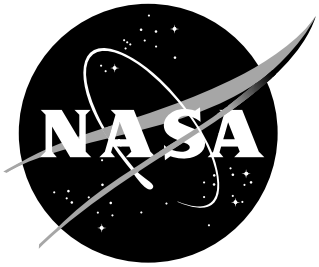
NASA/TM–2021–20210000561



Progress in Scheduling Algorithms for a Collaborative Distributed System for Flight Planning

James D. Phillips
Universities Space Research Association, Moffett Field, CA

Alexander V. Sadovsky
Aviation Systems Division, Moffett Field, CA

February 2021

NASA STI Program ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English- language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at 443-757-5803
- Phone the NASA STI Help Desk at 443-757-5802
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM–2021–20210000561

Progress in Scheduling Algorithms for a Collaborative Distributed System for Flight Planning

James D. Phillips
Universities Space Research Association, Moffett Field, CA

Alexander V. Sadovsky
Aviation Systems Division, Moffett Field, CA

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

February 2021

Acknowledgments

Thanks to Larry Meyn.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

This Technical Memorandum describes four contributions made by the authors to a larger team effort toward developing a distributed system for scheduling commercial flights at navigation fixes and/or airport runways. These contributions are as follows: (1) a proof of correctness for a scheduling algorithm published previously by Meyn [2], (2) an improvement of Meyn's algorithm from quadratic to linear time, (3) two independent implementations of the algorithm with test results identical to those published, and (4) an extension of Meyn's algorithm to support minimum usable time intervals.

1 Introduction

The context of this Memorandum is a team effort that is aimed at describing the development of a distributed computer system for scheduling a number of resources for use by multiple flights. The immediate application was scheduling arrival flights at a large airport or metroplex, but the system is agnostic to the intrinsic nature of transportation or resources. This system is called Collaborative Seamless Management of Airspace Resources and Traffic (CSMART).

CSMART was designed to consist of four types of web services, summarized in Table 1. The system is federated: each participant (flight) or subset of participants (e.g., an airline) is served by its own Flight Plan Service Supplier, and the schedule for the totality of the participants is computed by generalizations of the algorithm given in [1], which, in turn, is based on that in [2].

Table 1. CSMART components.

service	**function**
Directory	Supplies internet addresses for Resource Schedule, the Flight Information Management System, and the Flight Plan Service Supplier.
Resource Schedule	Keeps track, for each resource (e.g., a waypoint), of the time intervals reserved for the use of that resource.
Flight Plan Service Supplier	Provides operator flight schedules. An operator flight schedule is a list of desired time intervals for each scheduled resource in a flight's route.
Flight Information Management System	Provides flight plan information to the FAA or a simulation of the national airspace.

This paper documents improvements to Meyn's multi-point scheduling algorithm [2] that resulted during the initial implementation of the

CSMART Flight Plan Service Supplier. Meyn's algorithm has previously been used as a fundamental primitive in the implementation of an advanced first-come first-serve scheduler for flight schedules, for example [3], and for surface movement schedules, for example [4]. This illustrates the versatility of the algorithm.

The rest of this paper is organized as follows: we will give an overview of the approach and implementation by Meyn [2] in Section 2. As the algorithm in [2] outputs only the available time intervals at each resource and stops short of computing an actual schedule using those intervals, this opens the question whether at least one schedule exists. Section 3 documents a proof that the available time intervals computed by Meyn's algorithm [2] guarantee that the following schedule is viable: the one where, at each resource, the time of use is the earliest time of the earliest available interval. An improvement, in Meyn's algorithm [2], from quadratic to linear time is described in Section 4. The results of validating the authors' independent implementations of Meyn's algorithm are presented in Section 5. An extension of Meyn's algorithm to support minimum usable time intervals is presented in Section 6. Concluding remarks are presented in Section 7.

2 Meyn's Original Approach and Implementation

Meyn [2] gives a closed-form solution for multi-point scheduling of aircraft. This section is an overview of the algorithm. Meyn's terminology will be followed throughout the paper, with the following exception: what Meyn calls a *scheduling point*, we have called, and will continue to call, a *resource*. This preference is aimed at keeping this paper terminologically consistent with the project from which it emerged as a by-product.

In air traffic operations, multiple aircraft must cross the same waypoint or use the same runway, termed a *resource*, at different times. A resource may be unusable during a given time interval for any of multiple reasons. These reasons include poor weather, debris on a runway, and simply the resource having been reserved for use by another flight during the time interval in question. Such time intervals of non-use by the given flight will be called *blocked time intervals*. A time interval that fills the gap between adjacent non-overlapping blocked time intervals, or that fills the semi-infinite time interval after the latest blocked time interval or before the earliest blocked time interval, is an *available time interval*. Notionally, it is convenient to depict each resource as a time axis with the blocked time intervals marked on it. Such a depiction is shown in Fig. 1, where the resources are denoted–in keeping with Meyn's term *scheduling point*, and despite our use of a different term–by the symbol S with a subscript, and the blocked time intervals are shown in red.

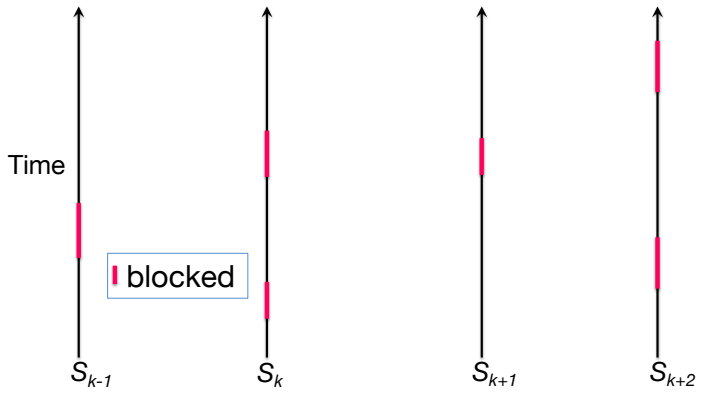


Figure 1. The blocked time intervals (shown in red) at each resource. The parts of the axes not covered by red are available time intervals.

