Noname manuscript No.
(will be inserted by the editor)

Optimal balanced chain decomposition of partially ordered sets with applications to operating cost minimization in aircraft routing problems

Radislav Vaisman · Ilya B. Gertsbakh

Received: date / Accepted: date

Abstract We consider the task of constructing a cost-effective daily flight schedule with minimum number of required aircrafts and maximum number of balanced flight routes, namely, routes with the same start and end spatial location. We suggest a solution strategy which is able to determine the problem's hardness by estimating the number of all flight plans with minimum number of required aircrafts. Provided that this number is not too large, the same algorithm is utilized for fully enumerating and detecting the set of solutions that have the maximum number of balanced routes. Our experimental study implies that the method is both effective and scalable in practice. For example, when applied to the Australian domestic flights timetable which is serviced by a total of eighty-eight aircrafts, our method manages to increase the number of balanced flight routes from nine to forty-two, while using only several minutes of computational time.

Keywords Optimal flight scheduling · deadheading flights · approximate counting · optimization · heuristic

1 Introduction

The necessity of returning an aircraft to its starting location by adding deadheading flights imposes a serious challenge to airline companies, since these trips are usually performed without receiving payments by corresponding flight operators. Despite the fact that the task of constructing schedules that both utilize the minimum number of required aircrafts and have the minimum number of deadheading trips is computationally hard, the problem's importance attracted a significant research effort [19,6,24,16].

School of Mathematics and Physics The University of Queensland, St Lucia QLD 4072, Australia E-mail: r.vaisman@uq.edu.au · Department of Mathematics, Ben-Gurion University, Beer-Sheva, 84105, Israel E-mail: elyager@bezeqint.net

It is of great interest to consider schedules that contain routes that *start* and *end* at the same spatial location; (such routes are called *balanced* routes; please see Stern and Gertsbakh [24]), since these schedules minimize the number of deadheading flights. In this study, we propose a novel two-step approach for addressing the problem of finding an *optimal balanced schedule* (OBS). Specifically, we introduce a randomized algorithm for approximate counting of *all* schedules that require the minimum number of aircrafts. The proposed procedure opens a way for obtaining optimal solutions via full enumeration, provided that the number of such schedules is not very large. That is, the method allows to extract schedules that have the maximum number of balanced routes. However, it is important to note that while obtaining the counting estimator is not a very hard task, one cannot expect that the full enumeration procedure will always be computationally feasible. Namely, when the total number of schedules is prohibitively large, we are forced to resort to alternative approaches. In order to deal with this scenario, we propose a simple and fast heuristic, which is able to obtain high-quality solutions to the OBS problem. One of the major advantages of the proposed framework is that it is capable of determining the problem's *hardness*. Namely, provided that the counting estimator in the first step shows a relatively small number, we know that the problem in hand is *easy*, in the sense that a set of *globally optimal* solutions can be obtained via full enumeration while using a reasonable computational effort.

Due to the problem importance, the (aircraft) routing problem has been addressed by many researchers in the past. While the majority of this research effort focused on applying *mathematical programming* and other heuristic tools [2, 18, 25, 15, 7, 1], in a recent work of Stern and Gertsbakh [24], the authors proposed to handle the problem of balanced scheduling by utilizing the *deficit function* approach; for details, please see [17]. Stern et al. suggest to decompose an aviation schedule of aircraft flights into aircraft chains (or routes). Then, the authors use the corresponding deficit function and show how one can construct a near optimal balanced schedule in a sequential manner, namely, a schedule that has many balanced routes. In this paper, we apply a more general approach with a view to develop generic algorithms for counting and optimization of balanced schedules.

In particular, as we show in Section 2, one can take advantage of the fact that the OBS task can be viewed as a generalization of the *minimum chain decomposition* problem of partially ordered sets [22]. The minimum chain decomposition problem was first proposed by [9], and the correspondence between the *Dilworth's problem* and the *maximum matching* problem in bipartite graphs was established by [11]. This correspondence allows to apply computationally efficient techniques in order to obtain the optimal (minimum) number of aircrafts needed to service a given set of routes, [13]. Moreover, by counting all maximum matchings in the corresponding bipartite graph, we can retrieve all possible schedules that use this minimum number of aircrafts. The detailed explanation of these ideas is specified in Section 2. Having in mind the relationship between maximum matchings in bipartite graphs and optimal solutions of the Dilworth's problem, we recognize that when the full enumeration of all

maximum matchings is computationally feasible, one can obtain an optimal solution to the OBS task. In fact, in this case, one can even retrieve the set of all optimal solutions. Alternatively, if the number of maximum matchings is too large for the full enumeration procedure, we propose a simple heuristic approach based on a local search in the corresponding maximum matching space, which is capable of optimizing the schedule with a view to increase the number of balanced routes. The heuristic method is detailed in Section 3.3.

It is important to note that the problem of counting of maximum matchings in bipartite graphs belongs to the #P complexity class [28, 20, 10, 8], that is, the problem is *computationally hard*. The #P complexity class consists of counting problems that are associated with the problem of counting the number of accepting paths of a polynomial-time non-deterministic Turing machine. To cope with the task of counting maximum matchings in bipartite graphs, we propose to apply an approximate counting procedure called the stochastic enumeration (SE) method. It was shown that for some counting problems, the SE algorithm has a provable variance reduction guarantee [14, 21, 26]. In addition, the SE method has a further advantage in the sense that the algorithm can also be used for fully enumerating maximum matchings, subject to a reasonable cardinality of the maximum matchings set. The proposed two-step framework exploits this property in the full enumeration step. The major contribution of this study is as follows.

1. Our first contribution is the introduction of a randomized algorithm that can estimate the *hardness* of an OBS problem instance under consideration. In addition, as stated above, the proposed counting algorithm can also fully enumerate the set of all feasible solutions, namely, the set of all schedules that require the minimum number of aircrafts, provided that the latter is of a reasonable cardinality. In this case, the method allows to obtain the set of *globally* optimal solutions to the OBS problem.
2. Our second contribution is a fast and simple heuristic that one can apply, if the number of feasible solutions is too large for the full enumeration procedure.
3. Finally, we provide a research software package that can both handle real-life scheduling problems, and is able to provide good solutions while using a reasonable computation time. To the best of our knowledge, there exists no other non-proprietary package that can operate under the OBS problem setting.

The rest the paper is organized as follows. In Section 2 we formally establish the OBS problem setting and provide the required background on the correspondence of scheduling problems, partially ordered sets, and maximum matchings in bipartite graphs. The proposed two-stage framework is described in Section 3. In particular, we give an overview of the SE algorithm and detail the local search heuristic procedure. In Section 4 we present an experimental study that demonstrates the performance of the proposed methods when applied to several scheduling problems, including a real-life application to the

Australian domestic flights timetable. Finally, in Section 5 we summarize our findings and discuss possible directions for future research.

2 Problem formulation

In order to formally define the OBS problem setting, we start with important definitions that are partly adopted from [17]. Let $\mathcal{T} = \{1, \ldots, m\}$ be a set of *terminals*, where terminals are related to spatial locations such as airports or bus stops. In order to define schedules, we proceed with a formal definition of *passages* and *timetables*.

2.1 Definitions

Definition 1 (Passages and timetables) A *passage* (or trip), is defined as a 4-tuple $s = (p, q, t_s, t_e)$, where $p \in \mathcal{T}$ and $q \in \mathcal{T}$ denote the departure and the arrival terminals, and $t_s \in \mathbb{R}^+$ and $t_e \in \mathbb{R}^+$, such as $t_s \leq t_e$, stand for departure and arrival times, respectively. It is assumed that each passage is performed by a single *resource* (such as an aircraft or a vehicle), and that each resource can service any passage. A set of passages:

$$S = \left\{ s_i \stackrel{\text{def}}{=} \left(p^{(i)}, q^{(i)}, t_s^{(i)}, t_e^{(i)} \right) \; : \; i \in \mathcal{I} \right\},$$

where $\mathcal{I} = \{1, \ldots, n\}$ is a set of passage indices, is called a *timetable*.

Definition 2 (Feasibly joined passages) A passage $s_i \in S$ is *feasibly joined* to a passage $s_j \in S$, if s_i and s_j can be serviced sequentially by one resource. In this case, conditions: 1) $q^{(i)} = p^{(j)}$, and 2) $t_e^{(i)} \leq t_s^{(j)}$, are satisfied. If s_i is feasibly joined to s_j, we say that s_i is a *predecessor* of s_j, and that s_j is a *successor* of s_i.

Note that conditions 1) and 2) in Definition 2 are natural in the sense that the arrival terminal of the passage s_i is equal to the departure terminal of the passage s_j and that the arrival (end) time of passage s_i is smaller or equal to the departure (start) time of the s_j-th passage. It is convenient to express the relationship between feasibly joined passages s_i and s_j, using a notation of *partially order sets* (POSET) [22]. In our setting, a POSET is a pair $(\mathcal{P}, \preceq)$ of a set $\mathcal{P}$ and a binary relation $\preceq$. In particular, given a timetable S, we define $\mathcal{P} \stackrel{\text{def}}{=} S$ and use the binary relation $s_i \preceq s_j$ to signify that s_i is a predecessor of s_j.

A set of feasibly joined passages, namely, a sequence of passages $s_1, s_2, \ldots, s_l$, $1 \leq l \leq n$, ordered in such a way that each adjacent pair of passages satisfy conditions 1) and 2) from Definition 2, is called a *chain* or a *block*. That is, a chain is a set of passages that can be serviced by a *single resource*. In order to indicate that a sequence of passages $s_1, s_2, \ldots, s_l$ corresponds to a chain, we can write $s_1 \preceq s_2 \preceq \cdots \preceq s_l$. Finally, we arrive at the definition of a *schedule*.

