Dynamising Interval Scheduling: The Monotonic Case

Alexander Gavruskin[1], Bakhadyr Khoussainov[1], Mikhail Kokho[1], and Jiamou Liu[2]

[1] Department of Computer Science, University of Auckland, New Zealand
[2] School of Computing and Mathematical Sciences, Auckland University of Technology, New Zealand
{a.gavruskin,m.kokho}@auckland.ac.nz, bmk@cs.auckland.ac.nz, jiamou.liu@aut.ac.nz

Abstract. We investigate dynamic algorithms for the interval scheduling problem. We focus on the case when the set of intervals is monotonic. This is when no interval properly contains another interval. We provide two data structures for representing the intervals that allow efficient insertion, removal and various query operations. The first dynamic algorithm, based on the data structure called compatibility forest, runs in amortised time $O(\log^2 n)$ for insertion and removal and $O(\log n)$ for query. The second dynamic algorithm, based on the data structure called linearised tree, runs in time $O(\log n)$ for insertion, removal and query. We discuss differences and similarities of these two data structures through theoretical and experimental results.

1 Introduction

Background. Imagine a number of processes all need to use a particular resource for a period of time. Each process i specifies a starting time $s(i)$ and a finishing time $f(i)$ between which it needs to continuously occupy the resource. The resource cannot be shared by two processes at any instance. One is required to design a scheduler which chooses a subset of these processes so that 1) there is no time conflict between processes in using the resource; and 2) there are as many processes as possible that get chosen.

The above is a typical set-up for the interval scheduling problem, one of the basic problems in the study of algorithms. Formally, given a collection of intervals on the real line all specified by starting and finishing times, the problem asks for a subset of maximal size consisting of pairwise non-overlapping intervals. The interval scheduling problem and its variants appear in a wide range of areas in computer science and applications such as in logistics, telecommunication, and manufacturing. They form an important class of scheduling problems and have been studied under various names and with application-specific constraints [9].

The interval scheduling problem, as stated above, can be solved by a *greedy* scheduler as follows [8]. The scheduler sorts intervals based on their finishing time, and then iteratively selects the interval with the least finishing time that

T. Lecroq and L. Mouchard (Eds.): IWOCA 2013, LNCS 8288, pp. 178–191, 2013.

is compatible with the intervals that have already been scheduled. The set of intervals chosen in this manner is guaranteed to have maximal size. This algorithm works in a *static* context in the sense that the set of intervals is given a priori and it is not subject to change.

In a *dynamic* context the instance of the interval scheduling problem is usually changed by a real-time events, and a previously optimal schedule may become not optimal. Examples of such real-time events include job cancelation, arrival of an urgent job, change in job processing time. 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 scheduling problem on the changed instances. In this dynamic context, the set of intervals change through a number of *update operations* such as insertion or removal. Our goal is to design data structures that allow us to solve the interval scheduling problem in a dynamic setting.

In our effort to dynamise the interval scheduling problem, we focus on a special class of interval sets which we call *monotonic interval sets*. In monotonic interval sets no interval is properly contained by another interval. Considering monotonic intervals is a natural setting for the problem. For example, if all processes require the same amount of time to be completed, then the set of intervals is monotonic. Moreover, monotonic interval sets are closely related to proper interval graphs. An *interval graph* is an undirected graph whose nodes are intervals and two nodes are adjacent if the two corresponding intervals overlap. A *proper interval graph* is an interval graph for a monotonic set of intervals. There exist linear time algorithms for representing a proper interval graph by a monotonic set of intervals [1,6,2]. Furthermore, solving the interval scheduling problem for monotonic intervals corresponds to finding a maximal independent set in a proper interval graph.

Related work. On a somewhat related work, S. Fung, C. Poon and F. Zheng [3] investigated an online version of interval scheduling problem for weighted intervals with equal length (hence, the intervals are monotonic), and designed randomised algorithms. We also mention that R. Lipton and A. Tompkins [5] initiated the study of online version of the interval scheduling problem. In this version a set of intervals are presented to a scheduler in order of start time. Upon seeing each interval the algorithm must decide whether to include the interval into the schedule.

A related problem on a set of intervals I asks to find a minimal set of points S such that every interval from I intersects with at least one point from S. Such a set S is called a *piercing set* of I. A dynamic algorithm for maintaining a minimal piercing set S is studied in [4]. The dynamic algorithm runs in time $O(|S| \log |I|)$. We remark here that if one has a maximal set J of disjoint intervals in I, one can use J to find a minimal piercing set of I, where each point in the piercing set corresponds to the finishing time of an interval in J in time $O(|J|)$. Therefore our dynamic algorithm can be adapted to one that maintains a minimal piercing set. Our algorithm improves the results in [4] when the interval set I is monotonic.