It is possible that blocked time intervals provided as input to the algorithm overlap or share endpoints. The first step of the algorithm is therefore to join every pair of blocked time intervals that share at least one point. The term "join" here means to replace the pair of blocked time intervals with their union. This process is continued until there are no overlapping or touching blocked time intervals. This property, termed *disjoint*, is thereafter preserved by the algorithm. An available time interval is the gap between two disjoint blocked time intervals or a semi-infinite time interval after the latest blocked time interval or before the earliest blocked time interval on the time axis. Thus, **at each stage of computation, the time intervals in a computed list of blocked time intervals, or of available time intervals, will always be disjoint**.

Each time instant at a resource lies either in a blocked time interval or in an available time interval. Note that the concepts of blocked and available time intervals are defined from the viewpoint of a given flight. What is available to one flight and reserved for it, becomes blocked to another.

On the route of each flight, between each consecutive pair of resources there is a *travel time constraint* specified as an input to the algorithm. A travel time constraint defines a pair of bounds–the minimum and the maximum–on the flight's travel time between those resources. These bounds are constant in time and apply whenever the flight happens to travel between consecutive resources. Different flights that share pairs of resources may have different travel time constraints, or different 3D paths between the resources, or both. They may or may not interfere with each other between the resources. The scheduling algorithm captures only the interference at the resources, not between two consecutive resources.

Thus, there are two types of constraints: those binding pairs of flights

(i.e., the time intervals blocked at a resource because another flight is already scheduled for that resource during that interval) and binding an individual flight (i.e., the travel time constraints and time intervals blocked because the resource is unavailable at that time for a reason other than a scheduling conflict with another flight; e.g., hazardous weather conditions).

The scheduling problem is to determine the available time intervals that satisfy all constraints at each resource for a given flight. Meyn's algorithm requires only two linear passes through the resources. However, the evaluation at each resource of some of the constraints requires computations of quadratic cost, as will be shown.

The constraints on the time (i.e., specific time instant) of traversal at each resource can be specified unambiguously by a list of blocked time intervals or by a list of available time intervals. Each list can be determined from the other with a single linear computational pass.

Meyn's implementation works with the available time intervals. There are two passes through the entire list of resources: a forward pass and a backward pass. In the forward pass, the available time intervals are propagated forward using the travel time constraints. The propagation of the travel time constraints from the current resource to the next results in wider available time intervals. This is because the aircraft, leaving the current resource at a specific time t, can arrive at the next resource at any time within the interval

$$[t + \text{minimal travel time}, t + \text{maximal travel time}]; \tag{1}$$

this constraint is illustrated notionally in Fig. 2(A).

Therefore, during forward propagation of an available interval [start, end] at the current resource, the start time of this interval is incremented by the minimum travel time, and the end time by the maximum travel time:

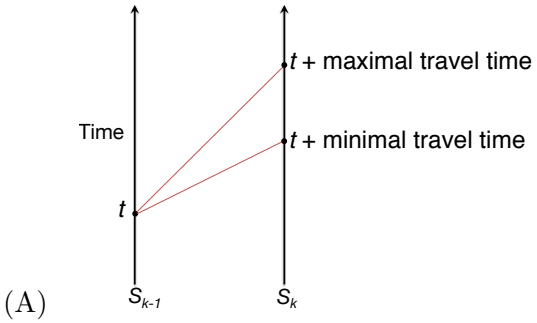
$$[\text{start} + \text{minimal travel time}, \text{end} + \text{maximal travel time}]. \tag{2}$$

This available time interval at the next resource is shifted with respect to, and longer than, the interval [start, end], as is illustrated in Fig. 2(B).

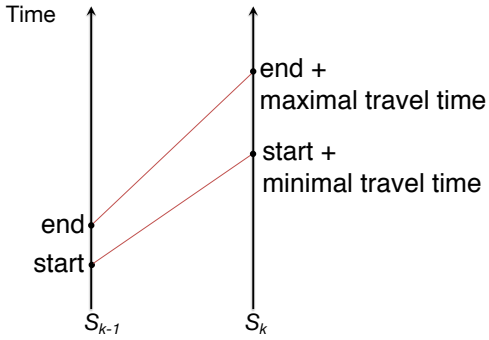
The next step is to perform an intersection operation between the intervals from the propagated list and from the existing list at the latter resource. In [2], this is implemented in Python with a double "for" loop through the two lists, applying the intersection operator to each pair of time intervals, and concatenating these results.

Note that this is an operation of quadratic cost, in the sense that the required number of elementary computations (such as a comparison between two numbers) is proportional to

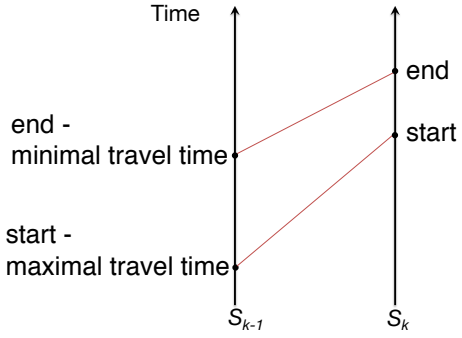
$$\left(\begin{array}{l} \text{the number of intervals} \\ \text{in the propagated list} \end{array} \right) \left(\begin{array}{l} \text{the number of intervals} \\ \text{in the existing list} \end{array} \right).$$



(A)



(B)



(C)

Figure 2. (A) Leaving the current resource, S_{k-1}, at time t, the flight can reach the next resource, S_k, only within time window (1). (B) Leaving S_{k-1} some time in the time interval [start, end], the flight can reach S_k only within time window (2). (C) The times at S_{k-1} from which S_k can be reached during the interval [start, end] fill the interval [start - maximal travel time, end - minimal travel time].