Definition 3 (Schedule) Given a timetable S, a *schedule* is defined to be a set of *disjoint* chains, in which each passage $s_i \in S$ is included in exactly one chain.

1. We further define an *optimal schedule* to be a schedule that results from a timetable partition (or decomposition), to a minimum number of chains.
2. The OBS problem is thus defined as a task of finding an optimal schedule, which also contains a maximum number of balanced chains. We note that the chain $s_1 \preceq s_2 \preceq \cdots \preceq s_l$ is balanced, if $p^{(1)} = q^{(l)}$ holds.

Given a timetable, the number of chains in a chain decomposition is equal to the number of required resources needed to service this timetable. In other words, the chain decomposition determines the minimum required *fleet size* [17,24]. Therefore, it will be reasonable to examine decompositions with a minimum number of chains. We would like to stress again, that such decompositions are optimal in the sense of the number of required resources (or minimum fleet size), since we only require one resource per chain.

As noted in the introductory section, the minimum chain decomposition problem was considered by [9]. In particular, we can show that the scheduling task essentially corresponds to a more general Dilworth's problem. The Dilworth's problem setting is as follows. Let $(\mathcal{P}, \preceq)$ be a POSET and let two elements $u \in \mathcal{P}$ and $w \in \mathcal{P}$ be comparable, if $u \preceq w$ or $w \preceq u$ holds. Otherwise, we say that u and w are *non-comparable*. Then, a subset $\mathcal{U} \subseteq \mathcal{P}$ is independent, if for all $u, w \in \mathcal{U}$, u and w are non-comparable. Such $\mathcal{U}$ is also called an *antichain*. In addition, a subset $\mathcal{U} \subseteq \mathcal{P}$ is a *chain*, if *for all* $u, w \in \mathcal{U}$, u and w are comparable; that is, every two elements of $\mathcal{U}$ are comparable.

Dilworth's theorem states that for any *finite* POSET, the number of elements in a largest antichain is equal to the number of chains in a minimum chain decomposition [9]. The Dilworth's problem objective is to find the minimum chain decomposition of $(\mathcal{P}, \preceq)$, such that each element of $\mathcal{P}$ belongs to exactly one chain. In other words, the Dilworth's problem corresponds to the optimal scheduling problem from Definition 3. An illustration of minimum chain decomposition of a POSET is provided in Example 1.

Example 1 Let $\mathcal{P} = \{1, 3, 7, 21\}$ be the set of divisors of the number 21, and let us define element divisibility as the partial order. That is, it holds that:

$$1 \preceq 3, \quad 1 \preceq 7, \quad 1 \preceq 21, \quad 3 \preceq 21, \quad 7 \preceq 21.$$

This is not very hard to see that the maximum cardinality antichain is $(3, 7)$, and that a corresponding minimum chain decomposition is $\{(1, 3, 21),\ (7)\}$. However, it is important to note that the minimum decomposition $\{(1, 3, 21),\ (7)\}$ is not unique. For example, the $\{(1, 3),\ (7, 21)\}$ decomposition is also a minimum decomposition. In Example 2, we will see that for this specific POSET, there are four minimum decompositions and that the set of all minimum decompositions is:

$$\left\{ \left\{ (1,3,21)\,,\ (7) \right\},\ \left\{ (1,3),\ (7,21) \right\},\ \left\{ (1,7),\ (3,21) \right\},\ \left\{ (1,7,21),\ (3) \right\} \right\}.$$

An optimal solution to the minimum chain decomposition problem can be obtained in polynomial time, since there exists a direct correspondence of the Dilworth's problem to the problem of finding a *maximum matching* in bipartite graphs [11,13]. We formally define a matching as follows. Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a *matching* is a set of edges $E' \subseteq E$, such that no two edges in E' share a common vertex. A *maximum (cardinality) matching* is a matching in G, that has the maximum number of edges.

The relation of Dilworth's problem to the maximum matching problem in bipartite graphs was established by [11]. To see this correspondence, consider the following construction of a bipartite graph from a POSET

$$\left(\mathcal{P} \stackrel{\text{def}}{=} \{u_1, \ldots, u_n\} \; ; \preceq\right).$$

For each element $u_i \in \mathcal{P}$, where $1 \leq i \leq n$, define two vertices a_i and b_i, and let $G = (V_a, V_b, E)$ be a bipartite graph, where $V_a = \{a_1, \ldots, a_n\}$, and $V_b = \{b_1, \ldots, b_n\}$. In addition, let $(a_i, b_j) \in E$, if it holds that $u_i \preceq u_j$, for $1 \leq i, j \leq n$. Now, find a maximum bipartite matching M in G, and let C be a family of chains formed by including u_i and u_j in the same chain if there exists an edge $(a_i, b_j) \in M$. Then, the maximum matching M in G, corresponds to the decomposition C of $(\mathcal{P}, \preceq)$, such that C is a minimum chain decomposition [11]. It is worth noting that a maximum matching in a bipartite graph can be found efficiently, for example via the Hopcroft-Karp algorithm, in $\mathcal{O}\left(|E|\sqrt{|V_a \cup V_b|}\right)$ time [13]. This correspondence between maximum matchings and minimum chain decompositions is illustrated in Example 2.

Example 2 Consider the bipartite graph in Fig. 1, and note that this graph was induced by the POSET from Example 1. A careful observation of Fig. 2, reveals that the figure shows all possible maximum matchings of the bipartite graph from Fig. 1. Having in mind that a family of chains formed by including u_i and u_j in the same chain, if there is an edge $(a_i, b_j) \in M$, it is not very hard to see that Figures 2 (a), 2 (b), 2 (c), and 2 (d), correspond to the minimum chain decompositions:

$$\left\{\left\{(1,3,21)\,,\ (7)\right\},\ \left\{(1,3),\ (7,21)\right\},\ \left\{(1,7),\ (3,21)\right\},\ \left\{(1,7,21),\ (3)\right\}\right\},$$

respectively.

2.2 A timetable example

We finalize this section by demonstrating the importance of obtaining high quality solutions to the OBS problem. In order to do so, we consider a flight scheduling example from [24].

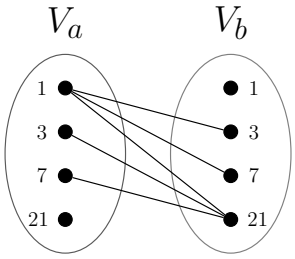

Fig. 1: A bipartite graph constructed from the POSET defined in Example 1.

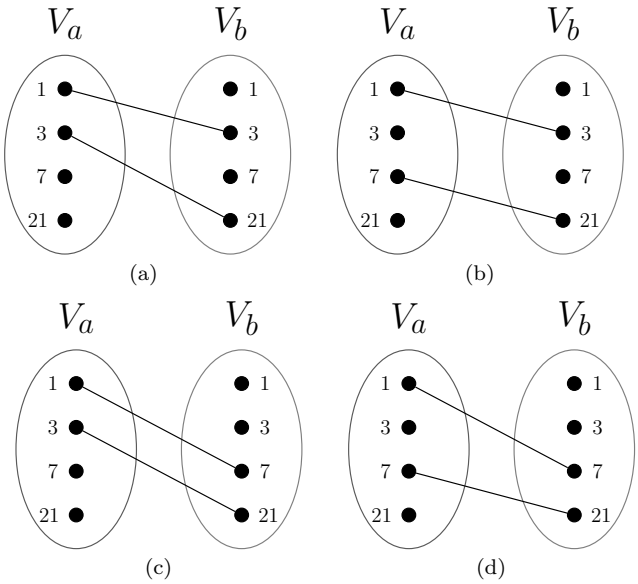

Fig. 2: All possible maximum matchings of the bipartite graph from Fig. 1.

Stern and Gertsbakh [24], examine the problem of finding a balanced flight schedule given a timetable with 30 passages and 4 terminals. The timetable data, which is denoted by F30, is summarized in Appendix A (Table 3). The authors show that the minimum chain decomposition has the cardinality of 12. That is, the minimum required aircraft fleet size that can service the F30 timetable, is equal to 12. Fig. 3 and Fig. 4 show two optimal schedules (or minimum decompositions). However, these schedules are different in the sense that there are 3 and 7 balanced chains in Fig. 3 (chains $1, 2$, and 3), and Fig. 4 (chains $1, 2, 3, 4, 5, 6$, and 7), respectively.

We already saw that aircrafts that service a balanced chain do not require a deadheading flight. In addition, the advantage of having many balanced chains is coherent from the maintenance point of view. For example, if required, an aircraft that services a balanced chain can undergo an overnight maintenance

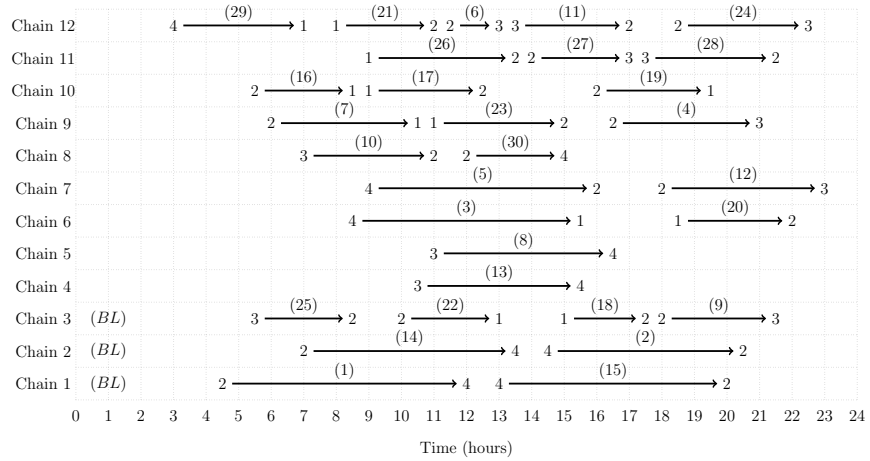

Fig. 3: A minimum decomposition of the F30 timetable. There are three balanced chains (chains $1, 2, 3$), denoted by (BL). Each flight is marked by an arrow and is identified by the flight number (over the arrow), and the start and the end terminals on the left and on the right sides of the arrow, respectively.

