Algorithmica

Dynamic Algorithms for Multimachine Interval Scheduling

--Manuscript Draft--

Manuscript Number:	ALGO-D-15-00018
Full Title:	Dynamic Algorithms for Multimachine Interval Scheduling
Article Type:	S.I. : ISAAC 2014
Keywords:	interval scheduling; fixed job scheduling; idle intervals
Corresponding Author:	Mikhail Kokho The University of Auckland Auckland, NEW ZEALAND
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	The University of Auckland
Corresponding Author's Secondary Institution:	
First Author:	Alexander Gavruskin
First Author Secondary Information:	
Order of Authors:	Alexander Gavruskin
	Bakhadyr Khoussainov
	Mikhail Kokho
	Jiamou Liu
Order of Authors Secondary Information:	
Abstract:	We study the dynamic scheduling problem for jobs with fixed start and end times on multiple machines. The problem is to maintain an optimal schedule under the update operations: insertions and deletions of jobs. Call the period of time in a schedule between two consecutive jobs in a given machine an idle interval. We show that for any set of jobs there exists a schedule such that the corresponding set of idle intervals forms a tree under the set-theoretic inclusion. Based on this result, we provide a data structure that updates the optimal schedule in O(d+log n) worst-case time, where d is the depth of the set idle intervals and n is the number of jobs. Furthermore, we show this bound to be tight for any data structure that maintains the nested set of idle intervals.

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

Dynamic Algorithms for Multimachine Interval Scheduling

A. Gavruskin · B. Khoussainov
M. Kokho · J. Liu

the date of receipt and acceptance should be inserted later

Abstract We study the dynamic scheduling problem for jobs with fixed start and end times on multiple machines. The problem is to maintain an optimal schedule under the update operations: insertions and deletions of jobs. Call the period of time in a schedule between two consecutive jobs in a given machine an *idle interval*. We show that for any set of jobs there exists a schedule such that the corresponding set of idle intervals forms a tree under the set-theoretic inclusion. Based on this result, we provide a data structure that updates the optimal schedule in $O(d+\log n)$ worst-case time, where d is the depth of the set idle intervals and n is the number of jobs. Furthermore, we show this bound to be tight for any data structure that maintains the nested set of idle intervals.

1 Introduction

Imagine an operator in a delivery company with two responsibilities. The first is to provide delivery service to clients who request specific times for delivery. The second is to schedule the requests for the drivers such that conflicting requests are assigned to different drivers. The goal of operator's work is to accept all client requests and to use as few drivers as possible. The work becomes harder if clients often cancel their requests or change the delivery times of their requests.

The example above is a basic setup for the interval scheduling problem, one of the well-known problems in the theory of scheduling [10,11]. Formally, the problem can be described as follows. An interval a is the usual closed

Alexander Gavruskin, Bakhadyr Khoussainov, Mikhail Kokho
The University of Auckland
E-mail: a.gavruskin@auckland.ac.nz, bmk@cs.auckland.ac.nz, m.kokho@auckland.ac.nz

Jiamou Liu
Auckland University of Technology
E-mail: jiamou.liu@aut.ac.nz

non-empty interval $[s(a), f(a)]$ on the real line, where $s(a)$ is the starting time and $f(a)$ is the finishing time. Naturally, intervals represents jobs that have to be completed in the specified periods. We say that two intervals a and b *overlap* if $a \cap b \neq \emptyset$; otherwise we say that they are *compatible*. We are given a set I of n intervals. The problem is to partition I into sets $S_1, \ldots, S_k$ such that intervals in every set S_i are pairwise compatible and the number k of the sets is as minimal as possible. The partition of jobs represent schedules for the machines.

Definition 1 A subset $J \subseteq I$ is *compatible* if the intervals in J are pairwise compatible.

Depending on the context, we view an interval a as either a process or as a set of real numbers. In the first case, we call $s(a)$ and $f(a)$ respectively the *start* and the *end* of a. In the second case, we refer to $s(a)$ and $f(a)$ as the *left* and *right endpoints* of a. We define the *depth* of I, denoted by $d(I)$, to be the maximal number of intervals in I that contain a common point. In the context of machines, the depth of I is the maximal number of processes that pass over any single point on the time line. We linearly order elements in the set I of intervals by their left endpoints. Namely, we define the order $\prec$ on the set I by setting $a \prec b$ whenever $s(a) < s(b)$ for all $a, b \in I$.

A *scheduling function* for the set of intervals I is a function $\sigma : I \to \{1, \ldots, k\}$ such that any two distinct intervals $a, b \in I$ where $\sigma(a) = \sigma(b)$ are compatible. The number k is called the *size* of the scheduling function. The scheduling function $\sigma : I \to \{1, \ldots, k\}$ partitions I into k *schedules* $S_1, \ldots, S_k$, where for each $i \in \{1, \ldots, k\}$ we have

$$S_i = \{a \in I \mid \sigma(a) = i\}.$$

It is easy to see that $d(I)$ is the smallest size of any scheduling function of I. This gives us the following important definition.

Definition 2 We call a scheduling function σ *optimal* if its size is $d(I)$.

We briefly describe two algorithms solving the basic interval scheduling problem. The standard greedy algorithm [7], which we call *Algorithm 1*, finds an optimal scheduling function σ for the given set of intervals I as follows. It starts with sorting the intervals in order of their starting time. Let a_1, a_2, …, a_n be the listing of the intervals in this order. Schedule a_1 into the first machine, that is, set $\sigma(a_1) = 1$. Then, for the given interval a_i and each $j < i$, if a_i and a_j overlap, exclude the machine $\sigma(a_j)$ for a_i. Schedule a_i into the first machine m that has not been excluded for a_i and set $\sigma(a_i) = m$. The correctness proof of this algorithm is an easy induction [7]. The algorithm runs in $O(n^2)$ time. It is important to observe that this algorithm works in a static context in the sense that the set of intervals I is given a priori and it is not subject to change.

The second algorithm due to Gupta et al. [5], which we call *Algorithm 2*, computes an optimal schedule in $O(n \log n)$ time. Gupta et al. also show that

Algorithm 2 is the best possible. In this algorithm, we work with endpoints of the intervals in I. Let $p_1, p_2, \ldots, p_{2n}$ be endpoints of intervals sorted in increasing order. Scan the endpoints from left to right. For each p_j, if p_j is the start of some interval a, find the first available machine and schedule a into that machine. Otherwise, p_j is the end of some interval b. Therefore, mark the machine $\sigma(b)$ as available. The correctness of the algorithm can be easily verified. Just as above, this algorithm works in a static context.

The problem setup. Our goal is to design data structures that allow us to solve the interval scheduling problem in a dynamic setting. In the dynamic context, the instance of the interval scheduling problem is changed by a real-time events, and a previously optimal schedule may become sub-optimal. Examples of real-time events include job cancellation, the arrival of an urgent job, and changes in job processing times. To avoid the repetitive work of rerunning the static algorithm every time when the problem instance has changed, there is a demand for efficient dynamic algorithms for solving the partitioning problem on the changed instances. In this dynamic context, the set of intervals changes through a number of update operations, such as insertion or removal. Thus, the *dynamic interval scheduling problem* is the problem of maintaining an optimal scheduling function σ for a set I of closed intervals, subject to update operations. The update operations are

- insert(a): insert an interval a into the set I,
- delete(a): delete an interval a from the set I (if it is already there).

