New Partitioning Techniques and Faster Algorithms for Approximate Interval Scheduling

Spencer Compton
Stanford University, CA, USA

Slobodan Mitrović
University of California Davis, CA, USA

Ronitt Rubinfeld
MIT, Cambridge, MA, USA

Abstract

Interval scheduling is a basic problem in the theory of algorithms and a classical task in combinatorial optimization. We develop a set of techniques for partitioning and grouping jobs based on their starting and ending times, that enable us to view an instance of interval scheduling on *many* jobs as a union of multiple interval scheduling instances, each containing only *a few* jobs. Instantiating these techniques in dynamic and local settings of computation leads to several new results.

For $(1+\varepsilon)$-approximation of job scheduling of n jobs on a single machine, we develop a fully dynamic algorithm with $O(\log n/\varepsilon)$ update and $O(\log n)$ query worst-case time. Further, we design a local computation algorithm that uses only $O(\log N/\varepsilon)$ queries when all jobs are length at least 1 and have starting/ending times within $[0, N]$. Our techniques are also applicable in a setting where jobs have rewards/weights. For this case we design a fully dynamic *deterministic* algorithm whose worst-case update and query time are $\mathrm{poly}(\log n, \frac{1}{\varepsilon})$. Equivalently, this is *the first* algorithm that maintains a $(1+\varepsilon)$-approximation of the maximum independent set of a collection of weighted intervals in $\mathrm{poly}(\log n, \frac{1}{\varepsilon})$ time updates/queries. This is an exponential improvement in $1/\varepsilon$ over the running time of a randomized algorithm of Henzinger, Neumann, and Wiese [SoCG, 2020], while also removing all dependence on the values of the jobs' starting/ending times and rewards, as well as removing the need for any randomness.

We also extend our approaches for interval scheduling on a single machine to examine the setting with M machines.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Approximation algorithms analysis

Keywords and phrases interval scheduling, dynamic algorithms, local computation algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.45

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2012.15002`

Funding S. Compton was supported in part by the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program. S. Mitrović was supported by the Swiss NSF grant No. P400P2_191122/1, NSF award CCF-1733808, and FinTech@CSAIL. R. Rubinfeld was supported by the NSF TRIPODS program (awards CCF-1740751 and DMS-2022448), NSF award CCF-2006664, and FinTech@CSAIL.

Acknowledgements We thank Benjamin Qi (MIT) for helpful discussions.

1 Introduction

Job scheduling is a fundamental task in optimization, with applications ranging from resource management in computing [21, 22] to operating transportation systems [14]. Given a collection of *machines* and a set of *jobs* (or tasks) to be processed, the goal of job scheduling is to assign those jobs to the machines while respecting certain constraints. Constraints set on

© Spencer Compton, Slobodan Mitrović, and Ronitt Rubinfeld;
licensed under Creative Commons License CC-BY 4.0

jobs may significantly vary. In some cases a job has to be scheduled, but the starting time of its processing is not pre-specified. In other scenarios a job can only be scheduled at a given time, but there is a flexibility on whether to process the job or not. Frequent objectives for this task can include either maximizing the number of scheduled jobs or minimizing needed time to process all the given jobs.

An important variant of job scheduling is the task of *interval scheduling*: here each job has a specified starting time and its length, but a job is not required to be scheduled. Given M machines, the goal is to schedule as many jobs as possible. More generally, each job is also assigned a *reward* or weight, which can be thought of as a payment received for processing the given job. If a job is not processed, the payment is zero, i.e., there is no penalty. We refer to this variant as *weighted interval scheduling.* This problem in a natural way captures real-life scenarios. For instance, consider an assignment of crew members to flights, where our goal is to assign (the minimum possible) crews to the specified flights. In the context of interval scheduling, flights can be seen as jobs and the crew members as machines [14, 17]. Interval scheduling also has applications in geometrical tasks – it can be seen as a task of finding a collection of non-overlapping geometric objects. In this context, its prominent applications are in VLSI design [13] and map labeling [1, 25].

The aforementioned scenarios are executed in different computational settings. For instance, some use-cases are dynamic in nature, e.g., a flight gets cancelled. Then, in certain cases we have to make online decisions, e.g., a customer must know immediately whether we are able to accept its request or not. While in some applications there might be so many requests that we would like to design extremely fast ways of deciding whether a given request/job can be scheduled or not, e.g., providing an immediate response to a user submitting a job for execution in a cloud. In this work, our aim is to develop methods for interval scheduling that can be turned into efficient algorithms across many computational settings:

Can we design unified techniques for approximating interval scheduling very fast?

In this paper we develop fast algorithms for the dynamic and local settings of computation. We also give a randomized black-box approach that reduces the task of interval scheduling on multiple machines to that of interval scheduling on a single machine by paying only $2 - 1/M$ in the approximation factor for unweighted jobs, where M is the number of machines, and e in approximation factor for weighted jobs. A common theme in our algorithms is partitioning jobs over dimensions (time and machines). It is well studied in the dynamic setting how to partition the time dimension to enable fast updates. It is also studied how to partition over the machines to enable strong approximation ratios for multiple-machine scheduling problems. We design new partitioning methods for the time dimension (starting and ending times of jobs), introduce a partitioning method over machines, and examine the relationship of partitioning over the time dimension and machines simultaneously in order to solve scheduling problems. We hope that, in addition to improving the best-known results, our work provides a new level of simplicity and cohesiveness for this style of approach.

1.1 Computation Models

In our work, we focus on the following two models of computation.

Dynamic setting. Our algorithms for the fully dynamic setting design data structures that maintain an approximately optimal solution to an instance of the interval scheduling problem while supporting insertions and deletions of jobs/intervals. The data structures also support queries of the maintained solution's total weight and whether or not a particular interval is used in the maintained solution.

Local computation algorithms (LCA). The LCA model was introduced by Rubinfeld et al. [20] and Alon et al. [2]. In this setting, for a given job J we would like to output whether J is scheduled or not, but we do not have a direct access to the entire list of input jobs. Rather, the LCA is given access to an oracle that returns answers to questions of the form: "*What is the input job with the earliest ending time among those jobs that start after time x?*" The goal of the LCA in this setting is to provide (yes/no) answers to user queries that ask "Is job i scheduled?" (and, if applicable, "On which machine?"), in such a manner that all answers should be consistent with the same valid solution, while using as few oracle-probes as possible.