Kaplan et al. in [7] studied a problem of maintaining a set of nested intervals with priorities. The problem asks for an algorithm that given a point p finds the interval with maximal priority containing p. Similarly to our dynamic algorithm, the solution in [7] also uses dynamic trees to represent a set of intervals.

Our results. We provide two dynamic algorithms for solving the interval scheduling problem on monotonic set of intervals. Both algorithms allow efficient insertion, removal and query operation. Formal explanation are in the next sections.

The first algorithm maintains the *compatibility forest* data structure denoted by CF. We say the *right compatible interval* of an interval i is the interval j such that $f(i) < s(j)$ and there does not exist an interval ℓ such that $f(i) < s(\ell)$ and $f(\ell) < f(j)$. The CF data structure maintains the right compatible interval relation. The implementation of the data structure utilises, nontrivially, the dynamic tree data structure of Sleator and Tarjan [10]. As a result, in **Theorem 4** of Section 3 we prove that the insert and remove operations take amortised time $O(\log^2 n)$ and the query operation takes amortised time $O(\log n)$.

The second dynamic algorithm maintains the *linearised tree* data structure denoted by LT. We say that intervals are *equivalent* if their right compatible intervals coincide. The LT data structure maintains both the right compatibility relation and the equivalence relation. Then, in **Theorem 9** of Section 4 we prove that the insertion, removal and query operations take time amortised $O(\log n)$. However, this comes with a cost. As opposed to the CF data structure that keeps a representation of an optimal set after each update operation, the linearised tree data structure does not explicitly represent the optimal solution.

To test the performance of our algorithms, we carried out experiments on random sequences of update and query operations. The experiments show that the two data structures CF and LT perform similarly. The reason for this is that the first dynamic algorithm based on CF reaches the bound of $\log^2 n$ only on specific sequences of operations, while on uniformly random sequences the algorithm may run much faster. Both algorithms outperform the modified naive algorithm (described in Sec. 2).

Organisation of the paper. Section 2 introduces the problem, monotonic interval sets and the modified naive dynamic algorithm. Section 3 and 4 describe the CF and LT data structures and present our dynamic algorithms. Section 5 extends the data structures by adding the report operation that outputs the full greedy solution. Section 6 discusses the experiments.

2 Preliminaries

Interval scheduling basics. An *interval* is a pair $(s(i), f(i)) \in \mathbb{R}^2$ with $s(i) < f(i)$, where $s(i)$ is the *starting time* and $f(i)$ is the *finishing time* of the interval. We abuse notation and write i for the interval $(s(i), f(i))$. Two intervals i and j are *compatible* if $f(i) < s(j)$ or $f(j) < s(i)$. Otherwise, these two intervals *overlap*. Given a collection of intervals $I = \{i_1, i_2, \ldots, i_k\}$, a *compatible set* of I is a subset $J \subseteq I$ such that the intervals in J are pairwise compatible. An *optimal set* of I

is a compatible set of maximal size. The *interval scheduling problem* consists of designing an algorithm that finds an optimal set.

We recall the greedy algorithm that solves the problem [8]. The algorithm sorts intervals by their finishing time, and then iteratively chooses the interval with the least finishing time compatible with the last selected interval. The set of thus selected intervals is optimal. The algorithm is in $O(n \log n)$ where n is the size of I. If the sorting is already given then the algorithm runs in linear time. Below, we formally define the greedy optimal set found by this greedy algorithm.

Let $\preceq$ be the ordering of the intervals by their finishing time. Throughout, by the *least interval*, the *greatest interval*, the *next interval*, the *previous interval*, we mean the least, greatest, next and previous interval with respect to $\preceq$. Without loss of generality we may assume that the intervals in I have pairwise distinct finishing times. Given the collection I, we inductively define the set $J = \{i_1, i_2, \ldots\}$, the *greedy optimal set* of I, as follows. The interval i_1 is the least interval in I. The interval i_{k+1} is the least interval compatible with i_k such that $i_k \prec i_{k+1}$. The set J obtained this way is an optimal set [8].

Dynamic setting. In this setting the collection I of intervals changes over time. Thus, the input to the problem is an arbitrary sequence $o_1, \ldots, o_m$ of update and query operations described as follows:

- *Update operations*: $\mathsf{insert}(i)$ inserts an interval i and $\mathsf{remove}(i)$ removes an interval i.
- *Query operation*: The operation $\mathsf{query}(i)$ returns true if i belongs to the greedy optimal set and false otherwise.

Our goal is to design algorithms for performing these operations that minimise the total running time. We will use the following data structures.