Contribution of the paper. There are three main technical contributions of the paper. The first contribution concerns the concept of idle intervals. An interval (t_0, t_1) is *idle* in a given schedule σ if some machine $\sigma(k)$ stays idle during the time period from t_0 to t_1. Intuitively, an idle interval is a place in the schedule where we can insert a new interval if its endpoints are between t_0 and t_1. Now, call the collection of all idle intervals *nested* if any two idle intervals either have no points in common or one interval is included in the other. Firstly, we prove in Lemma 3 that nested schedules are always optimal. Secondly, we prove in Theorem 1 that there are optimal schedules for which the set of idle intervals is nested. This theorem allows us to represent idle intervals of the schedule as a tree, and perform the update operations through maintaining the idle intervals of the schedule. Here we note that Diedrich et al. use idle intervals in [2], where they call them *gaps*, to approximate algorithms for scheduling with fixed jobs. In [2] idle intervals are static and do not depend on the schedule. On the contrary, we describe how to effectively maintain a dynamic set of idle intervals.

Our second contribution is that we provide an optimal data structure that represents nested schedules and supports insert and delete operations. The data structure and its efficiency is based on Theorem 1. Namely, it maintains the nestedness property of the schedules. Theorem 2 proves that all the update operations run in $O(d+log(n))$ in the worst-case. Note that if we naively make *Algorithm 1* or *Algorithm 2* dynamic, the update operations of such algorithms will be significantly slower.

Finally, our third contribution is that we prove in Theorem 4 that the bound $O(d+log(n))$ is tight for any data structure representing nested schedules.

Related work. There are many surveys on the interval scheduling problem and its variants, also known as "k-coloring of intervals", "channel assignment", "bandwidth allocation" and many others. For instance, the reader can consult surveys [10,11]. Gertsbakh and Stern [4] studied the basic problem of scheduling intervals on unlimited number of identical machines. Arkin and Silverber [1] described and solved a weighted version of the interval scheduling problem. In their work the number of machines is restricted and each job has a value. The goal is to maximize the value of completed jobs. A further generalization of the problem, motivated by maintenance of aircraft, was extensively studied by Kroon, Salomon and Wassenhove [12,13] and by Kolen and Kroon [8,9]. In this generalization, each job has a class, and each machine is of specific type. The type of a machine specifies which classes of jobs it can process. Since it was shown in [1] that the problem of scheduling classified jobs is NP-complete, the authors study approximation algorithms. Later, Spieksma [17] studied the question of approximating generalized interval scheduling problem.

2 Idle Intervals and Nested Scheduling

2.1 Idle Intervals

Recall that we have the order $\prec$ on interval by their starting time. Our next definition introduces the notion of idle interval which is crucial for this paper.

Definition 3 Let $J = \{a_1, a_2, \ldots, a_m\}$ be a compatible set of closed intervals such that $a_i \prec a_{i+1}$ for each $i \in \{1, \ldots, m-1\}$. Define the set of *idle intervals of* J as the following set

$$\mathrm{Idle}(J) = \bigcup_{i=1}^{m-1} \{\, [f(a_i), s(a_{i+1})] \,\} \cup \{\, [-\infty, s(a_1)] \,\} \cup \{\, [f(a_m), \infty] \,\}.$$

Note that an idle interval can start at $-\infty$ or end at ∞. Naturally, such intervals represent a period of time when a machine is continuously available before or after some moment of time.

Let $\sigma : I \to \{1, \ldots, k\}$ be a scheduling function of size k of the set I of intervals. Recall the sets $S_1, \ldots, S_k$ with respect to σ: $S_i = \{a \in I \mid \sigma(a) = i\}$. The idea behind considering the set of idle intervals is this: when we insert a new interval a into I, we would like to find a gap in some schedule S_i that fully covers a. Similarly, a deletion of an interval a from I creates a gap in the schedule $S_{\sigma(a)}$. Thus, intuitively the insertion and deletion operations are intimately related to the set of idle intervals of the current schedules S_1, …, S_k. Therefore, we need to have a mechanism that efficiently maintains the idle intervals of S_1, …, S_k.

Definition 4 The set of *idle intervals of* σ is

$$\mathrm{Idle}(\sigma) = \{[-\infty, \infty]\} \cup \mathrm{Idle}(S_1) \cup \mathrm{Idle}(S_2) \cup \ldots \cup \mathrm{Idle}(S_k).$$

Through the scheduling function σ, we can also enumerate the set of idle intervals. Namely, the *schedule number* $\sigma(b)$ of the idle interval $b \in \mathrm{Idle}(\sigma)$ is i if $b \in \mathrm{Idle}(S_i)$, and is $k+1$ if $b = [-\infty, \infty]$. The next lemma state that the depth of the idle interval set is greater than or equal to the depth of the interval set.

Lemma 1 *We have* $d(I) \le k \le d(\mathrm{Idle}(\sigma))$.

Proof For the first part, observe that for any sets of intervals I and J, the following inequality holds true:

$$d(I) \le d(I \cup J) \le d(I) + d(J).$$

Since the depth of each schedule S_i is 1, we have

$$d(I) \le d(S_1 \cup \cdots \cup S_k) \le \sum_{1 \le i \le k} d(S_i) = k$$

For the second part, let a_i be the $\preceq$-least interval in the schedule S_i. Then for each $i \in \{1, \ldots, k\}$ an interval $[-\infty, s(a_i)]$ belongs to $\mathrm{Idle}(\sigma)$. Now take a real number $x \in \mathbb{R}$ that is smaller than all the starting times of the intervals in I. In particular, x is smaller that the starting time of any a_i. Thus x intersects with k intervals in $\mathrm{Idle}(\sigma)$, i.e. $k \le d(\mathrm{Idle}(\sigma))$. □

Definition 5 A set J of intervals is *nested* if $[-\infty, \infty] \in J$ and for all $b_1, b_2 \in J$, it is either that b_1 covers b_2 or b_2 covers b_1 or b_1, b_2 are compatible.

Any nested set of intervals J defines a tree under set-theoretic inclusion $\subseteq$. Indeed, here the nodes in the tree are the intervals in J, and an interval b_2 is a descendent of another interval b_1 if $b_2 \subset b_1$. We call this tree the *nested tree* of J and denote it by $\mathrm{Nest}(J)$. We order siblings in $\mathrm{Nest}(J)$ by the left endpoints of the corresponding intervals. Recall that the *height* of a tree is the maximum number of edges in a path that goes from the root to any leaf.

Lemma 2 *For any nested set* J *of intervals, the depth of* J *equals to the height of the nested tree* $\mathrm{Nest}(J)$.

Proof Let J be a nested set of intervals and h be the height of the nested tree $\mathrm{Nest}(J)$. To show that $d(J) \le h$, we take a maximal path $b_0, b_1, \ldots, b_h$ in the nested tree. In this path $b_0 = [-\infty, \infty]$, and $b_{i+1} \subset b_i$ for all $i \in \{0, \ldots, h-1\}$. The interval b_h is fully covered by all other intervals. Therefore the starting point $s(b_h)$ intersects with h intervals. Hence $d(J) \le h$

To show the reverse inequality, take any real number $x \in \mathbb{R}$ and let C be the set of intervals in J that contain x. Since J is a nested set, C is a nested set as well. Therefore C contains a sequence $b_1, b_2, \ldots, b_\ell$ where $b_i \subset b_{i+1}$ for all $i \in \{1, \ldots, \ell\}$. This sequence defines a single path in the tree $\mathrm{Nest}(J)$. Thus $h \le d(J)$. □

In the next subsection we connect idle interval sets with the nested trees.

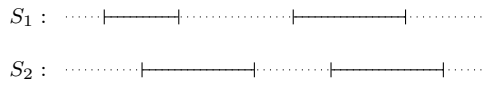

Fig. 1 Dotted lines define the idle interval set

2.2 Nested Scheduling

Definition 6 Let σ be a scheduling function of the set of intervals I. We say that σ is *nested schedule* if the set $\mathrm{Idle}(\sigma)$ of idle intervals is nested.