1.2 Our Results

Our first result, given in Section 4, focuses on designing an efficient dynamic algorithm for unweighted interval scheduling on a single machine. Prior to our work, the state-of-the-art result for this unweighted interval scheduling problem was due to [4], who design an algorithm with $O(\log n/\varepsilon^2)$ update and query time. We provide an improvement in the dependence on ε.

▶ **Theorem 1** (Unweighted dynamic, single machine). *Let $\mathcal{J}$ be a set of n jobs. For any $\varepsilon > 0$, there exists a fully dynamic algorithm for $(1+\varepsilon)$-approximate unweighted interval scheduling for $\mathcal{J}$ on a single machine performing updates in $O\left(\frac{\log(n)}{\varepsilon}\right)$ and queries in $O(\log(n))$ worst-case time.*

Theorem 1 can be seen as a warm-up for our most challenging and technically involved result, which is an algorithm for the dynamic *weighted* interval scheduling problem on a single machine. We present our approach in detail in the full version. As a function of $1/\varepsilon$, our result constitutes an exponential improvement compared to the running times obtained in [12]. We also remove all use of randomness, remove all dependence on the job starting/ending times (previous work crucially used assumptions on the coordinates to bound the ratio of jobs' lengths by a parameter N), and remove all dependence on the value of the job rewards.

▶ **Theorem 2** (Weighted dynamic, single machine). *Let $\mathcal{J}$ be a set of n weighted jobs. For any $\varepsilon > 0$, there exists a fully dynamic algorithm for $(1+\varepsilon)$-approximate weighted interval scheduling for $\mathcal{J}$ on a single machine performing updates and queries in worst-case time $T \in \mathrm{poly}(\log n, \frac{1}{\varepsilon})$. The exact complexity of T is given by*

$$O\left(\frac{\log^{12}(n)}{\varepsilon^7} + \frac{\log^{13}(n)}{\varepsilon^6}\right).$$

1.2.1 Implications in Other Settings

Local Computation Algorithms. We show that the ideas we developed to obtain Theorem 1 can also be efficiently implemented in the local setting, as we explain in detail in the full version and prove the following claim. This is the first non-trivial local computation algorithm for the interval scheduling problem.

▶ **Theorem 3** (Unweighted LCA, single machine). *Let $\mathcal{J}$ be a set of n jobs with length at least 1 and ending times upper-bounded by N. For any $\varepsilon > 0$, there exists a local computation algorithm for $(1+\varepsilon)$-approximate unweighted interval scheduling for $\mathcal{J}$ on a single machine using $O\left(\frac{\log N}{\varepsilon}\right)$ probes.*

Multiple machines. By building on techniques we introduced to prove Theorems 1 and 3, we show similar results in the full version in the case of interval scheduling on multiple machines at the expense of slower updates. To the best of our knowledge, these results initiate a study of dynamic and local interval scheduling in the general setting, i.e., in the setting of maximizing the total reward of jobs scheduled on multiple machines.

1.3 Related Work

The closest prior work to ours is that of Henzinger et al. [12] and of Bhore et al. [4]. [12] studies $(1+\varepsilon)$-approximate dynamic interval scheduling for one machine in both the weighted and unweighted setting. Unlike our main result in Theorem 2, they assume jobs have rewards within $[1, W]$, assume jobs have length at least 1, and assume all jobs start/end within times $[0, N]$. They obtain randomized algorithms with $O(\exp(1/\varepsilon)\log^2 n \cdot \log^2 N)$ update time for the unweighted and $O(\exp(1/\varepsilon)\log^2 n \cdot \log^5 N \cdot \log W)$ update time for the weighted case. They cast interval scheduling as the problem of finding a maximum independent set among a set of intervals lying on the x-axis. The authors extend this setting to multiple dimensions and design algorithms for approximating maximum independent set among a set of d-dimensional hypercubes, achieving a $(1+\varepsilon)2^d$-approximation in the unweighted and a $(4+\varepsilon)2^d$-approximation in the weighted regime.

The authors of [4] primarily focus on the unweighted case of approximating maximum independent set of a set of cubes. For the 1-dimensional case, which equals interval scheduling on one machine, they obtain $O(\log n/\varepsilon^2)$ update time, which is slower by a factor of $1/\varepsilon$ than our approach. They also show that their approach generalizes to the d-dimensional case, requiring $\operatorname{poly}\log n$ amortized update time and providing $O(4^d)$ approximation.

The problem of dynamically maintaining an exact solution to interval scheduling on one or multiple machines is studied by [11]. They attain a guarantee of $\tilde{O}(n^{1/3})$ update time for unweighted interval scheduling on $M = 1$ machine, and $\tilde{O}(n^{1-1/M})$ for $M \geq 2$. Moreover, they show an almost-linear time conditional hardness lower bound for dynamically maintaining an exact solution to the weighted interval scheduling problem on even just $M = 1$ machine. This further motivates work such as ours that dynamically maintains approximate solutions for weighted interval scheduling.

The authors of [9] consider dynamic interval scheduling on multiple machines in the setting in which all the jobs must be scheduled. The worst-case update time of their algorithm is $O(\log(n) + d)$, where d refers to the depth of what they call *idle intervals* (depth meaning the maximal number of intervals that contain a common point); they define an idle interval to be the period of time in a schedule between two consecutive jobs in a given machine. The same set of authors, in [10], study dynamic algorithms for the monotone case as well, in which no interval completely contains another one. For this setup they obtain an algorithm with $O(\log(n))$ update and query time.

In the standard model of computing (i.e. one processor, static), there exists an $O(n+m)$ running time algorithm for (exactly) solving the unweighted interval scheduling problem on a single machine with n jobs and integer coordinates bounded by m [8]. An algorithm with running time independent of m is described in [24], where it is shown how to solve this problem on M machines in $O(n\log(n))$ time. An algorithm is designed in [3] for weighted interval scheduling on M machines that runs in $O(n^2\log(n))$ time.

We refer a reader to [14] and references therein for additional applications of the interval scheduling problem.