For example, if there are 4 intervals in each constraint list, then 16 intersection operations are needed in general.

The forward pass continues until reaching the last resource. At this stage in the algorithm, each resource has a list of disjoint available time intervals resulting from both the existing ones at the current resource and those propagated from the previous one.

The backward propagation starts at the last resource and propagates the travel time constraints backward through the resources by subtracting the travel times to arrive at feasible start and end times at the previous resource (see Fig. 2(C)). Again, the travel time constraints are used to widen the available time intervals during propagation. Therefore, during backward propagation, the start time of an available interval subtracts the maximum travel time and the end time subtracts the minimum travel time. This produces a list of shifted time intervals at each previous resource. The same intersection operation is performed between the backward propagated available time intervals and the existing available time intervals from the forward pass.

At the end of the backward pass, there is a list of available time intervals that satisfy all time constraints at each resource. These time intervals can be used to build feasible schedules that satisfy all travel constraints and existing blocked time intervals.

The overall algorithm is linear in the number of resources, but quadratic when performing constraint intersection operations between lists of available time intervals.

3 How to Compute a Guaranteed Feasible Schedule from an Output of Meyn's Multi-Point Scheduling Algorithm

As its output, the algorithm assigns to each resource a list of available time intervals (see Section 2) that satisfy all constraints. This assignment and these constraints were covered in the overview in Section 2. We now discuss them in detail, together with their implications for the availability of a given time instant at a resource. A time interval is considered available if it satisfies two types of constraints: (i) not conflicting with existing blocked time intervals and (ii) being consistent with the flight's bounds on travel time. To illustrate (ii), a resource may not be reachable by a flight at a given time if the flight left the previous resource at an earlier given time, simply because the flight cannot fly between the two resources quickly enough or slowly enough. Figure 2(A) illustrates that, on leaving the current resource at a given time t, the flight can reach the next resource only during a certain time interval determined by the bounds on the travel time. Figure 3–which can be thought of as a superposition of Figs. 1 and 2(A)–illustrates a situation when, by leaving the current resource at a certain time t, the flight cannot reach

the next resource at any time available at that resource. This results in the unavailability of time t at the current resource.

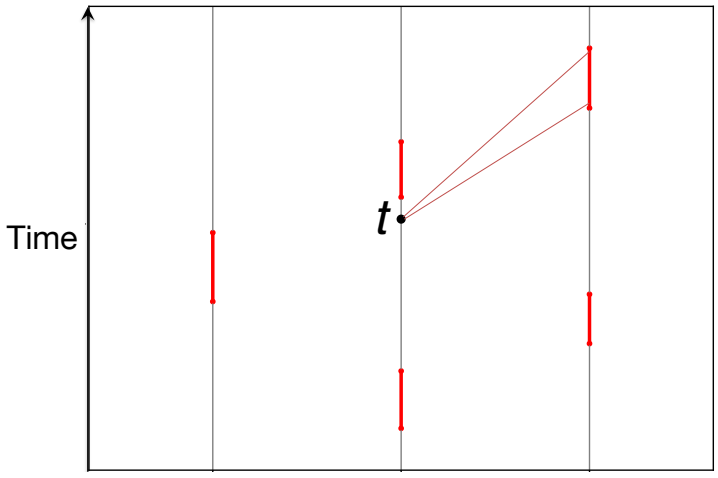


Figure 3. The flight's inability to reach the next resource during an available time by leaving the current resource at a given time, t.

As a result of these constraints, not all of the available time interval pairs at two consecutive resources are reachable one from the other. Whereas the causes of such unreachability are illustrated in Fig. 3, a notional example of such unreachability is shown in Fig. 4.

A schedule requires such a sequence of available time windows, one window per resource, that for every two time windows at two consecutive resources, the later one be reachable from the earlier one. Two such sequences of available time windows satisfying this requirement (hence each capable of being converted into a schedule) are shown notionally in Fig. 4, one using brown curves, the other purple curves.

The occurrences of unreachability, illustrated above, lead to the following concern:

Is Meyn's algorithm guaranteed to give a sequence of available time intervals, one per resource, from which a feasible schedule can be computed for the flight by picking a time instant in each time interval?

The rest of this section addresses this concern by computing such a schedule and proving its feasibility. Following the notation of [1], denote by f the flight for which a schedule is sought, and let $A_f^{k,j}$ denote the j-th earliest time window available to flight f at resource S_k; refer to Fig. 5[1].

Index k is used to number the scheduled resources along the flight's route. Thus, two resources whose indices differ by 1 appear on the route consecutively.

The available time windows, shown in green in Figs. 4 and 5, are

[1]In the paper [1], the k-th resource is denoted simply by k, not by S_k.