The next lemma shows the usefulness of the notion of nested schedules. In particular, nested schedules are optimal.

Lemma 3 *If $\sigma : I \to \{1, \dots, k\}$ is a nested scheduling function, then the depth of the idle intervals $\mathrm{Idle}(\sigma)$ coincides with the depth of I. In particular, every nested schedule is optimal.*

Proof Let $\sigma : I \to \{1, \dots, k\}$ be a nested scheduling function for I. By Lemma 1, $d(I) \leq d(\mathrm{Idle}(\sigma))$. To show that $d(\mathrm{Idle}(\sigma)) \leq d(I)$, by Lemma 2, it is sufficient to prove that the height h of the nested tree $\mathrm{Nest}(\mathrm{Idle}(\sigma))$ is at most $d(I)$.

Take a maximal path $b_0, b_1, \dots, b_h$ in $\mathrm{Nest}(\mathrm{Idle}(\sigma))$ such that $f(b_1) \neq \infty$. For each $i \in \{1, \dots, h\}$ let S_i be a schedule such that $b_i \in \mathrm{Idle}(S_i)$. We show that in every schedule S_i there exists an interval $a_i \in S_i$ such that $f(b_1)$ intersects with a_i.

For contradiction, assume that there exists a schedule S_j such that $f(b_1)$ does not intersect with any interval in S_j. Then there exists an idle interval $c \in \mathrm{Idle}(S_j)$ such that $f(b_1) \in c$. Therefore $s(c) < f(b_1)$. On the other hand, since $b_j \in \mathrm{Idle}(S_j)$, we have $s(b_1) < f(b_j) < s(c)$. These imply that the idle intervals b_1 and c overlap, which contradicts with the fact that $\mathrm{Idle}(\sigma)$ is a nested schedule. Thus h is at most $d(I)$. □

A natural question is whether the schedule constructed by either *Algorithm 1* or *Algorithm 2* is nested. The next simple example gives a negative answer to this question. Indeed, consider the set I of intervals presented in Figure 1. Both algorithms yield the same scheduling, which is not nested:

In the next section, however, we prove that every interval set I possesses a nested scheduling.

2.3 Extending Nestedness

One of the goals of this section is to prove that every interval set I possesses a nested scheduling. The proof will also provide a method, explained in the next section, that maintains the interval set I by keeping the nestedness property invariant under the update operations.

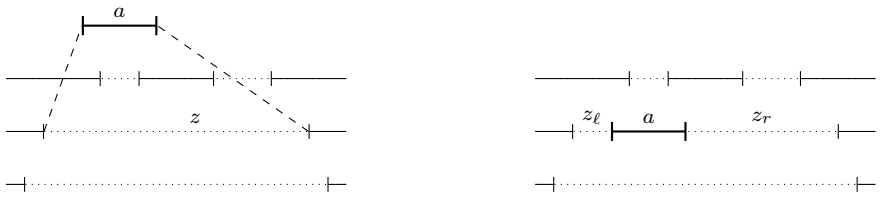

Fig. 2 Rescheduling when a does not overlap with idle intervals

Suppose that $\sigma : I \to \{1, \ldots, k\}$ is a nested scheduling function for the interval set I. Recall that we use $S_1, \ldots, S_k$ to denote the k schedules with respect to σ. Let a be a new interval not in I. We introduce the following notations and make several observations to give some intuition to the reader.

- Let $L \subset \text{Idle}(\sigma)$ be the set of all the idle intervals that contain $s(a)$, but do not cover a. The set L, as $\text{Idle}(\sigma)$ is nested, is a sequence of embedded intervals $x_1 \supset \cdots \supset x_\ell$, where $\ell \geq 1$. Note that L can be the empty set.
- Let $R \subset \text{Idle}(\sigma)$ be the set of all the idle intervals that contain $f(a)$, but do not cover a. The set R, as above, is a sequence of embedded intervals $y_1 \supset \cdots \supset y_r$, where $r \geq 1$. Again, R can be empty as well.
- Let z be the shortest interval in $\text{Idle}(\sigma)$ that covers a. Such an interval exists since $[-\infty, \infty] \in \text{Idle}(\sigma)$. To simplify the presentation, we set $x_0 = y_0 = z$. Note that $x_0 \supset x_1$ and $y_0 \supset y_1$.

Now our goal is to construct a new nested schedule based on σ and the content of the sets L and R. For that we consider several cases.

Case 1: L and R are empty sets

In this case we can easily extend σ to the domain $I \cup \{a\}$ and preserve the nestedness property. Indeed, as $a \subset z$, we simply extend σ by setting $\sigma(a) = \sigma(z)$. We show that the resulted schedule is nested.

After insertion of a, the idle interval z is split into two intervals $z_\ell = [s(z), s(a)]$ and $z_r = [f(a), f(z)]$. Consider an arbitrary idle interval u in $Idle(\sigma)$. If u and z are compatible then u is compatible with both z_r and z_ℓ. If $z \subset u$ then the intervals z_ℓ and z_r are now covered by u. If $z \supset u$ then u is either covered by z_ℓ or z_r, or u does not intersect with the new idle intervals. Thus the resulting set of idle intervals is nested. See an example in Figure 2.

Case 2: L is not empty, but R is not

If we simply set $\sigma(a) = \sigma(z)$ as in the previous case, some of the intervals in the new idle set will be overlapping. For example, the new idle interval $[s(z), s(a)]$ will intersect with x_1. Therefore we reorganize the schedule σ as follows.

We schedule interval a for the machine $\sigma(x_\ell)$. We move all the jobs d of the machine $\sigma(x_\ell)$ such that $d \succ x_\ell$ to machine $\sigma(x_{\ell-1})$. In the schedule $S_{\sigma(x_{\ell-1})}$ there are other jobs that start after $x_{\ell-1}$. To avoid collisions, we move these jobs to the machine $\sigma(x_{\ell-2})$. We continue this on until we reach the jobs scheduled for the machine $\sigma(z)$. Finally, we move the jobs d from the machine $\sigma(z)$ such that $d \succ z$ to the machine $\sigma(x_\ell)$. Note that if $f(z) = +\infty$ there is simply no jobs on the machine $\sigma(z)$ to reschedule. Example of this process is

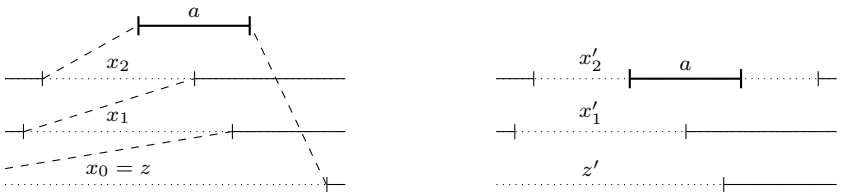

Fig. 3 Rescheduling when idle intervals overlapping a contains only $s(a)$

on Figure 3. Formally, we define the new scheduling function σ_1 as follows and claim that σ_1 is nested:

$$\sigma_1(d) = \begin{cases} \sigma(x_\ell) & \text{if } d = a, \text{ or} \\ & \sigma(d) = \sigma(z) \text{ and } d \succ z, \\ \sigma(x_{i-1}) & \text{if } \sigma(d) = \sigma(x_i) \text{ and } d \succ x_i, \\ & \text{where } 0 < i \leq \ell, \\ \sigma(d) & \text{otherwise.} \end{cases}$$

Claim A. The scheduling σ_1 defined is nested.

