Dynamic Interval Scheduling on a Single Machine

Manoj Gupta
Project Advisor | Assistant Professor
Indian Institute of Technology, Gandhinagar, India

Mrigankashekhar Shandilya
Undergraduate Student
Indian Institute of Technology, Gandhinagar, India

Abstract

In the realm of interval scheduling, real-time adaptability through dynamic updates is of paramount importance. This paper presents a novel algorithm for batch updates in the Interval Scheduling problem that substantially improves the time complexity for the Dynamic Interval Scheduling Problem on a single machine. We introduce an algorithm that operates with a time complexity of $\mathcal{O}(f \log n)$ for batch updates, leveraging this to develop an algorithm for dynamic updates with an amortized time complexity of $\mathcal{O}(\sqrt{n} \log n)$ per update.

While previous works such as [4, Gawrychowski, Pokorski, ICALP 2022] provide a much more efficient ($\mathcal{O}(\sqrt[3]{n} \log n)$ amortized time complexity per update) algorithm for the fully dynamic version of the problem, our approach is significantly simpler, making use of classical data structures. In contrast to other previous works that offer approximation solutions, such as the one proposed by [5, Henzinger, Neumann, Wiese, SoCG 2020], our approach is geared towards exact solutions. This advancement has significant implications for applications where interval scheduling with real-time adaptability is crucial.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases interval scheduling, data structures, greedy algorithms, dynamic problems, batch updates, single-machine environment

1 Introduction

The Interval Scheduling (IS) problem serves as an elementary but vital case study in greedy algorithms. This problem involves identifying the largest subset of intervals that do not overlap with each other, often termed as compatible intervals in this domain. The problem can be envisioned as a single-machine scheduling challenge where each interval represents an uninterruptible job demanding exclusive machine access. The aim is to maximize the number of jobs scheduled.

In the standard version of the problem, a well-known greedy algorithm offers a solution in $\mathcal{O}(n)$ time, assuming the intervals are pre-sorted by their end-points b_i. However, in practical applications, the set of intervals often undergoes dynamic updates, such as insertions and deletions. The challenge here is to maintain an optimal solution or its cost in real-time without the computational burden of recalculating the entire solution from scratch, which would take linear time relative to the input size at the very least. To this end, we propose maintaining auxiliary structures to facilitate efficient updates.

1.1 The Dynamic Interval Scheduling Problem

The primary problem under consideration is to find the cardinality, $|S|$, of the maximum size subset of disjoint intervals in a dynamically changing set of intervals $\mathcal{I}$. More formally, given a set $\mathcal{I} = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$, where $a_i, b_i \in \mathbb{R}$ and $a_i < b_i$, the objective is to find a subset $S \subseteq \mathcal{I}$ such that:

1. Any two intervals $[a_i, b_i]$ and $[a_j, b_j]$ in S are disjoint: $[a_i, b_i] \cap [a_j, b_j] = \emptyset$ for $i \neq j$, and

2. $|S|$ is maximized, where $|S|$ denotes the cardinality of set S.

The set $\mathcal{I}$ is subject to fully dynamic updates, which include insertions of new intervals and deletions of existing intervals. After each such update, the cardinality $|S|$ must be recalculated to maintain and report the optimal solution.

1.2 The 'batch updates to the Interval Scheduling' problem

An intriguing sub-variant of this problem involves performing a batch of updates on $\mathcal{I}$, say $\mathcal{U} = \{u_1, u_2, \dots, u_f\}$, where u_i is either an insertion or a deletion. The goal is to find the cardinality $|S'|$ of the maximum size subset of disjoint intervals in the updated set $\mathcal{I}'$ after all updates in $\mathcal{U}$ have been executed. Subsequently, the set is reverted to $\mathcal{I}$, and the process continues.

2 Previous Work

Surprisingly, there does not exist, to the best of our knowledge, any previous work which deals with the sub-variant of batch updates to the interval scheduling (IS) problem.

Prior contributions in the field of the Dynamic Interval Scheduling (DIS) problem include the algorithm proposed by Gawrychowski et al. [4] for fully dynamic interval scheduling across one or multiple machines, which achieves a time complexity of $\mathcal{O}(\sqrt[3]{n}\log n)$ per update. This can, of course, be extended to handle batch updates in $\mathcal{O}(f\sqrt[3]{n}\log n)$ time.