Other related work. There has also been a significant interest in job scheduling problems in which our goal is to schedule *all* the given jobs across multiple machines, with the objective to minimize the total scheduling time. Several variants have been studied, including setups which allow preemptions, or setting where jobs have precedence constraints. We refer a reader to [15, 7, 19, 23, 5, 18, 16] and references therein for more details on these and additional variants of job scheduling. Beyond dynamic algorithms for approximating maximum independent sets of intervals or hypercubes, [6] show results for geometric objects such as disks, fat polygons, and higher-dimensional analogs. After we had published a preprint of this work, [6] proved a result that captures Theorem 1 with a more general class of fat objects.

2 Overview of Our Techniques

Our primary goal is to present unified techniques for approximating scheduling problems that can be turned into efficient algorithms for many settings. In this section, we discuss key insights of our techniques.

In the problems our work tackles, partitioning the problem instance into independent, manageable chunks is crucial. Doing so enables an LCA to determine information about a job of interest without computing an entire schedule, or enables a dynamic data structure to maintain a solution without restarting from scratch.

2.1 Unweighted Interval Scheduling – Partitioning Over Time (Section 4)

For simplicity of presentation, we begin by examining our method for partitioning over time for just the unweighted interval scheduling problem on one machine (i.e., $M = 1$). In particular, we first focus on doing so for the dynamic setting.

Recall that in this setting the primary motivation for partitioning over time, is to divide the problem into independent, manageable chunks that can be utilized by a data structure to quickly modify a solution while processing an update. In our work, we partition the time dimension by maintaining a set of *borders* that divide time into some number of contiguous regions. By doing so, we divide the problem into many *independent regions*, and we ignore jobs that intersect multiple regions; equivalently, we ignore jobs that contain a border. Our goal is then to dynamically maintain borders in a way such that we can quickly recompute the optimal solution completely within some region, and that the suboptimality introduced by these borders does not affect our solution much. In Section 4, we show that by maintaining borders where the optimal solution inside each region, i.e., a time-range between two borders, is of size $\Theta(\frac{1}{\varepsilon})$, we can maintain a $(1+\varepsilon)$-approximation of an optimal solution as long as we optimally compute the solution within each region.

Here, the underlying intuition is that because each region has a solution of size $\Omega(\frac{1}{\varepsilon})$, we can charge any suboptimality caused by a border against the selected jobs in an adjacent region. Likewise, because each region's solution has size $O(\frac{1}{\varepsilon})$, we are able to recompute the optimal solution within some region quickly using a balanced binary search tree. We dynamically maintain borders satisfying our desired properties by adding a new border when a region becomes too large, or merging with an adjacent region when a region becomes too small. As only $O(1)$ regions will require any modification when processing an update, this method of partitioning time, while simple, enables us to improve the fastest known update/query time to $O(\log(n)/\varepsilon)$.[1] In Section 2.2 we build on these ideas to design an algorithm for the weighted interval scheduling problem.

2.2 Weighted Interval Scheduling

In our most technically involved result, we design the first deterministic $(1+\varepsilon)$ approximation algorithm for weighted interval scheduling that runs in $\mathrm{poly}(\log n, \frac{1}{\varepsilon})$ time. In this section we give an outline of our techniques and discuss key insights. For full details we refer a reader to the full version.

2.2.1 Job data structure

Let $\mathcal{E}$ be the set of all the endpoints of given jobs, i.e., $\mathcal{E}$ contains s_i and f_i for each job $[s_i, f_i]$. We build a hierarchical data structure over $\mathcal{E}$ as follows. This structure is organized as a binary search tree T. Each node Q of T contains value $\text{KEY}(Q) \in \mathcal{E}$, with "1-1" mapping between $\mathcal{E}$ and the nodes of T. Each node Q is *responsible for a time range.* The root of T, that we denote by Q_{root}, is responsible for the entire time range $(-\infty, \infty)$. Each node Q has at most two children, that we denote by Q_L and Q_R. If Q is responsible for the time range $[X, Y]$, then Q_L is responsible for $[X, \text{KEY}(Q)]$, while Q_R is responsible for $[\text{KEY}(Q), Y]$.

Jobs are then assigned to nodes, where a job J is assigned to every node Q such that J is contained within the Q's responsible time range.

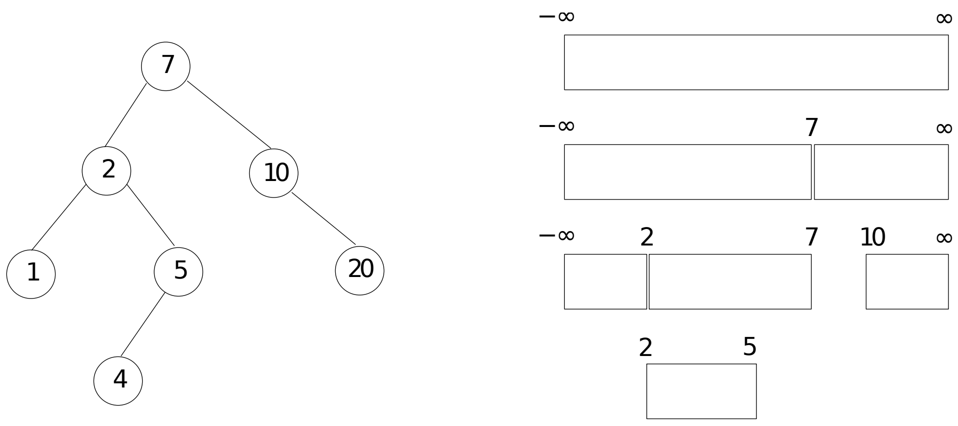

Figure 1 Visual example for hierarchical decomposition. Consider we are given jobs with the following ranges of $(1,5), (2,10), (7,20), (4,5)$. On the left is T, a balanced binary search tree over the set of all s_i and f_i. On the right is the hierarchical decomposition that corresponds to T. That is, in each row, the intervals on the right correspond to the $[l_Q, r_Q]$ for the nodes on the left. For instance, in the third row, $(-\infty, 2]$ corresponds to the node Q with $KEY(Q) = 1$.

[1] The main advantage of this techniques is that it leads to worst-case $O(\log{(n)}/\varepsilon)$ update time, as opposed to only an amortized one. We point out that it is not difficult to obtain such amortized guarantee in the following way: after each $\varepsilon \cdot OPT$ many updates, recompute the optimum solution from scratch. Given access to the balanced binary tree structure described above, this re-computation can be done in $O(OPT \cdot \log n)$ time.

