Algorithmica (2016) 76:1160–1180
DOI 10.1007/s00453-016-0148-5

Dynamic Algorithms for Multimachine Interval Scheduling Through Analysis of Idle Intervals

Alex Gavryushkin[1] · **Bakhadyr Khoussainov**[1] · **Mikhail Kokho**[1] · **Jiamou Liu**[2]

Received: 25 January 2015 / Accepted: 15 April 2016 / Published online: 27 April 2016

Abstract We study the dynamic scheduling problem for jobs with fixed start and end times on multiple machines. The problem is to design efficient data structures that support 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. We prove that any such schedule is optimal. Based on this result, we provide a data structure that maintains the updates the optimal schedule in $O(d + \log n)$ worst-case time, where d is the depth of the set of idle intervals and n is the number of jobs. Furthermore, we show this bound is tight.

Keywords Interval scheduling · Fixed job scheduling · 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.

✉ Mikhail Kokho
m.kokho@auckland.ac.nz

1 Centre for Computational Evolution, The University of Auckland, Auckland, New Zealand

2 Department of Computer Science, The University of Auckland, Auckland, New Zealand

The example above is a basic setup for the interval scheduling problem, one of the well-known problems in the theory of scheduling [12,13]. To be more formal, we bring in some easy definitions and notations. An interval a is the usual closed non-empty interval $[s(a), f(a)]$ on the real line $\mathbb{R}$, where $s(a)$ is the starting time and $f(a)$ is the finishing time. Here recall that a closed interval $[x, y]$ consist of all $t \in \mathbb{R}$ such that $x \leq t \leq y$. We consider *non-trivial intervals*, that is, interval a such that $s(a) < f(a)$. Naturally, closed intervals represent jobs that have to be completed in the specified periods. In this paper we often use open intervals; these are of the form (x, y) that consist of all t such that $x < t < y$.

Definition 1 We say that two intervals a and b *overlap* if the cardinality of the intersection $a \cap b$ is greater than 1; otherwise we say that they are *compatible*. Also, if $a \subset b$ then we say that b *covers* a.

Formally, we describe the scheduling problem as follows. We are given a set I of n intervals (representing jobs). The problem is to partition the set 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 small as possible. The partition of jobs represent schedules for the machines. An important definition is the following:

Definition 2 A subset J of the set I of intervals 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 $a, b \in I$. One small issue here is that it might be the case that $s(a) = s(b)$ for $a \neq b$. This is not a problem. One way to ensure that $\prec$ strictly orders the pairwise distinct intervals is this. If $a \neq b$ and $s(a) = s(b)$ then we set $a \prec b$ if $f(a) < f(b)$. However, for this paper, to simplify the presentation, we assume that left endpoints of all intervals are pairwise distinct.

Definition 3 A *scheduling function* for the set of intervals I is a function $\sigma : I \rightarrow \{1, \ldots, k\}$ 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.

Note that the scheduling function $\sigma : I \rightarrow \{1, \ldots, k\}$ 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 4 We call a scheduling function σ *optimal* if its size is $d(I)$.

We briefly describe the standard greedy algorithm solving the basic interval scheduling problem. The algorithm [9], which we call *Greedy*, 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 $a_1, a_2, \ldots, a_n$ 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 [9]. The naive implementation of the algorithm runs in $O(n^2)$ time.

There are many efficient implementations of *Greedy* algorithm. For instance, Gupta et al. [5] design an optimal schedule that runs in $O(n \log n)$ time. They show that their implementation is the best possible. In their implementation, one works 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. Note that if endpoints of the intervals are sorted, the algorithm takes $O(n)$ 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.

1.1 The Problem Setup

In the dynamic context, the instances of the interval scheduling problem are usually changed by real-time events, and a previously optimal schedule may become suboptimal. Examples of such real-time events include job cancellation, the arrival of an urgent job, and changes in job processing times. The *dynamic interval scheduling problem* is the problem of maintaining an optimal number of machines for a set of closed intervals, subject to update operations of insert and delete. Below we explain this in more detail.

Imagine that there is an agent that requests new jobs to be processed or some jobs to be deleted from the schedule. There is a scheduler whose goal is to satisfy the agent. In addition, the scheduler should

1. Perform the insert and delete operations efficiently.
2. Maintain the optimal number of machines (resources) to process the jobs.

In this paper, we address both (1) and (2). With respect to (1), we design efficient data structures that represent scheduling and support the following update operations:

– 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).