For the weighted version of the Dynamic Interval Scheduling Problem (Dynamic Weighted Interval Scheduling, or DWIS), which can naturally be reduced to the unweighted version, Henzinger et al. [5] introduced an efficient approximation algorithm that maintains a $(1+\epsilon)$-approximate solution in polylogarithmic time. Compton et al. [2] improved upon the ϵ-dependency in the approximate solution, reducing it from exponential to polynomial complexity.

Building upon these prior contributions, our research seeks to further optimize the time complexity associated with fully dynamic interval scheduling on a single machine. We have engineered an algorithm for batch updates with a time complexity of $\mathcal{O}(f\log n)$, thereby offering a more computationally efficient way of handling large sets of updates.

Leveraging this structure, we can also build an algorithm to handle fully dynamic updates to the interval scheduling problem with $\mathcal{O}(\sqrt{n}\log n)$ amortized time per update.

Thus, our work naturally progresses from existing literature by targeting even more efficient exact and amortized time complexities, while focusing specifically on a single-machine environment. The end goal is to achieve a logarithmic-time algorithm, positioning our research as a significant step toward that objective.

3 Our Contribution and Approach

3.1 Prerequisites

▶ **Definition 1.** *An interval α_i is a pair of two integers $[a_i, b_i]$ such that $a_i < b_i$.*

▶ **Definition 2.** *Two intervals α_i and α_j are disjoint ($\alpha_i \cap \alpha_j = \emptyset$) if either $b_i < a_j$ or $b_j < a_i$.*

▶ **Definition 3.** *An interval α_j is called the next compatible interval of α_i, NC_i, if $j = \operatorname{argmin}_x b_x \,\forall\, x \in \{x \mid \alpha_x \cap \alpha_i = \emptyset \wedge a_x > b_i\}$. If the set defined is empty, $NC_i = \phi$.*

▶ **Definition 4.** *The greedy solution set from an interval, α_i, denoted by GSS_i is the maximum size subset of disjoint intervals, S'', provided that $i = \operatorname{argmin}_x a_x \,\forall\, x \in \{x \mid \alpha_x \in S''\}$. It is important to note that $GSS_\phi = \emptyset$.*

▶ **Definition 5.** *The greedy solution from an interval, α_i, denoted by GS_i is the cardinality of the set GSS_i. It is important to note that $GS_\phi = 0$.*

▶ **Definition 6.** *The interval α_j is called the last compatible interval from α_y of the interval α_i, denoted by $LC_{\{i,\,y\}}$, if $j = \operatorname{argmax}_x b_x \,\forall\, x \in \{x \mid \alpha_x \in GSS_y \,\wedge\, b_x < a_i\}$.*

▶ **Definition 7.** *The heavy child, h_i, of any node i in a tree is the child whose subtree size is the largest among all of i's children. If i is a leaf node, $h_i = \phi$.*

▶ **Definition 8.** *The edge connecting a heavy child to its parent is called a heavy edge.*

▶ **Definition 9.** *A path consisting of just heavy edges in a tree is known as a heavy path.*

3.2 Efficiently performing Batch Updates

We look into the the folklore greedy solution of the interval scheduling problem through an analytical lens to observe that the greedy solution can be recursively defined as:

$$GS_i = GS_{NC_i} + 1$$

The primary idea upon which we build our algorithm is that given an insert update (an interval α_{n+k}), we can find if it fits our optimal greedy solution just by looking into the interval $\beta = LC_{\{n+k,\,t\}}$ such that $t = \operatorname{argmin}_x b_x \,\forall\, \alpha_x \in \mathcal{I}$, and comparing NC_β with the interval α_{n+k} to determine if the value of NC_β changes; specifically, if $b_{NC_\beta} < b_{n+k}$. It is obvious that finding the answer is not an issue when we find the above condition to be false, *i.e.* when α_{n+k} is not part of the optimal greedy solution. If it is a part of the solution, we claim that finding the solution is still *"easy"* using the recursive definition on NC_{n+k}.

