicates that a heuristic solution with *Caracal* was not necessary because the problem was solved in the root node.
- **Avg disp** is the displacement per displaced flight in minutes in the best integer solution obtained with *Caracal*.

For each instance, the fastest solution time is highlighted in **bold**.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
I067	3	8,145	8,145	40.2	1	8,145	8,145	5.3	1	-	8,145	8,145	**0.5**	1	8.9
I097	4	16,860	16,860	32.9	1	16,860	16,860	5.9	1	-	16,860	16,860	**0.5**	1	16.5
I029	5	1,575	1,575	54.7	1	1,575	1,575	5.3	1	-	1,575	1,575	**0.1**	1	5.5
I046	15	2,330	2,330	176.3	1	2,330	2,330	10.4	1	-	2,330	2,330	**0.1**	1	5.3
I045	16	5,775	5,775	191.1	1	5,775	5,775	10.8	1	-	5,775	5,775	**0.3**	1	8.1
I054	19	8,595	8,595	116.7	1	8,595	8,595	13.2	1	-	8,595	8,595	**0.4**	1	12.5
I090	22	6,415	6,415	285.6	1	6,415	6,415	11.3	1	-	6,415	6,415	**0.5**	1	8.4
I061	27	5,020	5,020	349.4	1	5,020	5,020	14.5	1	-	5,020	5,020	**0.4**	1	7.7
I010	31	9,035	9,035	316.4	1	9,035	9,035	21.0	1	-	9,035	9,035	**0.6**	1	7.5
I066	35	5,670	5,670	270.4	1	5,670	5,670	22.9	1	-	5,670	5,670	**0.5**	1	6.2
I035	76	7,085	7,085	501.8	1	7,085	7,085	34.5	1	-	7,085	7,085	**0.5**	1	5.7
I008	123	6,430	6,430	879.7	1	6,430	6,430	53.4	1	-	6,430	6,430	**0.4**	1	5.6
I069	137	4,130	4,130	911.5	1	4,130	4,130	51.5	1	-	4,130	4,130	**0.7**	1	6.5
I086	150	9,440	9,440	886.9	1	9,440	9,440	57.3	1	-	9,440	9,440	**0.7**	1	6.2
I003	158	1,265	1,265	1,051.5	1	1,265	1,265	56.5	1	-	1,265	1,265	**0.4**	1	5.0
I079	248	3,795	3,795	2,772.8	1	3,795	3,795	130.0	1	-	3,795	3,795	**0.6**	1	6.0
I059	264	3,260	3,260	1,695.4	1	3,260	3,260	80.0	1	-	3,260	3,260	**0.5**	1	6.7
I049	331	16,625	16,625	2,473.6	1	16,625	16,625	119.7	1	-	16,625	16,625	**0.8**	1	6.3
I065	403	3,000	3,000	2,805.2	1	3,000	3,000	135.6	1	-	3,000	3,000	**0.6**	1	5.7