With respect to (2), our data structures always maintain the optimal number of processes. Both goals (1) and (2) are achieved through the following observation. From the agent and the scheduler's view points all machines are equal in terms of various parameters, e.g. speed, capacity, memory. Hence, the agent and the scheduler are satisfied as long as the jobs are processed. Also, the scheduler is concerned with just (1) and (2). Thus, for both the agent and the scheduler it is not so crucial to know the identity of the machine that process any given job. More formally, it is not too important to know if a given job a is processed by machine i as long as (1) and (2) above are satisfied. We note that our data structure of course allows one to check if a is processed by machine i. This, however requires the usual $O(n)$ time, where n is the cardinality of I.

As a simplified example, consider a dynamic scheduling of virtual machines in a cloud host server. The host provides virtual machines for users. The user specifies the start and the end times during which a virtual machine is occupied. This time period can be viewed as a closed interval. The host handles these intervals (requests) by providing the users dedicated virtual machines. When several such intervals overlap, the computing capacity of the host is shared among all the virtual machines. In order to efficiently use the resources of the cloud host, it is desirable that the host utilises the smallest number of virtual machines. Since requests can come and go in a dynamic manner, one would like to optimise the number of virtual machines, while allowing insertion and deletion operations. To solve this problem the host does not need to explicitly know virtual machines assigned to the users. The important issue for the host is an efficient use of the resources which directly relates to optimising the number of virtual machines.

1.2 Contributions and Related Work

The paper belongs to the area of scheduling, and investigates the problem of dynamically maintaining a schedule for a set of intervals under the insertion and deletion operation. There are four 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. 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. We prove in Theorem 1 that nested schedules are always optimal. Here we note that Diedrich et al. use idle intervals in [2], where they call them *gaps*, to design approximation 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.
- The second contribution is that we provide methods that preserve nested schedules and support insert and delete operations. This is based on Theorem 2 that proves that there are always optimal schedules for which the set of idle intervals is nested. This theorem allow us to represent idle intervals of the schedule as a tree, and perform the update operations through maintaining the idle intervals of the schedule.
- The third contribution is that we design a data structure that maintains nested set of idle intervals as a tree. Theorem 4 proves that all the update operations run in

$O(d + log(n))$ in the worst-case. Note that if we naively make *Greedy* dynamic, the update operations of such algorithms will be significantly slower.

- Finally, the next contribution is that we prove in Theorem 6 that the bound $O(d + log(n))$ is tight for any data structure representing nested schedules.

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 [12,13]. Gertsbakh and Stern [4] studied the basic problem of scheduling intervals on unlimited number of identical machines. Arkin and Silverberg [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 maximise the value of completed jobs. A further generalisation of the problem, motivated by maintenance of aircrafts, was extensively studied by Kroon et al. [14,15] and by Kolen and Kroon [10,11]. In this generalisation, 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 [19] studied the question of approximating generalised interval scheduling problem.

Note that in our set-up, there is no a priori bound on the number of intervals (jobs) and machines. In this sense, it makes sense to develop a universal mechanism for scheduling infinitely many intervals. We plan to develop such a mechanism using finite automata based on the ideas in [7] and [8].

2 Idle Intervals and Nested Schedules

2.1 Idle Intervals

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

Definition 5 Let $J = \{a_1, a_2, \dots, a_m\}$ be a compatible set of closed intervals such that $a_i \prec a_{i+1}$ for each $i \in \{1, \dots, m-1\}$. Define the set of *idle intervals of J* as the following set

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

Clearly, compatibility assumption on J is important. We note that an idle interval can start at $-\infty$ or end at ∞. Intuitively, idle intervals represent a period of time when a machine is continuously available to process new jobs. We point out that all idle intervals are open intervals.

Let $\sigma : I \to \{1, \dots, k\}$ be a scheduling function of size k of the set I of intervals. Recall our definition of sets $S_1, \dots, S_k$ 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 $S_1, \dots, S_k$. Therefore, we need to have a mechanism that efficiently maintains the idle intervals of $S_1, \dots, S_k$.