As such, for insert updates, our problem is reduced to the following subproblems:
1. Finding $LC_{\{i,\,j\}}$ efficiently.
2. Finding NC_i efficiently.

For deletion of an interval, α_i, we simply proceed by temporarily changing b_i to ∞ and reversing this change at the end of our computation of the answer using the recursive definition on NC_β, where $\beta = LC_{\{i,t\}}$ and $t = \operatorname{argmin}_x b_x \,\forall\, \alpha_x \in \mathcal{I}$.

We shall later extend this idea to design an algorithm that makes the updates $\mathcal{U} = \{u_1, u_2, \ldots, u_f\}$ to the original set $\mathcal{I}$ efficiently.

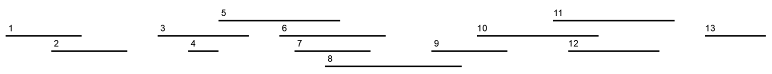
3.2.1 Modelling the set to a Greedy Tree

We can now model this recursive solution in the form of the path length from node i to the root of a tree constructed as follows:
- Every interval in the set $\mathcal{I}$ is represented as a node in the forest of trees.
- The parent of a node i is NC_i.

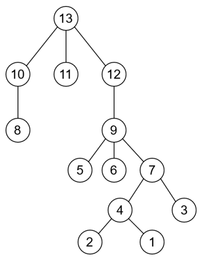
Notice that this forms a forest of trees such that every interval, $\alpha_i \in \mathcal{I}$, corresponds to a node in exactly one tree in the forest. Then:

$$GS_i = W_{\{i,\,root\}} + 1$$

where $W_{\{i,\,root\}}$ is the length of the path from node i to the *root* of the tree it is a part of. We shall first construct this forest of trees [refer to Figure 1].



(a) A given set of intervals



(b) A tree built on the given set of intervals according to the rules defined above

Figure 1 An example for a tree built on a set of intervals

3.2.2 An efficient way to compute the next compatible interval

In order to compute the next compatible interval, we make use of the classical data structure, segment trees [1]. We first sort the intervals present in our set, $\mathcal{I}$, by their starting times, *i.e.*, $a_i \,\forall\, \alpha_i \in \mathcal{I}$, in ascending order. We then build a segment tree on this sorted sequence of intervals where the value stored in each node (denoting a range) is the interval with least ending time in that range of intervals. More formally, the value stored in each node is α_l such that:

$$l = \operatorname{argmin}_x b_x \,\forall\, \alpha_x \in \mathcal{R}$$

where $\mathcal{R}$ is the set of intervals in the range represented by the corresponding node of the segment tree.

Notice that our aim is to process queries of the form $findNCbyInterval(\alpha_i)$ which is equivalent to processing queries of the form $findNCbyTime(b_i)$. By the definition of NC, we know that $NC_i \in \mathcal{R}' = \{\alpha_x \mid a_x > b_i\}$. We can easily observe that the set $\mathcal{R}'$ is a contiguous subsequence of the sequence over which we built our segment tree. As such, $\mathcal{R}'$ can be represented as a range of elements from the sorted sequence. This range can easily be found in $\mathcal{O}(\log n)$ by performing a binary search on the sorted sequence.

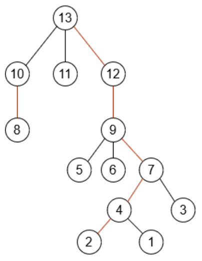
Moreover, the interval with the least ending time in this range is, by definition, the LC_i. Notice that we can find this interval just by querying the computed range in the segment tree. This, again, takes $\mathcal{O}(\log n)$ time.

Since sorting the intervals and building the segment tree each take $\mathcal{O}(n \log n)$ time, we can conclude that with a pre-processing of $\mathcal{O}(n \log n)$, we can compute queries of the form $findNC(\alpha_i)$ in $\mathcal{O}(\log n)$ time per query.