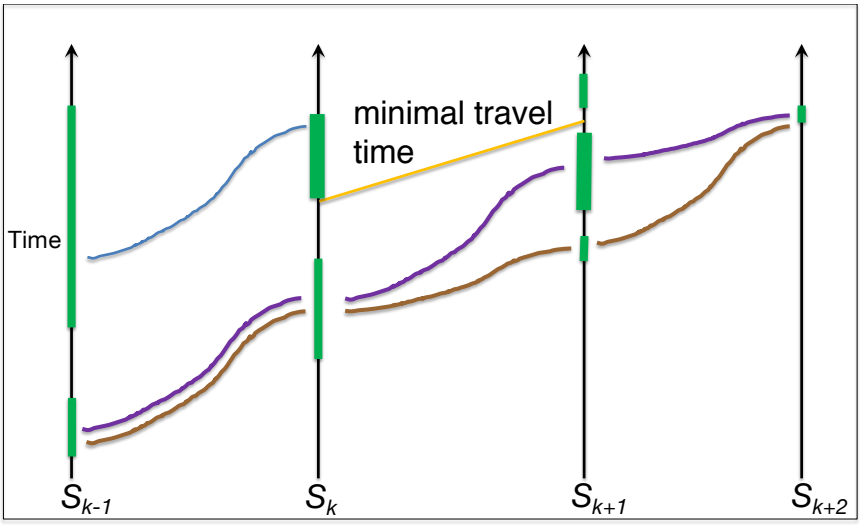


Figure 4. Pairings of intervals by reachability. The available time intervals are shown in green. The time interval shown as a thick green segment at resource S_{k+1} is unreachable from the thick green interval at resource S_k because the flight could not fly quickly enough. The lower bound on the flight's travel time between these two resources is notionally shown as the yellow segment. The pairs of time intervals satisfying reachability are connected with curves of the color brown, purple, and blue.

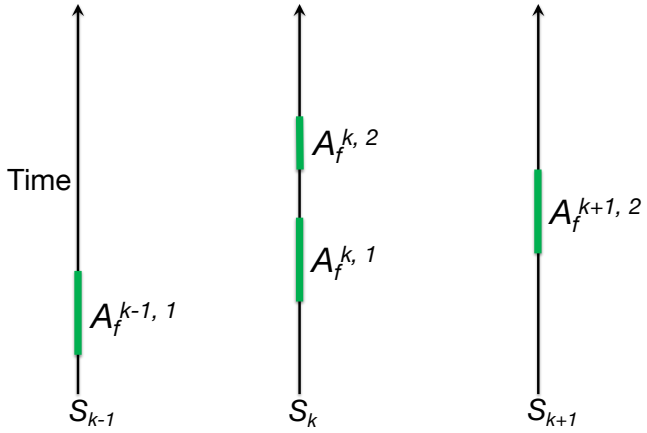


Figure 5. A sample output of Meyn's algorithm for a flight denoted f: the available time intervals at each resource are shown in green.

computed using the algorithm in [2]. We will show here that choosing the earliest time in the earliest time window, $A_f^{k,1}$, at each resource S_k produces a feasible schedule for the flight.

To show this, consider the earliest available time windows available to flight f at two consecutive resources: $A_f^{k-1,1}, A_f^{k,1}$. Denote earliest times in these intervals by t^{k-1} and t^k. If flight f could not reach resource S_k at time t^k by leaving resource S_{k-1} at time t^{k-1}, then t^k could not be the earliest time in an available time window at resource S_k. If flight f could not depart from resource S_{k-1} at time t^{k-1}, then t^{k-1} could not be the earliest time in an available time window at resource S_{k-1}.

4 Improvement of Efficiency for Meyn's Closed-Form Algorithm

In this section, we will describe performance improvements to the algorithm in [2].

Recall from Section 2 that Meyn's algorithm [2] works with the available time intervals. An alternative implementation, complementary to Meyn's, is to propagate the blocked time intervals instead of the available time intervals. Then at each resource the blocked time intervals are combined using union instead of intersection (see Fig. 6).

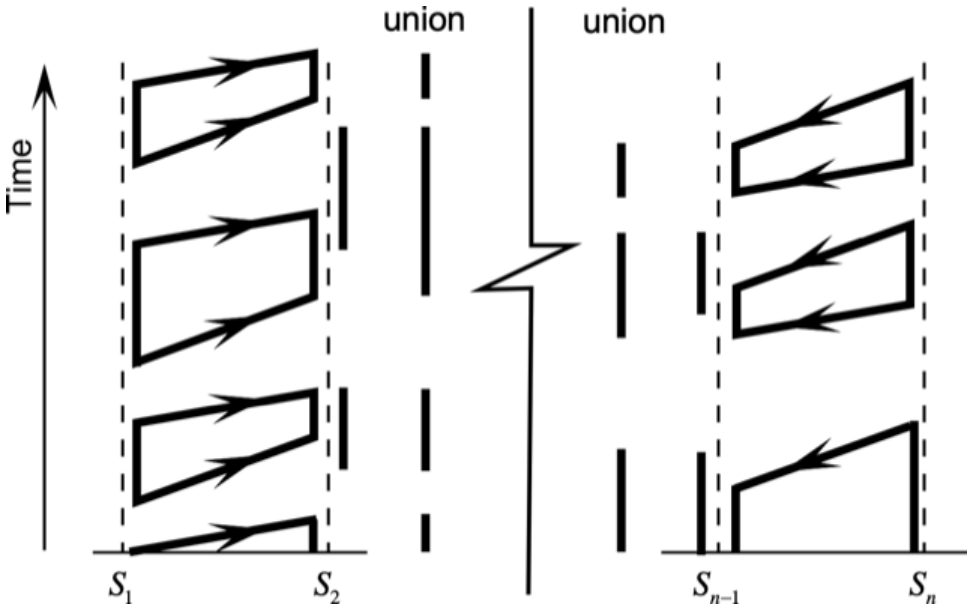


Figure 6. Propagation of blocked time intervals. The points on the horizontal axes, denoted by an S with an index, are the resources.

In Fig. 6, the left hand side shows the first propagation of the forward pass, whereas the right hand side (after the break line) shows the first

propagation of the backward pass. The thick vertical segments represent blocked time intervals. The thick lines with arrowheads represent the propagation of travel time constraints. The vertical gaps between blocked time intervals are the available time intervals.

The dashed vertical lines represent the resources (labeled S_1, S_2, etc.) with time increasing vertically up. In the forward pass, the intervals just to the right of the resource line are the existing input blocked time intervals. The blocked time intervals just to the left of a resource (see S_2) are the propagated blocked time intervals. The existing and propagated blocked time intervals are combined with a union operation shown under the "union" label. These are then propagated forward and the process repeats until reaching S_n.

