A near-linear time algorithm and a min-cost flow approach for determining the optimal landing times of a fixed sequence of planes

Bin Cao[*] Chao Xu[†]

Abstract

The aircraft landing problem (ALP) is an important issue of assigning an airport's runways to the arrival aircrafts as well as to schedule the landing time of these aircrafts in practice. A large number of the extant studies have tried to address such a practical problem with using various algorithms for one or more runways. For a static single-runway of the ALP, this paper proposes a new approach to develop an alternative powerful algorithm. For a given sequence of planes, we develop a faster algorithm for solving the ALP with the running time $O(n \log n)$, where n is the number of aircrafts in the schedule. Alternatively, we reduce the proposed problem of minimizing the total cost by determining the landing times for a given landing sequence into a min-cost flow problem. We conduct a set of experimental studies to compare the performance of our near-linear time algorithm to the quadratic time algorithm whose time complexity is $O(n^2)$, for computing the optimal landing times. The computational results show that the proposed heuristic based on our algorithm could be much faster than both such quadratic time algorithm and the one using linear programming.

Keywords: aircraft landing problem; static single-runway system; scheduling; min-cost flow; polynomial-time algorithm

1 Introduction

The aircraft landing problem (ALP) is an important topic that seeks to determine the sequence of aircraft landing on or taking off from the available runways at an airport, in order to minimize given objectives (costs) under some operational constraints (Bennell et al., 2013). Specifically, this problem can be addressed on one or more runways. In general, on each runway, the landing times of each aircraft should lie within target time windows, and the safety separation distances between two successive landings must be respected (Atkin et al., 2019; Faye, 2015; Ikli et al., 2021). In addition, if a plane lands before or after the target landing time, it will incur a cost, and the objective is to schedule the planes to land as close to their preferred landing time as possible, that is, minimizing the total cost of deviation from the target times. For solving the ALP, so far there is a large number of studies using different (heuristic) algorithms. As an initial work on the ALP, Beasley et al. (2000) presented a mixed-integer zero-one formulation of the problem for the single runway or multiple runway case, and then adopted the linear programming-based tree search approaches to schedule landing times. Moreover, they pointed out that the ALP is an NP-hard problem.

To overcome this challenge, the majority of the relevant papers had to use some heuristic algorithms to solve large instances. For example, Ernst et al. (1999) used a space search heuristic as well as a branch-and-bound method to solve both single- and multiple-runway problems. Fahle et al. (2003) compared four new exact and heuristic solution methods in terms of quality, speed, and flexibility. Hansen (2004) used genetic search algorithms to solve certain complexities associated with air traffic control. Pinol and Beasley (2006) solved the multiple runway case of the static ALP by scatter search and bionomic algorithms. Bianco et al. (2006) provided a fast dynamic

[*]School of Management, Jinan University, Guangzhou 510632, China. bincscut@gmail.com

[†]School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China 610054. thechaoxu@gmail.com

local search heuristic algorithm for air traffic control via a deterministic job-shop scheduling model. Soomer and Franx (2008) adopted a local search heuristic to obtain reasonable solutions for the single runway arrival problem of determining an arrival schedule. D'Ariano et al. (2012) addressed take-off and landing operations in a TMA (Terminal Maneuvering Area) and solved the proposed problem via a truncated branch and bound algorithm for fixed routes. Salehipour et al. (2013) designed a hybrid meta-heuristic algorithm to solve a mixed integer goal programming model for the ALP. Hancerliogullari et al. (2013) applied greedy heuristics and meta heuristics to obtain solutions in reasonable computation times for the Aircraft Sequencing Problem (ASP) over multiple runways. Based on an approximation of the separation time matrix to a rank two matrix, Faye (2015) used the formulation in Beasley et al. (2000) and presented a dynamic constraint generation algorithm to solve ALP with a time discretization approach. Moreover, Sama et al. (2019) considered a particularly challenging problem of efficiently scheduling take-off and landing operations at a busy terminal maneuvering area.

On the other hand, only a few studies have considered the dynamic case of ALP. Note that in this situation, decisions must be dynamically determined as time passes, and each new decision must consider the previous decision that was made (see, Beasley et al. (2004)). Within the literature, Lieder and Stolletz (2016) solved the aircraft scheduling problem with general runway configurations by using a dynamic programming approach. Ghoniem et al. (2015) developed an effective branch-and-price algorithm to address multiple-runway aircraft sequencing problems. Ji et al. (2017) proposed a dynamic sequence searching and evaluation, and an estimation of distribution algorithm and a heuristic search method to find the optimal landing sequence of the aircrafts. Using the real data of the Beijing Capital International Airport, they also compared the proposed method to other related algorithms. For the dynamic scheduling of aircraft landings, Bennell et al. (2017) developed an algorithm that periodically updates the previous schedule to account for newly available aircraft. Bennell et al. (2017) used a multi-objective approach to model the static/off-line aircraft landing problem and the dynamic/on-line version of the problem, and conducted dynamic programming, iterated descent, and simulated annealing algorithms to solve the problems. Other studies include such as Atkin et al. (2019), Ng et al. (2017) and Ghoniem and Farhadi (2015). For a more comprehensive related review, we refer the reader to Ikli et al. (2021) and the references therein.

In contrast to the previous studies, Lieder et al. (2015) used a dynamic programming approach and provided an exact algorithm to solve the ALP with multiple runways, positive target landing times, and time windows. In particular, Faye (2018) proposed a new approach by using a dynamic programming approach to effectively solve the static case of the ALP in a single runway. Note that the dynamic programming used to solve the ALP is commonly found in other literature, e.g., Psaraftis (1978), Psaraftis (1980), Brentnall (2006), and Balakrishnan and Chandran (2010). Importantly, according to this method, Faye developed a quadratic time algorithm to compute the optimal landing times for a given sequence of planes; that is, the time complexity of this algorithm is $O(n^2)$ where n is the number of planes of the sequence. In addition, Faye (2018) compared the performance of this quadratic time algorithm to the linear programming method studied in the literature (Beasley et al., 2000). Then, based on simulated annealing, the two methods are embedded into an iterative algorithm to seek a sequence of minimum cost. The experimental results showed that the dynamic programming algorithm generates significant improvements in CPU time compared with linear programming, which allows obtaining better solutions in a fixed amount of time.

Therefore, motivated by the work of Faye (2018) and Lieder et al. (2015), the purpose of our paper is to improve the running time of Faye's algorithm to solve the ALP more effectively. To do this, we provide generalizing the algorithm in Rote (2018), we develop an alternative approach and a faster algorithm for solving the ALP for a fixed sequence of planes, i.e., minimizing the total cost via determining the landing times of a fixed sequence of planes (denoted by FALP). The running time of the proposed algorithm we prove is $O(n \log n)$, where n is the number of planes. For given sequence of planes, our experiments show that the running time of our algorithm for FALP is much faster. Alternatively, we reduce FALP into a min-cost flow problem. In this regard, we can state that FALP is a special case of min-cost flow problem on series-parallel graphs. In Appendix A, we embed our proposed algorithm and the quadratic time algorithm whose time complexity is $O(n^2)$ studied in the literature into an iterative algorithm to heuristically finding a sequence of minimum cost, and numerically compare the performance of these algorithms. In this regard, the heuristics based on our algorithm could be much faster than both such quadratic time algorithm and the one using linear programming.

The paper's main contribution is that we develop a faster $O(n \log n)$ time algorithm for solving the aircraft landing problem for a given sequence of planes considered in the literature, e.g., Faye (2018). For practical purposes, we find a faster implementation of the dynamic programming algorithm in Faye (2018) by generalizing the result of Rote (2018) and avoiding complicated data structures. For theoretical elegance, we also show that

FALP is a special case of the min-cost flow problem on series-parallel graphs.