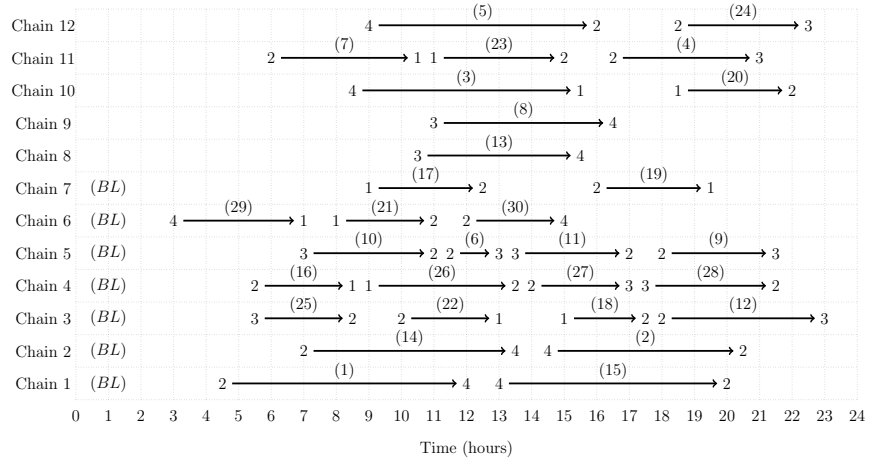

Fig. 4: A minimum decomposition of the F30 timetable. There are seven balanced chains (chains $1, \ldots, 7$), denoted by (BL). Each flight is marked by an arrow and is identified by the flight number (over the arrow), and the start and the end terminals on the left and on the right sides of the arrow, respectively.

work at its *home airport*. Moreover, a maximization of the number of aircrafts that return to the initial departure terminal at the end of the working day, can introduce additional benefits to *aircrew scheduling*. For instance, we can consider a *rest time* property of a chain. The rest time R is defined as the time between the last arrival time during the day and the next day departure. It

is crucial that such time is as long as possible, say at least 11 hours, since if the rest time is sufficient, the chain can be (conceivably) handled by the same aircrew during the next working day. For example, the rest times for chains $1, 2$ and 3 in Fig. 3 are: 8.5, 10.5, and 8 hours, respectively. On the other hand, the rest times for chains $1, \ldots, 7$ in Fig. 4 are: 8.5, 10.5, 6.5, 8, 9.5, 12, and 13.5 hours, respectively.

In the following section, we detail the proposed two-step procedure for handling OBS problems.

3 Methods

The proposed two-step procedure is summarized in Fig. 5. The input is a timetable S and a threshold value T, which we typically set to be $\mathrm{T} = 10^6$. Depending on the counting estimator value $\left(\widehat{\ell}\right)$, obtained in the first step, the execution of the second step results in either an *exact* or an *approximate* solution to the corresponding OBS problem.

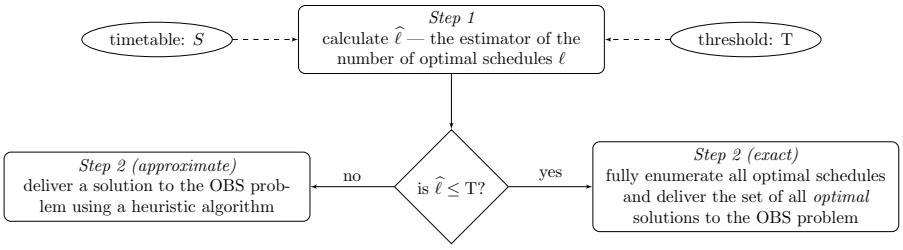

Fig. 5: The two-step framework for OBS problems.

Our first objective is to develop a randomized *counting* algorithm which is capable of delivering $\widehat{\ell}$, an estimator of the true number of optimal schedules ℓ. While the correspondence between optimal schedules and maximum matchings in bipartite graphs was established in Section 2, note that here we are concerned with the counting problem. Therefore, it is still required to establish that this reduction is *parsimonious*, namely, that the transformation from the original minimum chain decomposition problem, to the problem of counting of maximum matchings in the corresponding bipartite graph, preserves the number of solutions [12]. This result is established in Lemma 1.

Lemma 1 (Maximum matchings and minimum chain decompositions) The reduction between the minimum chain decomposition problem and the maximum matching problem is parsimonious.

Proof Recall that given a POSET

$$\left(\mathcal{P} \stackrel{\text{def}}{=} \{u_1, \ldots, u_n\} \; ; \; \preceq\right),$$

we define for all $u_i \in \mathcal{P}$ (where $1 \leq i \leq n$), two vertices a_i and b_i, and construct the corresponding bipartite graph $G = (V_a, V_b, E)$, where $V_a = \{a_1, \ldots, a_n\}$, $V_b = \{b_1, \ldots, b_n\}$, and $(a_i, b_j) \in E$, if it holds that $u_i \preceq u_j$ for $1 \leq i, j \leq n$. This construction can be performed in polynomial time in the size of the POSET. Specifically, the time complexity is $\mathcal{O}\left(n^2\right)$, since there are at most $\binom{n}{2} = \frac{n(n-1)}{2}$ binary relations in a POSET with n elements. Similarly, the polynomial complexity holds for the task of constructing the POSET $(\mathcal{P}, \preceq)$ from a bipartite graph $G = (V_a, V_b, E)$.

The one-to-one relationship between a minimum decomposition and a maximum matching is established from the construction process. In particular, recall that given a maximum matching M in G, we defined a family of chains C (where C is a minimum decomposition [11], formed by including u_i and u_j in the same chain of C, if $(a_i, b_j) \in M$). That is, by construction, each maximum matching M corresponds to a unique minimum chain decomposition C. In order to see that every minimum chain decomposition C corresponds to a unique maximum matching in G, consider two different minimum chain decompositions C and C', and suppose that both C and C' correspond to the same maximum matching M in G. Since $C \neq C'$, there exists at least one pair of POSET's elements u_i and u_j, such that u_i is a predecessor of u_j in some chain of C, and such that u_i is not a predecessor of u_j in any chain of C'. Therefore, it both holds that $(a_i, b_j) \in M$ and that $(a_i, b_j) \notin M$, which leads to a contradiction to the assumption that C and C' correspond to the same maximum matching M in G.

From Lemma 1, we conclude that the reduction to the maximum matchings problem is parsimonious. That is, in order to count minimum decompositions (or optimal schedules), one can instead count maximum matchings in the corresponding bipartite graph. We next show how the counting maximum matching problem can be regarded as a *tree counting problem*, and give a brief overview of the general SE method for counting trees.

3.1 The tree counting problem

Following the above discussion, we restrict our attention to counting maximum matchings in bipartite graphs. The problem is in #P [28, 20, 10, 8], and, therefore, our focus is on the development of an approximate counting technique. We start with the definition of the tree counting problem.

Definition 4 (The tree counting problem [26]) Consider a rooted tree $T = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$ (so that $|\mathcal{E}| = |\mathcal{V}| - 1$). We denote the root of the tree by v_0, and for any $v \in \mathcal{V}$, the subtree rooted at v is denoted by T_v. With each node v is associated a cost $c(v) \in \mathbb{R}$. Then, the tree counting problem is to calculate the total cost of the tree,

$$\text{Cost}(T) \stackrel{\text{def}}{=} \text{Cost}\left(T_{v_0}\right) = \sum_{v \in \mathcal{V}} c(v),$$

or more generally, the total cost of a subtree T_v denoted by $\text{Cost}(T_v)$.

We first show that the task of calculating the number of maximum matchings in a bipartite graph can be reduced to the tree counting problem from Definition 4. To see this, consider the following tree construction process. Given a bipartite graph $G = (V_a, V_b, E)$, where $V_a = \{a_1, \ldots, a_n\}$ and $V_b = \{b_1, \ldots, b_n\}$, suppose without loss of generality that $(a_1, \ldots, a_n)$ is an arbitrary vertex ordering of V_a. Next, define the tree root and suppose that the root node contains an empty matching prefix, namely $M = \emptyset$. In addition, assume that the root node is *associated* with the a_1 vertex and denote all edges incident to a_1 by $E_{a_1} = (e_1, \ldots, e_k)$. For each edge $e \in E_{a_1}$, examine if the inclusion of e in M, can eventually result in a maximum matching. Additionally, check if there is a possibility of not including the a_1 vertex in the maximum matching at all. Note that these subproblems are easy in the sense that they can be solved in polynomial time [13]. Each successful possibility detailed above, is extended to be a child of the tree node associated with a_1, which in turn will be associated with the next vertex in the ordering, namely, with a_2, and will contain the corresponding maximum matching prefix. The tree construction is then continued in the recursive fashion, until a leaf node is reached. Such leaf node will contain a matching that has a maximum cardinality, that is, a maximum matching. For each such leaf node, we assign a cost of 1; all other (non-leaf) nodes are being assigned with the zero cost. A formal consideration of the correctness of the proposed tree counting construction is given in Proposition 1.

An important concept which is required for further discussion is a *tree level*.

Definition 5 (The tree level) Consider a rooted tree $T = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$. The number of edges from the root node of T to a node $v \in \mathcal{V}$ is called a depth of v. We refer to the collection of all nodes that have the same depth as a tree level.