Proof The set of idle intervals $\mathrm{Idle}(\sigma_1)$ consists of the new interval $[f(a), f(z)]$ together with all the idle intervals of σ where the idle intervals x_ℓ, $x_{\ell-1}$, …, x_1, and z are changed to the following new idle intervals $[s(x_\ell), s(a)]$, $[s(x_{\ell-1}), f(x_\ell)]$, …, $[s(x_1), f(x_2)]$, and $[s(z), f(x_1)]$, respectively. We denote the set of changed intervals by L'.

Let u and v be two idle intervals of σ_1. We want to show that either $u \cap v = \emptyset$ or one of these two intervals is contained in the other. If both u and v are old or both u and v are new then we are done. So, say u is new, and v is old. First, assume that u is $[f(a), f(z)]$. Because by assumption $R = \emptyset$, the interval v either covers a or does not intersect with a. In the first case, $u \subset v$, because we have chosen z such that $z \subset v$. In the second case, if $f(v) < s(a)$ then v and u does not intersect. If $s(v) > f(a)$ then, because $\mathrm{Idle}(\sigma)$ was a nested set of intervals, we have that either $v \subset u$ or $u \cap v = \emptyset$.

Second, assume u is one of the changed intervals in L' and v is an old idle interval. If v contains $s(a)$ then v must contain z since v is old. Hence $u \subset v$. Otherwise, suppose that v contains a point $r \in [s(x_i), f(x_{i+1})]$ or $r \in [s(x_\ell), s(a)]$. Then either $r \in x_i$ or $r \in x_{i+1}$. Hence, $v \subset x_i$ or $v \subset x_{i+1}$. If the first case we have $v \subset [s(x_i), s(a)]$, and in the second case $v \subset [s(a), f(x_{i+1})]$. In either case, $v \subset [s(x_i), f(x_{i+1})]$. This proves the claim. □

Case 3: The set R is not empty, but $L = \emptyset$

This case is symmetric to the previous case: we need to reorganize the intervals, but we reorganize the intervals with respect to the finishing time of the new interval a. First, we set $\sigma(a)$ equal to $\sigma(z)$. The new idle interval $(f(a), f(z))$ now intersects with idle intervals $y_1, \ldots, y_r$. Therefore for every $1 \leq i < r$ we move intervals from the machine $\sigma(y_i)$ that start after y_i to the next machine $\sigma(y_{i+1})$. To avoid collision on the last machine $\sigma(y_r)$ we

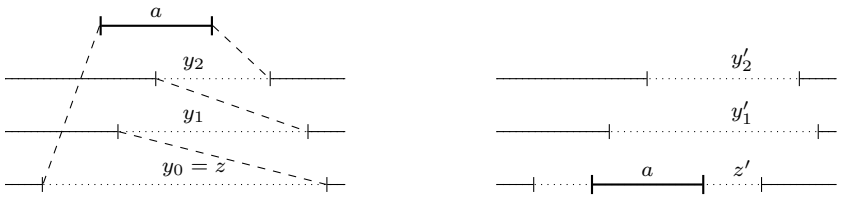

Fig. 4 Rescheduling when idle intervals overlapping a contains only $f(a)$

move intervals from this machine that start after y_r to the machine $\sigma(z)$. An example with two intervals y_1 and y_2 is shown on Figure 4. Formally, we define a new scheduling function σ_2 as follows:

$$\sigma_2(d) = \begin{cases} \sigma(y_\ell) & \text{if } d = a, \text{ or} \\ & \sigma(d) = \sigma(z) \text{ and } d \succ z, \\ \sigma(y_{i+1}) & \text{if } \sigma(d) = \sigma(y_i) \text{ and } d \succ y_i, \\ & \text{where } 0 \le i < r, \\ \sigma(d) & \text{otherwise.} \end{cases}$$

To see that σ_2 is nested, consider two arbitrary idle intervals u and v in the idle set $Idle(\sigma)$. If these intervals has not been changed by the schedule reorganization, then, by the nestedness of $Idle(\sigma)$, these two intervals are either compatible or nested. If both of the intervals has been changed, then by construction of σ_2 these intervals are nested. If one of the intervals has been changed, say u, then either $u \subset v$ or $v \subset u$ or they are compatible. We leave the details of this reasoning to the reader.

Case 4: Both sets L and R are non-empty.

We reorganize the schedule σ in two steps. In the first step, we proceed exactly as in *Case 2*. Namely, we move all the intervals d of the machine $\sigma(x_\ell)$ that start after x_ℓ to the machine $\sigma(x_{\ell-1})$; we continue this on by moving all the intervals of the machine $\sigma(x_i)$ that start after x_i to the machine $\sigma(x_{i-1})$. When we reach the machine $\sigma(x_0)$, we move all the jobs d of the machine $\sigma(x_0)$ such that $d \succ x_0$ to the machine $k+1$, that is, to the idle interval $[-\infty, +\infty]$. Denote the resulting schedule by σ_1. Note there are no collisions between the jobs in the resulted schedule, but it is not a nested schedule yet. A formal definition of σ_1 is as follows:

In the second step, starting from $\sigma_1(y_i)$, where $i = 1, \ldots, r-1$, we move all intervals of the machine $\sigma_1(y_i)$ that start after y_i to the machine $\sigma(y_{i+1})$:

Lemma 4 *The scheduling function σ_2 is nested.*

Proof Let K be the set of intervals in $\mathrm{Idle}(\sigma_2)$ that begins at or after $s(x_0)$ and ends at or before $f(x_0)$. In other words

$$K = \{d \in \mathrm{Idle}(\sigma_2) \mid d \subset x_0\}.$$

$$\sigma_1(d) = \begin{cases} \sigma(x_\ell) & \text{if } d = a, \\ \sigma(x_{i-1}) & \text{if } \sigma(d) = \sigma(x_i) \text{ and } d \succ x_i, \\ & \text{where } 0 < i \leq \ell, \\ k+1, & \text{if } \sigma(d) = \sigma(x_0) \text{ and } d \succ x_0, \\ \sigma(d) & \text{otherwise.} \end{cases}$$

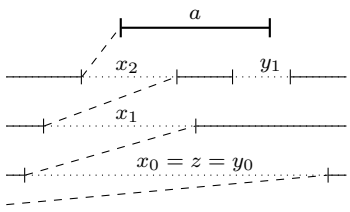

Fig. 5 The first step of rescheduling: defining σ_1.

$$\sigma_2(d) = \begin{cases} \sigma_1(a) & \text{if } \sigma_1(d) = \sigma_1(y_r) \text{ and } d \succ y_r, \\ \sigma_1(y_{i+1}) & \text{if } \sigma_1(d) = \sigma_1(y_i) \text{ and } d \succ y_i, \\ & \text{where } 0 < i < r, \\ \sigma_1(y_1) & \text{if } \sigma_1(d) = k+1, \\ \sigma_1(d) & \text{otherwise.} \end{cases}$$

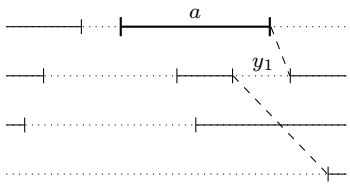

Fig. 6 The second step of rescheduling: defining σ_2.

By construction $\mathrm{Idle}(\sigma_2) \setminus K = \mathrm{Idle}(\sigma) \setminus K$, and by nestedness of $\mathrm{Idle}(\sigma)$, $\mathrm{Idle}(\sigma_2) \setminus K$ is also an nested set. Furthermore, it is clear that for any interval $p \in K$ and $q \in \mathrm{Idle}(\sigma) \setminus K$, it is either that p, q are compatible or $p \subset q$. Therefore it only remains to show that the set K is also a nested set. We show that any two intervals $p, q \in K$ are either compatible or one is covered by the other.