Definition 6 The set of *idle intervals* of the scheduling function $\sigma : I \to \{1, \dots, k\}$ is

$$\text{Idle}(\sigma) = \{(-\infty, \infty)\} \cup \text{Idle}(S_1) \cup \text{Idle}(S_2) \cup \cdots \cup \text{Idle}(S_k).$$

Through the scheduling function σ, we can enumerate the set of idle intervals:

Definition 7 For a scheduling function $\sigma : I \to \{1, \dots, k\}$, the *schedule number* $\sigma(b)$ of the idle interval $b \in \text{Idle}(\sigma)$ is i if $b \in \text{Idle}(S_i)$, 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 *Let $\sigma : I \to \{1, \dots, k\}$ be schedule. Then $d(I) \leq k \leq d(\text{Idle}(\sigma)) - 1$.*

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

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

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

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

For the second part, let a_i be the $\prec$–least interval in the schedule S_i. Then for each $i \in \{1, \dots, k\}$ an interval $[-\infty, s(a_i)]$ belongs to $\text{Idle}(\sigma)$. 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 overlaps with k intervals in $\text{Idle}(\sigma)$. In addition, x belongs to the interval $(-\infty, +\infty)$. Hence, $k \leq d(\text{Idle}(\sigma)) - 1$. □

Recall that open intervals are denoted by (x, y). Here is a key definition:

Definition 8 A set J of open 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 $\text{Nest}(J)$. Thus the root of the tree is the interval $(-\infty, \infty)$. We order siblings of any given node (interval) in $\text{Nest}(J)$ by their left endpoints. 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 the height of the nested tree* $\mathrm{Nest}(J)$ *plus 1.*

Proof Let J be a nested set of intervals and h be the height of the nested tree $\mathrm{Nest}(J)$. To show that $h + 1 \leq d(J)$, 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 any point that belongs to b_h also belongs to the intervals $b_0, \ldots, b_{h-1}$. Hence $h + 1 \leq d(J)$.

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 $d(J) \leq h + 1$. □

2.2 Nested Schedules

In this subsection we connect idle interval sets with the nested trees through the following definition:

Definition 9 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 result shows the usefulness of the notion of nested schedules. In particular, nested schedules are optimal.

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

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

Take a longest path $b_0, b_1, \ldots, b_h$ in $\mathrm{Nest}(\mathrm{Idle}(\sigma))$. Note that $b_0 = (-\infty, +\infty)$. Assume that $f(b_1) \neq +\infty$. If $f(b_1) = +\infty$ then $s(b_1) \neq -\infty$ and the reasoning below can be applied to $s(b_1)$ instead of $f(b_1)$. For each $i \in \{1, \ldots, h\}$ consider the schedule $S_{t(i)}$ such that b_i belongs $\mathrm{Idle}(S_i)$. We claim that in every schedule $S_{t(i)}$ there exists an interval $a_i \in S_i$ such that $f(b_1)$ belongs to a_i. This will prove $d(\mathrm{Idle}(\sigma)) - 1 \leq d(I)$.

To arrive at a contradiction, assume that there exists a schedule $S_{t(j)}$ such that the point $f(b_1)$ does not belong to any interval in $S_{t(j)}$. Then there exists an idle interval c in the set $\mathrm{Idle}(S_j)$ such that $f(b_1) \in c$. Therefore $s(c) < f(b_1)$. On the other hand, since b_j belongs to $\mathrm{Idle}(S_{t(j)})$, we have $s(b_1) < f(b_j) < s(c)$. This is because $b_j \subset b_1$, and b_j and c are idle intervals in $S_{t(j)}$. These imply that $s(b_1) < s(c) < f(b_1)$. Hence, we have $b_1 \cap c \neq \emptyset$ and $c \nsubseteq b_1$ and $b_1 \nsubseteq c$. This contradicts the fact that $\mathrm{Idle}(\sigma)$ is a nested schedule. Thus h is at most $d(I) + 1$. □

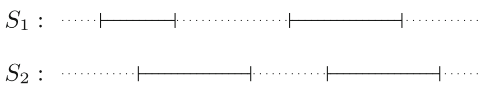

Fig. 1 *Dotted lines* define the idle interval set

A natural question is if the schedule constructed by the *Greedy* algorithm (explained in the introduction) is nested. The next simple example gives a negative answer to this question. Indeed, consider the set I of intervals presented in Fig. 1. The *Greedy* algorithm yields a scheduling which is not nested.

3 Preserving Nestedness After Insertion

One of the goals of this section is to prove that every interval set I possesses a nested schedule. 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.

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 $x_1 \supset \cdots \supset x_\ell$. 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 $y_1 \supset \cdots \supset y_r$. 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 $x_0 = y_0 = z$. These intervals do not belong to L and R. However $x_0 \supset x_1$ and $y_0 \supset y_1$ because of nestedness, if x_1 or y_1 exist.

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. In each case, we prove that the newly constructed schedule is nested.

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 $\sigma(a) = \sigma(z)$ if $z \neq (-\infty, +\infty)$. We show that the resulting schedule is nested.