The rest of the paper is organized as follows. Section 2 briefly goes over the model on the ALP in a static single-runway case. In Section 3, we design a faster algorithm for solving FALP and numerically compare this algorithm with that developed by Faye (2018) for the fixed sequence of planes by running the same experiment as the one in the literature. Section 4 reduces the ALP for a fixed sequence of planes in the schedule into a min-cost flow problem. Finally, Section 5 concludes the paper.

2 Preliminaries

In this paper, we consider the aircraft landing problem (ALP) under the static case in a single runway, which is adopted only for landing. This situation is commonly studied in the literature, e.g., Beasley et al. (2000) and Faye (2018). In particular, there is a sequence of n planes that will land on a runway in order. For ease of exposition, let $[n]$ be the set of positive integers up to n, that is, $[n] = \{1, \dots, n\}$, and x_i be the scheduled landing time for plane $i \in [n]$. For a fixed $i \in [n]$, the ith aircraft has a landing time that must be between an earliest landing time E_i and a latest landing time L_i for the aircraft; this is referred to as the *landing time constraints*. Formally, it is

$$E_i \le x_i \le L_i, \quad \forall i \in [n].$$

Here the earliest landing time represents the time at which the aircraft can land by flying at its fastest speed. Note that delaying the landing time can be done by such as decreasing the speed of the aircraft or the flight plan can be lengthened by circling. The latest landing time is referred to as the maximum landing time achievable because of these delaying mechanisms.

Let P be the set of all permutations of $[n]$. Moreover, $\pi \in P$ (i.e., a permutation π) can be seen as sequence of planes, and let $\pi(i)$ denote plane i's position in the landing sequence. Associated with each aircraft is a weight class that determines the minimum separation times between successive landings. Denote $S_{i,j}$ to be the minimum landing separation time between planes i and j. To ensure the minimum separation time between a plane and all its successors, it requires that for each $\pi(i) < \pi(j)$,

$$x_j \ge x_i + S_{i,j}.$$

This is referred to as the *separation time constraints*. In addition, within the time interval $[E_i, L_i]$, there is a target time T_i corresponding to the preferred landing time. There are costs associated with landing either earlier or later than this target landing time. Specifically, for each i, let b_i and a_i be the non-negative penalty costs per unit of time for landing before and after the target time T_i for plane i, respectively. Then, the cost of plane i landing at time x can be encoded as a function $c_i : [E_i, L_i] \to \mathbb{R}_{\ge 0}$, defined as

$$c_i(x) = \begin{cases} b_i(T_i - x) & \text{if } x \le T_i \\ a_i(x - T_i) & \text{if } x > T_i. \end{cases} \tag{1}$$

For convenience, we summarize the notation adopted in the model description in Table 1. In line with the literature

Table 1. Notations

n	the number of aircrafts in the schedule
x_i	the scheduled landing time for plane $i \in [n]$
E_i	the earliest landing time for plane $i \in [n]$
L_i	the latest landing time for plane $i \in [n]$
T_i	the target (preferred) landing time for plane $i \in [n]$
$S_{i,j}$	the minimum separation time between planes i and j where i lands before j on the same runway
a_i	the penalty cost per unit of time for landing after the target time T_i for plane i
b_i	the penalty cost per unit of time for landing before the target time T_i for plane i
P	the set of all permutations of $[n]$
$\pi(i)$	the position of plane i in the landing sequence, $\pi \in P$
$c_i(x)$	the cost for plane $i \in [n]$ landing at time x

(Beasley et al., 2000, 2001; Bennell et al., 2013; Faye, 2015, 2018; Pinol & Beasley, 2006), we consider the ALP as follows:

$$\begin{aligned} \min_{\pi \in P} \min_{x} \quad & \sum_{i \in [n]} c_i(x_i) \\ \text{s.t.} \quad & x_j \geq x_i + S_{i,j}, \quad i,j \in [n], \pi(j) > \pi(i) \\ & E_i \leq x_i \leq L_i, \qquad\qquad \forall i \in [n] \end{aligned}$$

Throughout this paper, we mainly focus on solving the ALP for a fixed sequence permutation π, denoted by FALP, which is mainly considered in Faye (2018). That is,

$$\begin{aligned} \min_{x} \quad & \sum_{i \in [n]} c_i(x_i) \\ \text{s.t.} \quad & x_j \geq x_i + S_{i,j}, \quad i,j \in [n], \pi(j) > \pi(i) \\ & E_i \leq x_i \leq L_i, \qquad\qquad \forall i \in [n] \end{aligned}$$

In particular, we can renumber the planes so $\pi(i) = i$. Similarly with the literature above, we also assume that the separation times satisfy the Triangle Inequality. That is, $S_{i,j} + S_{j,k} \geq S_{i,k}$ for all $i,j,k \in [n]$. Thus, we only have to check the separation between consecutive planes. Hence, the separation time constraints above can be simplified as follows:

$$x_{i+1} \geq x_i + S_{i,i+1}, \text{ for all } 1 \leq i \leq n-1.$$

For brevity, let $S_i = S_{i,i+1}$ in the remainder of the paper. Finally, the above problem can be simplified into the following optimization problem:

$$\begin{aligned} \min_{x} \quad & \sum_{i \in [n]} c_i(x_i) \\ \text{s.t.} \quad & x_{i+1} \geq x_i + S_i, \quad i \in [n-1] \\ & E_i \leq x_i \leq L_i, \qquad \forall i \in [n]. \end{aligned}$$

For ease of exposition, we refer to the above problem as the fixed-sequence ALP (FALP).

Note we make the assumption that the problem is feasible as it will simplify the implementation. In the next section, we will develop an alternative approach for solving FALP.

3 A near-linear time algorithm for FALP

We show how to implement the computation of Faye (2018) in $O(n \log n)$ time and $O(n)$ space by modifying the algorithm in Rote (2018). We use a simple data structure, a double-ended priority queue, which exists in both Java and C++, and various other platforms. In particular, `TreeMap` in Java is a known implementation of double-ended priority queue. Recall that in Faye (2018), the following sequence of functions was defined.

$$\begin{aligned} & f_1 : [E_1, L_1] \to \mathbb{R}, \text{ where } f_1(x) = c_1(x) \\ & g_i : [E_{i-1} + S_{i-1}, \infty) \to \mathbb{R}, \text{ where } g_i(x) = \min\{f_{i-1}(y) | x - y \geq S_{i-1}\} \quad \text{for all } 2 \leq i \leq n \\ & f_i : [E_i, L_i] \to \mathbb{R}, \text{ where } f_i(x) = c_i(x) + g_i(x) \quad \text{for all } 2 \leq i \leq n \end{aligned} \tag{2}$$

Note that we change the index and domain of g_i compared to Faye (2018). It does not change the computation, but only allows technically easier implementation. Intuitively, $f_i(x_i)$ is the minimum total cost for the sequence of planes from 1 to i when plane i lands at time x_i. The goal is to compute $\min_{x \in [E_n, L_n]} f_n(x)$. Faye represents each function in the above list as a list of breakpoints and additional slope information, computes the representation of each function f_i and g_i, and takes $O(n^2)$ time in total.

Faye's algorithm does look like the best possible: the total number of breakpoints in all these functions together is $\Omega(n^2)$. Hence, any algorithm that has to listing all the intermediate functions by their breakpoints would already take $\Omega(n^2)$ time. The crucial observation for the speed up is that the amount of changes between consecutive functions in the sequence is small. The goal is to modify the previous function to obtain the new function without touching most of the breakpoints.