Example 3 (A tree construction) To get a better understanding of the above tree construction process, consider the bipartite graph from Example 2. We saw that in this example, the cardinality of the maximum matching is 2. Suppose now that $(a_1, a_2, a_3, a_4) = (1, 3, 21, 7)$ is an arbitrary vertex ordering of $V_a = \{1, 3, 7, 21\}$, and let us examine the creation of the corresponding tree, which is shown in Fig. 6. We start with the root node, which is associated with a_1 and contains the empty maximum matching prefix $M = \emptyset$. Note that vertex a_1 should be included in all maximum matchings. To see this, note that if $a_1 = 1$ is not in the matching, then, the resulting matching will be of cardinality 1 since we only have two remaining edges, $(3, 21)$ and $(7, 21)$ (please see Fig. 1). Therefore, no child will contain the empty matching prefix $M = \emptyset$. Moreover, the edge $(1, 21)$ cannot be a part of a maximum matching too. That is, if we add $(1, 21)$ to the matching, then, it is not very hard to see (using Fig. 1), that no other edges can be added to this matching. Thus, the resulting matching is of cardinality 1 and this is not a maximum matching.

With this in mind, the root split will result in two children induced by edges $(1,3)$ and $(1,7)$ (these are the only possibilities that can potentially produce a maximum matching — please see Fig. 2). At the second level of the tree (which contains the $M = \{(1,3)\}$ and the $M = \{(1,7)\}$ nodes), we consider the set of edges incident to a_2. Taking a closer look at the leftmost child of the root (which contains the matching prefix $M = \{(1,3)\}$), we note that there is only one edge to consider, specifically, the $(3,21)$ edge (please see Fig. 2 (a)). This edge corresponds to the leftmost child at level 3, which also happens to be a leaf node since it contains the maximum matching $M = \{(1,3),(3,21)\}$. On the other hand, one can extend the $M = \{(1,3)\}$ matching without the a_2 vertex (please see Fig. 2 (b)), and this is the reason for having a child with the $M = \{(1,3)\}$ matching prefix at level 3; note that this is the rightmost child of the $M = \{(1,3)\}$ node at level 2. The recursive construction for each node continues in the similar fashion and we can readily verify that the Fig. 6 tree leaf nodes contain all possible maximum matchings (that were detailed in Example 2).

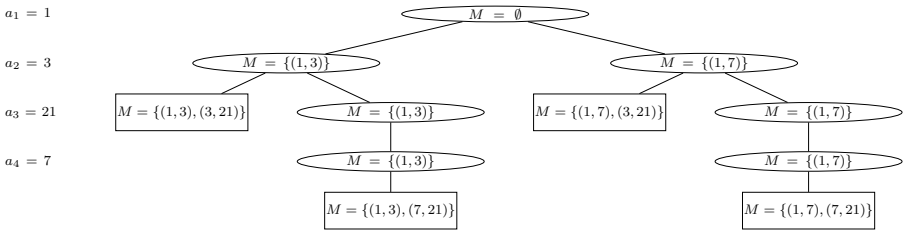

Fig. 6: The counting tree for the bipartite graph from Example 2.

Proposition 1 (The tree construction process) Given a bipartite graph $G = (V_a, V_b, E)$, and the proposed construction process of the corresponding counting tree T, it holds that:

$$\text{Cost}(T) = |\{M \,|\, M \text{ is a maximum matching in } G\}|.$$

Proof First, note that a tree leaf node $v \in \mathcal{V}$ such that $c(v) = 1$, will be associated with a matching of cardinality $|M|$; here, $|M|$ is the cardinality of a maximum matching in G. This follows from the construction process. Specifically, during the tree creation, a child is formed only if it can later be extended to a maximum matching. In addition, for each non-leaf tree node, the split leads to different matchings. To see this, let $a \in V_a$ be a vertex associated with a non-leaf tree node $u \in \mathcal{V}$. Note that for each tree child of u, it holds that a is either matched with some vertex $b \in V_b$, or not matched at all. In other words, each child of u contains a prefix of some distinct maximum matching in G.

In addition, for any maximum matching M, it is not very hard to show that there exists a path from the tree root to a leaf node, such that this leaf

node contains M. In order to reconstruct this path, fix a maximum matching M and consider, without loss of generality, the $(a_1, \ldots, a_n)$ vertex ordering of V_a. Now, start the reconstruction process from the tree root. Given M, there are two possibilities, that is, for some $2 \leq j \leq n$, either $(a_1, b_j) \in M$, or $(a_1, b_j) \notin M$. If it holds that $(a_1, b_j) \in M$, we consider the corresponding child of the root node, which contains the matching prefix $\{(a_1, b_j)\}$. Otherwise, if $(a_1, b_j) \notin M$, consider the child of the root node that contains the empty matching prefix ($M = \emptyset$). Note that such children should exist based on the construction process definition. By continuing recursively in the same fashion at each tree level associated with the $(a_2, \ldots, a_n)$ ordering, we arrive to the leaf node that contains the desired matching M.

Unfortunately, the corresponding tree can be prohibitively large, and, therefore, one cannot usually construct or inspect it explicitly. However, it is possible to traverse such tree in a stochastic manner, namely, to perform a random walk starting from the tree root and ending in one of the tree leaves. Our objective is to develop an estimator for the true cost of the tree, while restricting ourselves to the usage of random walks. Under our setting, the estimator will provide an approximate cardinality of the set of all maximum matchings. In order to accomplish this task, we proceed with a brief overview of the SE algorithm, which provides an unbiased estimator and is basically a generalization of Knuth's estimator for counting trees [14, 21, 26].

3.2 The stochastic enumeration method for counting trees

Definition 6 sets the stage by establishing the required notation.

Definition 6 (Hyper nodes and forests [26])
Consider a rooted tree $T = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$, where $|\mathcal{E}| = |\mathcal{V}| - 1$. Let $\{v_1, \ldots, v_r\} \in \mathcal{V}$ be a subset of tree nodes such that $r \leq |\mathcal{V}|$.

- We call a collection of distinct nodes in the same level (depth) of the tree $\mathbf{v} = \{v_1, \ldots, v_r\}$ a *hyper node* of cardinality $|\mathbf{v}| = r$.
- Let $\mathbf{v}$ be a hyper node. Generalizing the tree node cost, we define the cost of the hyper node as $c(\mathbf{v}) = \sum_{v \in \mathbf{v}} c(v)$.
- Let $\mathbf{v}$ be a hyper node. Define the set of successors of $\mathbf{v}$ as $S(\mathbf{v}) = \bigcup_{v \in \mathbf{v}} S(v)$, where $S(v)$ is the set of successors (children) of node v.
- Let $\mathbf{v}$ be a hyper node and let $B \in \mathbb{N}$, $B \geq 1$. Define

$$H(\mathbf{v}) = \begin{cases} \{S(\mathbf{v})\} & \text{if} \quad |S(\mathbf{v})| \leq B \\ \{\mathbf{w} \mid \mathbf{w} \subseteq S(\mathbf{v}),\ |\mathbf{w}| = B\} & \text{if} \quad |S(\mathbf{v})| > B, \end{cases} \tag{1}$$

  to be the set of all possible hyper nodes having cardinality $\max\{B, |S(\mathbf{v})|\}$ that can be formed from the set of $\mathbf{v}$'s successors. Note that if $|S(\mathbf{v})| \leq B$, we get a single hyper node with cardinality $|S(\mathbf{v})|$.

– For each hyper node $\mathbf{v}$ let $T_{\mathbf{v}} = \bigcup_{v \in \mathbf{v}} T_v$, be the forest of trees rooted at $\mathbf{v}$. An example of hyper node $\mathbf{v} = \{v_1, v_2, v_3, v_4\}$ and its corresponding forest $T_{\mathbf{v}} = \{T_{v_1}, T_{v_2}, T_{v_3}, T_{v_4}\}$ is shown in Fig. 7.
– For each forest rooted at hyper node $\mathbf{v}$, define its total cost as $\text{Cost}\,(T_{\mathbf{v}}) = \sum_{v \in \mathbf{v}} \text{Cost}(T_v)$.

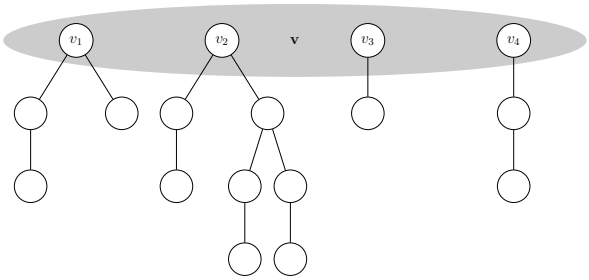

Fig. 7: Hyper node $\mathbf{v}$ that contains regular tree nodes v_1, v_2, v_3 and v_4 with their corresponding subtrees.

Based on Definition 6, we can state the main SE procedure, which is summarized in Algorithm 1.