After insertion of the interval a, the idle interval z is split into two intervals $z_\ell = (s(z), s(a))$ and $z_r = (f(a), f(z))$. In addition, if $z = (-\infty, +\infty)$ then we need to add $(-\infty, +\infty)$ to the set of new idle intervals. Consider an arbitrary idle interval u in the set $Idle(\sigma)$. 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 has no points in common with the new idle intervals. This is easily proved using the assumption that both L and R are empty. Thus the resulting set of idle intervals is nested. See Fig. 2.

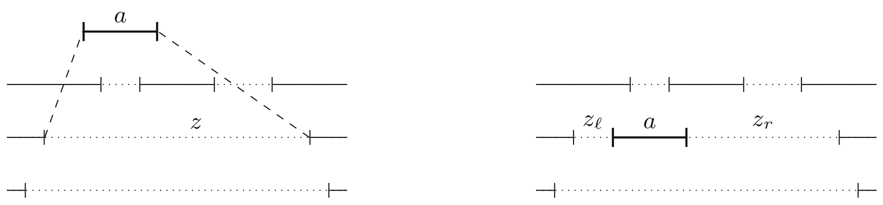

Fig. 2 Rescheduling when a does not overlap with idle intervals

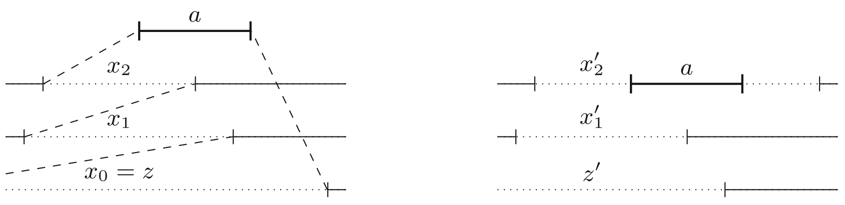

Fig. 3 Rescheduling when idle intervals overlapping a contains only $s(a)$

If $z = (-\infty, +\infty)$, we update the function by setting $\sigma(a) = d(I) + 1$ and $\sigma(z) = d(I) + 2$. It is not hard to see that nestedness is preserved.

Case 2: L **is not empty, but** $R = \emptyset$.
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, it might be the case that the new idle interval $(s(z), s(a))$ and x_1 will violate the nestedness property. Therefore we need to reorganise the schedule σ.

We schedule interval a for the machine $\sigma(x_\ell)$. We move all the jobs d of the machine $\sigma(x_\ell)$ such that $f(x_\ell) \leq s(d)$ to machine $\sigma(x_{\ell-1})$. In the schedule $S_{\sigma(x_{\ell-1})}$ there are other jobs that start after $f(x_{\ell-1})$. 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 $f(z) \leq s(d)$ 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 in Fig. 3.

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

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

Lemma 3 *The scheduling function* σ_1 *is nested.*

Proof The set of idle intervals Idle(σ_1) consists of the new interval $(f(a), f(z))$ together with all the idle intervals of σ where the idle intervals $x_\ell, x_{\ell-1}, \ldots, x_1$, and z are changed to the following new idle intervals $(s(x_\ell), s(a))$, $(s(x_{\ell-1}), f(x_\ell))$, …,

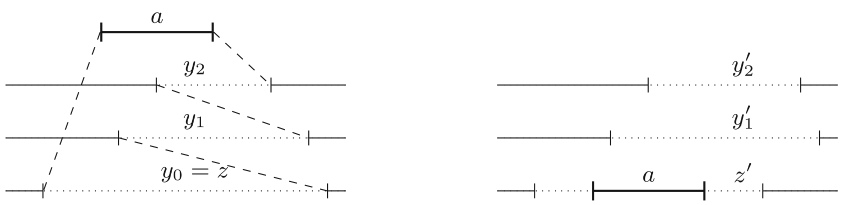

Fig. 4 Rescheduling when idle intervals overlapping a contains only $f(a)$

$(s(x_1), f(x_2))$, and $(s(z), f(x_1))$, respectively. In addition, if $z = (\infty, +\infty)$, then we again need to add the interval $(-\infty, +\infty)$ to the new set of idle intervals that we denote 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 has the empty intersection 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 $v \cap u = \emptyset$. If $s(v) > f(a)$ then, because Idle(σ) 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 $r \in x_i, r \notin x_\ell$ or $r \in x_\ell$. Hence, $v \subset x_i$ or $v \subset x_\ell$. If the first case we have $v \subset (s(x_i), s(a))$, and in the second case $v \subset (s(a), f(x_\ell))$. In either case, $v \subset (s(x_i), f(x_{i+1}))$. Hence, either $u \cap v = \emptyset$ or $v \subset u$. This proves the lemma. □

Case 3: The set R is not empty, but $L = \emptyset$.
This case is symmetric to the previous case: we need to reorganise the intervals, but we reorganise 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 overlaps with idle intervals $y_1, \ldots, y_r$. Therefore for every $1 \le i < r$ 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 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 in Fig. 4.