2.2.2 Organizing computation

We now outline how the structure T is used in computation. As a reminder, our main goal is to compute a $(1+\varepsilon)$-approximate weighted interval scheduling. This task is performed by requesting Q_{root} to solve the problem for the range $(-\infty, \infty)$. However, instead of computing the answer for the entire range $(-\infty, \infty)$ directly, Q_{root} *partitions* the range $(-\infty, \infty)$ into:

- a number of ranges over which it is relatively easy to compute approximate solutions, such ones are called *sparse*, and
- the remaining ranges over which it is relatively hard to compute approximate solutions at the level of Q_{root}.

These hard-to-approximate ranges are deferred to the children of Q_{root}, and are hard to approximate because any near-optimal solution for the range contains many jobs. On the other hand, solutions in sparse ranges are of size $O(1/\varepsilon)$. As we discuss later, approximate optimal solutions within sparse ranges can be computed very efficiently; for details, see the paragraph *Approximate dynamic programming* below.

In general, a child Q_C of Q_{root} might receive *multiple* ranges from Q_{root} for which it is asked to find an approximately optimal solution. Q_C performs computation in the same manner as Q_{root} did – the cell Q_C partitions each range it receives into "easy" and "hard" to compute subranges. The first type of subranges is computed by Q_C, while the second type if deferred to the children of Q_C. The same as in Section 2.3, these "hard" ranges have large weight and allow for drawing a boundary and hence dividing a range into two or more *independent* ranges. We now discuss how the partitioning into ranges is undertaken.

2.2.3 Auxiliary data structure

To divide a range into "easy" and "hard" ranges at the level of a node Q, we design an auxiliary data structure, which relates to a rough approximation of the problem. This structure, called $Z(Q)$, maintains a set of points (we call these points *grid endpoints*) that partition Q into *slices of time*. We use slice to refer to a time range between two *consecutive* points of $Z(Q)$. Recall how for unweighted interval scheduling, we maintained a set of borders and ignored a job that crossed any border. In the weighted version, we will instead use $Z(Q)$ as a set of partitions from which we will use *some subset* to divide time. Our method of designing $Z(Q)$ reduces the task of finding a partitioning over time $Z(Q)$ within a cell for the $(1+\varepsilon)$-approximate weighted interval scheduling problem to finding multiple partitionings for the $(1+\varepsilon)$-approximate unweighted problem.

It is instructive to think of $Z(Q)$ in the following way. First, we view weighted interval scheduling as $O(\log n)$ independent instances of unweighted interval scheduling – instance i contains the jobs having weights in the interval $(w_{max}(Q)/2^{i+1}, w_{max}(Q)/2^i]$. Then, for each unweighted instance we compute borders as described in Section 2.1. $Z(Q)$ constitutes a subset of the union of those borders across all unweighted instances. We point out that the actual definition of $Z(Q)$ contains some additional points that are needed for technical reasons, but in this section we will adopt this simplified view. In particular, as we will see, $Z(Q)$ is designed such that the optimal solution within each slice has small total reward compared to the optimal solution over the entirety of Q. This enables us to partition the main problem into subproblems such that the suboptimality of discretizing the time towards slices, that we call *snapping*, is negligible.

However, a priori, it is not even clear that such structure $Z(Q)$ exists. So, one of the primary goals in our analysis is to show that there exists a near-optimal solution of a desirable structure that can be captured by $Z(Q)$. The main challenge here is to detect/localize sparse and dense ranges efficiently and in a way that yields a fast dynamic algorithm. As an oversimplification, we define a solution as having *nearly-optimal sparse structure* if it can be generated with roughly the following process:

- Each cell Q receives a set of disjoint time ranges for which it is supposed to compute an approximately optimal solution using jobs assigned to Q or its descendants. Each received time range must have starting and ending time in $Z(Q)$.
- For each time range $\mathcal{R}$ that Q receives, the algorithm partitions $\mathcal{R}$ into disjoint time ranges of three types: sparse time ranges, time ranges to be sent to Q_L for processing, and time ranges to be sent to Q_R for processing. In particular, this means that subranges of $\mathcal{R}$ are deferred to the children of Q for processing.
- For every sparse time range, Q computes an optimal solution using at most $1/\varepsilon$ jobs.
- The union of the reward/solution of all sparse time ranges on all levels must be a $(1+\varepsilon)$-approximation of the globally optimal solution without any structural requirements.

Moreover, we develop a *charging method* that enables us to partition each cell with only $|Z(Q)| = \text{poly}(1/\varepsilon, \log(n))$ points and still have the property that it contains a $(1+\varepsilon)$-approximately optimal solution with nearly-optimal sparse structure. Then, we design an approximate dynamic programming approach to efficiently compute near-optimal solutions for sparse ranges. Combined, this enables a very efficient algorithm for weighted interval scheduling. On a high-level, $Z(Q)$ enables us to eventually decompose an entire solution into sparse regions.

2.2.4 The charging method

We now outline insights of our charging arguments that enable us to convert an optimal solution OPT into a near-optimal solution OPT' with nearly-optimal sparse structure while relaxing our partitioning to only need $|Z(Q)| = \text{poly}(1/\varepsilon, \log(N))$ points. For a visual aid, see Figure 2.

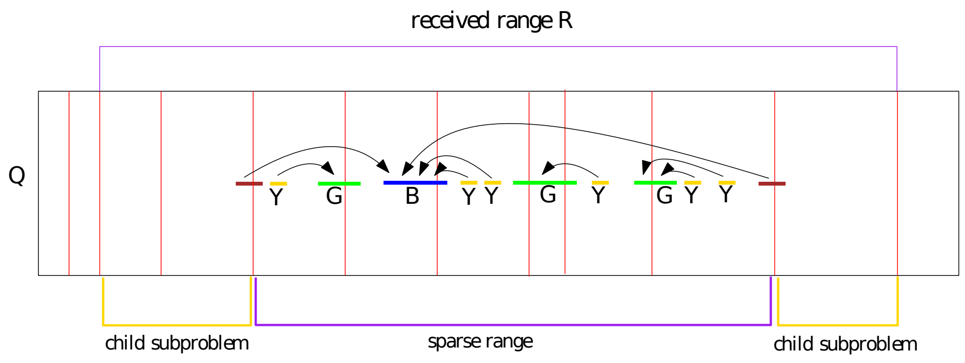

Figure 2 Visual example for charging argument.