Algorithm 1: The stochastic enumeration algorithm

Input: A forest $T_{\mathbf{v}}$ rooted at a hyper node $\mathbf{v}$, a cost function $c : \mathcal{V} \to \mathbb{R}$, and a budget $B \geq 1$
Output: An unbiased estimator of $\text{Cost}(T_{\mathbf{v}})$

1 Set $k \leftarrow 0$, $D \leftarrow 1$, $\mathbf{X}_0 \leftarrow \mathbf{v}$ and $C_{\text{SE}} \leftarrow c(\mathbf{X}_0)/|\mathbf{X}_0|$.
2 Let $S(\mathbf{X}_k)$ be the set of all children of $\mathbf{X}_k$.
3 **if** $S(\mathbf{X}_k) = \emptyset$ **then**
4 | **return** $|\mathbf{v}|\, C_{\text{SE}}$ as an estimator of $\text{Cost}(T_{\mathbf{v}})$.
5 **end**
6 Choose hyper node $\mathbf{X}_{k+1} \in H(\mathbf{X}_k)$ at random (note that $H(\mathbf{X}_k)$ is defined in (1) and depends on the budget B), each choice being equally likely.
7 Set $D_k \leftarrow |S(\mathbf{X}_k)|/|\mathbf{X}_k|$, $D \leftarrow D_k D$, and

$$C_{\text{SE}} \leftarrow C_{\text{SE}} + \left(\frac{c(\mathbf{X}_{k+1})}{|\mathbf{X}_{k+1}|} \right) D.$$

8 Set $k \leftarrow k + 1$ and return to line 2.

It can be shown that Algorithm 1 outputs a random variable

$$C_{\mathrm{SE}} = \frac{c(\mathbf{X}_0)}{|\mathbf{X}_0|} + \frac{|S(\mathbf{X}_0)|}{|\mathbf{X}_0|}\frac{c(\mathbf{X}_1)}{|\mathbf{X}_1|} + \frac{|S(\mathbf{X}_0)|}{|\mathbf{X}_0|}\frac{|S(\mathbf{X}_1)|}{|\mathbf{X}_1|}\frac{c(\mathbf{X}_2)}{|\mathbf{X}_2|} + \cdots \tag{2}$$
$$\cdots + \left(\prod_{0\le j\le \tau-1} \frac{|S(\mathbf{X}_j)|}{|\mathbf{X}_j|}\right)\frac{c(\mathbf{X}_\tau)}{|\mathbf{X}_\tau|},$$

where $\tau \le h$ (here, h is the height of the forest's deepest tree), is the random variable that represents the length of the random walk [26]. The term

$$\left(\prod_{0\le j\le k-1} \frac{|S(\mathbf{X}_j)|}{|\mathbf{X}_j|}\right)\frac{c(\mathbf{X}_k)}{|\mathbf{X}_k|} = D\frac{c(\mathbf{X}_k)}{|\mathbf{X}_k|},$$

(which is calculated in line 7 of Algorithm 1), is the estimator of the total cost of vertices at the k-th tree level. In addition, this estimator is unbiased; that is, for a tree T rooted at v_0, and for $\mathbf{v}_0 = \{v_0\}$, $\mathbb{E}\left(C_{\mathrm{SE}}\right)) = \mathrm{Cost}\left(T\right)$. For the proof of unbiasedness, efficiency considerations, and examples, we refer to [27, 26].

While Algorithm 1 outputs an unbiased estimator of the true counting value, a common practice is to repeat this algorithm for $N_{\mathrm{SE}} \ge 1$ independent replications, obtain independent unbiased estimators $C_{\mathrm{SE}}^{(1)}, \ldots, C_{\mathrm{SE}}^{(N_{\mathrm{SE}})}$, and report the average:

$$\widehat{\ell} = \frac{1}{N_{\mathrm{SE}}}\sum_{r=1}^{N_{\mathrm{SE}}} C_{\mathrm{SE}}^{(r)}.$$

As soon as these N_{SE} estimators are available, one can also measure the accuracy of the final estimator $\widehat{\ell}$, by calculating the estimator's relative error (RE), which is equal to $\sqrt{\mathrm{Var}\left(\widehat{\ell}\right)/\left(\mathbb{E}[\ell]\right)^2}$ [21]. Since the exact RE is generally not available, a common practice is to estimate this quantity via

$$\widehat{\mathrm{RE}} = \sqrt{\frac{\frac{1}{N_{\mathrm{SE}}-1}\sum_{r=1}^{N_{\mathrm{SE}}}\left(C_{\mathrm{SE}}^{(r)} - \widehat{\ell}\right)^2}{\widehat{\ell}^2}}.$$

The major advantage of the SE algorithm is due to its built-in splitting mechanism [21, Chapter 4]. The latter can bring an enormous variance reduction, and in some cases, to achieve a *zero-variance* estimation [3]. In fact, if we set the SE algorithm budget parameter to be greater than or equal to the maximum tree width [4], the SE algorithm provides a zero-variance estimator and can be used for fully enumerating all leaf nodes. In this case, all maximum matchings will be placed at the tree leaf nodes that will be present in the hypernode $\mathbf{X}_h$ in the final iteration of Algorithm 1. Thus, a set of optimal solutions can readily be obtained by reviewing all nodes of the $\mathbf{X}_h$ hypernode.

Example 4 We consider the tree from Example 3 and execute the SE Algorithm 1 with budget $B = 1$. With the view to counting leaf nodes, we further assume that the cost function is

$$\begin{cases} c(v) = 1 & \text{if } v \text{ is a leaf node,} \\ c(v) = 0 & \text{otherwise.} \end{cases}$$

Note that there are four random walks from the tree root to the tree leaves. Moreover, it is not very hard to see that each random walk is chosen with probability 1/4. Finally, the value of the random variable D is always equal to $1 \times 2 \times 2 = 4$, since for any random walk, there are exactly two splits at levels 1 and 2, and for each split, there are two children. We conclude that the SE algorithm outputs $C_{\mathrm{SE}} = 1 \times D = 4$, regardless of the chosen path. In this case, it is clear that $\mathbb{E}[C_{\mathrm{SE}}] = 4$, and that

$$\begin{aligned} \mathbb{E}\left[C_{\mathrm{SE}}^2\right] &= (1 \times 4)^2 \frac{1}{4} + (1 \times 4)^2 \frac{1}{4} + (1 \times 4)^2 \frac{1}{4} + (1 \times 4)^2 \frac{1}{4} = 16 \\ &\Rightarrow \mathrm{Var}(C_{\mathrm{SE}}) = \mathbb{E}\left[C_{\mathrm{SE}}^2\right] - \left(\mathbb{E}[C_{\mathrm{SE}}]\right)^2 = 16 - 4^2 = 0. \end{aligned}$$

That is, we obtained a zero-variance estimator. Please note that this favorite scenario is not very common in practice. In particular, we managed to obtain a zero-variance estimator while using the smallest possible budget $B = 1$, because the tree from Example 3 is highly *symmetric*. This rarely happens in reality, and the user needs to increase the budget B in order to reduce the estimator's variance. For additional examples, we refer to Section 3 and the Appendix in [27]. Finally, we note that the maximum width of the tree in Fig. 6 is equal to 4, so, by setting $B = 4$, the SE algorithm will reveal the entire set of leaf nodes (which will be present in the hypernode $\mathbf{X}_4$ in the final iteration of Algorithm 1).

In order to complete the description of the two-step framework from Fig. 5, we need to consider the case in which the counting estimator has a large value; under this setting, the full-enumeration procedure will be computationally infeasible. Having in mind that the problem of finding balanced chains is NP-hard [29, 24], we suggest to use a heuristic method to obtain the desired solution. The proposed heuristic is detailed next.

3.3 The heuristic

In this section, we are concerned with the task of constructing a heuristic procedure that can handle the OBS problem. Specifically, our major objective is to maximize the number of balanced chains. However, in addition, we would like to consider a secondary objective that was discussed in Section 2.2, specifically, the maximization of the number of chains with the rest time that is greater than or equal to some predefined threshold. Moreover, the aim is to construct a procedure that is fast, namely, we expect the method to be

able to handle relatively big timetables that emerge from real-life problems. Finally, the procedure should be designed in such a way that it can be easily extended to handle additional optimization criteria. For example, we might like the flight time of each aircraft, and in particular the total flight time in each chain, to be as similar as possible.

With this in mind, we propose to apply the heuristic approach which is summarized in Algorithm 2. The heuristic in Algorithm 2 is essentially a *local search* procedure. First, recall that there exists a correspondence between decompositions and maximum matchings. Algorithm 2 starts with an arbitrary maximum matching M (line 3). Then, using a randomized procedure, a new maximum matching M' is produced. Since there exist decompositions d and d' that correspond to M and M', respectively, one can choose and accept the best decomposition (and matching) (see lines 8-11). The procedure continues in this fashion until some stooping condition is met.

Algorithm 2: The heuristic

Input: A timetable S, and a binary relation $\preceq_{\text{os}}$ over the sets of all optimal schedules
Output: An optimized schedule

```
1  Create a POSET (P, ⪯) from S.
2  Construct a bipartite graph G = (V_a, V_b, E) from the POSET (P, ⪯); see Section 2
3  Find a maximum matching M of G = (V_a, V_b, E)
4  Construct a chain decomposition d from the maximum matching M
5  while stop condition is not met do
6  |  Create maximum matching M' by modifying the maximum matching M using
   |    Algorithm 3
7  |  Construct a chain decomposition d' from the maximum matching M'
8  |  if d ⪯_os d' then
9  |  |  M ← M'
10 |  |  d ← d'
11 |  end
12 end
13 return d – an optimized schedule
```

In order to complete the description of the heuristic Algorithm 2, we need to specify the termination condition in line 5, the binary relation $\preceq_{\text{os}}$ in line 8, and the creation of a modified maximum matching in line 6.