In the backward pass, the intervals just to the left of the resource line are the blocked time intervals from the forward pass. These are propagated backwards and placed just to the right of the previous resource (see S_{n-1}). The existing and propagated blocked time intervals are combined with a union operation shown under the "union" label. These are then propagated backward and the process repeats until reaching S_1.

Once S_1 is reached, each resource has a stack of blocked time intervals between which are feasible available time intervals satisfying all schedule constraints. It takes a linear pass through the blocked time intervals to collect the available time intervals.

For airline scheduling, it is common to prohibit starting the flight earlier than the scheduled departure time (origin of the vertical axis). This implies a semi-infinite blocked time interval from $-\infty$ to 0. During the forward propagation, the semi-infinite blocked time interval end time grows by the minimum travel time between resources. During the backward propagation, the existing blocked time interval from the forward propagation dominates, as shown on the lower right hand side of Fig. 6.

Examining the forward pass side of Fig. 6, we see characteristic trapezoids formed by propagating the existing blocked time intervals forward. The lower edges have steeper slopes, corresponding to the maximum travel time. The upper edges have gradual slopes, corresponding to the minimum travel time.

These trapezoids always taper increasing the gaps at the next resource. The blocked time intervals always decrease or remain the same in size and shift by the same amount. This means that if the existing blocked time intervals are sorted and disjoint (as noted in Section 2, recall that at each stage of computation, the time intervals in a computed list of blocked time intervals, or of available time intervals, will always be disjoint), then the propagated blocked time intervals are also sorted and disjoint. We would not have this guarantee if we propagated the available time intervals, because they can increase in size.

In the special case where the minimum travel time equals the maximum travel time, the trapezoids turn into parallelograms, and the original blocked time intervals are simply shifted downstream.

With the guarantee that both lists of blocked time intervals to be combined are sorted and disjoint, the union operation can be performed using a linear process similar to the merge in merge sort (see Fig. 7). Two examples of merging are shown in Fig. 7. The first example (left hand side) shows a case where the two lists do not intersect at all. The second example (right hand side) shows a more complicated case where some intervals between the two lists intersect. The preconditions for merging are that each list is sorted and disjoint. Once these two invariants are established with the initial blocked time intervals, the lists never need to be sorted again. Both propagation and merge preserve these invariants with a linear algorithm.

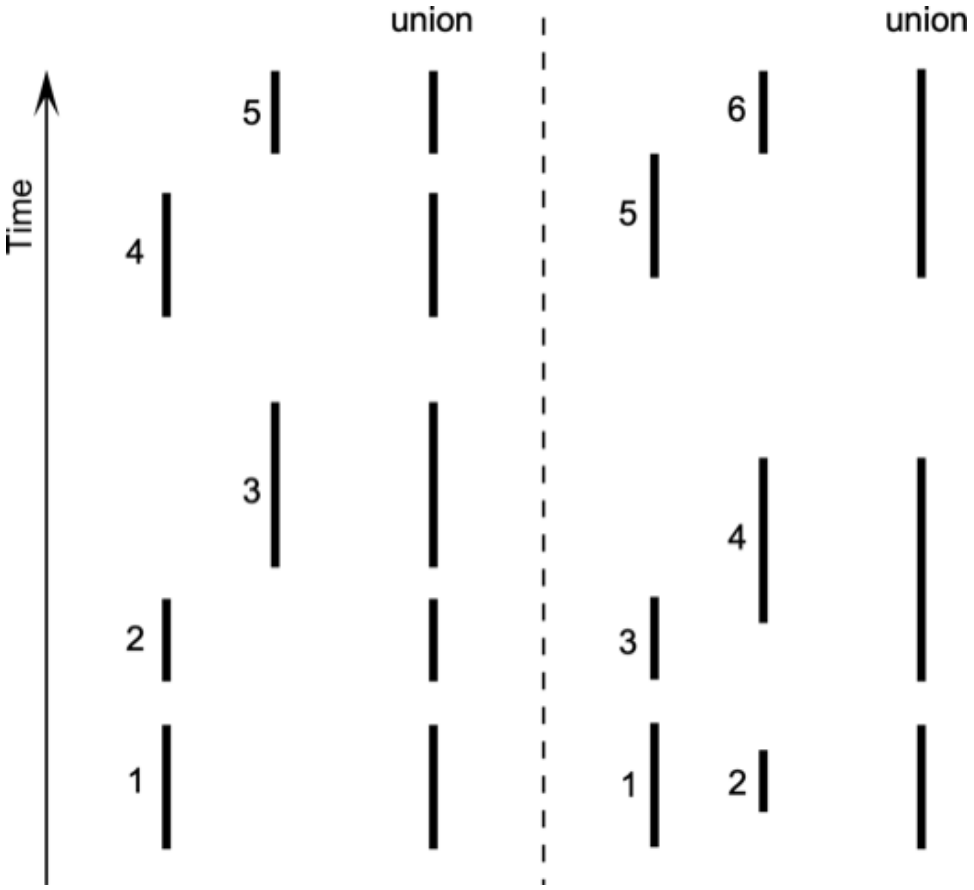


Figure 7. Merging lists of blocked time intervals.

For the distributed airline scheduling algorithm, a separate process provides the currently reserved intervals already sorted and disjoint to the scheduling algorithm. So the scheduling algorithm never has to sort the intervals at all. In any event, the scheduler would only have to sort and join the input lists of blocked time intervals once at the start and this would be the most expensive operation in terms of order of magnitude. Sorting lists in the worst case is linearithmic ($n \log n$).