Our algorithm is inspired by Rote (2018)'s algorithm that takes care of a special case of the FALP, the *isotonic regression problem*. A instance of the FALP is called an isotonic regression problem if for each i, $S_i = 0$, $a_i = b_i$, $L_i = \infty$, and $E_i = -\infty$. One can dive into the detail of Rote's algorithm, and realize that it also works for the case where a_i and b_i are not equal. However, it cannot solve the most general FALP. The difficulties are twofold: taking care of the case when S_i is non-zero, which requires *shifting*; and, L_i and E_i can take on finite values, which requires *restriction* on the function. Note that for computational purposes, when we write ∞, we can replace it with a large value M instead.

Let $\langle f \rangle$ be the number of breakpoints in f. Here, the breakpoints include the boundary of the domain. We need to represent a piecewise-linear convex function that allows a certain set of operations in a particular running time. When we say "compute" a function, we mean compute some representations of the function, which will be specified later.

3.1 The slope difference form

To start, we describe how piecewise-linear convex functions are represented. The specific form we call the *slope difference form*. This is the same abstract representation described in Rote (2018). Intuitively, we store the function by the *breakpoints* and the *difference between consecutive slopes* around the breakpoint. Rote recognized adding two functions in the slope difference form is very fast: For f and g (of the same domain), the slope difference form of $(f+g)$ differs from the slope difference form of f in only $\langle g \rangle$ positions. So instead of spending $(\langle f \rangle + \langle g \rangle)$ time, we spend $\langle g \rangle$ time, which is negligible if $\langle g \rangle$ is much smaller than $\langle f \rangle$. Finding min transform and the minimum was also handled by Rote's data structure. However, restriction and shifting are not handled. The goal is to make sure that in any of the operations, the algorithm should spend $O(\log n)$ time per *change*. Here, a change is either adding or removing a breakpoint. So we will progressively wrap the data structure to eventually handle all these operations.

Formally, let $s_{x,\Delta}$ be the function such that $s_{x,\Delta}(y) = \max(0, \Delta(y-x))$. We call this a *simple function*. Simple functions are certainly piecewise-linear convex functions with constant number of breakpoints. Assume that we have a piecewise-linear convex function $f : [a,b] \to \mathbb{R}$. Let $a = x_1, \ldots, x_n = b$ be the sequence of increasing breakpoints of the function f, then for $x \in [a,b]$, we have $f(x) = z + \sum_{i=1}^{n-1} s_{x_i,\Delta_i}(x)$ for some Δ_i. Observe that Δ_i is precisely the difference in the slope around breakpoint x_i.

In order to represent f, we just need z and the set of the associated simple functions. A simple function can be represented by its unique non-infinity breakpoint. In particular, we will let the representation of f consist of:

- f.breakpoints, a data structure storing $(x_1, \Delta_1), \ldots, (x_n, \Delta_n)$. Here $\Delta_n = 0$.
- f.startValue, storing z.
- f.lastSlope, storing $\sum_{i=1}^{n-1} \Delta_i$.

We remark that f.lastSlope can be derived from f.breakpoints and f.startValue. It is there only to speedup computations. For simplicity, we write f that is represented by $(z; (x_1, \Delta_1), \ldots, (x_n, \Delta_n))$.

3.2 Basic operations on slope difference form

We define the five basic methods on slope difference form as follows.

Definition The five basic methods given a slope difference form of a piecewise-linear convex function f with breakpoints $x_1, \ldots, x_n$ are:

1. f.REMOVELEFTMOSTBREAKPOINT(): restrict the function f to $[x_2, x_n]$.
2. f.REMOVERIGHTMOSTBREAKPOINT(): restrict the function f to $[x_1, x_{n-1}]$.
3. f.INSERTBREAKPOINT(x, Δ): update the function to $f + s_{x,\Delta}$.
4. f.UB(): returns the right most breakpoint.
5. f.LB(): returns the left most breakpoint.

REMOVELEFTMOSTBREAKPOINT and REMOVERIGHTMOSTBREAKPOINT are *breakpoint removals*, and INSERTBREAKPOINT is *breakpoint insertion*. Together, these 3 operations are called *breakpoint updates*. Breakpoint removal always decreases the number of breakpoints. Breakpoint insertion increases the number of breakpoints if the insertion point does not already exist as a breakpoint.

```
$f$.UB():
    return (key of $f$.breakpoints.getMax())
$f$.LB():
    return (key of $f$.breakpoints.getMin())
$f$.INSERTBREAKPOINT($x, \Delta$):
    $f$.breakpoints.update($x, \Delta$)
    $f$.lastSlope $\leftarrow f$.lastSlope $+\Delta$
```

```
$f$.REMOVELEFTMOSTBREAKPOINT():
    $(x_1, \Delta_1) \leftarrow f$.breakpoints.removeMin()
    $f$.lastSlope $\leftarrow f$.lastSlope $-\Delta_1$
    $(x_2, \Delta_2) \leftarrow f$.breakpoints.getMin()
    $f$.InsertBreakpoint($x_2, \Delta_1$)
    $f$.startValue $\leftarrow f$.startValue $+ \Delta_1 \cdot (x_2 - x_1)$
$f$.REMOVERIGHTMOSTBREAKPOINT():
    $f$.breakpoints.removeMax()
    $(x_{n-1}, \Delta_{n-1}) \leftarrow f$.breakpoints.getMax()
    $f$.lastSlope $\leftarrow f$.lastSlope $-\Delta_{n-1}$
```

Figure 3.1. The implementation of the five basic methods.

We show how to implement the five basic methods in the desired running time using a known data structure – a double-ended priority queue (Sahni, 2004).

A double-ended priority queue stores a set of elements S. For $(x, v) \in S$, x is called the *key* and v is called the *value*. The pairs are ordered by the first element. That is, $(x, v) \le (y, u)$ if $x \le y$. All the pairs have distinct keys. Let S be the set we are maintaining, and it has n elements, then each operation below takes $O(\log |S|)$ time.

- update(x, v): if there exist some $(x, v') \in S$, remove (x, v') from S and add $(x, v' + v)$ into S. Otherwise, add (x, v) into S.
- getMin(): return $\min(S)$.
- getMax(): return $\max(S)$.
- removeMin(): return $\min(S)$ and remove it from S.
- removeMax(): return $\max(S)$ and remove it from S.
- size(): return $|S|$.

Theorem 3.1. *If f has n breakpoints, then the five basic methods can be implemented in $O(\log n)$ time.*

Proof: We will store our sequence $(x_1, \Delta_1), \ldots, (x_n, \Delta_n)$ by a double-ended priority queue. Namely, the key is the breakpoint, and the value is the difference in slopes. The implementations are show in Figure 3.1. The operations are not only direct translation of the operation on the sequence to double-ended priority queue, but also maintaining the last slope as the aggregate of $\sum_{i=1}^{n} \Delta_i$.

Other than the constant time operations, each of the method queries the double-ended priority queue a constant number of times. Hence, each method takes $O(\log n)$ time. □

3.3 Operations on slope difference form allow restrictions

Now, we describe the data structure for piecewise-linear convex functions, so it supports restrictions. The running time of the operations is always bounded by $O(\ell \log n)$, where ℓ is the number of breakpoint updates, which can be expressed by the difference of the input and output functions.