1. The termination condition in line 5 depends on the user preferences. Specifically, one can decide on a certain number of iterations of the while loop (lines 5 – 12), or alternatively, to specify a predefined runtime limit for the algorithm's execution.
2. While the binary relation $\preceq_{\text{os}}$ can be easily modified or extended according to user's needs, in this study, the relation $\preceq_{\text{os}}$ is defined as follows. Given a minimum chain decomposition d, let $B(d)$ and $R(d)$ be the number of balanced chains and the number of chains with rest time greater than or equal to some threshold (say eleven hours), respectively. Then, for two

Algorithm 3: The maximum matching sampler

Input: A bipartite graph $G = (V_a, V_b, E)$, and a maximum matching $M \subseteq E$
Output: A maximum matching

```
1  k ← |M|
2  Select an edge e ∈ M uniformly at random and remove e from M
3  while |M| < k do
4  |  Select an edge e = (u, v) ∈ E \ M uniformly at random
   |  /* We say that a vertex w ∈ V_a ∪ V_b is matched in M, if there exists
   |     an edge e = (w, z) ∈ M, where w ∈ V_a and z ∈ V_b, or w ∈ V_b and
   |     z ∈ V_a, holds.                                                  */
5  |  if u is matched in M and v is matched in M then
   |  |  /* there exist w, z ∈ V_a ∪ V_b such that (u, w) ∈ M and (v, z) ∈ M
   |  |     --- do nothing.                                               */
6  |  end
7  |  else if u is not matched in M and v is not matched in M then
   |  |  /* there is no w ∈ V_a ∪ V_b and z ∈ V_a ∪ V_b such that (u, w) ∈ M and
   |  |     (v, z) ∈ M. Namely, it is safe to add the e = (u, v) edge to the
   |  |     matching M.                                                   */
8  |  |  M ← M ∪ {(u, v)}
9  |  end
10 |  else if u is matched to some w ∈ V_a ∪ V_b in M and v is unmatched in M then
   |  |  /* there exist w ∈ V_a ∪ V_b such that (u, w) ∈ M and there is no
   |  |     z ∈ V_a ∪ V_b such that (v, z) ∈ M --- exchange (u, w) with (u, v) */
11 |  |  M ← M \ {(u, w)}
12 |  |  M ← M ∪ {(u, v)}
13 |  end
14 |  else if v is matched to some w ∈ V_a ∪ V_b in M and u is unmatched in M then
   |  |  /* there exist w ∈ V_a ∪ V_b such that (v, w) ∈ M and there is no
   |  |     z ∈ V_a ∪ V_b such that (u, z) ∈ M --- exchange (v, w) with (u, v) */
15 |  |  M ← M \ {(v, w)}
16 |  |  M ← M ∪ {(u, v)}
17 |  end
18 end
19 return M
```

minimum chain decompositions d and d' we write:

$$d \preceq_{\text{os}} d' \text{ if: } B(d') > B(d), \text{ or if: } B(d') = B(d) \text{ and } R(d') \geq R(d). \quad (3)$$

In other words, we set a preference to the maximization of the balanced chains, and then, to the maximization of chains with rest time greater than or equal to eleven hours.

3. Algorithm 3 summarizes the maximum matching sampler, which is used in line 6 of Algorithm 2. This algorithm closely follows the Markov chain of [5]; for additional details, we refer to [23].

4 Experimental study

In this section, we focus on the performance evaluation of the proposed framework. Specifically, for each model under consideration, we apply the SE method

and obtain the counting estimator for the number of minimum chain decompositions. If the estimator delivers a manageable value, that is, if the estimator's value is less or equal to the $T = 10^6$ threshold, we perform the full enumeration procedure to obtain the set of optimal solutions to the OBS problem. If the counting estimator is larger than this threshold, we apply the heuristic from Section 3.3. Our benchmark study shows that the proposed framework is very useful for determining the problem hardness, and that it is effective and scalable in practice. Specifically, we consider the following experiments.

1. Our first case study is the F30 timetable example with 30 passages and 4 terminals from [24]. By applying the proposed two-step framework, we show that this problem is easy (there are only 576 solutions to consider), in the sense that the full enumeration of all optimal schedules is computationally feasible, and thus we can obtain an optimal solution to the corresponding OBS problem. In addition, since we know the optimal solution, this case study allows to benchmark the heuristic algorithm from Section 3.3.
2. The purpose of the second case study is to show that one should not be tempted to estimate a problem hardness by considering timetable parameters such as the number of terminals and the number of passages, only. To see this, we generate a random timetable, which is similar to the F30 example, in the sense that it has 30 passages and 4 terminals. However, we show that the counting estimator's value is quite large (about 7.19×10^8), and, therefore, a full enumeration is difficult to achieve in a reasonable time. As a consequence, we apply the proposed heuristic in order to obtain a (possibly sub-optimal) solution to the corresponding OBS problem.
3. In the third case study, we show that the proposed framework is suitable for handling large instances. In particular, we benchmark the method performance on a real-life Australian domestic flights timetable with 20 airports and 290 passages. In this benchmark, we show that this timetable can be serviced by 88 aircrafts, and by applying Algorithm 2, we were able to increase the number of balanced flight routes from 9 to 65.

Experimental setup

We implemented the proposed framework in `C++` packages called *SeMaxMatching* (Algorithm 1), and *TTOpt* (Algorithm 2 and Algorithm 3). These packages are freely available on the author's website under `https://people.smp.uq.edu.au/RadislavVaisman/#software`. The software was compiled using `GNU g++` with full optimization for speed (using the -O3 flag). All timing measures were instrumented directly into the code. All tests were executed on an Intel Core i7-6920HQ CPU 2.90GHz processor with 32GB of RAM running 64 bit Ubuntu 18.04 LTS. We did not implement parallelization, so all the software is single-threaded. However, due to the nature of the SE algorithm, parallelization would be relatively easy to include. To ensure reproducibility of the reported results, unless stated otherwise, we use a fixed seed of `12345` when executing all algorithms. For the SE algorithm, we need two parameters

B – the budget, and N_{SE} – the number of replications. The heuristic requires a single parameter N – the number of iterations. As mentioned above, the two-step framework in Fig. 5, also requires the threshold parameter T. For all case studies, we solve the OBS problem using the two-step procedure. Our main objective is to maximize the number of balanced chains. However, since the framework was specifically designed to handle additional optimization criteria, we also consider the maximization of the number of chains with rest time R, such that $R \geq 11$ hours.

4.1 The FS30 case study

We start with the case study from [24]. The timetable consists of 30 passages and 4 terminals. For this instance, we benchmark the SE algorithm with different budgets and different number of replications.

1. *Step 1:* The SE results for the FS30 instance are summarized in Table 1. The data in Table 1 is informative in the sense that it shows how the $\widehat{\mathrm{RE}}$ decreases as the budget B grows. In addition, it is clear that the number of solutions is very small as compared to the threshold $\mathrm{T} = 10^6$. In fact, the last row (with $B = 1000$), gives us a zero-variance estimator, so we conclude that there are exactly 576 solutions. The latter implies that the problem is easy, that is, one can fully enumerate all solutions and deliver the set of all optimal balanced schedules.

Table 1: The SE algorithm counting estimators for the FS30 benchmark using different budgets and different number of replications.

B	N_{SE}	$\widehat{\ell}$	$\widehat{\mathrm{RE}}$	Time (sec)
1	1000	583.2	0.0289	3.91
100	10	571.8	0.0172	2.27
1000	1	576.0	0.0000	0.99

2. *Step 2:* By performing the full enumeration, we found that an optimal OBS solution has seven balanced chains, and there exist optimal OBS solutions with five chains with rest time R, which is greater than or equal to eleven hours. We utilize this knowledge to benchmark the *TTOpt* heuristic. In particular, we execute Algorithm 2 for $N = 10^5$ iterations. The corresponding runtime for the completion of the *TTOpt* heuristic is about 0.56 seconds. Fig. 8 depicts a typical performance of the heuristic algorithm.

In summary, the FS30 instance is easy, so one can find an optimal solution by performing full enumeration. However, it will be incorrect to conclude that the instance hardness is determined by the number of passages and the number of terminals. To see this, we consider another (randomly generated) timetable with 30 passages and 4 terminals. Nevertheless, we will show that this instance has many optimal schedules.

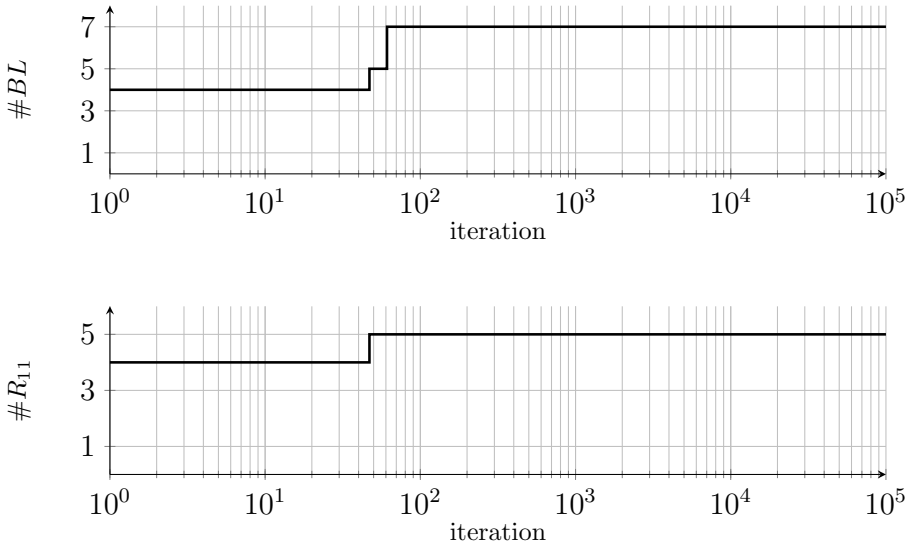

Fig. 8: Convergence of the *TTOpt* heuristic algorithm for the FS30 instance. The upper and the lower figures show the number of balanced chains ($\#BL$), and the number of chains with rest time which is greater than or equal to eleven hours ($\#R_{11}$), as a function of the algorithm's iteration number, respectively.

4.2 A random timetable with thirty passages and four terminals

The random timetable was generated in the following way. For each passage, we selected the departure and the arrival terminals uniformly at random. That is, each terminal from the $\{1, 2, 3, 4\}$ set is selected with probability $1/4$. The start time was selected uniformly at random from the 6:00 AM to 7:00 PM time interval. Finally, we set the flight time for each passage to have a fixed length of one hour.