As outlined in our overview of the nearly-optimal sparse structure, each cell Q receives a set of disjoint time ranges, with each time range having endpoints in $Z(Q)$, and must split them into three sets: sparse time ranges, time ranges for Q_L, and time ranges for Q_R. We will now modify OPT by deleting some jobs. This new solution will be denoted by OPT' and will have the following properties:

(1) OPT' exhibits nearly-optimal sparse structure; and

(2) OPT' is obtained from OPT by deleting jobs of total reward at most $O(\varepsilon \cdot w(OPT))$.

We outline an example of one such time range a cell Q may receive in Figure 2, annotated by "received range $\mathcal{R}$". We will color jobs in Figure 2 to illustrate aspects of our charging argument, but note that jobs do not actually have a color property beyond this illustration. Since our structure only allows a cell Q to use a job within its corresponding time range, any relatively valuable job that crosses between Q_L and Q_R must be used now by Q putting it in a sparse time range. One such valuable job in Figure 2 is in blue marked by "B". To have "B" belong to a sparse range, we must divide the time range $\mathcal{R}$ somewhere, as otherwise our solution in the received range will be dense. If we naively divide $\mathcal{R}$ at the partition of $Z(Q)$ to the left and right of the job "B", we might be forced to delete some valuable jobs; such jobs are pictured in green and marked by "G". Instead, we expand the division outwards in a more nuanced manner. Namely, we keep expanding outwards and looking at the job that contains the next partition point (if any). If the job's value exceeds a certain threshold, as those pictured as green and marked by "G" in Figure 2, we continue expanding. Otherwise, the job crossing a partition point is below a certain threshold, pictured as brown and not marked in Figure 2, and its deletion can be charged against the blue job. We delete such brown jobs and the corresponding partition points, i.e., the vertical red lines crossing those brown jobs, constitute the start and the end of the sparse range. By the end, we decided the starting and ending time of the sparse range, and what remains inside are blue job(s), green job(s), and yellow job(s) (also marked by "Y"). Note that yellow jobs must be completely within a partition slice of $Z(Q)$. Since we define $Z(Q)$ such that the optimal total reward within any grid slice is small, the yellow jobs have relatively small rewards compared to the total reward of green and blue jobs that we know must be large. Accordingly, we can delete the yellow jobs (to help make this time range's solution sparse) and charge their cost against a nearby green or blue job. In Figure 2, an arrow from one job to another represents a deleted job pointing towards the job who we charge its loss against. Finally, each sparse range contains only green job(s) and blue job(s). If there are more than $1/\varepsilon$ jobs in such a sparse range, we employ a simple sparsifying step detailed in the full proof.

It remains to handle the time ranges of the received range that were not put in sparse ranges. These will be time ranges that are sent to Q_L and Q_R. In Figure 2, these ranges are outlined in yellow and annotated by "child subproblem". However, the time ranges do not necessarily align with $Z(Q_L)$ or $Z(Q_R)$ as is required by nearly-optimal sparse structure. We need to adjust these ranges such that they align with $Z(Q_L)$ or $Z(Q_R)$ so we can send the ranges to the children. See Figure 3 for intuition on why we cannot just immediately "snap" these child subproblems to the partition points in $Z(Q_L)$ and $Z(Q_R)$. (We say that a range $\mathcal{R}$ is *snapped* inward (outward) within cell Q if $\mathcal{R}$ is shrunk (extended) on both sides to the closest points in $Z(Q)$. Inward snapping is illustrated in Figure 3.) Instead, we employ a similar charging argument to deal with snapping. As an analog to how we expanded outwards from the blue job for defining sparse ranges, we employ a charging argument where we contract inwards from the endpoints of the child subproblem. In summary, these charging arguments enabled us to show a solution of nearly-optimal sparse structure exists even when only partitioning each cell Q with $|Z(Q)| = \text{poly}(1/\varepsilon, \log(n))$ points.

2.2.5 Approximate dynamic programming

Now, we outline our key advance for more efficiently calculating the solution of nearly-optimal sparse structure. This structure allows us to partition time into ranges with sparse solutions. More formally, we are given a time range and we want to approximate an optimal solution within that range that uses at most $1/\varepsilon$ jobs. We outline an approximate dynamic programming approach that only requires polynomial time dependence on $1/\varepsilon$.