Theorem 3.2. *There is a data structure that stores a piecewise-linear convex function, such that the following operations are supported.*

1. f.RESTRICT(x, y) *returns the representation of* $h = f|_{[x,y]}$ *in* $O((\langle f \rangle - \langle h \rangle) \log n)$ *time.*
2. f.ADD(g) *returns the representation of* $h = f + g$, *which takes* $O((\langle g \rangle + (\langle f \rangle + \langle g \rangle - \langle h \rangle)) \log n)$ *time.*
3. f.MINTRANSFORM() *returns* h *such that* $h(x) = \min_{y \le x} f(y)$ *in* $O((\langle f \rangle - \langle h \rangle) \log n)$ *time.*
4. f.MIN() *returns* $\min_x f(x)$ *in* $O(\langle f \rangle \log n)$ *time.*

```
f.Restrict(a, b):
    f.InsertBreakpoint(a, 0)
    f.InsertBreakpoint(b, 0)
    while f.lb() < a:
        f.RemoveLeftMostBreakpoint()
    while f.ub() > b:
        f.RemoveRightMostBreakpoint()
```

Figure 3.2. The implementation restriction.

```
f.Add(g):
    lb ← max(f.lb(), g.lb())
    ub ← min(f.ub(), g.ub())
    f.Restrict(lb,ub)
    g.Restrict(lb,ub)
    for each $(x, \Delta)$ in g.breakpoints
        f.InsertBreakpoint($x, \Delta$)
    f.startValue ← f.startValue + g.startValue
```

Figure 3.3. The implementation of addition.

```
f.MinTransform():
    while f.lastSlope > 0 and f.breakpoints.size() > 1
        f.RemoveRightMostBreakpoint()
    f.InsertBreakpoint(f.ub(), -f.lastSlope)
    f.InsertBreakpoint($\infty$,0)
```

Figure 3.4. The implementation of MinTransform

```
f.Min():
    $S \leftarrow \{f$.startValue$\}$
    while f.breakpoints.size() > 1
        f.RemoveLeftMostBreakpoint()
        $S \leftarrow S \cup \{f$.startValue$\}$
    return min($S$)
```

Figure 3.5. The implementation of Min.

```
$f$.ADD($g$):
    ⟨⟨Note in our algorithm $g$ has shift of 0⟩⟩
    ⟨⟨manually shift $g$ by shifting g.unshifted⟩⟩
    $p \leftarrow f$.shift
    $z' \leftarrow f$.unshifted.startvalue
    $(x_1, \Delta_1), \ldots, (x_n, \Delta_n) \leftarrow g$.unshifted.breakpoints
    $g' \leftarrow (z'; (x_1 - p, \Delta_1), \ldots, (x_n - p, \Delta_n))$
    $f$.unshifted.ADD($g'$)
$f$.SHIFTEDMINTRANSFORM($p$):
    $f$.unshifted.MINTRANSFORM()
    $f$.shift $\leftarrow f$.shift + $p$
$f$.MIN():
    return $f$.unshifted.MIN()
```

Figure 3.6. The implementation of the methods for shifted functions

Here n is the maximum number of breakpoints involved in any function through out the lifetime of the data structure.

Proof: Restriction. Let $h = f|_{[a,b]}$. To find h, if a and b are breakpoints of f, then we can apply REMOVELEFTMOSTBREAKPOINT() and REMOVERIGHTMOSTBREAKPOINT() sequentially until we obtain h. This would take $O((\langle f \rangle - \langle h \rangle) \log n)$ time. To make sure that a and b are breakpoints, we insert dummy breakpoints $(a, 0)$ and $(b, 0)$ into f's list of breakpoints $(x_1, \Delta_1), \ldots, (x_n, \Delta_n)$. This takes $O(\log n)$ time. Hence, the total running time is $O((\langle f \rangle - \langle g \rangle) \log n)$. See the pseudocode in Figure 3.2.

Addition. In order to add two functions, we need both functions to be in the same domain. Hence we can first apply restriction to ensure that both function are in the same domain. We can assume the inputs of the addition are functions f and g with representation $(z; (x_1, \Delta_1), \ldots, (x_n, \Delta_n))$ and $(z'; (x'_1, \Delta'_1), \ldots, (x'_k, \Delta'_k))$ where $x_1 = x'_1$ and $x_n = x'_k$.

We apply f.INSERTBREAKPOINT(x, Δ) repeatedly, where (x, Δ) is in g.breakpoint. Finally, we update the startValue to $z + z'$. See Figure 3.3. The running time spent outside RESTRICTION is therefore $O(\langle g \rangle \log n)$. The time spent in RESTRICTION is $O(\log n)$ per breakpoint removed, hence upper bounded by $O((\langle f \rangle + \langle g \rangle - \langle h \rangle) \log n)$ time.

Min Transform. Let h be a function such that $h(x) = \min_{y \le x} f(y)$. Min transform is equivalent to remove breakpoints from f from the right end one by one until the slope is no larger than 0. At that point, extend the function to infinity with a line of slope 0. The only exception is when f is an increasing function on $[a, b]$. One can verify in that case, the output should be the constant function with value $f(a)$ in $[a, \infty)$.

Hence, we have to call f.REMOVERIGHTMOSTBREAKPOINT() $(\langle f \rangle - \langle h \rangle)$ times. The running time is therefore bounded by $O((\langle f \rangle - \langle h \rangle) \log n)$. See the implementation in Figure 3.4.

Min. We obtain the minimum by looking at the value of the function at each breakpoint. We obtain this by applying REMOVELEFTMOSTBREAKPOINT() until there is nothing left of the function. Since we always store the value of the function at the first breakpoint, this will compute the value at all breakpoints. The running time is $O(\langle f \rangle \log n)$. See the implementation in Figure 3.5. □

3.4 Main Data Structure for Shifted Functions

Finally, we wrap around the data structure of the previous section so it supports shifted representation of the piecewise-linear functions.

Theorem 3.3. *There exists a data structure that stores piecewise-linear convex functions, such that for two such functions f and g, we have:*

- f.ADD(g): *For function g, returns representation of the function $h = f + g$, which takes $O((\langle g \rangle + (\langle f \rangle + \langle g \rangle - \langle h \rangle)) \log n)$ time.*

```
Compute($c_1, \ldots, c_n, S_1, \ldots, S_n$):
  $f_1 \leftarrow c_1$
  for $i$ from 2 to $n$:
      $g_i \leftarrow f_{i-1}.\textsc{ShiftedMinTransform}(S_{i-1})$
      $f_i \leftarrow g_i.\textsc{Add}(c_i)$
  return $f_n.\textsc{Min}()$
```

Figure 3.7. Pseudocode to find the optimal value.

- $f.\textsc{ShiftedMinTransform}(s)$: *Returns a function h such that $h(x) = \min_{x-y \geq s} f(y)$. Running time is $O((\langle f \rangle - \langle h \rangle) \log n)$.*
- $f.\textsc{Min}()$: *Returns $\min_{x \in \mathbb{R}} f(x)$. Running time is $O(\langle f \rangle \log n)$.*

Here n is the largest number of breakpoints of any function involved in the life time of the data structure.

Proof: We are going to store functions in *shifted representation*, which wraps around our data structure for the slope difference form in the previous section. Define Shift(g,p) to be the function $f : [a+p, b+p] \to \mathbb{R}$ such that $f(x) = g(x-p)$. Computing g from f in slope difference form would take $O(\langle f \rangle)$ time, as we have to read the entire function. Shifted slope difference form avoids this by storing the shift as an extra parameter, thus it allows one to obtain Shift(g,p) from g in $O(1)$ time.

In order to represent $f = \text{Shift}(g,p)$, we store:

- f.unshifted: the representation of g in slope difference form,
- and f.shift is p.

The data structure in Figure 3.7 is working with shifted representation of the functions. Therefore, we have to implement the 3 method called in that function.

Now we show the data structure has the running time described in our theorem.