- *Interval tree.* We maintain the ordered set of intervals I in a balanced binary search tree. We call this tree the *interval tree* and denote it by $T(I)$. The interval tree supports all operations of a binary search tree and performs them in $O(\log n)$ worst-case time.
- *Splay tree.* A *splay tree* is a self-balancing binary search tree for storing linearly ordered objects. In addition to the standard binary search tree operations, the splay tree data structure supports the following operations. Operation $\mathsf{splay}(u)$ reorganises a splay tree so that u becomes the root. Operation $\mathsf{join}(A, B)$ merges two splay trees A and B, such that any element in A is less than any element in B, into one tree. Finally, operation $\mathsf{split}(A, u)$ divides a splay tree into two splay trees $R(u)$ and $L(u)$, where $R(u) = \{x \in A \mid u \leq x\}$ and $L(u) = \{x \in A \mid x < u\}$. All the operations for splay trees take $O(\log n)$ amortised time [11].
- *Dynamic trees.* This data structure maintains a forest. Basic update operations are $\mathsf{link}(v, w)$, which creates an edge from a root v to a vertex w (thus v becomes a child of w) and $\mathsf{cut}(v)$, which deletes the edge from v to its parent. Query operations for dynamic tree depend on specific application. Usually, a query operation searches for a node or an edge on a path from a given node.

For example, operation findmin(u) returns an edge with a minimal value on a path from u to a root. These operations have $O(\log n)$ amortised time complexity [10].

Monotonic Interval Sets. The set I of intervals is called *monotonic* if no interval in I contains another interval. Since I changes over time through update operations, to preserve monotonicity we assume that the insert(i) operation never adds an interval i which contains or is contained in an existing interval. Recall that the *right compatible interval* of i, denoted by rc(i), is the least interval j compatible with i such that $i \prec j$. Similarly, the *left compatible interval* of i, written lc(i), is the greatest interval j compatible with i such that $j \prec i$.

Monotonicity of I implies an important property of the interval tree $T(I)$: if an interval $i \in T(I)$ is not compatible with an interval j, then the left subtree of i does not contain rc(j) and the right subtree of i does not contain lc(j). This allows us to define two efficient operations: right_compatible(j), which is defined below, and left_compatible(j), which is similar except we replace "$\preceq$" with "$\succeq$" and swap "left" and "right".

Algorithm 1. right_compatible(i)

```
1: r ← nil
2: j ← the root in the interval tree T(I).
3: while j ≠ nil do
4:     if j ⪯ i or j overlaps i then
5:         j ← the right child of j
6:     else
7:         r ← j
8:         j ← the left child of j
9: return r
```

Lemma 1. *On monotonic set I of intervals the operations* right_compatible(i) *and* left_compatible(i) *run in time $\Theta(\log n)$ and return* rc(i) *and* lc(i) *respectively.*

To prove the lemma we observe that for a monotonic set I of intervals and $i, j \in I$, if i overlaps j, then each of the intervals between i and j overlaps both i and j.

Proof. We only prove the lemma for right_compatible. The operation takes time $\Theta(\log n)$ as the length of paths from a leaf to the root in $T(I)$ is $\lfloor \log n \rfloor + 1$.

For the correctness of right_compatible, we use the following loop invariant: *If I contains* rc(i)*, then the subtree rooted at j contains* rc(i) *or r equals* rc(i)*.*

Initially, j is the root of $T(I)$, so the invariant holds. Each iteration of the **while** loop executes either line 5 or lines 7-8 of Alg. 1. If line 5 is executed, then we have $j \preceq i$ or j overlaps i. If $j \preceq i$ then all intervals in the left subtree of j are less than i. If $j \succeq i$ but j overlaps i, then by the observation above, all intervals between i and j overlap i. In both cases, none of the intervals in the

left subtree of j is $\mathsf{rc}(i)$. Therefore setting j to be the right child of j preserves the invariant.

If lines 7-8 are executed, then we have $j \succeq i$ and j is compatible with i. If there exists an interval that is less than j and compatible with i, then such an interval is in the left subtree of j. If such an interval does not exist, j is the smallest interval which is compatible with i. Therefore setting r to be j and j to be the right child of j preserves the invariant.

Thus, the algorithm outputs $\mathsf{rc}(i)$ if it exists and outputs nil otherwise. Indeed, the loop terminates when $j = \mathsf{nil}$. Hence if the set of intervals I contains $\mathsf{rc}(i)$ then $r = \mathsf{rc}(i)$. If I does not contain $\mathsf{rc}(i)$ then line 5 is executed at every iteration, so $r = \mathsf{nil}$. □