The meaning of $<$ for blocked time intervals is that all times in one interval are strictly less than all times in the other interval. The lists could therefore be sorted using any time in the interval but it will be convenient to use start time. It is not necessary to consider both start time and end time for the purpose of this algorithm.

The basic idea of the merge algorithm is to alternate comparing segments between lists in sorted order. If they intersect, combine them. Otherwise, add the earliest one to the output list (it must be disjoint).

In the example on the left hand side of Fig. 7 none of the intervals intersect. The algorithm simplifies to exactly the merge in Merge Sort. We could use just two indices to keep track of the earliest unprocessed interval in each list. At each step, we output the earliest interval and increment the index for that list. So we would compare 1 to 3, output 1, then compare 2 to 3, output 2, then compare 3 to 4, output 3, then compare 4 to 5, and output 4. Since the first list is complete, we can output the rest of the second list, which in this case is just interval 5.

The right hand side shows a more complex case where some of the intervals intersect between the two lists. The full algorithm, which handles combining intervals, if necessary, is shown in three pseudocode sections (see Algorithm 1, Algorithm 2, Algorithm 3).

Algorithm 1 Merge algorithm initialization

Require: L_1, L_2 : sorted and disjoint; $n_1 = |L_1|, n_2 = |L_2|$

```
if n_1 = 0 then
    return L_2
else if n_2 = 0 then
    return L_1
end if
i_1 ← 0
i_2 ← 0
if L_1[i_1].start < L_2[i_2].start then
    C = L_1[i_1]
    i_1 ← i_1 + 1
else
    C = L_2[i_2]
    i_2 ← i_2 + 1
end if
```

The input to the merge algorithm is two lists of blocked time intervals: L_1 and L_2. The merge algorithm keeps track of the earliest unprocessed interval in each list using indices: i_1 and i_2. It also tracks two intervals: the current interval, C, and the next interval, N. C is used to accumulate intervals as they overlap. It is only output to the merged list, M, when the next interval is disjoint from it.

The steps to initialize the merge algorithm are shown in Algorithm 1. The preconditions are that the two input blocked time interval lists, L_1

and L_2, are sorted by start time and disjoint. Disjoint means that the intervals within each list do not overlap each other.

If one of the lists is empty, simply return the other one. This also handles the case when both lists are empty.

Initialize the list indices with the earliest interval in each list, then initialize C with the earliest interval from both lists (line 9 or 12) and increment by 1 (line 10 or 13) the index for the list which contains this earliest interval.

The merge algorithm main loop is shown in Algorithm 2.

Algorithm 2 Merge algorithm main loop

```
1:  while i_1 < n_1 and i_2 < n_2 do
2:     if L_1[i_1].start < L_2[i_2].start then
3:        N = L_1[i_1]
4:        i_1 ← i_1 + 1
5:     else
6:        N = L_2[i_2]
7:        i_2 ← i_2 + 1
8:     end if
9:     if C < N then
10:       M.append(C)
11:       C = N
12:    else
13:       C = C ∪ N
14:    end if
15: end while
```

The first step is to compare the start times of the indexed intervals. The earliest of these is assigned to the next interval (line 3 or 6). Its start time is guaranteed to be equal or greater than the start time of the current interval.

The next step is to determine if the two intervals intersect. This can be done with just one time comparison. If the next interval's start time is strictly greater than the current interval's end time, that means all times in the current interval are less than all times in the next interval and the two intervals are disjoint. In this case, save the current interval to the output list, M, (line 10) and assign the next interval to the current interval (line 11). Otherwise, the two intervals must touch or intersect. In that case, combine the two intervals and save the combination as the current interval (line 13).

Combining is defined as creating the shortest interval that contains both intervals. In general, this would require two comparisons: minimum of the start times and maximum of the end times. But we already know that the current interval's start time is the minimum of start times because the two lists are sorted by start times. So we only have to compare the end times to create the combined interval. Since the intervals

intersect, combining is technically a union.

The main loop uses just two time comparisons if current and next are disjoint and only one additional time comparison if they intersect. We are taking advantage of the two invariants we maintain: the lists are always sorted and disjoint.

The maximum number of passes through the main loop is the sum of the two list sizes minus one. The minimum number of passes through the main loop is the size of the shorter list. This is exactly the same as the merge in Merge Sort. For example, if each list contains 4 items, the maximum and minimum number of passes through the main loop is 7 and 4, respectively.

Algorithm 3 Merge algorithm completion

```
1: if i_1 = n_1 then
2:    while i_2 < n_2 do
3:       if C < L_2[i_2] then
4:          break
5:       end if
6:       C = C ∪ L_2[i_2]
7:       i_2 ← i_2 + 1
8:    end while
9:    M.append(C)
10:   M.append(L_2[i_2 : n_2])
11: else
12:   while i_1 < n_1 do
13:      if C < L_1[i_1] then
14:         break
15:      end if
16:      C = C ∪ L_1[i_1]
17:      i_1 ← i_1 + 1
18:   end while
19:   M.append(C)
20:   M.append(L_1[i_1 : n_1])
21: end if
22: return M
```

Ensure: M is sorted and disjoint

Recall that the original algorithm would take 16 times through its loops for this same example (see Section 2). That is more than a factor of 2 for relatively short lists.

The main loop always finishes by reaching the end of one of the lists. This leaves the last value of the current interval unassigned to the output list and possibly many intervals unprocessed in the other list. Algorithm 3 shows the handling of the last interval after the main loop is exited.

The current accumulated interval may overlap with the remaining

intervals in the other list. We need to loop through those remaining intervals until the accumulated interval is disjoint from the next interval or we reach the end of the other list. At that point we can add the rest of the other list because we know those remaining intervals are sorted and disjoint.