- $f.\textsc{Add}(g)$ takes $O(\langle g \rangle \log n)$ time to obtain all the unshifted breakpoints; and then, we apply shift manually to these breakpoints, and create a unshifted function g', takes another $O(\langle g \rangle \log n)$ time. Finally, the algorithm calls Add on unshifted functions, which takes $O((\langle g \rangle + (\langle g \rangle + \langle f \rangle - \langle h \rangle)) \log n)$ time.
- The running time for $f.\textsc{ShiftedMinTransform}$ is the same as MinTransform, which is $O((\langle f \rangle - \langle g \rangle) \log n)$.
- The running time for $f.\textsc{Min}$ for shifted representation carries over from the running time of unshifted representation, which is $O(\langle f \rangle \log n)$.

□

3.5 Compute the optimum

Recall the functions we have to compute in Equation 2. The algorithm is a direct translation of the definitions. To compute f_i, we use $g_i.\textsc{Add}(c_i)$. To compute g_i, we use $f_{i-1}.\textsc{ShiftedMinTransform}(S_{i-1})$. Finally, we just have to call $f_n.\textsc{Min}()$ to return the optimal value. The algorithm is conceptually simple. See Figure 3.7 for the pseudocode.

Theorem 3.4. *The running time of the algorithm in Figure 3.7 is $O(n \log n)$.*

Proof: The algorithm takes the previous function, and we transform it to the next function. Observe that the algorithm spends $O(\log n)$ time per breakpoint updated or removed.

We start with function f_1 with 3 breakpoints. In the sequence of functions, the breakpoints can increase only through addition. The function c_i always has 3 breakpoints, and f_i created from g_i can have at most 3 more breakpoints than g_i. Hence, there can be at most $O(n)$ breakpoints in any of the functions. Also, this proves that the number of breakpoints inserted in the entire algorithm is $O(n)$. Hence, this shows that the number of breakpoints removed is also $O(n)$, as one can remove a breakpoint only if it is inserted at some point of the algorithm. Hence the total running time is $O(n \log n)$. □

3.6 Running time experiments

The algorithm was implemented in Java, and can be found freely distributed on Github (Xu, 2019). We consider the problem size from 1,000 to 15,000 planes. For each size, we run 200 times of the test cases of that size. Each test case consists of 100 generated randomized input. The implementation was run on a MacBook Pro M3 Max with 128G of memory. We plot the average time it takes to complete a test case of a particular size. The result can be seen in Table 2. Figure 3.8 gives a comparison of the running times between our proposed algorithm and the algorithm of Faye (2018), and Figure 3.9 shows the running time of our algorithm by itself, so one can observe the growth rate. Our experiments demonstrate that our algorithm is faster, and as the size of the input grow, the gap becomes much larger.

Table 2. Comparison of the average running time of 200 runs in terms of miliseconds between our proposed algorithm and Faye's algorithm.

Size	Faye	Ours
1000	0.314371625	0.110645041
2000	1.201333375	0.19880425
3000	3.066012834	0.289033584
4000	6.129292334	0.37863475
5000	10.14854733	0.540683041
6000	15.20594188	0.717499792
7000	21.29048888	0.86567025
8000	27.80487892	0.945232833
9000	35.51039579	1.116850458
10000	44.02129504	1.239354458
11000	53.35887433	1.373465
12000	63.93943188	1.489784875
13000	75.02833608	1.71857025
14000	88.92440863	1.882641042
15000	106.4652665	2.129256416

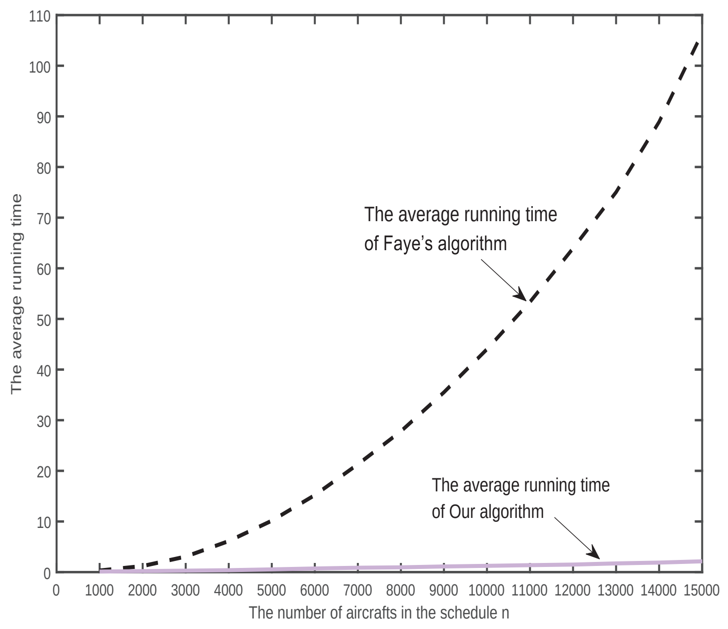

Figure 3.8. Comparison between our proposed algorithm and Faye's algorithm for solving FALP.

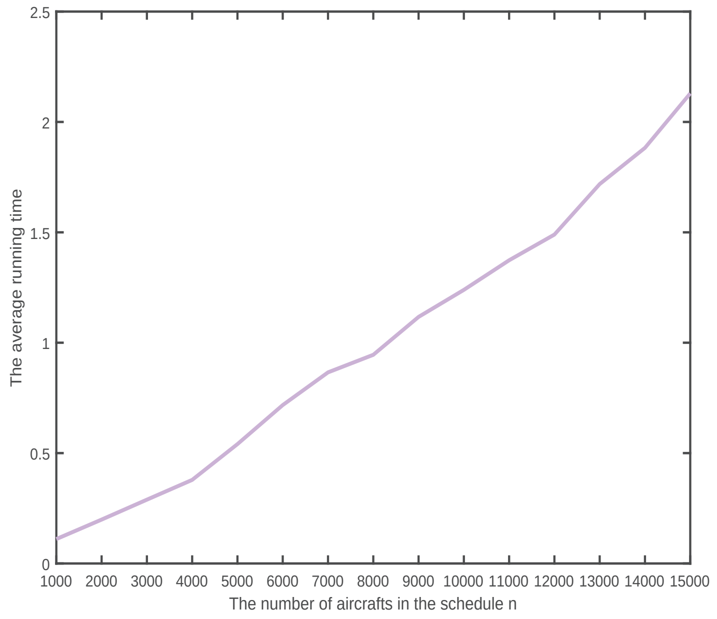

Figure 3.9. Impact of n on the average running time of our proposed algorithm.

4 FALP as a min-cost flow problem on a series-parallel graph

4.1 Preliminaries

To reduce the proposed FALP into a min-cost flow problem, we first introduce problem and describe some results on the classical series-parallel graphs in the following subsections.

4.2 Min-cost flow

In this subsection, we introduce the the min-cost flow problem using terminologies in Booth and Tarjan (1993). Specifically, a *network* is a tuple (G, ℓ, u, c), where $G = (V, E)$ is a graph and $\ell, u, c : \mathrm{E} \to \mathbb{R}$ are *real functions* over its edges. Here $\ell(e), u(e)$ and $c(e)$ are the *capacity lower bound*, *capacity upper bound* and *cost per unit of flow* on edge e, respectively. We emphasize that both capacities and cost are *allowed to be negative*.

Let $\delta^{in}(v)$ denote the set of incoming edges of the vertex v, and $\delta^{out}(v)$ denote the set of outgoing edges of the vertex v. For a function $f : E \to \mathbb{R}$, let $f^{in}(v) = \sum_{e \in \delta^{in}(v)} f(e)$ be the *incoming flow*, and $f^{out}(v) = \sum_{e \in \delta^{out}(v)} f(e)$ be the *outgoing flow*. For two vertices s and t, a function $f : E \to \mathbb{R}$ is a st*-flow*, if the following two conditions are satisfied:

- Capacity constraint: $\ell(e) \leq f(e) \leq u(e)$ for all $e \in E$.
- Flow conservation: $f^{in}(v) = f^{out}(v)$ for all $v \in V \setminus \{s, t\}$.