Suppose p contains $s(a)$. Then the start of p is $s(x_i)$ for some $0 \leq i \leq \ell$. Moreover, the end of p is $s(a)$, if $i = \ell$, and $f(x_{i+1})$, otherwise. Note that $p \subset x_i$. Consider two cases with respect to q:

- Case 1: q contains $s(a)$. Then, similarly to p, the start of q is $s(x_j)$ for some $0 \leq j \leq \ell$. If $x_j \prec x_i$ then q covers p. Otherwise, p covers q.
- Case 2: q does not contain $s(a)$. Let r be a real number such that $r \in q \cap p$. If such r does not exists, then p and q are compatible. Otherwise $r \in [s(x_i), s(a))$ or $r \in (s(a), f(x_{i+1})]$. Since $\mathrm{Idle}(\sigma)$ is a nested set, we have that either $q \subset x_i$ or $q \subset x_{i+1}$. Hence $q \subset p$.

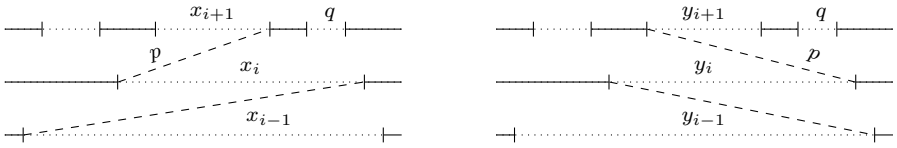

Fig. 7 Nestedness is preserved

Now suppose p contains $f(a)$. Then the end of p is $f(y_i)$ for some $0 \leq i \leq r$. Moreover, the start of p is $f(a)$, if $i = r$, and $f(y_{i+1})$, otherwise. Similarly to

the previous case, if q contains $f(a)$ then, depending on the end of q, one of the intervals covers the other. If q does not contain $f(a)$, there are two cases:

- Case 1: q is covered by y_r. If $a \prec q$ or $y_i \prec y_r$ then p covers q. Otherwise p and q are compatible.
- Case 2: p is covered by y_j for some $0 \leq j < r$ but not covered by y_{j+1}. If $y_{i+1} \prec q$ or $y_i \prec y_j$, then p coveres q. Otherwise, p and q are compatible.

Finally, suppose that neither p nor q contain $s(a)$ or $f(a)$. Then, by construction of σ_2, p and q are in $\mathrm{Idle}(\sigma)$. Therefore they are either compatible or one covers the other. Thus the set K is nested and hence σ_2 is a nested scheduling function. □

Theorem 1 *For any set of closed intervals I there is a scheduling function σ such that* $\mathrm{Idle}(\sigma)$ *is a nested set.*

Proof We prove by induction on the size $|I|$ of I. When $|I| = 1$ it is clear that $\mathrm{Idle}(I)$ is nested. The inductive step follows directly from the construction of σ_2 and Lemma 4. □

2.4 Restoring nestedness after deletion

In this section we show how to effectively restore nestedness of the schedule after deletion of an interval. We will need the technique described here to develop the delete operation of a dynamic algorithm.

We use the same notations as in the previous section: a denotes the deleted interval, L and R are the sets of idle intervals that intersect with $s(a)$ and $f(a)$ respectively, z is the shortest idle interval that covers a. Note that the sets L and R always contain the idle intervals x_ℓ and y_r, respectively, which are adjacent to the deleted interval a.

Case 1: L and R are empty sets

In this case we can easily delete a from the domain of σ and preserve the nestedness property. Indeed, after the deletion we have the new idle interval $b = [s(x_\ell), f(y_r)]$. Let v be an old idle interval. Suppose, v and a are compatible intervals. By the nestedness of σ, v is either covered by x_ℓ or y_r or is compatible with them. Therefore v is either covered by b or is compatible with it. Now suppose that v and a are not compatible. If $v \subset a$ then $v \subset b$. If $a \subset v$ then $x_\ell \subset v$ and $y_r \subset q$, which implies that $b \subset v$. Thus, in either case the nestedness is preserved.

Case 2: $|L| \geq 1$, but $|R| = 1$

In this case, after deletion of the interval a, the idle interval $[s(x_\ell), f(y_r)]$ intersects with $x_{\ell-1}$. Therefore we move all the jobs d of the machine $\sigma(x_\ell)$ such that $d \succ x_\ell$ to the machine $\sigma(x_1)$. Then we move all the jobs d of the machine $\sigma(x_{\ell-1})$ such that $d \succ x_{\ell-1}$ to the machine $\sigma(x_\ell)$. We repeat this process on every machine $\sigma(x_i)$. We stop after we moved the jobs of the machine $\sigma(x_1)$

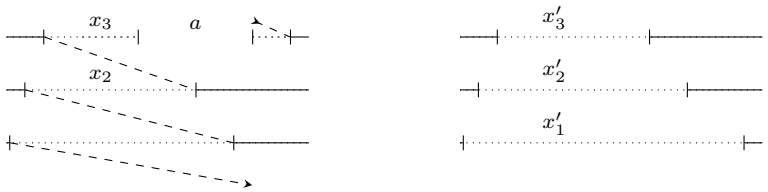

Fig. 8 Rotation of the schedules preserves the nestedness

to the machine $\sigma(x_2)$. Denote the new scheduling function by σ_3. An example of the rescheduling is shown in Figure 8.

Claim B. The scheduling σ_3 defined is nested.

Proof Let u and v be two idle intervals in $\mathrm{Idle}(\sigma_1)$. Similarly to the case in the previous subsection, if u and v are both old or both new idle intervals, then they are either compatible or one is contained in the other. So suppose u is a new idle interval, and v is old. Let t be a real number such that $t \in u \cap v$.

Suppose $t \in y_r$. Then $u = [s(x_\ell), f(y_r)]$. Moreover, by the nestedness of $\mathrm{Idle}(\sigma)$, it is either $v \subset y_r$ or $y_r \subset v$. In the first case, we have that $v \subset u$. In the second case, because $v \notin R$, v covers x_ℓ. Therefore v covers u.

Now suppose $t \notin y_r$. Then u is one of the intervals $x'_i = [s(x_i), f(x_{i-1})]$ or $x'_1 = [s(x_1), f(y_r)]$. We look at v and its relation to the interval $x_i \in \mathrm{Idle}(\sigma)$:

- $v \subset x_i$. Then $v \subset u$.
- $v \supset x_i$. Since v is old, $v \supset a$. Since $\mathrm{Idle}(\sigma)$ is nested, $v \supset y_r$. Therefore $v \supset u$.
- $f(v) < s(x_i)$. Then v and u are compatible.
- $s(v) > f(x_i)$. If $s(v) < f(x_{i-1})$, by nestedness of $\mathrm{Idle}(\sigma)$, v is covered by x_{i-1}. Therefore $v \subset u$. If $s(v) > f(x_{i-1})$ then v and u are compatible. □

Case 3: $|L| = 1$**, but** $|R| \geq 1$

This case is symmetric to the previous case. So, we leave the details to the reader.