1. *Step 1:* By running the SE algorithm with $B = 100$ and $N_{\mathrm{SE}} = 100$, the method delivered the counting estimator value of 7.19×10^8 and the $\widehat{RE}$ of 0.0489; the execution time is about 47 seconds. In this case, the full enumeration is obviously problematic, and, therefore, we apply the heuristic procedure in the second step of the two-step framework.
2. *Step 2:* We run the *TTOpt* heuristic for $N = 10^5$ iterations. The corresponding runtime for the completion of the *TTOpt* heuristic is about 0.45 seconds. Fig. 9 depicts a typical performance of the heuristic algorithm.
   We also tried to run the heuristic with ten different seeds (specifically, by starting the algorithm with seeds: $1, 2, \ldots, 10$), and increased the number of iterations N to 10^6. Nevertheless, the algorithm always returned the same result; namely, four balanced chains and eleven chains with the rest time $R \geq 11$ hours.

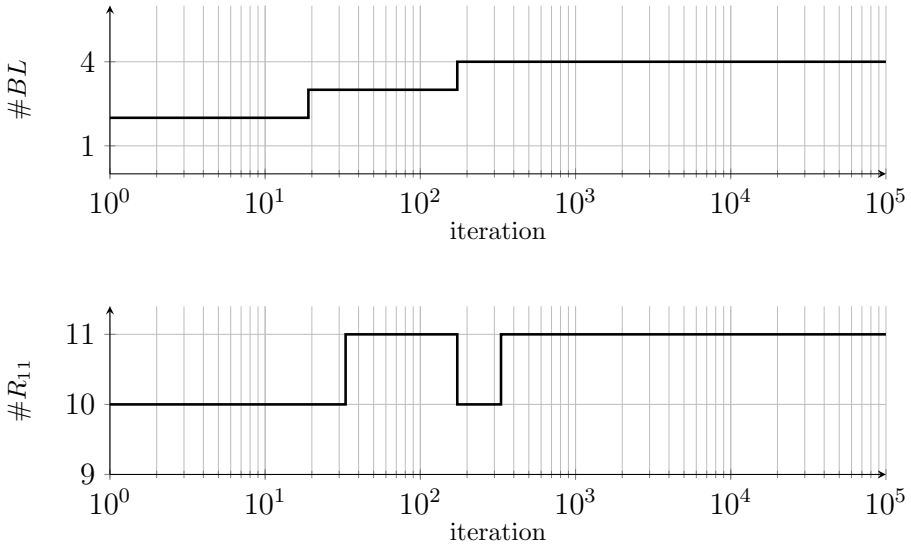

Fig. 9: Convergence of the *TTOpt* heuristic algorithm for the random timetable with 30 passages and 4 terminals. The upper and the lower figures show the number of balanced chains ($\#BL$), and the number of chains with rest time which is greater than or equal to eleven hours ($\#R_{11}$), as a function of the algorithm's iteration number, respectively.

Remark 1 (The required number of iterations N of the heuristic algorithm and the non-monotonic behavior of $\#R_{11}$) While it is not easy to specify the required number of iterations N, we recommend to execute the heuristic until the values of $\#BL$ and $\#R_{11}$ stop changing for some predefined number of iterations, say several thousands. We would also like to note that the lower figure which corresponds to the *rest time*, can show a non-monotonic behavior, while the upper plot which measures the number of balanced chains is monotonically non decreasing. This behavior is due to the usage of the performance function defined in (3). Specifically, this happens because we treat the number of balanced chains objective as the major one.

4.3 The Australian domestic flight timetable

The third case study is motivated by the fact that the Australian domestic flight timetable is large enough to represent a realistic problem. While there are 44 domestic airports in Australia, some major airports with the heaviest traffic are shown in Fig. 10. The set of busiest airports include the airports of Sydney, Melbourne, Brisbane, Perth, and Adelaide. It is worth noting that the flight time between Perth and Brisbane exceeds five hours, so the corresponding deadheading flight is quite expensive. This further emphasizes the importance of obtaining good solutions to the OBS problem.

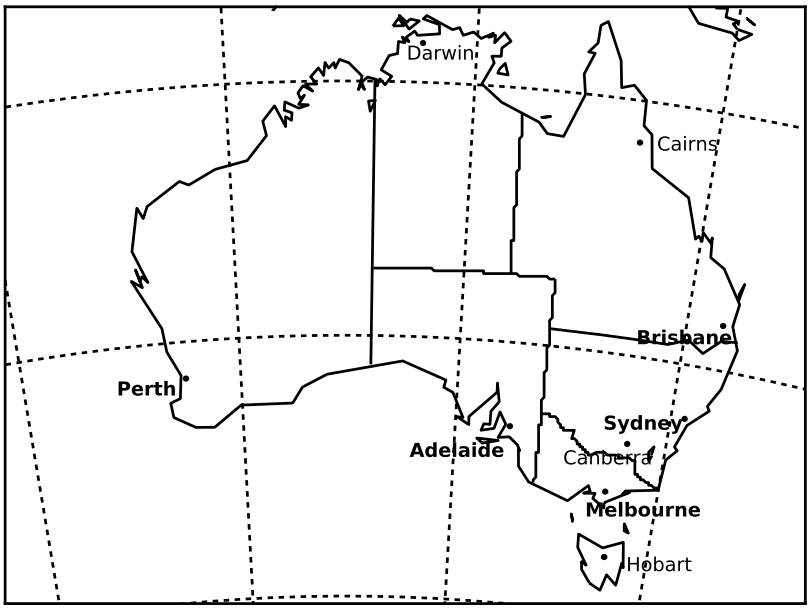

Fig. 10: Major domestic airports in Australia.

In this case study we consider the top twenty airports. The data was extracted from the open data-source (Monthly Airport Traffic Data for top twenty airports: January 2009 to current), which is available from `https://www.bitre.gov.au/sites/default/files/documents`. The list of top twenty airports (in alphabetical order) is: Adelaide, Alice springs, Ballina, Brisbane, Cairns, Canberra, Darwin, Gold coast, Hobart, Karratha, Launceston, Mackay, Melbourne, Newcastle, Perth, Proserpine, Rockhampton, Sunshine coast, Sydney, and Townsville.

We used the *google maps* data to calculate the flight time for each ordered pair of airports while considering only direct flights. In addition, we collected the information about the *total* number of corresponding daily flights. Next, we benchmark the two-step methodology on a timetable that was constructed as follows. First, we note that there are five main domestic airline carriers: *Qantas*, *Virgin Australia*, *Jetstar*, *Tiger Airways*, and *Regional Express Airlines*, and that our aim is to build a good schedule for a specific company (say for *Qantas*). As a consequence, we set the number of daily flights between two airports to be equal to the value of a ceiling function applied on the *total* number of daily flights divided by five. Next, for each pair of airports, the flights departure times were generated uniformly at random from the 6:00 AM to 10:00 PM time interval. The resulting timetable has 290 passages and 20 terminals.

1. *Step 1:* By executing a couple of SE iterations applied on this problem, we arrive to the conclusion that the problem is very hard. In particular, the

SE counting estimators shows that the order of magnitude of the number of optimal schedules is 10^{120}. During the execution of the SE algorithm, we also learn that a minimum chain decomposition has 88 chains, that is, we need a minimum number of 88 aircrafts to service the Australian domestic flight timetable.

2. *Step 2:* Clearly, the full-enumeration procedure is infeasible in this case and, therefore, we resort to the heuristic approach. We run the *TTOpt* heuristic for $N = 10,000,000$ iterations. The corresponding runtime for the completion of the *TTOpt* heuristic is about 32 minutes. Fig. 11 depicts a typical performance of the heuristic algorithm.

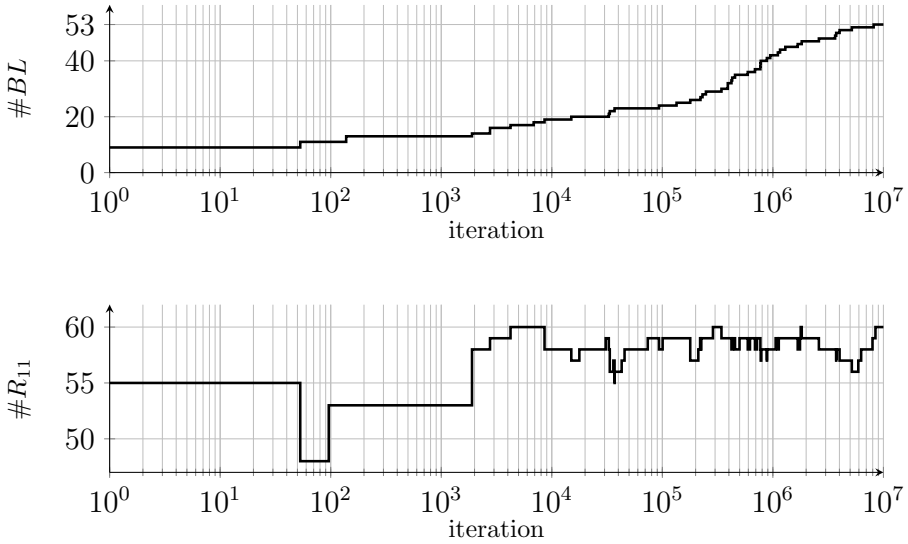

Fig. 11: Convergence of the *TTOpt* heuristic algorithm for the Australian domestic airline instance. The upper and the lower figures show the number of balanced chains ($\#BL$), and the number of chains with rest time which is greater than or equal to eleven hours ($\#R_{11}$), as a function of the algorithm's iteration number, respectively.