Then, the *cost of the flow* f is given as $\sum_{e \in E} c(e) f(e)$. In addition, we say that a st-flow is a *min-cost* st*-flow* if it is a st-flow of minimum cost. The *min-cost flow problem* takes a input network and vertices s and t, and output a min-cost st-flow. Currently, the fastest algorithm solving the min-cost flow problem has $m^{1+o(1)}$ running time on a m edge network (Brand et al., 2023).

4.3 Series-parallel graphs

Next we define two-terminal series-parallel graphs (s, t, G), where s and t are vertices of graph G. The vertex s is called the *source* and t is called the *sink*. The graph $G = (\{s, t\}, \{st\})$ is a two-terminal series-parallel graph. The remaining two-terminal series-parallel graphs are defined recursively. Let (s_1, t_1, G_1) and (s_2, t_2, G_2) be two-terminal series-parallel graphs, then (s, t, G) is a two-terminal series-parallel graph if it is obtained through either of the following two compositions.

- Series composition: G is obtained from G_1 and G_2 by identifying t_1 with s_2, and $s = s_1$, $t = t_2$.

- Parallel composition: G is obtained from G_1 and G_2 by identifying s_1 with s_2, t_1 with t_2, and $s = s_1$, $t = t_1$.

A graph G is a *series-parallel graph* if (s,t,G) is a two-terminal series-parallel graph for some vertices s and t in G (Eppstein, 1992). A network is series-parallel if its underlying graph is series-parallel.

4.4 The network N

For the input of the FALP, we construct the network $N = (G = (V,E), \ell, u, c)$. The vertices $V = \{s, v_1, \ldots, v_n, t\}$. Let $v_{n+1} = t$ for convenience. There is an edge e_i^s from s to v_i for each $1 \le i \le n$, and there are three edges from v_i to v_{i+1} for each $1 \le i \le n$, where the three edges are denoted by e_i^+, e_i^0 and e_i^-, respectively. See Figure 4.1 for an example.

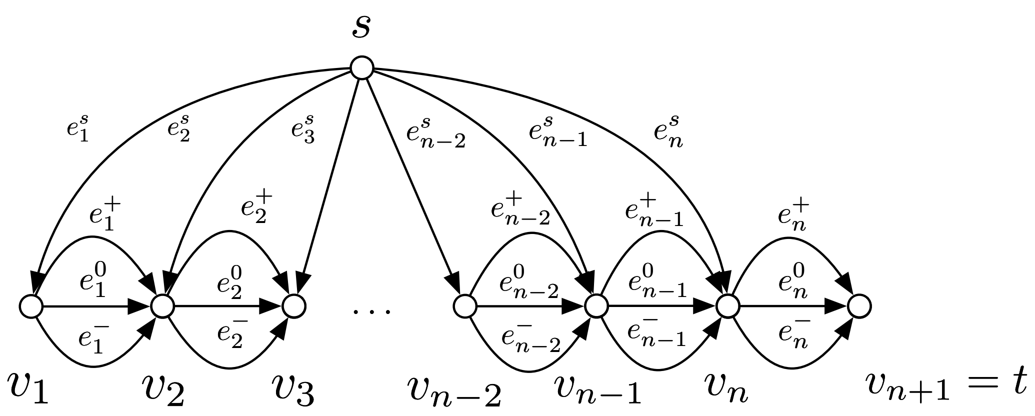

Figure 4.1. The graph G described in section 4.4.

Now we assign capacities to the edges as follows. Let $S_0 = 0$. For each $i \in [n]$, we have

- $\ell(e_i^s) = S_{i-1}$ and $u(e_i^s) = \infty$,
- $\ell(e_i^0) = T_i = u(e_i^0)$,
- $\ell(e_i^-) = E_i - T_i$ and $u(e_i^-) = 0$,
- $\ell(e_i^+) = 0$ and $u(e_i^+) = L_i - T_i$.

Finally, we give a cost function on the edges. The cost is defined as $c(e_i^+) = a_i$, $c(e_i^-) = -b_i$, and $c(e_i^s) = c(e_i^s) = 0$ for all i.

4.5 Reduction to min-cost flow

The next lemma characterizes the cost of the flow f in network N defined in the previous subsection.

Lemma 4.1. *The cost of the flow f in network N is $\sum_{i=1}^n c_i(f^{out}(v_i))$ if for each i, either $f(e_i^+)$ or $f(e_i^-)$ is 0.*

Proof: Recall the definition of $c_i(\cdot)$ in (1). Indeed, if $f(e_i^+)$ is 0, then $f(e_i^-) = f^{out}(v_i) - T_i$. The cost of the flow on edge e_i^- is $-b_i f(e_i^-)$, which is precisely $b_i(T_i - f^{out}(v_i))$. Similarly, if $f(e_i^-)$ is 0, then $f(e_i^+) = f^{out}(v_i) - T_i$. The cost of the flow on edge e_i^+ is $a_i f(e_i^+)$, which is precisely $a_i(f^{out}(v_i) - T_i)$. □

With the results above, we obtain the following theorem, which shows that solving the problem FALP is equivalent to finding a min-cost st-flow in N.

Theorem 4.1. *Suppose that f is a min-cost st-flow in the network N, then $x_i = f^{out}(v_i)$ is an optimal solution of the* FALP.

Proof: We prove the theorem in two steps. First, we show that if $(x_1, \ldots, x_n)$ is a feasible solution of the FALP with cost y, then there is a st-flow f in N with cost y. For simplicity, we define x_0 to be 0. To see this, we construct a st-flow f as follows.

- Let $f(e_i^0) = T_i$.
- If $x_i > T_i$, then $f(e_i^+) = x_i - T_i$ and $f(e_i^-) = 0$. Otherwise, $f(e_i^+) = 0$ and $f(e_i^-) = x_i - T_i$.
- Moreover, $f(e_i^s) = x_i - x_{i-1}$.

The upper and lower bounds of each edge are satisfied. Indeed, we know $E_i \le x_i \le L_i$.

- $\ell(e_i^+) = 0 \le f(e_i^+) = x_i - T_i \le L_i - T_i = u(e_i^+)$,
- $\ell(e_i^-) = E_i - T_i \le x_i - T_i = f(e_i^-) \le 0 = u(e_i^-)$,
- $\ell(e_i^s) = S_{i-1} \le x_i - x_{i-1} = f(e_i^s)$.

In addition, the flow conservation is satisfied. For each $2 \le i \le n-1$

$$f^{in}(v_i) = f(e_{i-1}^+) + f(e_{i-1}^0) + f(e_{i-1}^-) + f(e_i^s) = x_{i-1} + (x_i - x_{i-1}) = x_i = f^{out}(v_i).$$

If $i = 1$, $f^{in}(v_1) = f(e_1^s) = x_1 = f^{out}(v_1)$. Lemma 4.1 implies that the cost of the flow f is equal to the FALP solution $(x_1, \dots, x_n)$.

Second, we show that if f is a min-cost st-flow in N with cost y, then $x_i = f^{out}(v_i)$ is a feasible solution of the FALP with cost y. Fixing an $i \in [n]$. Note that if f is a min-cost st-flow, then we can assume either $f(e_i^+)$ or $f(e_i^-)$ is 0. On one hand, $f^{out}(v_i) = f(e_i^+) + f(e_i^-) + f(e_i^0) = f(e_i^+) + f(e_i^-) + T_i$. In addition, note that $0 \le f(e_i^+) \le L_i - T_i$ and $E_i - T_i \le f(e_i^-) \le 0$. Thus, it follows that $E_i \le f^{out}(v_i) = x_i \le L_i$. On the other hand, by flow conservation, we can obtain that $f^{out}(v_i) = f^{in}(v_i)$. Note that $f^{in}(v_i) = f^{out}(v_{i-1}) + f(e_i^s)$. Then, since $f(e_i^s) \ge S_{i-1}$, we have $x_i = f^{out}(v_i) \ge f^{out}(v_{i-1}) + S_{i-1} = x_{i-1} + S_{i-1}$. By Lemma 4.1, we complete the proof. $\square$

4.6 Series-Parallel Property

Finally, we can further show that the graph G is a series-parallel graph with $O(n)$ edges, which is stated in the following theorem.