Formally, we define a new scheduling function σ_2 as follows:

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

To see that σ_2 is nested, consider two arbitrary idle intervals u and v in the idle set $Idle(\sigma)$. If these intervals have not been changed by the schedule reorganisation,

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

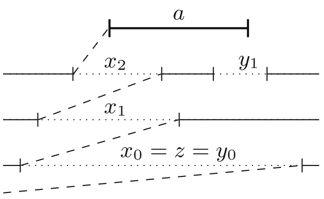

Fig. 5 The first step of rescheduling: defining σ_1

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

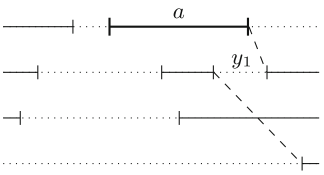

Fig. 6 The second step of rescheduling: defining σ_2

then these two intervals are either compatible or nested by the nestedness of $Idle(\sigma)$. If both of the intervals have been changed, then by construction of σ_2 these intervals are nested. If one of the intervals has been changed, say u, and the other has not, say v, then either $u \subset v$ or $v \subset u$ or they are compatible. We leave the details of this reasoning, which is similar to the previous case, to the reader.

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

We reorganise 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 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 $f(x_0) \le s(d)$ 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 resulting schedule, but it is not a nested schedule yet. A formal definition of σ_1 is given in Fig. 5.

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 $\sigma(y_{i+1})$ (see Fig. 6).

Lemma 4 *The scheduling function σ_2 is nested.*

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

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

By construction $\text{Idle}(\sigma_2)\backslash K = \text{Idle}(\sigma)\backslash K$, and by nestedness of $\text{Idle}(\sigma)$, $\text{Idle}(\sigma_2)\backslash K$ is also an nested set. Furthermore, it is clear that for any interval $p \in K$ and $q \in \text{Idle}(\sigma)\backslash K$, 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 (see Fig. 7).

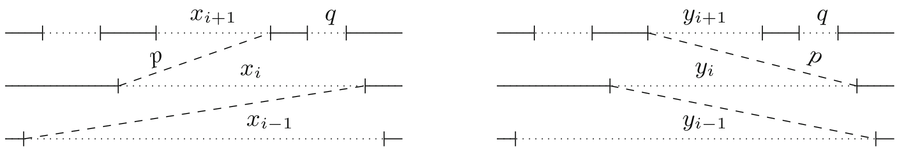

Fig. 7 Nestedness is preserved

Suppose p contains $s(a)$. Then the start of p is $s(x_i)$ for some $0 \leqslant i \leqslant \ell$. Moreover, the end of p is $s(a)$, if $i = \ell$, and $s(x_{i+1})$, 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 $0 \leqslant j \leqslant \ell$. 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 Idle(σ) is a nested set, we have that either $q \subset x_i$ or $q \subset x_{i+1}$. Hence $q \subset p$.

Now suppose p contains $f(a)$. Then the end of p is $f(y_i)$ for some $0 \leqslant i \leqslant r$, and the start of p is $f(a)$, if $i = r$, and $f(y_{i+1})$, otherwise. Consider two cases with respect to q:

- Case 1: q contains $f(a)$. Then the end of q is $f(y_j)$ for some $0 \leq j \leq r$. If $y_j \prec y_j$, then q covers p. Otherwise, q is covered by p.
- Case 2: q does not contain $f(a)$. 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 $r \in (f(y_{i-1}), f(a))$ or $r \in (f(a), f(y_i))$. Since Idle(σ) is a nested set, we have that either $q \subset y_{i-1}$ or $q \subset y$. Hence $q \subset p$.