Table 4: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the (0, 1] bracket.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
I050	12	19,205	19,205	129.1	1	19,205	19,205	17.3	1	19,205	19,205	19,205	**1.9**	3	10.8
I093	14	12,145	12,145	127.2	1	12,145	12,145	11.9	1	-	12,145	12,145	**2.9**	1	8.7
I032	21	18,215	18,215	215.3	1	18,215	18,215	31.5	1	-	18,215	18,215	**2.5**	1	13.6
I056	23	12,130	12,130	308.0	1	12,130	12,130	40.3	6	12,130	12,130	12,130	**1.4**	9	14.4
I051	25	19,435	19,435	255.7	1	19,435	19,435	96.0	21	19,435	19,435	19,435	**8.8**	11	10.2
I034	29	17,385	17,385	257.1	1	17,385	17,385	14.8	1	-	17,385	17,385	**3.4**	1	8.3
I092	47	21,730	21,730	220.1	1	21,730	21,730	45.5	1	21,730	21,730	21,730	**6.0**	21	11.8
I040	48	19,190	19,190	321.4	1	19,190	19,190	48.4	1	19,190	19,190	19,190	**5.1**	5	9.8
I076	49	5,470	5,470	400.2	1	5,470	5,470	31.7	1	-	5,470	5,470	**1.1**	1	6.4
I009	57	21,260	21,260	434.1	1	21,260	21,260	31.2	1	-	21,260	21,260	**2.3**	1	9.1
I098	58	28,580	28,580	337.8	75	28,580	28,580	119.6	96	28,660	28,580	28,580	**8.3**	63	10.6
I015	60	17,240	17,240	651.0	1	17,240	17,240	41.4	1	-	17,240	17,240	**1.0**	1	7.3
I070	67	30,760	30,760	606.7	1	30,760	30,760	67.0	1	-	30,760	30,760	**6.6**	1	9.7
I053	73	30,295	30,295	646.1	1	30,295	30,295	37.6	1	-	30,295	30,295	**3.5**	1	7.3
I083	95	16,740	16,740	756.2	1	16,740	16,740	69.5	1	16,740	16,740	16,740	**2.8**	3	6.7
I094	100	8,300	8,300	694.4	1	8,300	8,300	61.9	1	8,300	8,300	8,300	**1.5**	3	6.5
I073	105	11,340	11,340	1,065.3	1	11,340	11,340	158.5	1	11,340	11,340	11,340	**4.4**	9	8.9
I037	107	17,825	17,825	634.2	1	17,825	17,825	40.3	1	-	17,825	17,825	**2.0**	1	6.5
I063	119	11,055	11,055	1,033.4	1	11,055	11,055	53.5	1	-	11,055	11,055	**1.5**	1	8.4
I018	121	18,415	18,415	1,040.3	1	18,415	18,415	54.9	1	18,485	18,415	18,415	**2.4**	5	8.7
I030	131	16,630	16,630	1,806.7	1	16,630	16,630	124.7	1	16,630	16,630	16,630	**1.9**	3	7.7
I000	132	22,365	22,365	871.5	1	22,365	22,365	54.4	1	-	22,365	22,365	**1.9**	1	6.6
I060	134	18,250	18,250	1,692.6	1	18,250	18,250	96.3	1	-	18,250	18,250	**4.6**	1	6.9
I033	135	23,980	23,980	1,436.5	1	23,980	23,980	54.6	1	-	23,980	23,980	**2.1**	1	6.4
I084	151	44,190	44,190	915.9	1	44,190	44,190	65.3	1	-	44,190	44,190	**5.5**	1	8.9
I023	165	17,825	17,825	2,057.0	1	17,825	17,825	134.8	1	-	17,825	17,825	**1.7**	1	5.8
I038	178	12,430	12,430	1,141.8	1	12,430	12,430	69.3	1	-	12,430	12,430	**1.5**	1	6.6
I096	178	43,270	43,270	1,677.6	1	43,270	43,270	145.1	1	-	43,270	43,270	**8.2**	1	9.2
I014	182	16,160	16,160	1,678.0	1	16,160	16,160	121.3	1	-	16,160	16,160	**4.8**	1	9.1
I089	192	21,235	21,235	1,490.9	1	21,235	21,235	88.3	1	-	21,235	21,235	**5.4**	1	9.3
I006	237	10,150	10,150	1,470.6	1	10,150	10,150	96.9	1	-	10,150	10,150	**2.1**	1	7.6
I099	276	15,945	15,945	1,499.6	1	15,945	15,945	157.9	1	15,945	15,945	15,945	**4.4**	5	6.2
I004	321	26,490	26,490	2,293.6	1	26,490	26,490	124.0	1	-	26,490	26,490	**3.8**	1	6.4
I057	368	22,690	22,690	4,107.8	1	22,690	22,690	195.9	1	-	22,690	22,690	**7.9**	1	6.7
I042	379	18,965	18,965	3,736.1	1	18,965	18,965	158.4	1	19,065	18,965	18,965	**2.9**	5	5.3

Table 5: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the (1, 10] bracket.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
I021	31	63,835	63,835	581.9	479	63,835	63,835	75.5	250	64,435	63,835	63,835	**14.6**	29	22.0
I025	42	36,785	36,785	334.4	1	36,785	36,785	120.7	461	36,965	36,785	36,785	**56.2**	251	9.3
I072	53	58,975	58,975	533.1	1	58,975	58,975	138.9	17	59,190	58,975	58,975	**13.3**	11	11.1
I019	54	20,665	20,665	361.0	1	20,665	20,665	69.5	18	21,125	20,665	20,665	**48.2**	749	7.6
I078	61	40,310	40,310	539.9	1	40,310	40,310	37.8	1	-	40,310	40,310	**20.8**	1	9.3
I005	62	51,105	51,105	545.7	1	51,105	51,105	39.9	1	-	51,105	51,105	**11.3**	1	12.9
I016	110	43,000	43,000	1,287.8	1	43,000	43,000	125.1	1	-	43,000	43,000	**30.1**	1	12.1
I024	136	62,145	62,145	1,188.1	1	62,145	62,145	94.7	1	-	62,145	62,145	**17.4**	1	8.4
I017	143	108,395	108,395	1,454.7	1	108,395	108,395	105.8	1	-	108,395	108,395	**50.4**	1	9.6
I031	144	77,110	77,110	1,471.3	1	77,110	77,110	125.2	1	-	77,110	77,110	**34.7**	1	8.9
I001	152	69,850	69,850	947.4	1	69,850	69,850	94.1	1	69,850	69,850	69,845	**49.5**	3	10.0
I055	153	31,150	31,150	1,216.2	1	31,150	31,150	70.7	1	-	31,150	31,150	**10.0**	1	7.6
I064	157	65,085	65,085	787.2	1	65,085	65,085	67.8	1	-	65,085	65,085	**18.6**	1	9.8
I044	166	30,990	30,990	1,616.2	1	30,990	30,990	152.3	1	-	30,990	30,990	**14.8**	1	6.5
I087	183	43,145	43,145	1,105.6	1	43,145	43,145	84.1	1	-	43,145	43,145	**21.9**	1	8.4
I095	186	42,480	42,480	1,969.1	1	42,480	42,480	200.2	1	-	42,480	42,480	**56.4**	1	9.3
I028	192	41,555	41,555	1,029.5	1	41,555	41,555	77.2	1	41,555	41,555	41,555	**15.9**	3	6.3
I022	194	44,275	44,275	917.3	1	44,275	44,275	80.7	1	-	44,275	44,275	**14.3**	1	7.4
I013	214	30,800	30,800	2,102.4	1	30,800	30,800	111.4	1	-	30,800	30,800	**10.2**	1	7.5
I075	216	80,270	80,270	2,509.3	1	80,270	80,270	553.1	1	80,270	80,270	80,270	**48.4**	3	10.7
I011	232	30,735	30,735	2,748.5	1	30,735	30,735	403.5	1	-	30,735	30,735	**30.1**	1	8.2
I047	245	87,970	87,970	1,695.8	1	87,970	87,965	255.8	1	88,055	87,970	87,965	**20.0**	19	7.6
I041	260	94,760	94,760	3,022.5	1	94,760	94,760	1,462.6	1	94,795	94,760	94,760	**59.0**	9	9.8
I043	263	46,255	46,255	2,616.8	1	46,255	46,255	370.6	1	-	46,255	46,255	**55.0**	1	8.9
I077	282	14,880	14,880	2,982.1	1	14,880	14,880	311.8	1	-	14,880	14,880	**14.4**	1	7.9
I002	312	70,310	70,310	2,022.7	1	70,310	70,310	171.1	1	70,310	70,310	70,310	**44.0**	3	9.3
I088	445	65,155	65,155	3,086.8	1	65,155	65,155	918.7	1	65,160	65,155	65,155	**32.2**	13	7.9