Case 4: $|L| \geq 1$ **and** $|R| \geq 1$

We reschedule the intervals in two passes. We start as in the Case 2. Namely, for every machine $\sigma(x_i)$ we move the intervals $d \succ \sigma(x_i)$ to the machine $\sigma(x_{i+1})$ if $1 \leq i < \ell$, and to the machine $\sigma(x_1)$, if $i = \ell$. Denote the resulted schedule by σ_1. Note that all the idle intervals in R except y_r are preserved. The interval y_r is changed to $y'_r = [s(x_\ell), f(y_r)]$. This interval now intersects with all other intervals in R. To restore the nestedness, we move all the intervals $d \succ y_i$ of the machine $\sigma_1(y_i)$ to the machine $\sigma_1(y_{i-1})$ if $1 < i \leq r$, and to the machine $\sigma(y'_r)$ if $i = 1$. Denote the final schedule by σ_4. We give an example of the rescheduling in Figure 9, leaving the formal description of σ_4 to the reader.

Lemma 5 *The scheduling σ_4 is nested*

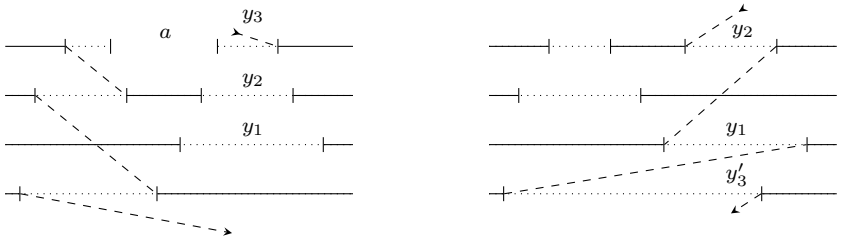

Fig. 9 Rescheduling after deletion of an interval a.

Proof Let K be a set of idle intervals that are start after$s(x_1)$ and finish before $f(y_1)$. Then the set of idle intervals $\text{Idle}(\sigma_4) \setminus K$ is exactly the set $\text{Idle}(\sigma) \setminus K$. Therefore $\text{Idle}(\sigma_4) \setminus K$ is nested. We show that K is also a nested set.

Let u and v be two idle intervals in K. If u, v are both old then, by the nestedness of $\text{Idle}(\sigma)$, they are compatible or one covers the other. If u, v are both new then, by construction of σ_4, they are compatible or one covers the other. So suppose u is new and v is old.

- $u = [s(x_1), f(y_1)]$. In this case, u is the longest interval in K. Therefore u covers v.
- u is one of the changed intervals in L. That is, $u = [s(x_i), f(x_{i_1})]$ for some $1 < i \leq \ell$. For contradiction, assume that u and v intersect. Then v contains either $s(x_i)$ or $f(x_{i-1})$. Moreover, since $v \notin L$, $s(a) \notin v$. In other words, v starts and finishes before or after the point $s(a)$. Therefore, v intersects with either x_i or x_{i-1}, which contradicts with the fact that $\text{Idle}(\sigma)$ is nested.
- u is one of the changed intervals in R. That is, $u = [s(y_{i-1}), f(y_i)]$ for some $1 < i \leq r$. Similarly, assume for contradiction that u and v intersect. Then v contains $s(f_i)$ and the endpoint $f(v)$ is between $f(a)$ and $s(y_i)$, or v contains $f(y_{i-1})$ and the start point is between $f(a)$ and $f(y_{i-1})$. Either case contradicts with the nestedness of $\text{Idle}(\sigma)$.

This proves that K is a nested set. By construction of K, an interval from K is either compatible with or covered by an interval in $\text{Idle}(\sigma_4) \setminus K$. Hence $\text{Idle}(\sigma_4)$ is a nested set of idle intervals. □

3 Tight Complexity Bound for Nested Scheduling

While various data structures [6,3] can be used for maintaining a set of nested intervals, they are not optimal in maintaining the nested schedule. Recall that a nested schedule depends on the set of interval I. Therefore when we insert or delete an interval from I, we need to update $O(d)$ intervals in a nested schedule.

This section contains three subsections. In the first subsection we show that maintaining nested scheduling as a set of a self-balancing tree requires

$O(d \cdot \log n)$ time for update operations. In the second subsection we improve the complexity of update operations to $O(d + \log n)$. Finally, in the third subsection we show the $O(d + \log n)$ bound to be tight for any data structure representing nested schedules.

3.1 Straightforward Implementation

We maintain a nested schedule σ of a set of intervals by maintaining the nested tree of σ. We assume that every endpoint is linked to two corresponding intervals, real and idle. That is if a schedule contains intervals $[3, 6]$ and $[8, 9]$ subsequently, we have a direct access from the potion 6 to the real interval $[3, 6]$ and to the idle interval $[6, 8]$.

The straightforward implementation of a nested tree is denoted by T. The nodes of T are idle intervals. The root of T is the interval $[-\infty, \infty]$. The children of a node v is a set $Sub(v)$ of intervals u that are directly covered by v, i.e. there is no interval w such that $v \supset w \supset u$. The children are ordered from left to right by the starting times. The set $Sub(v)$ is represented as a self-balancing binary search tree, that supports join and split operations. The root of $Sub(v_{i+1})$ keeps a pointer to its parent v_i in T.

First we describe three auxiliary operations: intersect, shortenLeft and shortenRight. The *intersect* operation returns a path in the tree T consisting of idle intervals intersecting a given point q. The operation *shortenLeft* set the starting time of a given idle interval i to the new value. We assume that a new value is between $s(i)$ and $f(i)$. Thus the interval i becomes shorter at the left end. The operation *shortenRight* is similar to shortenLeft operation, but it changes the finishing time of an idle interval.

For the intersect operation, we start at the root and at every node v_i we perform a search in $Sub(v_i)$ for a child v_{i+1} that contains q. Since the children are stored in a binary search tree, the search takes $O(\log n)$ time. Note that by the nestedness property, there is at most one such child. If we found one, we add it to the path and continue in the subtree of v_{i+1}. Otherwise, we stop and return the constructed path of intervals. Note that q defines a unique path in T. As the height of T is at most d, the time complexity of the search is $O(d \log n)$.

For the shortenLeft and shortenRight operations, after shortening an idle interval, we need to update the children of the updated interval. Let v be an idle interval and p be new value. We split the children of v into two sets A and B. The set A contains idle intervals in $Sub(v)$ whose finishing time is less that p, and the set B contains idle intervals whose starting time is greater that p. We assume that p does not intersect any of the children of v. Therefore $A \cup B = Sub(v)$. If the interval has been shortened at the left, we add intervals in A to the parent of v and set B as the children of the update intervals. If the interval has been shortened to the right, A becomes the children of v and B is merged with the children of v's parent. Shortening operation takes $O(\log n)$

time. An example in Figure 10 shows the result of applying *shortenRight* to the interval x_i.

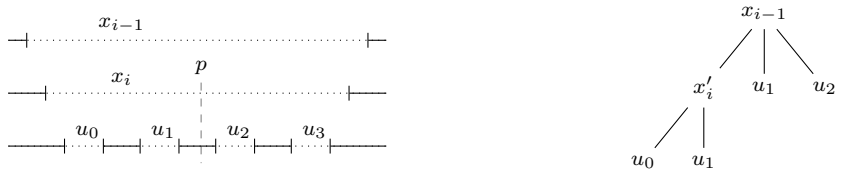

Fig. 10 Changes in the nested tree after applying *shortenRight*(x_i, p).

Now we are ready to describe insertion and deletion of intervals. Let a be the inserted interval. Let $P_L = \{v_0, \ldots, v_k = z, x_1, \ldots, x_\ell\}$ and $P_R = \{v_0, \ldots, v_k = z, y_1, \ldots, y_r\}$ be the paths returned by *intersect*$(s(a))$ and *intersect*$(f(a))$, respectively. Following the construction of σ_2 in the previous section, we shorten the interval x_ℓ at the right to the point $s(a)$. Then, we shorten each idle interval x_i at the right to the point $f(x_{i+1})$. Similarly, we shorten intervals $y_r, y_{r-1}, \ldots, y_1$ at the left to the points $f(a), s(y_r), \ldots, s(y_2)$.