Finally, suppose that neither p nor q contain $s(a)$ or $f(a)$. Then, by construction of σ_2, p and q are in Idle(σ). 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 2 *For any set of closed intervals I there is a scheduling function σ such that* Idle(σ) *is a nested set.*

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

3.1 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 follow the same notations as in the previous section: a denotes the deleted interval (instead of an interval to be inserted), L is the set of idle intervals that contains

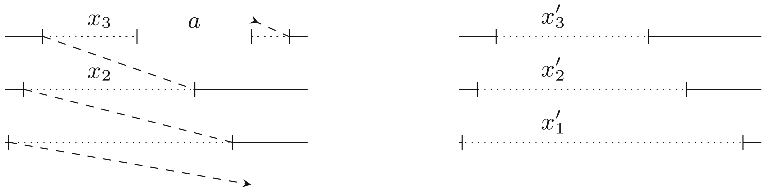

Fig. 8 Rotation of the schedules preserves the nestedness

$s(a)$, but not $f(a)$, and R is the sets of idle intervals that contains $f(a)$, but not $s(a)$, 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: $|L| = 1$ **and** $R = |1|$.
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 $y_r \subset v$, 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 $(s(x_\ell), f(y_r))$ has a non-empty intersection 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 $\sigma(x_1)$ to the machine $\sigma(x_2)$. Denote the new scheduling function by σ_3. An example of the rescheduling is shown in Fig. 8.

Lemma 5 *The scheduling function* σ_3 *is nested.*

Proof Let u and v be two idle intervals in $\text{Idle}(\sigma_1)$. 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 $u = (s(x_1), f(y_r))$. Moreover, by the nestedness of $\text{Idle}(\sigma)$, 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_1. 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 $x_i \in \text{Idle}(\sigma)$:

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

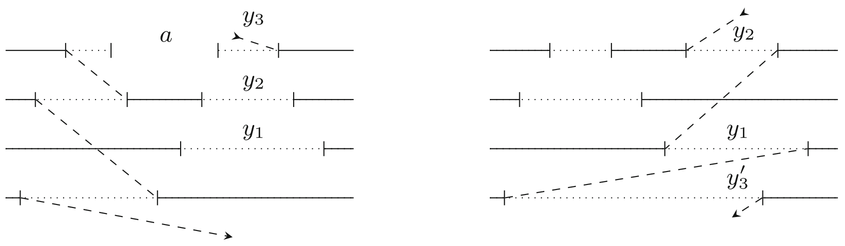

Fig. 9 Rescheduling after deletion of an interval a

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 $d \succ x_i$ to the machine $\sigma(x_{i+1})$ if $1 \leq i < \ell$, and to the machine $\sigma(x_1)$, if $i = \ell$. Denote the resulting schedule by σ_1. Note that all the idle intervals in R except y_r are preserved. The interval y_r is changed to $y_r' = (s(x_1), f(y_r))$. This interval now overlaps 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 $1 < i \leqslant r$, 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 6 *The scheduling function σ_4 is nested.*

Proof Let K be a set of idle intervals that start after $s(x_1)$ and finish before $f(y_1)$. Then the set of idle intervals $\text{Idle}(\sigma_4)\backslash K$ is exactly the set $\text{Idle}(\sigma)\backslash K$. Therefore $\text{Idle}(\sigma_4)\backslash K$ 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 overlap. 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 overlaps with either x_i or x_{i-1}, which contradicts 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 $1 < i \leq r$. For contradiction, assume that u and v overlap. That is, v either contains the left endpoint of u, but not its right endpoint, or v contains the right endpoint of u, but not its left endpoint. Because v is old and it does not cover u, then it does not include $f(a)$. Then v contains $s(y_i)$ 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 $f(y_{i-1})$. In other words, there exists two idle intervals in the old idle interval set that overlap. However, this contradicts the assumption the Idle(σ) is nested.

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 Idle(σ_4)$\backslash K$. Hence Idle(σ_4) is a nested set of idle intervals. □

4 Tight Complexity Bound for Nested Schedules

Our goal is to design data structures that store idle intervals in $Idle(\sigma)$ where σ is a nested schedule of a set I of closed intervals. The data structures naturally depend on the set of intervals I and the nested schedule σ. Although the data structures explicitly maintain idle intervals, we point out that our data structures allow easy access to intervals in I and the corresponding nested schedule σ: With every idle interval x, we keep two pointers to the closed intervals immediately to the left and to the right of x, respectively. For example, suppose I contains two intervals $[3, 6]$ and $[8, 9]$ which are assigned by σ to the same machine, the idle interval $(6, 8)$ has a pointer to $[3, 6]$ and a pointer to $[8, 9]$. Of course, if the idle interval starts at $-\infty$, then the left pointer is empty, and if the idle interval ends at $+\infty$, then the right pointer is empty. In this way the data structure is a representation of the nested schedule σ.

While various data structures [3,6] can be used for maintaining a set of nested intervals, they are not optimal in maintaining the nested schedule. This section contains three subsections. In the first subsection we show that maintaining nested schedules 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.