In the example on the right hand side of Fig. 7, some of the intervals intersect. The algorithm proceeds as follows:

1. Compare 1 to 2. Initialize current to 1.
2. Compare 2 to 3. Choose 2 as next.
3. Compare current to next. Since they intersect, combine current and next (1 and 2) and save as current.
4. Compare 3 to 4. Choose 3 as next.
5. Compare current to next. They are disjoint so output current (which is the combination of 1 and 2) and save next (which is 3) as current.

This process continues, outputting the combination of 3 and 4 and saving 5 as current. Once this is done, the main loop terminates because we have reached the end of the left hand side list. The completion algorithm must handle the remaining intervals in the right hand side list which in this case is just interval 6. Since the first list finished first, this corresponds to lines 2 through 10 in Algorithm 3.

The first comparison in the while loop (line 3 in Algorithm 3) compares current (which is 5) to 6. Since they intersect, combine current and 6 and save as current. We have now reached the end of the right-hand side list. So output current (combination of 5 and 6) and the rest of the right hand list (which is now empty). That completes the algorithm for the right hand side example of Fig. 7.

In summary, the new blocked time interval merge algorithm reduces the order of magnitude from quadratic to linear over the original implementation. This difference will be noticeable even for relatively short blocked time interval lists. Note that the overall algorithm is still linear in the number of resources, just like the original algorithm.

5 Testing the Consistency of Outputs from Two Independent Implementations of Meyn's Algorithm

The algorithm given in [2] was implemented independently by each author of the present paper, in two different programming languages, Java and Python. In what follows, the following parameters and notation are used:

Table 2. The parameters used in the software implementations described in Section 5.

notation	**value**	**definition**
MAX	9223372036854775807	The maximal integer value used in Java.
TOL	1.0×10^{-7}	The tolerance for floating point arithmetic used in Python.

Phillips chose Java because the CSMART software is implemented in Java. Sadovsky chose Python because the original implementation by Meyn is in Python and the simulation supporting [1] was also implemented in Python. Each of these implementations was run on the two examples presented in Section B and in Figure 4 of Ref. [2]. The computed results matched each other and those in [2]. These results are shown in Tables 3 and 4. In Table 4, the last two time windows computed by the Python implementation as effectively available at the last resources, I, are the open intervals (26.0, 28.0) and (28.0, inf). By contrast, for the same resource, the Java implementation reports the interval [26.0, MAX]. The reason for this difference is that, unlike the Java implementation, the Python one treats the available time intervals as *open* intervals and does not post-process the available intervals to make their list disjoint (see the third paragraph of Section 2).

6 Modification to Support Minimum Usable Time Intervals

Meyn's algorithm finds feasible intervals of time to form a schedule between multiple resources. The size of these intervals is strictly greater than zero, but can be arbitrarily small. Very small intervals do not allow for uncertainty, even if feedback control is somehow used to implement the schedules. The spacing between scheduled intervals is also arbitrarily small.

If only an approximate schedule is desired, then a single time at each resource is adequate. However, if the schedule is being used to separate aircraft in time, then the presence of wind, if nothing else, can introduce some uncertainty. Realistically, a contracted trajectory should therefore specify time intervals, not single time points.

In the context of separating aircraft landing on a runway, there are required separations related to wake turbulence that are specified as distances. The required separation is a function of both aircraft types. These separation distances can be approximately converted to the required time separation using the approach speed of each aircraft.

Table 3. The effectively available time windows that result in the example in Section B of [2]. Parameters MAX and TOL are defined in Table Table 2.

author		Phillips	Sadovsky
language		Java	Python
units		milliseconds	seconds
data type		long	float
resource	A	[0, 0]	[(-TOL, TOL)]
	B	[3000, 5000], [6000, 7000], [9000, MAX]	[(3.0, 5.0), (6.0, 7.0), (9.0, inf)]
	C	[6000, 7000], [9000, 10000], [11000, MAX]	[(6.0, 7.0), (9.0, 10.0), (11.0, inf)]
	D	[8000, 9000], [10000, 11000], [13000, MAX]	[(8.0, 9.0), (10.0, 11.0), (13.0, inf)]

Table 4. The effectively available time windows that result in the example in Figure 4 of [2]. Parameters MAX and TOL are defined in Table Table 2.

		author	Phillips	Sadovsky
		language	Java	Python
		units	milliseconds	seconds
		data type	long	float
resource	A		[0, 0]	[(-TOL, TOL)]
	B		[3000, 5000], [6000, 7000], [9000, 12000], [13000, MAX]	[(3.0, 5.0), (6.0, 7.0), (9.0, 12.0), (13.0, inf)]
	C		[6000, 7000], [9000, 10000], [12000, 14000], [16000, MAX]	[(6.0, 7.0), (9.0, 10.0), (12.0, 14.0), (16.0, inf)]
	D		[8000, 8200], [10000, 11000], [14000, 16000], [17500, MAX]	[(8.0, 8.2), (10.0, 11.0), (14.0, 16.0), (17.5, inf)]
	E		[10000, 10200], [13000, 14000], [17000, 18500], [20500, MAX]	[(10.0, 10.2), (13.0, 14.0), (17.0, 18.5), (20.5, inf)]
	F		[11000, 12200], [15000, 16000], [19000, 20500], [22000, MAX]	[(11.0, 12.2), (15.0, 16.0), (19.0, 20.5), (22.0, inf)]
	G		[13000, 15000], [17000, 19000], [21000, 23000], [25000, MAX]	[(13.0, 15.0), (17.0, 19.0), (21.0, 23.0), (25.0, inf)]
	H		[15000, 16000], [19000, 21000], [23000, 25000], [26000, MAX]	[(15.0, 16.0), (19.0, 21.0), (23.0, 25.0), (26.0, inf)]
	I		[18000, 19000], [22000, 24000], [26000, MAX]	[(18.0, 19.0), (22.0, 24.0), (26.0, 28.0), (28.0, inf)]

There are also separation requirements specified in nautical miles under the FAA visual flight rules. These can also be approximately converted to the required time separation using the airspeed of the aircraft.