Modified naive dynamic algorithm. A naive dynamic algorithm for the interval scheduling problem is to keep intervals sorted and construct the greedy optimal set from scratch at each query operation. Another modified yet still naive dynamic algorithm is this. Store the greedy optimal set in a self-balancing binary search tree T. After each $\mathsf{insert}(i)$ or $\mathsf{delete}(i)$ operation search for the greatest interval $j_0 \in T$ such that $f(j_0) < s(i)$. Then insert a sequence $j_1 = \mathsf{rc}(j_0), \ldots, j_k = \mathsf{rc}(j_{k-1})$ of intervals into T. The sequence ends with the interval j_k such that $\mathsf{rc}(j_k)$ does not exist or is already in T. While inserting, we remove all intervals between j_0 and j_{k+1} from T. The $\mathsf{query}(i)$ operation of this algorithm takes $O(\log n)$ worst-case time. The $\mathsf{insert}(i)$ and $\mathsf{remove}(i)$ operations take $O(k \log n)$ worst-case time, where k is the number of intervals inserted into T. In Section 6 we compare this modified algorithm with the algorithms provided by the CF and LT data structures.

3 Compatibility Forest Data Structure (CF)

Building the data structure. Let I be a set of intervals. We define the *compatibility forest* as a graph $\mathcal{F}(I) = (V, E)$ where $V = I$ and $(i, j) \in E$ if $j = \mathsf{rc}(i)$. By a forest we mean a directed graph where the edge set contains links from nodes to their parents. We use $p(v)$ to denote the parent of node v. The *roots* and *leaves* are standard notions that we do not define. Figure 1 shows an example of a monotonic set of intervals with its compatibility forest. We note that for every forest one can construct in a linear time a monotonic set of intervals whose compatibility forest coincides (up to isomorphism) with the forest.

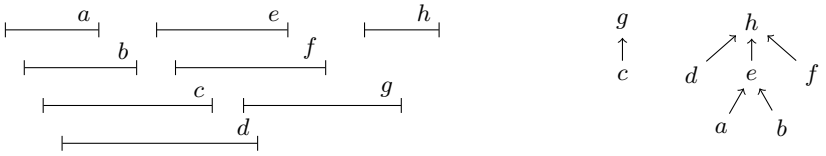

Fig. 1. Example of a monotonic set of intervals and its compatibility forest

A *path* in the compatibility forest $\mathcal{F}(I)$ is a sequence of nodes $i_1, i_2, \ldots, i_k$ where $(i_t, i_{t+1}) \in E$ for any $t = 1, \ldots, k-1$. It is clear that any path in the forest $\mathcal{F}(I)$ consists of compatible intervals. Essentially, the forest $\mathcal{F}(I)$ connects nodes by the greedy rule: for any node i in the forest $\mathcal{F}(I)$, if the greedy rule is applied to i, then the rule selects the parent j of i in the forest. Hence, the longest paths in the compatibility forest correspond to optimal sets of I. In particular, the path starting from the least interval is the greedy optimal set. Our first dynamic algorithm amounts to maintaining this path in the forest $\mathcal{F}(I)$.

We explain how we maintain paths in the compatibility forest $\mathcal{F}(I)$. The representation of the forest is developed from the dynamic tree data structure as in [10]. The idea is to partition the compatibility forest into a set of node-disjoint paths. Paths are defined by two types of edges, *solid edges* and *dashed edges*. Each node in the compatibility forest is required to have at most one incoming solid edge. A sequence of edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$ where each (u_i, u_{i+1}) is a solid edge is called a *solid path*. A solid path is *maximal* if it is not properly contained in any other solid path. Therefore, the solid edges in $\mathcal{F}(I)$ form several maximal solid paths in the forest. Furthermore, the data structure ensures that each node belongs to some maximal solid path. There is an important subroutine in the dynamic tree data structure called the *expose* operation [10]. The operation starts from a node v and traverses the path from v to the root: while traversing, if the edge $(x, p(x))$ is dashed, we declare $(x, p(x))$ solid and declare the incoming solid edge (if it exists) incident to $p(x)$ dashed. Thus, after exposing node v, all the edges on the path from v to the root become solid. Note that in CF data structure the $p(x)$ and $\mathsf{rc}(x)$ are the same.

To represent CF we use two data structures. The first is the interval tree $T(I)$. The operation right_compatible computes the outgoing dashed edges of the compatibility forest. The second is a set of splay trees. Each splay tree stores the nodes of a maximal solid path in the compatibility forest with the underlying order $\preceq$. We denote by ST_u the splay tree containing the node u.

Dynamic Algorithm 1. We now describe algorithms for maintaining compatibility forest data structure. We call the algorithms queryCF, insertCF and removeCF for the query, insertion, and removal operations, respectively.

The operation queryCF*:* To perform this operation on an interval i, we first find in the interval tree $T(I)$ the minimum element m. We then check if i belongs to the splay tree ST_m. We return true if $i \in \mathrm{ST}_m$; otherwise we return false.