4.1 Straightforward Implementation

The straightforward implementation stores a nested tree of $Idle(\sigma)$, which 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 form 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 that overlap a given point q. The operation *shortenLeft* sets the starting time of a given idle interval i to the new value. We assume that the 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 the *shortenLeft* operation, but it changes the finishing time of an idle interval.

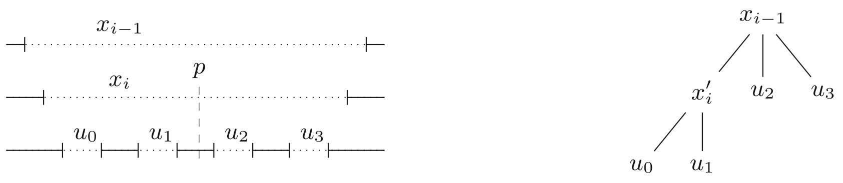

Fig. 10 Changes in the nested tree after applying *shortenRight*(x_i, p)

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 the 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 than p, and the set B contains idle intervals whose starting time is greater than p. We assume that p does not overlap 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 updated 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 Fig. 10 shows the result of applying *shortenRight* to the interval x_i.

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 $1 \le i < \ell$ 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 3 *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 overlap 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 takes $O(d \log n)$ time. The correctness of the operations follows from Lemmas 4 and 6 □

4.2 Optimal Data Structure

We present here another data structure for storing $Idle(\sigma)$. After each update of I we need to restore nestedness of the idle interval set. Therefore when we insert or delete an interval a, we update all idle intervals that overlap with the endpoints of a. Below we describe how to maintain an idle interval tree and perform update operations in $O(d + \log n)$ worst-case time.

We store idle intervals in an *interval tree* [16]. 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 Fig. 11.

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 overlapping a given point p, we 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 $p > split(v_i)$, and the left child of v_i otherwise. At every node v_i, scan $I(v_i)$ and report all intervals containing p. Note that p overlaps 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 (ℓ, r). If v is the left child, the range of $p(v)$ after rotation becomes $(split(v), r)$. If v is the right child, the range of p shortens at the other end and becomes $(\ell, split(v))$. Therefore all intervals in $I(p(v))$ that overlaps with $split(v)$ 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 overlap 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 $split(v_i) \in x_j \subset range(v_i)$. 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 overlap

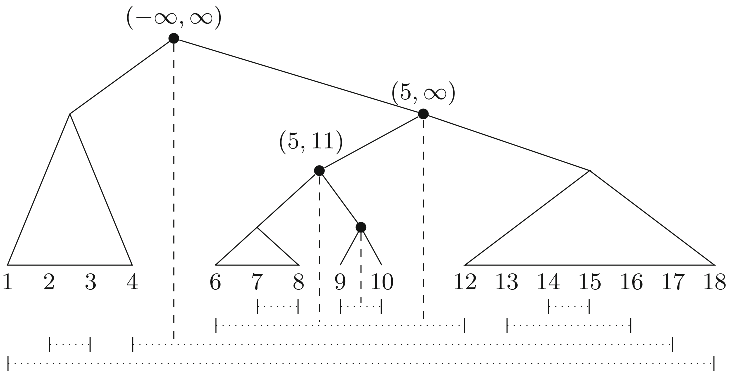

Fig. 11 The nested set of intervals represented by the interval tree data structure. As indicated by the *dotted lines*, the split of the root with range $(-\infty, +\infty)$ is 5; The split of the right child of the root, with range $(5, +\infty)$ is 11; The split of the internal node with range $(5, 11)$ is 8.5

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 $range(v)$. 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 4 *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 Lemmas 4 and 6. □

4.3 Lower Bound

In this subsection we show that the 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. More precisely this lower bound holds for any data structure that maintains two pieces of information:

(1) the depth of the set of idle intervals and
(1) each idle interval of the nested schedule explicitly

First we recall a lower bound for the static interval scheduling problem:

Theorem 5 (Shamos and Hoey [17]) $\Omega(n \log n)$ *is a lower bound on the time required to determine if* n *intervals on a line are pairwise disjoint.*

Lemma 7 $\Omega(\log n)$ *is a lower bound on the time required to update a data structure that maintains (1) and (2) above.*

Proof For contradiction, assume that there is a data structure with a complexity $f(n) \in o(\log n)$. We may iteratively insert n intervals to this data structure. The n intervals do not overlap if and only if in the resulting data structure the depth of the idle interval tree is 1. However, the time taken is $n \cdot f(n) \in o(n \log n)$, which contradicts Theorem 5. □

Lemma 8 *If* σ *and* τ *are nested scheduling functions for the same interval set* I*, then* $\text{Idle}(\sigma) = \text{Idle}(\tau)$.