Theorem 4.2. *The graph G is a series-parallel graph of $O(n)$ edges.*

Proof: Recall the directed graph G in Figure 4.1. We show that such a graph is a two-terminal series-parallel graph with source s and sink t. To see this, it is sufficient to show this graph can be produced by a sequence of series and parallel compositions.

Let X_i to be the graph with vertices v_i and v_{i+1} and 3 parallel edges e_i^+, e_i^0 and e_i^-. Clearly, (v_i, v_{i+1}, X_i) is a two-terminal series-parallel graph. Similarly, we define Y_i to be the single edge graph $(\{s, v_i\}, \{e_i^s\})$. We construct the graph G by alternatively applying series composition and parallel composition using the previous graph and either the graph X_i or Y_i. Let the base graph $G_1' = Y_0$. Let G_i to be the series composition of (s, v_i, G_i') and (v_i, v_{i+1}, X_i). Let G_i' be the parallel composition of (s, v_{i+1}, G_i) and the single edge (s, v_{i+1}, Y_{i+1}). It is clear that $G_n = G$, which shows that G is a series-parallel graph. $\square$

5 Conclusion

In this paper, we consider a static single-runway Aircraft Landing Problem (ALP), and mainly focus on computing the optimal landing times for a fixed sequence of planes. To solve this problem, this paper presents an algorithm with $O(n \log n)$ running time, which is theoretically more efficient than the quadratic time algorithm in Faye (2018) via a dynamic programming approach. For theoretical curiosity, we find that the proposed FALP can be reduced to a min-cost flow problem on a series-parallel graph allowing negative capacities. Furthermore, using numerical simulations of the same set of input of Faye, we implement our proposed algorithm and then compare them. The result also shows that one of the algorithm has a 36 times speed improvement over Faye's algorithm for large inputs. In this regard, the heuristics based on our algorithm could be much faster than both the quadratic time algorithm and the one using linear programming.

Acknowledgments

The authors would like to thank the Editor and three anonymous referees for their valuable comments and suggestions, which significantly improved the quality and presentation of this paper. This work was supported by the National Natural Science Foundation of China (Grants 72371117, 72071082, 72074082), the National Social Science Foundation of China (Grant 22BGL120), and the Ministry of Education of China, Humanities and Social Sciences Project (Project 23YJC630196).

References

Atkin, J., Hoogeveen, H., & Stolletz, R. (2019). Airport operations management. *OR Spectrum*, *41*(3), 613–614. https://doi.org/10.1007/s00291-019-00562-z

Balakrishnan, H., & Chandran, B. G. (2010). Algorithms for scheduling runway operations under constrained position shifting. *Operations Research*, *58*(6), 1650–1665. http://www.jstor.org/stable/40984034

Beasley, J. E. (1996). Obtaining test problems via internet. *Journal of Global Optimization*, *8*(4), 429–433. https://doi.org/10.1007/BF02404002

Beasley, J. E., Krishnamoorthy, M., Sharaiha, Y. M., & Abramson, D. (2000). Scheduling aircraft landings–the static case. *Transportation Science*, *34*(2), 180–197. https://doi.org/10.1287/trsc.34.2.180.12302

Beasley, J. E., Krishnamoorthy, M., Sharaiha, Y. M., & Abramson, D. (2004). Displacement problem and dynamically scheduling aircraft landings. *The Journal of the Operational Research Society*, *55*(1), 54–64. http://www.jstor.org/stable/4101827

Beasley, J. E., Sonander, J., & Havelock, P. (2001). Scheduling aircraft landings at london heathrow using a population heuristic. *Journal of the Operational Research Society*, *52*(5), 483–493. https://doi.org/10.1057/palgrave.jors.2601129

Bennell, J. A., Mesgarpour, M., & Potts, C. N. (2013). Airport runway scheduling. *Annals of Operations Research*, *204*(1), 249–270. https://doi.org/10.1007/s10479-012-1268-1

Bennell, J. A., Mesgarpour, M., & Potts, C. N. (2017). Dynamic scheduling of aircraft landings. *European Journal of Operational Research*, *258*(1), 315–327. https://doi.org/https://doi.org/10.1016/j.ejor.2016.08.015

Bianco, L., Dell'Olmo, P., & Giordani, S. (2006). Scheduling models for air traffic control in terminal areas. *Journal of Scheduling*, *9*(3), 223–253. https://doi.org/10.1007/s10951-006-6779-7

Booth, H., & Tarjan, R. (1993). Finding the Minimum-Cost Maximum Flow in a Series-Parallel Network. *Journal of Algorithms*, *15*(3), 416–446. https://doi.org/10.1006/jagm.1993.1048

Brand, J. V. D., Chen, L., Kyng, R., Liu, Y. P., Peng, R., Gutenberg, M. P., Sachdeva, S., & Sidford, A. (2023). A deterministic almost-linear time algorithm for minimum-cost flow. *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, 503–514. https://doi.org/10.1109/FOCS57990.2023.00037

Brentnall, A. R. (2006). *Aircraft arrival management* [Doctoral dissertation, University of Southampton].

D'Ariano, A., Pistelli, M., & Pacciarelli, D. (2012). Aircraft retiming and rerouting in vicinity of airports. *IET Intelligent Transport Systems*, *6*(4), 433–443. https://doi.org/10.1049/iet-its.2011.0182

Eppstein, D. (1992). Parallel recognition of series-parallel graphs. *Information and Computation*, *98*(1), 41–55. https://doi.org/10.1016/0890-5401(92)90041-D

Ernst, A. T., Krishnamoorthy, M., & Storer, R. H. (1999). Heuristic and exact algorithms for scheduling aircraft landings. *Networks*, *34*(3), 229–241. https://doi.org/10.1002/(SICI)1097-0037(199910)34:3<229::AID-NET8>3.0.CO;2-W

Fahle, T., Feldmann, R., Götz, S., Grothklags, S., & Monien, B. (2003). The aircraft sequencing problem. In R. Klein, H.-W. Six, & L. Wegner (Eds.), *Computer science in perspective: Essays dedicated to thomas ottmann* (pp. 152–166). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-36477-3_11

Faye, A. (2015). Solving the aircraft landing problem with time discretization approach. *European Journal of Operational Research*, *242*(3), 1028–1038. https://doi.org/10.1016/j.ejor.2014.10.064

Faye, A. (2018). A quadratic time algorithm for computing the optimal landing times of a fixed sequence of planes [Extending the OR Horizons]. *European Journal of Operational Research*, *270*(3), 1148–1157. https://doi.org/https://doi.org/10.1016/j.ejor.2018.04.021

Ghoniem, A., & Farhadi, F. (2015). A column generation approach for aircraft sequencing problems: A computational study. *Journal of the Operational Research Society*, *66*(10), 1717–1729. https://doi.org/10.1057/jors.2014.131

Ghoniem, A., Farhadi, F., & Reihaneh, M. (2015). An accelerated branch-and-price algorithm for multiple-runway aircraft sequencing problems. *European Journal of Operational Research*, *246*(1), 34–43. https://doi.org/https://doi.org/10.1016/j.ejor.2015.04.019