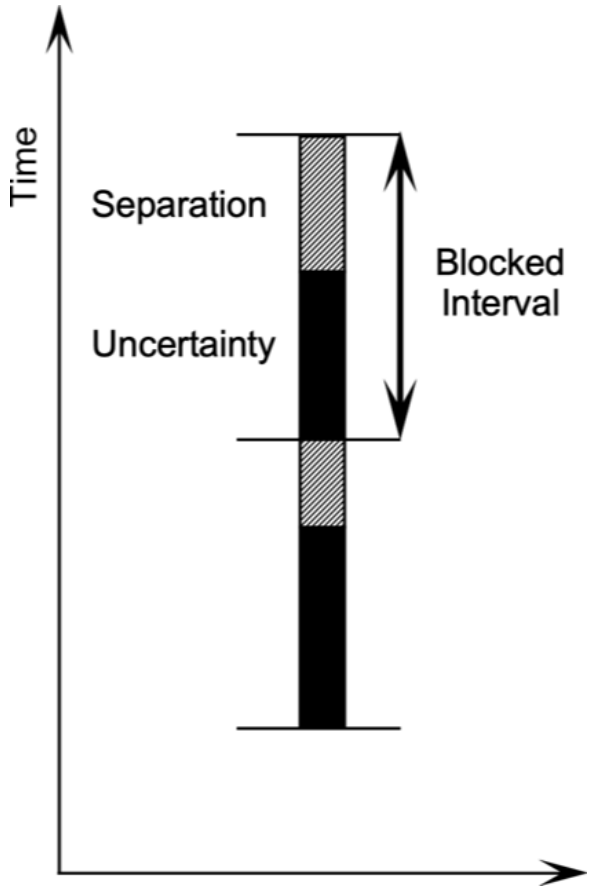


Figure 8. Uncertainty and separation.

These separation requirements are not related to uncertainty. In order to guarantee the required separation and allow for uncertainty, the two must be summed. Figure 8 illustrates the relationship between uncertainty, separation, and blocked time intervals.

Figure 8 shows two blocked time intervals corresponding to different flights. The intervals can be contiguous because the required separation is incorporated into the interval as shown. The black portion of the blocked time intervals is the required time of arrival: the contracted interval that must contain the actual crossing time. It represents the uncertainty in the trajectory. The aircraft may cross at any time during the black interval but not outside. The diagonal hatched segments correspond to the required time separation to the next flight. Note that the two flights can differ in the amount of separation and in the amount of uncertainty.

The algorithm described in this paper can be easily modified to en-

force a minimum time interval. Line 9 in Algorithm 2 and lines 3 and 13 in Algorithm 3 determine whether one interval is disjoint from another. If not, the intervals are combined. This is done by determining if the end of the earlier interval is strictly less than the start of the later interval. All we need to do is modify this definition to include the minimum time interval. Then if the time gap between intervals is too small, the intervals will be combined and the scheduler will naturally not schedule in the small gap.

7 Concluding Remarks

In support of a distributed system for scheduling flights, we have made algorithmic improvements to a closed-form multi-point scheduling algorithm by Meyn [2]. One of those improvements allows enforcing a minimum time interval. This is useful for modeling uncertainty.

We have also proved that the earliest times in the output from the scheduler must be a feasible schedule. This is also a useful schedule since it represents the earliest possible arrival times at each resource.

We have verified for consistent outputs two independent implementations of Meyn's algorithm using examples from his paper.

This work contributes to solving the general problem of scheduling flights that must share airspace resources in a collaborative manner. [3,4] describe two such applications of the original algorithm in [2]. The more recent projects at NASA, e.g. [5], require scheduling flights collaboratively. For example, a recent research goal is the development of a distributed system where multiple participants must share the same resources to schedule their operations. Such is also the project that gave rise to the present work. This work is relevant to all contexts in which vehicles must share resources and are constrained as to how they can move.

The general problem of fitting a finite number of flights into a set of resources with limited capacity is a problem of high computational complexity, akin to the *job-shop problem*. In this context, studies of simplifying assumptions and improvements to the efficiency of schedule computation add value to flight scheduling research.

References

1. Sadovsky, Alexander; and Windhorst, Robert: *A Scheduling Algorithm Compatible with a Distributed Management of Arrivals in the National Airspace System.* 38th Digital Avionics Systems Conference (DASC), San Diego, California, September 8-12, 2019.

2. Meyn, Larry: *A Closed-Form Solution to Multi-Point Scheduling Problems.* American Institute of Aeronautics and Astronautics

(AIAA) Guidance, Navigation, and Control (GNC) Conference and Modeling and Simulation Technologies (MST) Conference, Toronto, Canada, August 2-5, 2010.

3. Park, Chunki; Lee, Hak-Tae; and Meyn, Larry: *Computing Flight Departure Times using an Advanced First-Come First-Served Scheduler.* AIAA 2012-5675, 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Indianapolis, Indiana, September 17-19, 2012.

4. Park, Bay-Seon; Lee, Hyeonwoong; Kang, Seon Young; and Lee, Hak-Tae: *Airport Surface Movement Scheduling with Route Assignment Using First-Come First-Served Approach.* AIAA 2017-4259, 17th AIAA Aviation Technology, Integration, and Operations Conference, Denver, Colorado, June 5-9, 2017.

5. Rios, Joseph L.; Smith, Irene S.; Venkatesan, Priya; Smith, David R.; Baskaran, Vijayakumar; Jurcak, Sheryl M.; Iyer, Shankar K.; and Verma, Punam: *UTM UAS Service Supplier Development: Sprint 2 Toward Technical Capability Level 4.* NASA Technical Memorandum, NASA/TM-2018-220024, 2018.