The operation expose*:* To expose an interval i, we find the maximum element j in the splay tree ST_i. Then find the right compatible interval $i' = \mathsf{rc}(j)$. If i' does not exist (that is, j is a root in the compatibility forest), we stop the process. Otherwise, (j, i') is a dashed edge. We split the splay tree at i' into trees $L(i')$ and $R(i')$ and join ST_i with $R(j')$. We then repeat the process taking i' as i.

The operation insertCF*:* To insert an interval i, we add i into the tree $T(I)$. Then we locate the next interval r of i in the ordering $\preceq$. If such r exists, we access r in the splay tree ST_r and find the interval j such that (j, r) is a solid edge. If such a j exists and j is compatible with i, we delete the edge (j, r) and create a new

edge (j, i) and declare it solid. We restore the longest path of the compatibility forest by exposing the least interval in $T(I)$.

The operation removeCF*:* To delete an interval i, we delete the incoming and outgoing solid edges of i if such edges exist. We then delete i from the tree $T(I)$. We restore the longest path of the CF by exposing the least interval in $T(I)$.

Correctness of the operations. For correctness, we use the following invariants.

(A1) Every splay tree represents a maximal path formed from solid edges.
(A2) Let m be the least interval in I. The splay tree ST_m contains all intervals on the path from m to the root.

Note that (A2) guarantees that the query operation correctly determines if a given interval i is in the greedy optimal set. The next lemma shows that (A1) and (A2) are invariants indeed and that the operations correctly solve the dynamic monotonic interval scheduling problem.

Lemma 2. *(A1) and (A2) are invariants of* insertCF*,* removeCF*, and* queryCF*.*

Proof. For (A1), first consider the operation of joining two splay trees A and B via the operation expose(i). Let j be the maximal element in A and j' be the minimum element in B. In this case, j' is obtained by the operation right_compatible(j). It is clear that (j, j') is an edge in the forest $\mathcal{F}(I)$. Next, consider the case when we apply insertCF(i) into the splay tree A. In this case, A is $L(r)$ where r is the next interval of i in I. Let j be the previous interval of r in the tree ST_r. By (A1), before inserting i, (j, r) is an edge in $\mathcal{F}(I)$ and thus $r = \mathsf{rc}(j)$. Note we only insert i to $L(r)$ when j is compatible with i. Since $i < r$, after inserting i, i becomes the new right compatible interval of j. So, joining $L(r)$ with i preserves (A1). Operations removeCF(i) and queryCF(i) do not create new edges in splay trees. Thus, (A1) is preserved under all operations.

For (A2), the expose(i) operation terminates when it reaches a root of the compatibility forest. As a result, ST_i contains all nodes on the path from i to the root. Since expose(minimum$(T(I))$) is called at the end of both insertCF(i) and removeCF(i) operations, (A2) is preserved under every operation. □

Complexity. Let n be the number of intervals in I. As discussed in Section 2, all operations for the interval tree have $O(\log n)$ worst case complexity, and all operations for splay trees have $O(\log n)$ amortised complexity. The query operation, involves finding the minimum interval in $T(I)$ and searching i in a splay tree. Hence, the query operation runs in amortised time $O(\log n)$. For each insert and remove operation, we perform a constant number of operations on $T(I)$ and the splay trees plus one expose operation.

To analyse expose operation, define the size size(i) of an interval i to be the number of nodes in the subtree rooted at i in $\mathcal{F}(I)$. Call an edge (i, j) in $\mathcal{F}(I)$ *heavy* if $2 \cdot \mathsf{size}(i) > \mathsf{size}(j)$, and *light* otherwise. It is not hard to see that this partition of edges has the following properties:

($\star$) Every node has at most one incoming heavy edge.
($\star\star$) Every path in the compatibility forest consists of at most $\log n$ light edges.

Lemma 3. *In a sequence of k update operations, the total number of dashed edges, traversed by* expose *operation, is $O(k \log n)$.*

Proof. The number of iterations in expose operation is the number of dashed edges in a path from the least interval to the root. A dashed edge is either heavy or light. From ($\star\star$), there are at most $\log n$ light dashed edges in the path. To count the number of heavy edges, consider the previous update operations. After deletion of i, all children of i become children of the next interval of i. After inserting i, the children of the next interval of i that are compatible with i become children of i. Figure 2 illustrates these structural changes. Thus, an update operation transforms at most $\log n$ light dashed edges to heavy dashed edges in each path, starting at the next interval or the right compatible interval of i. Execution of expose in an update operation creates at most $\log n$ heavy dashed edges from heavy solid edges. Hence, the total number of heavy dashed edges created after k update operations is $O(k \log n)$. □