Finally, we split the interval z into two intervals $z_1 = [s(z), f(x_1)]$ and $z_2 = [s(y_1), f(z)]$. If x_1 or y_1 do not exist, we set $z_1 = [s(z), s(a)]$ and $z_2 = [f(z), f(a)]$. These two intervals will be new nodes in T. We split the children of z as well. We set the children of z_1 and z_2 to be those intervals in $Sub(z)$ that finish before $f(x_1)$ and start after $s(y_1)$, respectively. We delete node z from $Sub(v_{k-1})$ and in its place we insert, preserving order, intervals z_1, z_2 and the intervals in $Sub(z)$ not covered by z_1 or z_2.

Now we describe the deletion operation. Let a be a deleted interval, P_L and P_R be the paths returned by *intersect*$(s(a))$ and *intersect*$(f(a))$, respectively. Note that the $x_\ell \in P_L$ and $y_r \in P_R$ are idle intervals adjacent to a.

First, we delete idle intervals x_ℓ and y_r from T. We move the children of these intervals to the children of their parents. Then for every $1 \leq i < \ell$ we shorten x_i at the left to $s(x_{i+1})$. Similarly, we shorten intervals $y_{r-1}, \ldots, y_1$ at the right to $f(y_r), \ldots, f(y_2)$, respectively. We also add children of y_r to the children of y_{r-1}.

Finally, we add the new idle interval $b = [s(x_1), f(y_1)]$. We add b as a child of v_k, the interval that covers both x_1 and y_1. We search for the intervals in $Sub(v_k)$ that are covered by b. We remove these intervals from $Sub(v_k)$ and set them to be children of b.

Theorem 2 *The data structure described above maintains the optimal scheduling and supports insertions and deletions in $O(d \cdot \log n)$ worst-case time.*

Proof When we update or delete an interval, we change idle intervals that intersect with at most two points. It takes $O(d \log n)$ time to find these idle intervals. Once found, we change endpoints of every idle interval. To change

endpoint of an idle interval in T takes $O(\log n)$ time, since we need to split and join children of the interval and its parent. We change endpoints of at most $2d$ intervals. Thus in total an update operation take $O(d \log n)$ time. The correctness of the operations follows from Lemma 4 and Lemma 5 □

3.2 Optimal Data Structure

Our data structure stores idle intervals $Idle(\sigma)$ that depends on the scheduling function σ and the set of intervals I. We assume that every endpoint is linked to two corresponding intervals, real and idle. That is if a schedule contains intervals $[3, 6]$ and $[8, 9]$ subsequently, we store the idle interval $[6, 8]$. After each update of I we restore the nestedness of the idle interval set. Therefore when we insert or delete an interval a, we update all idle intervals that intersect with the endpoints of a. Below we describe how maintain a nested schedule and perform update operations in $O(d + \log n)$ worst-case time.

We store idle intervals in an *interval tree* [14]. An interval tree is a leaf-oriented binary search tree where leaves store endpoints of the intervals in increasing order. Intervals themselves are stored in the internal nodes as follows. For each internal node v the set $I(v)$ consists of intervals that contain the *split point* of v and are covered by the *range* of v. The split point of v, denoted by $split(v)$, is a number such that the leaves of the left subtree of v store endpoints smaller than $split(v)$, and the leaves of the right subtree of v store endpoints greater than $split(v)$. The range of v, denoted by $range(v)$, is defined recursively as follows. The range of the root is $(-\infty, \infty]$. For a node v, where $range(v) = (l, r]$, the range of the left child of v is $(l, split(v)]$, and the range of the right child of v is $(split(v), r]$. An example of an interval tree is shown in Figure 11.

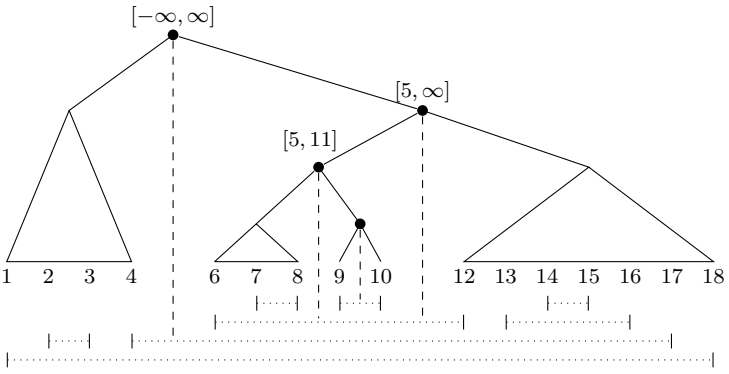

Fig. 11 Nested set of intervals represented by interval tree data structure.

We represent each set $I(v)$ as a linked list. The intervals in $I(v)$ are stored in order of their left endpoints. Since the set is nested, every interval in a list

covers all the subsequent intervals in the list. To search for all intervals intersecting a given point p do the following. Start at the root and visit the nodes $v_0, \ldots, v_k$, where v_{i+1} is the right child of v_i if $p > \mathit{split}(v_i)$, and the left child otherwise. At every node v_i, scan $I(v_i)$ and report all intervals containing p. Note that p intersects with at most d intervals and the length of the path is $O(\log n)$. Thus the search takes $O(d + \log n)$ time.

To allow updates of the interval tree, we represent it as a red-black tree. In a red-black tree, insertion or deletion of a node takes $O(\log n)$ time plus the time for at most 3 rotations to restore the balance. When performing a rotation around an edge $(v, p(v))$ the sets $I(v)$ and $I(p(v))$ change. Let the range of $p(v)$ be $(\ell, r]$. If v is the left child, the range of $p(v)$ after rotation becomes $[\mathit{split}(v), r]$. If v is the right child, the range of p shortens at the other end and becomes $[\ell, \mathit{split}(v)]$. Therefore all intervals in $I(p(v))$ that intersects with $\mathit{split}(v)$ must be moved to $I(v)$. Note that ranges of other nodes are not affected. Since there are at most d intervals in each of the internal interval sets, rotation takes $O(d)$ time. Thus in total we need $O(d + \log n)$ time to insert or delete a node.

Now we describe the update operations. Let a be the inserted interval. Recall that when we insert an interval a, we need to update idle intervals that intersect with the endpoints of a. Let L be the set of idle intervals that contain $s(a)$, but not $f(a)$. Let R be the set of idle intervals that contain $f(a)$, but not $s(a)$. Let z be the shortest idle interval that contains both endpoints of a. We show how to update intervals in L. The update of intervals in R is similar.

Let v_0 be a node such that $z \in I(v_0)$. This node is our starting position. To find intervals in L, we walk down a path $v_0, \ldots, v_k$ defined by $s(a)$. When visiting a node v_i, we iterate through $I(v_i)$ and put intervals that contains $s(a)$ into L. We delete intervals from $I(v_i)$ that we put in L. We stop when we reach a leaf node.

Let $x_1 \supset \cdots \supset x_\ell$ be intervals we have put in L. We iterate through L and walk up the path we have traversed. We start iteration from the last interval x_ℓ. For an interval x_j, we set $s(x_j) = s(x_{j-1})$. Then we check if x_j belongs to $I(v_i)$, i.e. if $\mathit{split}(v_i) \in x_j \subset \mathit{range}(v_i)$. If x_j satisfies these conditions, we put x_j at the beginning of $I(v_i)$ and remove it from L . Otherwise, we walk up the path until we find a node with a satisfactory split point and range. Note that no interval in $I(v_i)$ contains $s(a)$, since on the way down we removed all such intervals. Therefore, by the nestedness of idle intervals, x_j covers all intervals in $I(v_i)$.