We tried to execute the heuristic using $N = 10^8$ iterations. Table 2 is instructive in the sense that it shows the dynamics of the improvement one can expect to get when increasing the number of iterations. The algorithm continues to improve the objective as the number of iterations grows. However, in order to execute $N = 10^8$ iterations, we need more than five hours of computation time. The latter requires a careful consideration, especially if the rescheduling task needs to be performed regularly. In this case, a future work of developing faster algorithms is required.

Table 2: The obtained results with the *TTOpt* heuristic for the Australian domestic airlines timetable when using different number of iterations N.

N	$\#BL$	$\#R_{11}$	Time (sec)
10^0	9	55	0.0024
10^1	9	55	0.0042
10^2	11	53	0.0241
10^3	13	53	0.1682
10^4	19	58	1.8523
10^5	24	58	19.762
10^6	42	58	203.98
10^7	53	60	1965.1
10^8	65	59	18809

5 Conclusions and future research

In this manuscript we proposed a two-step framework for obtaining optimal balanced schedules. Specifically, in the first step, we developed a randomized counting algorithm that delivers an estimator for the number of optimal schedules, that are not necessarily optimally balanced. We showed that this procedure can be utilized for determining the problem's hardness and for finding the set of optimal solutions to the optimal balanced scheduling problem, when the number of optimal schedules is not very large. For hard problems, namely, for problems with a large number of optimal schedules, we proposed a simple heuristic approach, which is very efficient in the sense that it can handle hard problem instances within a reasonable computational time. We studied the performance of this heuristic procedure, and found that it was effective when applied to the Australian domestic flights timetable, which contains several hundreds of passages and twenty terminals. Finally, we developed a freely available software package that implements the two-step procedure that was introduced in this paper.

For the future research, we believe that the following directions are of interest.

1. While the proposed heuristic is both fast and shows a reasonable performance on the problem instances examined in the numerical section of this manuscript, it is important to consider alternative approaches. For example, *evolutionary algorithms* can potentially outperform this heuristic method. However, to handle bigger instances, such as the Australian domestic flight timetable scheduling problem from Section 4, we presume that it will be necessary to provide a parallel implementation of such methods.
2. Of great interest is to consider an extension of the optimal balanced scheduling problem, in which the timetable is not fixed in advance. Namely, we should investigate a scenario where an operator can modify flight departure times. In this case, an attractive research direction is to utilize the stochastic enumeration method with a view to fluctuate flights in order to *minimize* the total number of optimal schedules. The main idea is to min-

imize this number until it reaches some relatively small threshold value. Then, we can obtain a globally optimal OBS solution via the full enumeration procedure.
3. It will be important to extend the optimal balanced scheduling problem to a stochastic setting. Specifically, one can consider a quite realistic assumption, under which a passage has both random departure and random arrival times. This stochastic behavior is usually caused by circumstances that are beyond our control. In this case, a considerable number of attractive research questions arise. For example, we might be interested to construct the most reliable balanced schedule. That is, to construct an optimal schedule that has the maximum possible number of balanced chains with high probability.
4. Finally, while we provide a single-threaded implementation of the stochastic enumeration algorithm package, from the practical point of view, it would be of interest to develop a parallel software implementation that is capable of running on multiple central processing units or a graphics processing unit. Such parallel software package will allow practitioners to handle large real-life timetable instances.

Acknowledgments

We are thoroughly grateful to the anonymous reviewers for their valuable and constructive remarks and suggestions. This paper is dedicated to the memory of Professor Ilya B. Gertsbakh (1933–2020). This work was supported by the Australian Research Council Centre of Excellence for Mathematical & Statistical Frontiers, under CE140100049 grant number.

Disclosure statement

We confirm that we are not aware of any conflict of interest for this study.

Data availability

Software packages and the research data is available on the author's website under:
`https://people.smp.uq.edu.au/RadislavVaisman/sw/scheduling/software.zip`

References

1. Ahmed, M.B., Mansour, F.Z., Haouari, M.: Robust integrated maintenance aircraft routing and crew pairing. Journal of Air Transport Management **73**, 15–31 (2018)
2. Arnold, F., Gendreau, M., Sörensen, K.: Efficiently solving very large-scale routing problems. Computers and Operations Research **107**, 32–42 (2019)

3. Asmussen, S., Glynn, P.: Stochastic Simulation: Algorithms and Analysis, *Stochastic Modelling and Applied Probability*, vol. 57. Springer, New York, NY (2007)
4. Bertelé, U., Brioschi, F.: Contribution to nonserial dynamic programming. Journal of Mathematical Analysis and Applications **28**(2), 313–325 (1969)
5. Broder, A.: How hard is it to marry at random? (on the approximation of the permanent). In: Proceedings of the eighteenth annual ACM symposium on theory of computing, STOC '86, pp. 50–58. ACM (1986)
6. Ceder, A., Stern, H.I.: Deficit function bus scheduling with deadheading trip insertions for fleet size reduction. Transportation Science **15**(4), 338–363 (1981)
7. Cortes, J.D., Suzuki, Y.: Vehicle routing with shipment consolidation. International Journal of Production Economics **227**, 107–120 (2020)
8. Curticapean, R., Dell, H., Fomin, F., Goldberg, L.A., Lapinskas, J.: A fixed-parameter perspective on #BIS. Algorithmica **81**, 3844–3864 (2019)
9. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics **51**, 161–166 (1950)
10. Dyer, M., Goldberg, L.A., Greenhill, C., Jerrum, M.: The relative complexity of approximate counting problems. Algorithmica **38**, 471–500 (2004)
11. Fulkerson, D.R.: Note on Dilworth's decomposition theorem for partially ordered sets. Proceedings of the American Mathematical Society **7**, 701–702 (1956)
12. Goldreich, O.: Computational complexity: a conceptual perspective. ACM SIGACT News **39**(3), 35–39 (2008)
13. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. **2**(4), 225–231 (1973)
14. Knuth, D.E.: Estimating the efficiency of backtrack programs. Mathematics of Computation **29**(129), 122–136 (1975)
15. Lan, S., Clarke, J.P., Barnhart, C.: Planning for robust airline operations: Optimizing aircraft routings and flight departure times to minimize passenger disruptions. Transportation Science **40**(1), 15–28 (2006)
16. Lei, Z., Zhe, L., Chun-An, C., Wanpracha Art, C.: Airline planning and scheduling: Models and solution methodologies. Front. Eng. Manag. **7**, 1–26 (2020)
17. Liu, T., Ceder, A.: Deficit function related to public transport: 50 year retrospective, new developments, and prospects. Transportation Research Part B: Methodological **100**, 1–19 (2017)
18. Marla, L., Vaze, V., Barnhart, C.: Robust optimization: Lessons learned from aircraft routing. Computers and Operations Research **98**, 165–184 (2018)
19. Nasibov, E., Eliiyi, U., Ertac, M.O., Kuvvetli, U.: Deadhead trip minimization in city bus transportation: A real life application. Promet - Traffic & Transportation **25**(2), 137–145 (2013)
20. Provan, J.S., Ball, M.O.: The complexity of counting cuts and of computing the probability that a graph is connected. SIAM J. Comput. **12**, 777–788 (1983)
21. Rubinstein, R.Y., Ridder, A., Vaisman, R.: Fast Sequential Monte Carlo Methods for Counting and Optimization. John Wiley & Sons, New York (2013)
22. Simovici, D.A., Djeraba, C.: Mathematical Tools for Data Mining: Set Theory, Partial Orders, Combinatorics. Advanced Information and Knowledge Processing. Springer London, London (2014)
23. Sinclair, A.J.: Randomised algorithms for counting and generating combinatorial structures (1988)
24. Stern, H.I., Gertsbakh, I.B.: Using deficit functions for aircraft fleet routing. Operations Research Perspectives **6**, 100–104 (2019)
25. Stojković, G., Soumis, F., Desrosiers, J., Solomon, M.M.: An optimization model for a real-time flight scheduling problem. Transportation Research Part A: Policy and Practice **36**(9), 779–788 (2002)
26. Vaisman, R., Kroese, D.P.: Stochastic enumeration method for counting trees. Methodology and Computing in Applied Probability **19**(1), 31–73 (2017)
27. Vaisman, R., Kroese, D.P., Gertsbakh, I.B.: Improved sampling plans for combinatorial invariants of coherent systems. IEEE transactions on reliability **65**(1), 410–424 (2016)
28. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
29. Vazirani, V.V.: Approximation algorithms. Springer, Berlin, Germany (2003)

6 Appendices

A The F30 timetable from [24]

Table 3: The F30 flight timetable with 30 passages. For each passage, the table displays the flight number, the departure and the arrival terminals, and the departure and the arrival times. Here, a fraction such as 4.5 in the first row, stands for the 4:30AM time.

flight number	departure terminal	arrival terminal	departure time	arrival time
1	2	4	4.5	12
2	4	2	14.5	20.5
3	4	1	8.5	15.5
4	2	3	16.5	21
5	4	2	9	16
6	2	3	11.5	13
7	2	1	6	10.5
8	3	4	11	16.5
9	2	3	18	21.5
10	3	2	7	11
11	3	2	13.5	17
12	2	3	18	23
13	3	4	10.5	15.5
14	2	4	7	13.5
15	4	2	13	20
16	2	1	5.5	8.5
17	1	2	9	12.5
18	1	2	15	17.5
19	2	1	16	19.5
20	1	2	18.5	22
21	1	2	8	11
22	2	1	10	13
23	1	2	11	15
24	2	3	18.5	22.5
25	3	2	5.5	8.5
26	1	2	9	13.5
27	2	3	14	17
28	3	2	17.5	21.5
29	4	1	3	7
30	2	4	12	15