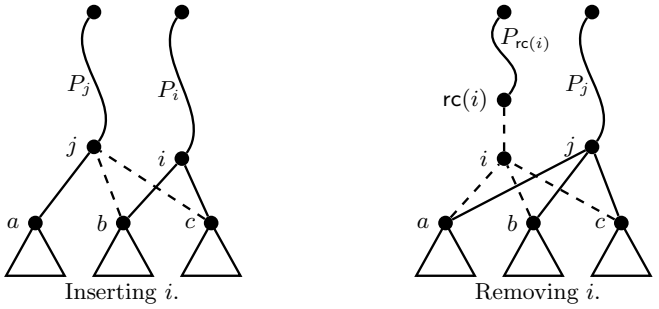

Fig. 2. Redirections of edges in CF, where j is the next interval of i

Lemma 2 and Lemma 3 give us the following theorem:

Theorem 4. *The algorithms* queryCF, insertCF *and* removeCF *solve the dynamic monotonic interval scheduling problem. The algorithms perform insert interval and remove interval operations in $O(\log^2 n)$ amortised time and query operation in $O(\log n)$ amortised time, where n is the size of the set I of intervals.*

Remark. Tarjan and Sleator's dynamic tree data structure has amortised time $O(\log n)$ for update and query operations. To achieve this, the algorithm maintains dashed edges explicitly. Their technique cannot be adapted directly to CF because insertion or removal of intervals may result in redirections of a linear number of edges. Therefore, more care should be taken; for instance,

one needs to maintain dashed edges implicitly in $T(I)$ and compute them calling right_compatible operation.

Proposition 5 (Sharpness of the $\log^2 n$ bound). *In* CF *data structure there exists a sequence of k update operations with $\Theta(k \log^2 n)$ total running time.*

Proof. Consider a sequence which creates a set of $n < k$ intervals. We assume that $n = 2^{h+1} - 1$ for an $h \in \omega$. The first n operations of the sequence are insertCF such that the resulted compatibility forest is a perfect binary tree T_n, that is, each internal node of T_n has exactly two children and the height of each leaf in T_n is h. The next $k - n$ operations starting form T_n are pairs of insertCF followed by removeCF. At stage $s = n+2m+1$, insertCF inserts an interval i_s into T_s producing the tree T_{s+1}. The interval i_s is such that in T_{s+1} the path from i_s to the root is of length $h + 1$ and the path consists of dashed edges only. Then, at stage $s + 1$ we delete i_s. This produces a tree T_{s+2} which is a perfect binary tree of height h. We repeat this $k - n$ times. We can select i_s as desired since each perfect binary tree T_s always has a path of length h consisting of dashed edges only. Therefore a sequence of k such operations takes time $\Theta(k \log^2 n)$. □

4 Linearised Tree Data Structure (LT)

Building the data structure. We describe a second dynamic algorithm for solving the monotonic interval scheduling problem. Our goal is to improve the running time for the update operations by introducing the *linearised tree* data structure.

We say that intervals i and j are *equivalent*, written as $i \sim j$, iff $\mathsf{rc}(i) = \mathsf{rc}(j)$. Denote the equivalence class of i by $[i]$. Thus, two intervals are in the same equivalence class if they are siblings in the compatibility forest. In the linearised tree we arrange all intervals in an equivalence class in a path using the $\preceq$-order. The linearised tree consists of all such "linearised" equivalence classes joined by edges. Hence, there are two types of edges in the linearised tree. The first type connects intervals in the same equivalence class. The second type joins the greatest interval in an equivalence class with its right compatible interval. Formally, the *linearised tree* $\mathcal{L}(I)$ is a triple $(I; E_\sim, E_c)$, where $E_\sim$ and E_c are disjoint set of edges such that:

- $(i, j) \in E_\sim$ if and only if $i \sim j$ and i is the previous interval of j. Call i the *equivalent child* of j.
- $(i, j) \in E_c$ if and only if i is the greatest interval in $[i]$ and $j = \mathsf{rc}(i)$. Call i the *compatible child* of j.

Figure 3 shows an example of a linearised tree. We stress three crucial differences between the CF and LT data structures. The first is that a path in a linearised tree may not be a compatible set of intervals. The second is that linearised trees are binary. The third is when we insert or remove an interval we need to redirect at most two existing edges in the linearised tree. We explain the last fact in more details below when we introduce the dynamic algorithm.

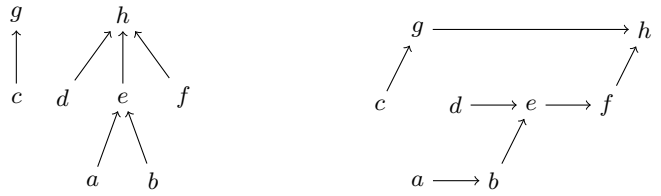

Fig. 3. Example of a compatibility forest (left) and linearised tree (right)

We use the dynamic tree data structure to represent the linearised tree. We also maintain the interval tree $T(I)$ as an auxiliary data structure. The interval tree is used to compute previous and next intervals as well as left compatible and right compatible intervals of a given interval.