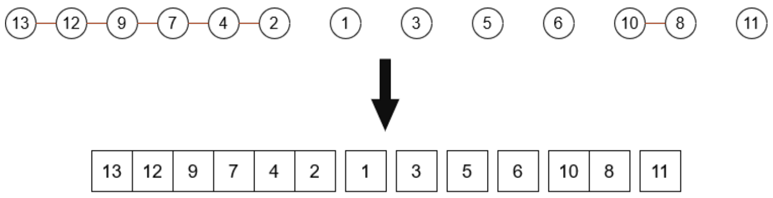
This also naturally extends to the fact that we can therefore build the Greedy Tree in $\mathcal{O}(n \log n)$ time by querying the next compatible interval, NC_i, for each interval, α_i, and setting NC_i as the parent of α_i.

3.2.3 An efficient way to compute the last compatible interval from some arbitrary interval

In order to compute $LC_{\{i,y\}}$, we first decompose our greedy tree according to the rules of heavy light decomposition [6] and store the disjoint heavy paths as a a sequence, $\mathcal{S}$ [refer to figure 2]. As already known, any path in the tree consists of at most $\mathcal{O}(\log n)$ distinct heavy paths.



(a) The greedy tree with heavy paths highlighted in red



(b) A modified traversal of the greedy tree leading to the sequence, $\mathcal{S}$

Figure 2 Heavy-light decomposition of the Greedy Tree

We can notice that in order to compute $LC_{\{i,y\}}$, we must investigate GSS_y which is the path from α_y to the root in the greedy tree [as mentioned in section 3.2.1]. This path can now be thought of as the union of at most $\mathcal{O}(\log n)$ distinct subsequences (heavy paths) of the sequence, $\mathcal{S}$. We can now attempt the find the interval α_j such that $j = \text{argmax}_x\, b_x \,\forall\, x \in \{x \mid \alpha_x \in GSS_y \wedge b_x < a_i\}$ by doing a binary search on the appropriate subsequence since the intervals in a path are sorted by both, a_x and b_x. In order to find the appropriate subsequence, one may simply traverse through the $\mathcal{O}(\log n)$ subsequences investigating their first and last entries (since the subsequences are sorted). As such, we can find $LC_{\{i,y\}}$ in $\mathcal{O}(\log n)$ time since both operations (finding the appropriate subsequence and performing a binary search) take $\mathcal{O}(\log n)$ time.

3.2.4 Performing the updates in one batch and finding the answer

As described in section 3.2, we shall use the recursive definition to find the answer. Note that GS_{NC_i} is not yet computed and thus, can handle updates, but, on the other hand, already processed intervals can't handle any more updates. As such, it is necessary to process the intervals in an ascending order (by ending time). Thus, given a batch of updates, $\mathcal{U}$, we first sort these updates by the ending time of the interval that needs to be added or deleted in ascending order.

Initially, we consider a dynamic answer set, $\mathcal{A}$, as the originial GSS_i such that $i = \text{argmin}_x\, b_x \,\forall\, x \in \{x \mid \alpha_x \in \mathcal{I}\}$ and dynamic answer, $\mathcal{A}_n$, as the cardinality of $\mathcal{A}$. We also proceed to initialize the start of dynamic answer, Γ, as i. Next, in order to process a given update (in the sorted sequence) with the interval, α_j, we look into $\mathcal{A}$ and find $\beta = \alpha_l = LC_{\{j,\Gamma\}}$ to find if or if not α_{n+k} belongs to the actual answer by comparing b_j and b_{NC_l}. We can see that the answer till β is now final, irrespective of whether or not α_j belongs to the solution.

In order to evaluate an insert update when α_j belongs to the solution, we can then update $\Gamma \leftarrow findNCbyTime(b_j)$; and update $\mathcal{A}$ and $\mathcal{A}_n$ accordingly.

In order to evaluate a delete update when α_j belongs to the solution, we can then update the value of $b_j \leftarrow \infty$ in the segment tree, and then update $\Gamma \leftarrow findNCbyTime(b_l)$; and update $\mathcal{A}$ and $\mathcal{A}_n$ accordingly. Then we can reverse the update on the segment tree.