Proof For contradiction, assume that there exist two nested scheduling functions σ and τ for I such that $\text{Idle}(\sigma) \neq \text{Idle}(\tau)$. Then there exist two idle intervals $a_0 \in \text{Idle}(\sigma)$

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 $\mathrm{Idle}(\tau)$. Since I is finite, this process eventually stops and one of the scheduling functions appears to be not nested. □

Theorem 6 *An update operation in a data structure maintaining (1) and (2) takes at least $\Omega(\log n + d)$ time.*

Proof Let I be an interval set and $\mathrm{Nest}(I)$ be the nested tree of I. By Lemma 8, $\mathrm{Nest}(I)$ is unique. Let $v_0 v_1 \dots v_d$ be the longest path in $\mathrm{Nest}(I)$. Now consider an interval a, which starts in the middle of v_d and finishes after the end of v_1. Clearly, $s(a)$ overlaps with exactly d idle intervals. Therefore the trees $\mathrm{Nest}(I)$ and $\mathrm{Nest}(I \cup a)$ differ in $\Omega(d)$ nodes. Taking into account Lemma 7, an update operation of this data structure requires $\Omega(\log n + d)$ time. □

References

1. Arkin, E.M., Silverberg, E.B.: Scheduling jobs with fixed start and end times. Discrete Appl.Math. **18**(1), 1–8 (1987)
2. Diedrich, F., Jansen, K., Pradel, L., Schwarz, U.M., Svensson, O.: Tight approximation algorithms for scheduling with fixed jobs and nonavailability. ACM Trans. Algorithms **8**(3), 27 (2012)
3. Gavruskin, A., Khoussainov, B., Kokho, M., Liu, J.: Dynamising Interval Scheduling: The Monotonic Case. Combinatorial Algorithms. Springer, Berlin (2013)
4. Gertsbakh, I., Stern, H.I.: Minimal resources for fixed and variable job schedules. Oper. Res. **26**(1), 68–85 (1978)
5. Gupta, U.I., Lee, D.T., Leung, J.T.: An optimal solution for the channel-assignment problem. IEEE Trans. Comput. **100**(11), 807–810 (1979)
6. Kaplan, H., Molad, E., Tarjan, R.E.: Dynamic rectangular intersection with priorities. In: Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, pp. 639–648. ACM (2003)
7. Kharlampovich, O., Khoussainov, B., Miasnikov, A.: From automatic structures to automatic groups. arXiv preprint arXiv:1107.3645 (2011)
8. Khoussainov, B., Nerode, A.: Open Questions in the Theory of Automatic Structures. Bull. EATCS, **94**, 181–204 (2008)
9. Kleinberg, J., Tardos, E.: Algorithm Design. Pearson Education India (2006)
10. Kolen, A.W., Kroon, L.G.: On the computational complexity of (maximum) class scheduling. Eur. J. Oper. Res. **54**(1), 23–38 (1991)
11. Kolen, A.W., Kroon, L.G.: An analysis of shift class design problems. Eur. J. Oper. Res. **79**(3), 417–430 (1994)
12. Kolen, A.W., Lenstra, J.K., Papadimitriou, C.H., Spieksma, F.C.: Interval scheduling: a survey. Nav. Res. Logist. **54**(5), 530–543 (2007)
13. Kovalyov, M.Y., Ng, C.T., Cheng, T.C.: Fixed interval scheduling: models, applications, computational complexity and algorithms. Eur. J. Oper. Res **178**(2), 331–342 (2007)
14. Kroon, L.G., Salomon, M., Van Wassenhove, L.N.: Exact and approximation algorithms for the operational fixed interval scheduling problem. Eur. J. Oper. Res. **82**(1), 190–205 (1995)
15. Kroon, L.G., Salomon, M., Van Wassenhove, L.N.: Exact and approximation algorithms for the tactical fixed interval scheduling problem. Oper. Res. **45**(4), 624–638 (1997)
16. Mehlhorn, K.: Data Structures and Algorithms, Multi-dimensional Searching and Computational Geometry, vol. 3. Springer, Berlin (1984)

17. Shamos, M. I., & Hoey, D.: Geometric intersection problems. In: 17th Annual Symposium on Foundations of Computer Science, 1976, pp. 208–215. IEEE (1976)
18. Sleator, D., Tarjan, R.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
19. Spieksma, F.C.: On the approximability of an interval scheduling problem. J. Sched. **2**(5), 215–227 (1999)