Dynamic Algorithm 2. We now describe algorithms for maintaining linearised tree data structure. We call the algorithms queryLT, insertLT and removeLT for the query, insertion, and removal operations, respectively.

The operation queryLT*:* To detect if an interval i is in the greedy optimal set, consider the path P from the least node m to the root in the linearised tree $\mathcal{L}(I)$. If $i \notin P$, return false. Otherwise, consider the direct predecessor j of i in the path P. If j does not exist or $(j, i) \in E_c$, return true. Otherwise, we return false.

Algorithm 2. queryLT(i)

```
1: m ← minimum(T(I))
2: if i = m then                                  ▷ i is the least interval
3:     return true
4: expose(m)                          ▷ Make the path from m to the root solid
5: if i ≠ find(ST_m, i) then          ▷ i is not on the path from m to the root
6:     return false
7: j ← predecessor(ST_m, i)                       ▷ (j, i) is an edge in LT
8: if i is compatible with j then
9:     return true
10: else
11:     return false
```

Lemma 6. *The operation* queryLT(i) *returns* true *if and only if a given interval i belongs to the greedy optimal set of I.*

The operation insertLT*:* Given i, we insert i into $T(I)$. If i is the greatest interval in $[i]$, then we add the edge $(i, \mathsf{rc}(i))$ into E_c. Otherwise, we add the edge (i, j) to $E_\sim$, where j is the next interval equivalent to i. If i has an equivalent child k then we add the edge (k, i) to $E_\sim$ and delete the old outgoing edge from k in case such edge exists. If i has a compatible child ℓ then we add the edge (ℓ, i) to E_c and delete the old outgoing edge in case such edge exists.

Lemma 7. *The operation* insertLT(i) *preserves linearised tree data structure.*

The operation removeLT*:* Given i, we delete i from $T(I)$. We delete an edge from i to the parent of i and redirect the edge from the equivalent child j of i to the parent of i. Then we redirect an edge from the compatible child ℓ of i. Removing i may add new intervals to the equivalence class of ℓ. Therefore if ℓ is still the greatest interval in the updated equivalence class, we add an edge $(\ell, \mathsf{rc}(\ell)$ to E_c. Otherwise, we add the edge (i, j) to $E_\sim$, where j is the next interval of ℓ.

Lemma 8. *The operation* removeLT(i) *preserves linearised tree data structure.*

Lemmas 6-8 lead us to the following theorem:

Theorem 9. *The* queryLT*,* insertLT *and* removeLT *operations solve the dynamic monotonic interval scheduling problem in* $O(\log n)$ *amortised time, where* n *is the size of the set* I *of intervals.*

Note. The time complexity of the operations above depends on the type of dynamic trees, representing paths of LT. We can achieve the worst-case bound instead of amortized if we use globally biased trees instead of splay trees [10]. However, after each operation we must ensure that for every pair of edges (v, u) and (w, u) of the linearised tree, nodes v and u are in the same dynamic tree if and only if the numbers of nodes in the subtree rooter at v is greater or equal to the number of nodes in the subtree rooted at u.

5 Extending Functionality of CF and LT Data Structures

The operations queryCF and queryLT detect if a given interval i belongs to the current greedy optimal set. Alternatively, another intuitive meaning of the query operation is to report the full greedy optimal set. The report operation, given a set I of monotonic intervals, outputs all the intervals (with their starting and finishing times) in the greedy optimal set. It turns out, our data structures allow an efficient implementation of reportCF and reportLT operations.

In the CF data structure, the greedy schedule is the set of intervals on the path from the least node m to the root. This path is represented by the splay tree ST_m and is maintained after every update operation. Therefore the reportCF amounts to in-order traversal of ST_m. The only thing we need to remember is the root of ST_m after every update operation.

Theorem 10. *The amortised complexity of the* reportCF *operation is* $O(|\mathrm{ST}_m|)$*, where* ST_m *is the greedy optimal set.*

The theorem above also shows a subtle difference between the two data structures CF and LT. In the LT data structure, in order to perform the reportLT operation, one needs to examine the path P starting form the minimal element in the tree $\mathcal{L}(I)$. But the path might contain nodes that are not necessarily in the greedy optimal solution. Namely, we need to filter out those nodes v in P

for which there exists a $u \in P$ such that $(u, v) \in E_{\sim}$. Hence, reportLT runs in linear time on the size of P, where in the worst case $P = I$.