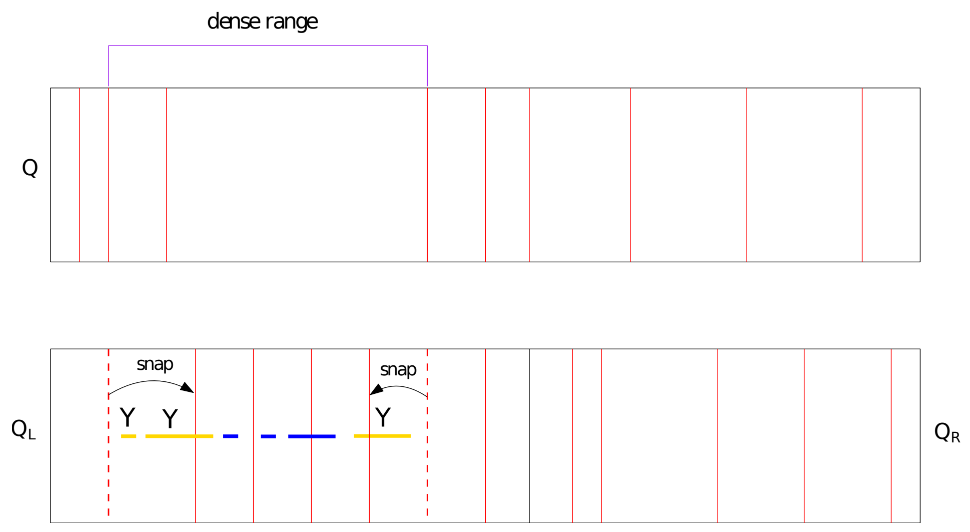

Figure 3 This example illustrates why the snapping we perform has to be done with care. The horizontal segments in this figure represent jobs. We show an initial dense range (outlined in purple) with endpoints in $Z(Q)$. With dashed vertical lines, we show where these endpoints are in Q_L. Importantly, they are not aligned with $Z(Q_L)$, i.e., the vertical dashed lines do not belong to $Z(Q_L)$. However, our structure *requires* that dense ranges align with $Z(Q_{child})$, so we must address this. If we were to naively snap the endpoints of the dense range inwards to the endpoints of $Z(Q_L)$, then we would need to delete some jobs (these deleted jobs are colored in yellow and marked by "Y"), while some other jobs would not be affected (like the remaining jobs in this example, those colored in blue). While this naive snapping may be fine in some cases, it will incur significant loss in cases in which the "Y" jobs have large weight. Notice that naively snapping outward to define a new region corresponding to the purple one is not a solution neither, as this could cause the dense time range to overlap with a previously selected sparse time range. Having overlapping ranges can cause us to choose intersecting jobs, and thus an invalid solution. Thus, we detail a more comprehensive manner of dealing with snapping.

The relatively well-known dynamic programming approach for computing weighted interval scheduling is to maintain a dynamic program where the state is a prefix range of time and the output is the maximum total reward that can be obtained in that prefix range of time. However, for our purposes, there are too many possibilities for prefix ranges of time to consider. Instead, we invert the dynamic programming approach, and have a state referencing some amount of reward, where the dynamic program returns the minimum length prefix range of time in which one can obtain a given reward. Unfortunately, there are also too many possible amounts of rewards. We observe that we do not actually need this exact state, but only an approximation. In particular, we show that one can round this state down to powers of $(1 + \varepsilon^2)$ and hence significantly reduce the state-space. In the full version, we show how one can use this type of observation to quickly compute approximate dynamic programming for a near-optimal sparse solution inside any time range.

2.2.6 Comparison with Prior Work

The closest to our work is the one of [12]. In terms of improvements, we achieve the following: we remove the dependence on N and $w_{\max}$ in the running-time analysis; we obtain a deterministic approach; and, we design an algorithm with $\text{poly}(1/\varepsilon, \log n)$ update/query time, which is exponentially faster in $1/\varepsilon$ compared the prior work.

In this prior work, jobs are assumed to have length at least 1 and belong in the time-interval $[1, N]$. To remove the dependence on N and such assumptions, we designed a new way of bookkeeping jobs. Instead of using a complete binary tree on $[1, N]$ to organize jobs as done in the prior work, we employ binary balanced search tree on the endpoints of jobs. A complete binary tree on $[1, N]$ is oblivious to the density of jobs. On the other hand, and intuitively, our approach allows for "instance-based" bookkeeping: the jobs are in a natural way organized with respect to their density. Resorting to this approach incurs significant technical challenges. Namely, the structure of solution our tree maintains is hierarchically organized. However, each tree update, which requires node-rotations, breaks this structure which requires additional care in efficiently maintaining approximate solution after an update, as well as requiring an entirely different approach for maintaining a partitioning of time $Z(Q)$ within cells. Moreover, we show how to further leverage these ideas to obtain a deterministic approach.

In our work, we use borders to define the so-called sparse and dense ranges. This idea is inspired by the work of [12]. We emphasize, though, that one of our main contributions and arguably the most technically involved component is showing how to algorithmically employ those borders in running-time only polynomially dependent on $1/\varepsilon$, while [12] require exponential dependence on $1/\varepsilon$.

Our construction of auxiliary data structure $Z(Q)$ enables us to boost an $O(\log(n))$-approximate solution into a decomposition enabling a $(1+\varepsilon)$-approximate solution is inspired by the approach of [12]. They similarly develop $Z(Q)$ to boost an instead $O(1)$-approximation that fundamentally relies on the bounded coordinate assumptions of jobs being within $[1, N]$ and having length at least 1. Our different approach towards $Z(Q)$ enables simplification of some arguments as well as not relying on randomness, or on length or bounded coordinate assumptions. Further, we note that the dynamic programming approach for sparse regions that we develop is significantly faster than the enumerative approach used in the prior work, that eventually enables us to obtain a $\text{poly}(1/\varepsilon)$ dependence in the running time. The way we combine solutions over sparse regions is similar to the way it is done in the prior work.

2.3 Localizing the Time-Partitioning Method

We also show that this method of partitioning over time can be used to develop local algorithms for interval scheduling. Here, we desire to answer queries about whether a particular job is in our schedule. We hope to answer each of these queries consistently (i.e., they all agree with some approximately optimal schedule) and in less time than it would take to compute an entire schedule from scratch. Partitioning over time seems helpful for this setting, because this would enable us to focus on just the region of the job being queried. However, our previously mentioned method for maintaining borders does so in a sequential manner that we can no longer afford to do in this model of computation. Instead, we use a hierarchical approach to more easily compute the locations of borders that create regions with solutions not too big or too small.

For simplicity, we again focus on the unweighted setting with only one machine. In the standard greedy algorithm for computing unweighted interval scheduling on one machine, we repeatedly select the job $successor(x)$: "*What is the interval with the earliest endpoint, of those that start after point x?*" (where x is the endpoint of the previously chosen job). As reading the entire problem instance would take longer than desired, an LCA requires some method of probing for information about the instance. Our LCA utilizes such successor probes to do so. For further motivation, see the full version. We outline a three-step approach towards designing an LCA that utilizes few probes:

Hierarchizing the greedy. Instead of just repeatedly using $\textit{successor}(x)$ to compute the solution as the standard greedy does, we add hierarchical structure that adds no immediate value but serves as a helpful stepping stone. Consider a *binary search tree* (BST) like structure, where the root node corresponds to the entire time range $[0, N]$. Each node in the structure has a left-child and a right-child corresponding to the 1st and the 2nd half, respectively, of that node's range. Eventually, leaf nodes have no children and correspond to a time range of length one unit. At a high-level, we add hierarchical structure by considering jobs contained in some node's left-child, then considering jobs that go between the node's left-child and right-child, and then considering jobs contained in the node's right-child. This produces the same result as the standard greedy, but we do so with a hierarchical structure that will be easier to utilize.