3.3 An efficient solution to the Dynamic Interval Scheduling Problem

The batch updates algorithm can be used to process $\sqrt{n}$ queries without updating the underlying structures, after which we can rebuild the structures from scratch. This shall, in total, take $\mathcal{O}(n \log n)$ time which when amortized over $\sqrt{n}$ updates gives an amortized time complexity of $\mathcal{O}(\sqrt{n} \log n)$ per update.

In order to do this, we would need to maintain a predecessor/successor data structure (such as a balanced binary search tree [3]), $\mathcal{D}_{\mathcal{U}}$, to maintain the updates in ascending order of their ending time. We can now follow algorithm 1 in order to compute updates and return $|S|$. We identify the set of all insertion updates as $\mathcal{U}_I$ and the set of all deletion updates as $\mathcal{U}_D$.

Algorithm 1 Dynamic Query Processing

```
1: $n_u \leftarrow 0$                                        ▷ Counter for updates
2: procedure ProcessUpdate($u$)
3:     $I_D \leftarrow [\,]$
4:     $\mathcal{A}_n \leftarrow 0$
5:     $\Gamma \leftarrow 0$          ▷ assuming 0 to be index of interval with least ending time
6:     $j \leftarrow n + 1$
7:     if $n_u < \sqrt{n}$ then
8:         $\mathcal{D}_{\mathcal{U}}$.append($u$)
9:         for each update $u_i$ in $\mathcal{D}_{\mathcal{U}}$ do
10:            $\alpha_j \leftarrow u_i$.interval
11:            $\alpha_l \leftarrow findLC(j, \Gamma)$
12:            $\alpha_k \leftarrow findNC(l)$
13:            if $u_i \in \mathcal{U}_I \ \wedge \ b_k \geq b_j$ then
14:                $\mathcal{A}_n \leftarrow \mathcal{A}_n +$ size of $GSS_\Gamma$ till $\alpha_l + 1$
15:                $\Gamma \leftarrow findNCbyTime(b_j)$
16:            else if $u_i \in \mathcal{U}_D$ then
17:                $\mathcal{A}_n \leftarrow \mathcal{A}_n +$ size of $GSS_\Gamma$ till $\alpha_l$
18:                $I_D$.append($j$)
19:                $b_j \leftarrow \infty$ in segment tree
20:                $\Gamma \leftarrow findNCbyTime(b_l)$
21:            end if
22:            $j \leftarrow j + 1$
23:        end for
24:        for each $m$ in $I_D$ do
25:            $b_m$ in segment tree $\leftarrow b_m$
26:        end for
27:        Return $\mathcal{A}_n$
28:    else
29:        Reconstruct all the structures (including the Greedy Tree and the Segment Tree)
           from scratch while incorporating the updates in $\mathcal{D}_{\mathcal{U}}$
30:        Clear $\mathcal{D}_{\mathcal{U}}$
```

```
31:         Return size of $GSS_0$
32:     end if
33: end procedure
```

4 Conclusion and Impact

In conclusion, this research undertakes a rigorous exploration of the fully dynamic interval scheduling problem, specifically focusing on optimizing time complexity in a single-machine environment. By introducing an algorithm with a time complexity of $\mathcal{O}(f \log n)$, where $f = |\mathcal{U}|$, for batch updates, this work makes a considerable contribution to the domain, especially for applications where batch updates are prevalent.

The algorithmic advancements in this research possess the potential to revolutionize real-time scheduling systems, ranging from job scheduling in data centers to resource allocation in cloud computing and real-time communication systems. By reducing the time complexity, these systems can function more efficiently and responsively, thereby reducing operational costs and increasing throughput.

References

1 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

2 Spencer Compton, Slobodan Mitrović, and Ronitt Rubinfeld. New partitioning techniques and faster algorithms for approximate interval scheduling, 2023. arXiv:2012.15002.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

4 Paweł Gawrychowski and Karol Pokorski. Sublinear dynamic interval scheduling (on one or multiple machines), 2022. arXiv:2203.14310.

5 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles. In Sergio Cabello and Danny Z. Chen, editors, *36th International Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12209, doi:10.4230/LIPIcs.SoCG.2020.51.

6 Benjamin Qi and Andrew Cheng. Heavy-light decomposition. URL: https://usaco.guide/plat/hld?lang=cpp.