In the modified naive algorithm, reporting of the greedy optimal set J takes $O(|J|)$ time. However, to maintain the set J, update operations of the algorithm take $O(k \log n)$ time as described in Section 2, where k is the number of changes in J. In the worst case, $k = \Theta(n)$.

6 Experiments

Here we experimentally compare the naive (N), modified naive (MN), CF and LT data structure algorithms. We implemented these algorithms in Java and run experiments on a laptop with *4GB of RAM* memory and *Intel Core 2 Duo 2130 Mhz, 3MB of L2 cache memory* processor.

In our tests, we measure the average running time in a randomly chosen sequence of $m = n + rn + qn$ operations on initially empty interval set, where n, rn and qm are the number of insert, remove and query operations respectively. Here both q and r are parameters. For insert(i) operation, the starting time $s(i)$ is a random number in $[0, 1]$ and the finishing time is $s(i) + 1/n$. The operations remove and query are always randomly applied to the current set I of intervals. The summary of our experiments are the following:

- CF performs similarly to LT in spite of the fact that CF takes $O(\log^2 n)$ in average as opposed to $O(\log n)$ of LT data structure. Figure 4 shows the results of an experiment with $q = n$ and $r = 0.5n$. Here our sequences of operations do not contain report operation.
- CF data structure performs better than LT if we replace q query operations with q report operations. This confirms our remarks at the end of Section 5.

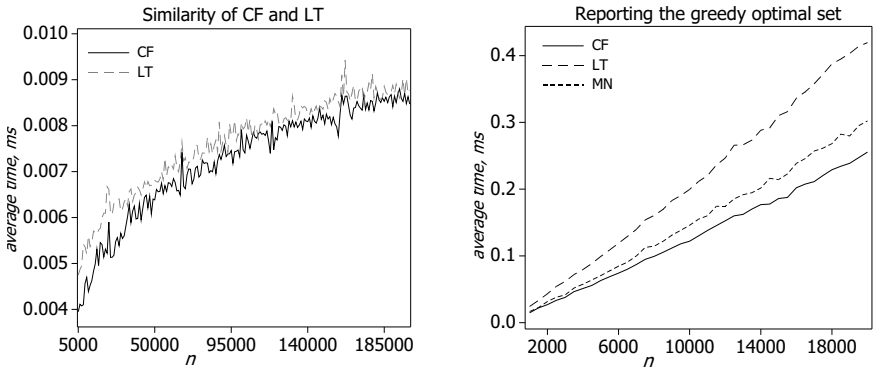

Fig. 4. The parameters of the experiment on the left plot are $q = n$ and $r = 0.5$. The parameters of the experiment on the right plot are $q = n$ and $r = 0$.

- More surprisingly, even MN performs better than LT if we replace q query operations with q report operations.
- CF outperforms MN if we replace q query operations with q report operations.
- N is outperformed by all algorithms in most of the settings.

7 Conclusions and Open Problems

Several directions for further research remain open. One of them is to remove the monotonic restriction and allow intervals to be contained in other intervals. To treat this general case a result in line with Lemma 1 would perhaps play a crucial role. Another direction is to allow an arbitrary, but fixed number of available resources. Data structures, solving these more general interval scheduling problems, would be valuable in practical applications.

References

1. Corneil, D.: A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. Discrete Applied Mathematics 138(3), 371–379 (2004)
2. Deng, X., Hell, P., Huang, J.: Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. SIAM Journal on Computing 25(2), 390–403 (1996)
3. Fung, S.P.Y., Poon, C.K., Zheng, F.: Online interval scheduling: Randomized and Multiprocessor cases. In: Lin, G. (ed.) COCOON 2007. LNCS, vol. 4598, pp. 176–186. Springer, Heidelberg (2007)
4. Katz, M.J., Nielsen, F., Segal, M.: Maintenance of a piercing set for intervals with applications. Algorithmica 36(1), 59–73 (2003)
5. Lipton, R., Tompkins, A.: Online interval scheduling. In: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 302–311 (1994)
6. Heggernes, P., Meister, D., Papadopoulos, C.: A new representation of proper interval graphs with an application to clique-width. Electronic Notes in Discrete Mathematics 32, 27–34 (2009)
7. Kaplan, H., Molad, E., Tarjan, R.: Dynamic rectangular intersection with priorities. In: Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, pp. 639–648 (June 2003)
8. Kleinberg, J., Tardos, E.: Algorithm Design (2006)
9. Kolen, A., Lenstra, J.K., Papadimitriou, C.H., Spieksma, F.C.: Interval scheduling: A survey. Naval Research Logistics 54(5), 530–543 (2007)
10. Sleator, D., Tarjan, R.: A Data Structure for Dynamic Trees. Journal of Computer and System Sciences 26(3), 362–391 (1983)
11. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. Journal of the ACM 32(3), 652–686 (1985)