Approximating the hierarchical greedy. Now, we modify the hierarchical greedy so that it is no longer exactly optimal but is instead an approximation. At first this will seem strictly worse, but it will yield an algorithm that is easier to localize. When processing each node, we will first check whether it is the case that both the left-child and the right-child have optimal solutions of size $> \frac{1}{\varepsilon}$. A key observation here is that checking whether a time range has an optimal solution of size $> \frac{1}{\varepsilon}$ can be done by making at most $1 + \frac{1}{\varepsilon}$ successor probes (i.e., one does not necessarily need to compute the entire optimal solution to check if it is larger than some relatively small threshold). If both the left-child and the right-child would have optimal solutions of size $> \frac{1}{\varepsilon}$, then we can afford to draw a border at the midpoint of our current node and solve the left-child and right-child independently. Jobs intersecting a border are *ignored*, and we charge the number of such ignored jobs, i.e., the number of drawn borders, to the size of solution in the corresponding left- and right-child. Ultimately, we show that the addition of these borders makes our algorithm $(1 + \varepsilon)$-approximate. Moreover, and importantly, these borders introduce *independence* between children with large solutions.

Localizing the approximate, hierarchical greedy. Finally, we localize the approximate, hierarchical greedy. To do so, we note that when some child of a node has a small optimal solution, then we can get all the information we need from that child in $O(\frac{1}{\varepsilon})$ probes. As such, if a node has a child with a small optimal solution, we can make the required probes from the small child and recurse to the large child. Otherwise, if both children have large solutions, we can draw a border at the midpoint of the current node and only need to recurse down the child which contains the job the LCA is being queried about.

With these insights, we have used our partitioning method over time for local algorithms to produce an LCA only requiring $O(\frac{\log(N)}{\varepsilon})$ successor probes.

3 Problem Setup

In the interval scheduling problem, we are given n jobs and M machines. With each job j are associated two numbers s_j and $l_j > 0$, referring to "start" and "length" respectively, meaning that the job j takes l_j time to be processed and its processing can only start at time s_j. While prior work such as [12] used assumptions such as $s_j \geq 0, l_j \geq 1$ and have an upper-bound N on $s_j + l_j$, we utilize such assumptions *only in our LCA results*. In addition, with each job j is associated weight/reward $w_j > 0$, that refers to the reward for processing the job j. The task of *interval scheduling* is to schedule jobs across machines while maximizing the total reward and respecting that each of the M machines can process at most one job at any point in time.

4 Dynamic Unweighted Interval Scheduling on a Single Machine

In this section we prove Theorem 1. As a reminder, Theorem 1 considers the case of interval scheduling in which $w_j = 1$ for each j and $M = 1$, i.e., the jobs have unit reward and there is only a single machine at our disposal. This case can also be seen as a task of finding a maximum independent set among intervals lying on the x-axis. The crux of our approach is in designing an algorithm that maintains the following invariant:

▶ **Invariant 1.** *The algorithm maintains a set of borders such that an optimal solution schedules between $1/\varepsilon$ and $2/\varepsilon$ intervals within each two consecutive borders.*

We will maintain this invariant unless the optimal solution has fewer than $1/\varepsilon$ intervals, in which case we are able to compute the solution from scratch in negligible time. We aim for our algorithm to maintain Invariant 1 while keeping track of the optimal solution between each pair of consecutive borders. The high level intuition for this is that if we do not maintain too many borders, then our solution must be very good (our solution decreases by size at most one every time we add a new border). Furthermore, if the optimal solution within borders is small, it is likely easier for us to maintain said solutions. We prove that this invariant enables a high-quality approximation:

▶ **Lemma 4.** *A solution that maintains an optimal solution within consecutive pairs of a set of borders, where the optimal solution within each pair of consecutive borders contains at least K intervals, maintains a $\frac{K+1}{K}$-approximation.*

Proof. For our analysis, suppose there are implicit borders at $-\infty$ and $+\infty$ so that all jobs are within the range of borders. Consider an optimal solution OPT. We will now design a K-approximate optimal solution OPT' as follows: given OPT, delete all intervals in OPT that overlap a drawn border. Fix an interval J appearing in OPT but not in OPT'. Assume that J intersects the i-th border. Recall that between the $(i-1)$-st and the i-th border there are at least K intervals in OPT'. Moreover, at most one interval from OPT intersects the i-th border. Hence, to show that OPT' is a $\frac{K+1}{K}$-approximation of OPT, we can charge the removal of J to the intervals appearing between the $(i-1)$-st and the i-th border in OPT'. ◀

Not only does Invariant 1 enable high-quality solutions, but it also assists us in quickly maintaining such a solution. We can maintain a data structure with $O(\frac{\log(n)}{\varepsilon})$ updates and $O(\log(n))$ queries that moves the borders to maintain the invariant and thus maintains an $(1+\varepsilon)$-approximation as implied by Lemma 4.

▶ **Theorem 1** (Unweighted dynamic, single machine)**.** *Let $\mathcal{J}$ be a set of n jobs. For any $\varepsilon > 0$, there exists a fully dynamic algorithm for $(1+\varepsilon)$-approximate unweighted interval scheduling for $\mathcal{J}$ on a single machine performing updates in $O\left(\frac{\log(n)}{\varepsilon}\right)$ and queries in $O(\log(n))$ worst-case time.*

Proof. Our goal now is to design an algorithm that maintains Invariant 1, which by Lemma 4 and for $K = 1/\varepsilon$ will result in a $(1+\varepsilon)$-approximation of Maximum-IS.

On a high-level, our algorithm will maintain a set of borders. When compiling a solution of intervals, the algorithm will not use any interval that contains any of the borders, but proceed by computing an optimal solution between each two consecutive borders. The union of those between-border solutions is the final solution. Moreover, we will maintain the invariant that the optimal solution for every contiguous region is of size within $[\frac{1}{\varepsilon}, \frac{2}{\varepsilon})$.

In the rest, we show how to implement these steps in the claimed running time.

Maintained data-structures. Our algorithm maintains a balanced binary search tree T_{all} of intervals sorted by their starting points. Each node of T_{all} will also maintain the end-point of the corresponding interval. It is well-known how to implement a balanced binary search tree with $O(\log n)$ worst-case running time per insertion, deletion and search query. Using such an implementation, the algorithm can in $O(\log n)$ time find the smallest ending-point in a prefix/suffix on the intervals sorted by their starting-points. That is, in $O(\log n)$ time we can find the interval that ends earliest, among those that start after a certain time.