Table 6: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the (10, 60] bracket.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
I091	29	45,395	45,395	498.4	942	45,395	45,395	734.9	3,192	45,675	45,395	45,395	**123.7**	465	20.0
I082	44	56,385	56,385	554.9	1	56,385	56,385	**133.8**	1,064	56,385	56,385	56,385	153.4	1,213	15.9
I007	143	196,200	196,200	1,599.8	1	196,200	196,200	**135.4**	1	-	196,200	196,200	201.5	1	11.9
I074	144	136,985	136,985	1,388.5	1	136,985	136,980	162.4	1	-	136,985	136,985	**72.1**	1	12.5
I071	194	29,935	29,935	1,749.9	1	29,935	29,935	105.7	1	-	29,935	29,935	**76.8**	1	6.5
I068	215	97,050	97,050	1,607.3	1	97,050	97,050	**233.5**	5	105,990	97,050	97,050	315.8	53	12.0
I026	218	115,060	115,055	2,122.1	1	115,060	115,060	292.8	1	115,060	115,060	115,055	**98.7**	15	8.9
I085	330	96,990	96,990	2,323.5	1	96,990	96,990	187.2	1	-	96,990	96,990	**64.2**	1	9.6
I058	414	133,855	133,855	4,132.0	1	133,855	133,855	487.8	1	-	133,855	133,855	**113.3**	1	12.4

Table 7: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the (60, 300] bracket.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
I020	37	25,735	25,735	**389.5**	1	25,735	25,735	391	1,619	26,035	25,735	25,735	2,689	13,007	10.6
I062	72	48,105	48,105	3,393	101,768	48,105	48,105	**2,191**	7,029	49,615	48,400	47,165	*	4,575	14.7
I027	198	212,485	212,485	2,065	1	212,485	212,485	**304**	1	-	212,485	212,485	423	1	13.3
I048	207	517,755	517,715	2,304	1	517,755	517,730	**723**	12	557,295	517,755	517,710	2,267	85	23.2
I080	251	100,365	100,365	3,077	63	100,365	100,365	3,228	45	100,645	100,365	100,360	**656**	91	10.5
I052	323	162,160	162,160	4,462	762	162,160	162,160	2,340	186	162,160	162,160	162,150	**383**	309	9.6
I012	363	289,025	289,025	4,344	196	289,025	289,025	**630**	1	314,155	289,025	289,015	1,805	95	17.5

Table 8: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the (300, 10k] bracket.

		Baseline				Preprocessing				Caracal					
Instance	NZ	UB	LB	Time	Nodes	UB	LB	Time	Nodes	Heur	UB	LB	Time	Nodes	Avg disp
l081	118	167,820	161,440	*	31,319	167,015	163,440	*	20,863	167,495	167,495	160,695	*	1,545	15.4
l039	131	129,595	120,840	*	31,088	129,510	122,455	*	20,703	129,580	129,580	121,060	*	1,589	17.1
l036	137	217,145	163,120	*	3,190	217,315	166,610	*	5,349	209,420	209,420	168,240	*	197	22.6

Table 9: Comparison of Baseline, Preprocessing and *Caracal* on synthetic instances in the $>$ 10k bracket.