Hancerliogullari, G., Rabadi, G., Al-Salem, A. H., & Kharbeche, M. (2013). Greedy algorithms and metaheuristics for a multiple runway combined arrival-departure aircraft sequencing problem. *Journal of Air Transport Management*, *32*, 39–48. https://doi.org/10.1016/j.jairtraman.2013.06.001

Hansen, J. V. (2004). Genetic search methods in air traffic control. *Computers & Operations Research*, *31*(3), 445–459. https://doi.org/https://doi.org/10.1016/S0305-0548(02)00228-9

Ikli, S., Mancel, C., Mongeau, M., Olive, X., & Rachelson, E. (2021). The aircraft runway scheduling problem: A survey. *Computers & Operations Research*, *132*, 105336. https://doi.org/https://doi.org/10.1016/j.cor.2021.105336

Ji, X.-P., Cao, X.-B., Du, W.-B., & Tang, K. (2017). An evolutionary approach for dynamic single-runway arrival sequencing and scheduling problem. *Soft Computing*, *21*(23), 7021–7037. https://doi.org/10.1007/s00500-016-2241-8

Lieder, A., Briskorn, D., & Stolletz, R. (2015). A dynamic programming approach for the aircraft landing problem with aircraft classes. *European Journal of Operational Research*, *243*(1), 61–69. https://doi.org/https://doi.org/10.1016/j.ejor.2014.11.027

Lieder, A., & Stolletz, R. (2016). Scheduling aircraft take-offs and landings on interdependent and heterogeneous runways. *Transportation Research Part E: Logistics and Transportation Review*, *88*, 167–188. https://doi.org/https://doi.org/10.1016/j.tre.2016.01.015

Ng, K., Lee, C., Chan, F. T., & Qin, Y. (2017). Robust aircraft sequencing and scheduling problem with arrival/departure delay using the min-max regret approach. *Transportation Research Part E: Logistics and Transportation Review*, *106*, 115–136. https://doi.org/https://doi.org/10.1016/j.tre.2017.08.006

Pinol, H., & Beasley, J. (2006). Scatter search and bionomic algorithms for the aircraft landing problem. *European Journal of Operational Research*, *171*(2), 439–462. https://doi.org/10.1016/j.ejor.2004.09.040

Psaraftis, H. N. (1980). A dynamic programming approach for sequencing groups of identical jobs. *Operations Research*, *28*(6), 1347–1359. https://doi.org/10.1287/opre.28.6.1347

Psaraftis, H. N. (1978). *A dynamic programming approach to the aircraft sequencing problem* (tech. rep.). Massachusetts Institute of Technology, Flight Transportation Laboratory. Cambridge, Massachusetts.

Rote, G. (2018). Isotonic Regression by Dynamic Programming. In J. T. Fineman & M. Mitzenmacher (Eds.), *2nd symposium on simplicity in algorithms (sosa 2019)* (1:1–1:18, Vol. 69). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.SOSA.2019.1 Keywords: Convex functions, dynamic programming, convex hull, isotonic regression.

Sahni, S. (2004). *Data structures, algorithms, and applications in java* (2nd). Silicon Press.

Salehipour, A., Modarres, M., & Naeni, L. M. (2013). An efficient hybrid meta-heuristic for aircraft landing problem. *Computers & Operations Research*, *40*(1), 207–213. https://doi.org/10.1016/j.cor.2012.06.004

Sama, M., D' Ariano, A., Palagachev, K., & Gerdts, M. (2019). Integration methods for aircraft scheduling and trajectory optimization at a busy terminal manoeuvring area. *OR Spectrum*, *41*(3), 641–681. https://doi.org/10.1007/s00291-019-00560-1

Soomer, M., & Franx, G. (2008). Scheduling aircraft landings using airlines' preferences. *European Journal of Operational Research*, *190*(1), 277–291. https://doi.org/10.1016/j.ejor.2007.06.017

Xu, C. (2019). Airplane.java. https://gist.github.com/chaoxu/a92e1f8e7e4c89b37a5f962451689a0d

Appendix A

In this appendix, we further consider using our algorithm for ALP itself and compare it with that of Faye (2018). In order to show the difference in running time, we should match the same set of input as Faye (2018). It consists of 10 tests from the publicly available OR-library (Beasley, 1996) involving from 10 to 500 aircrafts. One can find the description of the input in Table 1 of Faye (2018). Note that the experiments are run on the ALP problem instead of the FALP, where the algorithm is a simulated annealing algorithm, and each iteration it runs the subroutine for the FALP.

All the parameters for simulated annealing are also the same as Faye's dynamic programming algorithm; see Table 2 in Faye (2018). However, instead of the same running time as Faye, which does t seconds and repeat 20 runs, we use $t/10$ seconds and repeat 200 runs. The simulated annealing algorithm's convergence is related to the number of iteration, not actual amount of time spent. The total running time over all tests is still the same. We consider two measures of improvement. Gains on objective value compared to Faye (2018), which is already the state of the art, and gains in speed. Speed is measured by the number of iterations per second.

Table 3. Numerical results obtained by using our near-linear algorithm.

Instance	Time limit	CPU time	Iterations	Objective min	Objective avg	Objective max	SD
S1	0.1	0.1	31800	700	700	700	0
S2	0.1	0.1	18157	1480	1490.6	1500	9.98
S3	0.1	0.1	12399	820	820	820	0
S4	0.1	0.1	15239	2520	2520	2520	0
S5	0.1	0.1	14106	3100	3100	3100	0
L1	3	3.001	77055	5607.74	5866.47	6629.42	223.28
L2	6	6.001	98336	12292.44	13149.71	15274.21	465.88
L3	8	8.002	95114	12412.83	12873.28	14523.70	345.19
L4	10	10.003	89295	16144.67	16942.46	18864.39	448.30
L5	15	15.005	74539	37690.84	38954.85	41121.73	580.10

In Table 3, CPU time is the average CPU time, iterations are the average total number of iterations using our proposed algorithm, objective min (avg, and max) refers to the minimum (resp., average, and maximum) values obtained over the runs, and SD is the standard deviation. In Table 4, Gain (%)= $\frac{\text{Objective value in Faye - Objective value in ours}}{\text{Objective value in Faye}} \times 100$, and Gain (×)= $\frac{\text{Iterations in ours}}{\text{Iterations in Faye}}$. For objective value, we find equal or strictly better results for all but two instances. In particular, we improve the best known result for instance L1, L3 and the largest instance L5. However, both the gains and loss are not significant. This is expected because the search algorithm is exactly the same, we only improved the running time. Our experiments demonstrate that the running time for the FALP is at least 36 times faster for the large instances used in Faye (2018), and at least 5 times faster on smaller instances. This is to be expected, as we can see more improvements when the size of the instance grows larger.[1]

Table 4. Comparison between our proposed algorithm and Faye's algorithm.

Instance	Objective value			Iteration/second		
	Faye	Ours	Gain (%)	Faye	Ours	Gain (×)
S1	700	700	0.00	3432	31800	9.27
S2	1480	1480	0.00	2962	18157	6.13
S3	820	820	0.00	2242	12399	5.53
S4	2520	2520	0.00	2512	15239	6.07
S5	3100	3100	0.00	2522	14106	5.59
L1	5611.7	5607.74	0.07	543.9	25685	47.22
L2	12292.2	12292.44	0.00	414.92	16389.33	39.50
L3	12423.1	12412.83	0.08	293.21	11889.25	40.55
L4	16134.4	16144.67	-0.06	246.17	8929.5	36.27
L5	37983	37690.84	0.77	122.75	4969.26	40.52

[1]We did not obtain a improved solution for L2 and L4. It is uncertain why. Since the algorithm is randomized, each run the result can be different. However, recall we only claiming this is a faster implementation of Faye's algorithm.