In addition, the algorithm also maintains a balanced binary search tree T_{borders} of the borders currently drawn.

Also, we will maintain one more balanced binary search tree T_{sol} that will store the intervals that are in our current solution.

We will use that for any range with optimal solution of size S, we can make $O(S)$ queries to these data structures to obtain an optimal solution for the range in $O(S \cdot \log n)$ time.

Update after an insertion. Upon insertion of an interval J, we add J to T_{all}. We make a query to T_{borders} to check whether J overlaps a border. If it does, we need to do nothing; in this case, we ignore J even if it belongs to an optimal solution. If it does not, we recompute the optimal solution within the two borders adjacent to J. If after recomputing, the new solution between the two borders is too large, i.e, it has at least $\frac{2}{\varepsilon}$ intervals, then draw/add a border between the $\frac{1}{\varepsilon}$-th and the $(1 + \frac{1}{\varepsilon})$-th of those intervals.

Update after a deletion. Upon deletion of an interval J, we delete J from T_{all}. If J was not in our solution, we do nothing else. Otherwise, we recompute the optimal solution within the borders adjacent to J and modify T_{sol} accordingly. Let those borders be the i-th and the $(i+1)$-st. If the new solution between borders i and $i+1$ now has size less than $1/\varepsilon$ (it would be size exactly $1/\varepsilon$), we delete an arbitrary one of the two borders (thus combining this region with an adjacent region). Then, we recompute the optimal solution within the (now larger) region J is in. If this results in a solution of size at least $2/\varepsilon$, we will need to split the newly created region by adding a border. Before splitting, the solution will have size upper-bounded by one more than the size of the solutions within the two regions before combining them as an interval may have overlapped the now deleted border (one region with size exactly $\frac{1}{\varepsilon} - 1$ and the other upper-bounded by $\frac{2}{\varepsilon} - 1$). Thus, the solution has size at in range $[2/\varepsilon, \frac{3}{\varepsilon})$. We can add a border between interval $1/\varepsilon$ and $1/\varepsilon + 1$ of the optimal solution, and will have a region with exactly $1/\varepsilon$ intervals and another with $[1/\varepsilon, 2/\varepsilon)$ intervals, maintaining our invariant.

In all of these, the optimal solution for each region has size $O(1/\varepsilon)$, so recomputing takes $O(\log(n)/\varepsilon)$ time.

For queries, we will have maintained T_{sol} in our updates such that it contains exactly the intervals in our solution. So each query we just need to do a lookup to see if the interval is in T_{sol} in $O(\log n)$ time. ◀

This result improves the best-known time complexities [4, 12]. Unfortunately, it does not immediately generalize well to the weighted variant. In the full version, we show our more technically-challenging result for the weighted variant.

References

1 Pankaj K Agarwal and Marc J Van Kreveld. *Label placement by maximum independent set in rectangles*, volume 1998. Utrecht University: Information and Computing Sciences, 1998.

2 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1132–1139. Society for Industrial and Applied Mathematics, 2012.

3 Esther M Arkin and Ellen B Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987.

4 Sujoy Bhore, Jean Cardinal, John Iacono, and Grigorios Koumoutsos. Dynamic geometric independent set. *arXiv preprint*, 2020. `arXiv:2007.08643`.

5 Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2012.

6 Jean Cardinal, John Iacono, and Grigorios Koumoutsos. Worst-case efficient dynamic geometric independent set. In *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204, page 25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

7 José R Correa and Andreas S Schulz. Single-machine scheduling with precedence constraints. *Mathematics of Operations Research*, 30(4):1005–1021, 2005.

8 A FRANK. Some polynomial algorithms for certain graphs and hypergraphs. In *Proceedings of the 5th British Combinatorial Conference, 1975*. Utilitas Mathematica, 1975.

9 Alexander Gavruskin, Bakhadyr Khoussainov, Mikhail Kokho, and Jiamou Liu. Dynamic interval scheduling for multiple machines. In *International Symposium on Algorithms and Computation*, pages 235–246. Springer, 2014.

10 Alexander Gavruskin, Bakhadyr Khoussainov, Mikhail Kokho, and Jiamou Liu. Dynamic algorithms for monotonic interval scheduling problem. *Theoretical Computer Science*, 562:227–242, 2015.

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

12 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic approximate maximum independent set of intervals, hypercubes and hyperrectangles. In *36th International Symposium on Computational Geometry (SoCG 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

13 Dorit S Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM (JACM)*, 32(1):130–136, 1985.

14 Antoon WJ Kolen, Jan Karel Lenstra, Christos H Papadimitriou, and Frits CR Spieksma. Interval scheduling: A survey. *Naval Research Logistics (NRL)*, 54(5):530–543, 2007.

15 Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

16 Elaine Levey and Thomas Rothvoss. A (1+ epsilon)-approximation for makespan scheduling with precedence constraints using lp hierarchies. *SIAM Journal on Computing*, pages STOC16–201, 2019.

17 Aristide Mingozzi, Marco A Boschetti, Salvatore Ricciardelli, and Lucio Bianco. A set partitioning approach to the crew scheduling problem. *Operations Research*, 47(6):873–888, 1999.

18 Michael Pinedo. *Scheduling*, volume 29. Springer, 2012.

19 Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 491–500, 2008.

20 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. *arXiv preprint*, 2011. `arXiv:1104.1377`.

21 Pinal Salot. A survey of various scheduling algorithm in cloud computing environment. *International Journal of Research in Engineering and Technology*, 2(2):131–135, 2013.

22 Raksha Sharma, Vishnu Kant Soni, Manoj Kumar Mishra, and Prachet Bhuyan. A survey of job scheduling and resource management in grid computing. *world academy of science, engineering and technology*, 64:461–466, 2010.

23 Martin Skutella and Marc Uetz. Stochastic machine scheduling with precedence constraints. *SIAM Journal on Computing*, 34(4):788–802, 2005.

24 Eva Tardos and Jon Kleinberg. Algorithm design, 2005.

25 Bram Verweij and Karen Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In *European Symposium on Algorithms*, pages 426–437. Springer, 1999.