Finally, we insert $s(a)$ into the tree. Once inserted, we search for the lowest common ancestor v of the leaves containing $s(x_\ell)$ and $s(a)$. We add interval $[s(x_\ell), s(a)]$ into $I(v)$.

The deletion of an interval a is similar to insertion. First we delete the intervals x_ℓ and y_r and the endpoints $s(a)$ and $f(a)$ from the interval tree. Then we traverse the path defined by $s(a)$. Recall that $x_{\ell-1}, \ldots, x_1$ are the idle intervals that intersect with $s(a)$. We change the starting time of all of them. Suppose v is a node such that $x_i \in I(v)$. Clearly, x_{i+1} is in the $\mathit{range}(v)$.

For every interval x_i we encountered we set $s(x_i)$ to be $s(x_{i+1})$. We move the changed intervals x'_i to the node v such that $split(v) \in x'_i \subset range(v)$. Note that v is on the path we are traversing. Similarly, we traverse the path defined by $f(a)$, update and move intervals y_i. Finally, we add a new idle interval $[s(x_1), f(y_1)]$ into the interval tree.

Theorem 2 *The data structure described above maintains the optimal scheduling and supports insertions and deletions in $O(d + \log n)$ worst-case time.*

Proof When we insert or delete an interval, we update only two sets L and R of idle intervals. These two sets corresponds to two paths of length at most $O(\log n)$. Furthermore, all intervals in each set share a common point. Therefore the size of each set is at most d. Since the intervals in internal nodes are ordered, it takes $O(d)$ time to add intervals into L and R. When we put updated intervals back, we add them at the beginning of the lists. Therefore it takes $O(d)$ time to add intervals from L and R into the internal nodes. Finally, we insert or delete at most two leaves. Thus, an update takes $O(d+\log n)$ time.

The optimality of scheduling follows from Lemma 4 and Lemma 5. □

3.3 Lower Bound

In this subsection we show that complexity of any data structure that maintains a nested tree is at least $\Omega(\log n + d)$, where d is the height of the nested tree. First we recall a lower bound for the static interval scheduling problem:

Theorem 3 (Shamos and Hoey [15]) *$\Omega(n \log n)$ is a lower bound on the time required to determine if n intervals on a line are pairwise disjoint.*

Lemma 6 *$\Omega(\log n)$ is a tight bound on the time required to update a data structure that maintains a nested tree.*

Proof For contradiction, assume that there is a data structure with a complexity $f(n) \in o(\log n)$. We create a nested tree of n intervals using this data structure. If the height of the tree is 1, then the intervals do not intersect. However, the time taken is $n \cdot f(n) \in o(n \log n)$, which contradicts Theorem 3. □

Lemma 7 *If σ and τ are nested scheduling functions then* $\mathrm{Idle}(\sigma) = \mathrm{Idle}(\tau)$.

Proof For contradiction, assume that there exist two nested scheduling functions σ and τ such that $\mathrm{Idle}(\sigma) \neq \mathrm{Idle}(\tau)$. Then there exist two idle intervals $a_0 \in \mathrm{Idle}(\sigma)$ and $b_0 \in \mathrm{Idle}(\tau)$ such that they have the same non-infinite starting time, but different finishing times, i.e. $s(a_0) = s(b_0) \neq -\infty$ and $f(a_0) \neq f(b_0)$. Without loss of generality, suppose that $f(a_0) < f(b_0)$. Now we take an interval b_1 from $\mathrm{Idle}(\tau)$ that finishes at $f(a_0)$. If its starting time is less than $s(b_0)$ then intervals b_0 and b_1 overlap, which contradicts the nestedness of τ. Otherwise, we continue to $\mathrm{Idle}(\sigma)$ and take an interval a_1 that starts at $s(b_1)$.

If $f(a_1) > f(a_0)$ then a_1 and a_0 overlap and it is a contradiction. Otherwise, we continue in the same manner to Idle(τ). Since I is finite, this process eventually stops and one of the scheduling functions appears to be not nested. □

Theorem 4 *An update operation in a data structure representing a nested tree takes at least $\Omega(\log n + d)$ time.*

Proof Let I be an interval set and Nest(I) be the nested tree of I. By Lemma 7, Nest(I) is unique. Let $v_0 v_1 \dots v_d$ be longest path in Nest(I). Now consider an interval a, which starts in the middle of v_d and finishes after the end of v_1. Clearly, $s(a)$ intersects with exactly d idle intervals. Therefore the trees Nest(I) and Nest($I \cup a$) differ in $\Omega(d)$ nodes. Taking into account Lemma 6, an update operation of a nested tree requires $\Omega(\log n + d)$ time. □

References

1. Arkin, E. M., & Silverberg, E. B. (1987). Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1), 1-8.
2. Diedrich, F., Jansen, K., Pradel, L., Schwarz, U. M., & Svensson, O. (2012). Tight approximation algorithms for scheduling with fixed jobs and nonavailability. *ACM Transactions on Algorithms (TALG)*, 8(3), 27.
3. Gavruskin A., Khoussainov B., Kokho M. & Liu J. (2013). Dynamising Interval Scheduling: The Monotonic Case. *Combinatorial Algorithms* (pp. 178-191). Springer Berlin Heidelberg.
4. Gertsbakh, I., & Stern, H. I. (1978). Minimal resources for fixed and variable job schedules. Operations Research, 26(1), 68-85.
5. Gupta, U. I., Lee, D. T., & Leung, J. T. (1979). An optimal solution for the channel-assignment problem. *Computers, IEEE Transactions on*, 100(11), 807-810.
6. Kaplan, H., Molad, E., & Tarjan, R. E. (2003). Dynamic rectangular intersection with priorities. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing* (pp. 639-648). ACM.
7. Kleinberg, J., & Tardos, E. *Algorithm design*. Pearson Education India, 2006.
8. Kolen, A. W., & Kroon, L. G. (1991). On the computational complexity of (maximum) class scheduling. *European Journal of Operational Research*, 54(1), 23-38.
9. Kolen, A. W., & Kroon, L. G. (1994). An analysis of shift class design problems. *European Journal of Operational Research*, 79(3), 417-430.
10. Kolen, A. W., Lenstra, J. K., Papadimitriou, C. H., and Spieksma, F. C. (2007). Interval scheduling: A survey. *Naval Research Logistics (NRL)*, 54(5), 530-543.
11. Kovalyov, M. Y., Ng, C. T., & Cheng, T. C. (2007). Fixed interval scheduling: Models, applications, computational complexity and algorithms. *European Journal of Operational Research*, 178(2), 331-342.
12. Kroon, L. G., Salomon, M., & Van Wassenhove, L. N. (1995). Exact and approximation algorithms for the operational fixed interval scheduling problem. *European Journal of Operational Research*, 82(1), 190-205.
13. Kroon, L. G., Salomon, M., & Van Wassenhove, L. N. (1997). Exact and approximation algorithms for the tactical fixed interval scheduling problem. *Operations Research*, 45(4), 624-638.
14. Mehlhorn, K. *Data structures and algorithms, Volume 3: Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.
15. Shamos, M. I., & Hoey, D. (1976). Geometric intersection problems. *Foundations of Computer Science*, 1976., 17th Annual Symposium on (pp. 208-215). IEEE.
16. Sleator, D. & Tarjan, R. (1983) A Data Structure for Dynamic Trees, *Journal of computer and system sciences*, 26, 3, 362–391.
17. Spieksma, F. C. (1999). On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2(5), 215-227.