Dynamic Scheduling Algorithms: Theory and Practice

Mikhail Kokho

A thesis submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy in Computer Science,
the University of Auckland, 2015.

December 2015

Abstract

In this thesis we investigate the dynamic problems of interval scheduling. The emphasis is on interval scheduling in the following contexts: single machine, multiple machines, and elastic mixed-criticality real-time systems. We analyze the complexity of these problems and find dynamic algorithms that efficiently maintain the interval set in the presence of real-time events.

Acknowledgements

Foremost, I am deeply grateful to my supervisor, Bakhadyr Khoussainov, for the continuous support, advice and encouragement. His guidance and intuition helped me to choose the direction of the research. He has taught me the art of research and critical thinking.

I would like to thank my colleague and co-author, Jiamou Liu, for the help during the first year of my study. The thesis would not have started without his mentorship.

I am indebted to my friends, Sasha and Sasha, for their time, discussions, suggestions and endless enthusiasm for mathematics.

Last, but not the least, I would like to thank my wife, Anastasia. My dear, you are a rock upon which I stand. Thank you for the support and inspiration.

Contents

List of Figures

Chapter 1

Introduction

We live in a world of limited resources. Time, space, materials, energy, knowledge, money, services, staff – everything we possess is finite. Having this in mind, we always want to get as much as possible from a resource. Searching for the best options forces people to make decisions constantly. Business sharks choose which sectors to invest in; airport dispatchers assign the order of landings and take-offs; computers distribute resources to maintain its processes; individuals daily plan how to manage their own time, which is, sadly, limited to only twenty-four hours a day. The list of problems, when one needs to allocate the possessed resources to various tasks, is endless.

While resources may take many forms – they can be lecture halls in a university, processors of a computer, gates in an airport, even mechanics in a repair shop – the range of possible tasks is also enormous. These can be classes at a university, threads of a computer program, landing aircrafts at an airport, or servicing of a car in a mechanics shop. Indeed, *scheduling*, the process of allocating limited resources to tasks, can be applied to a number of various areas. Clearly, the objective of any scheduling problem is to maximize the outcome of the completed activities, while not exceeding the available resources.

We investigate scheduling problems in *dynamic environment*. In contrast to static environment, where the number of available resources and the number of planned tasks are fixed, in dynamic environment these numbers are constantly changing. The number of resources changes in the event of a processor failure, illness of a lecturer, overloading of a server. The number of tasks changes when an unexpected task arrives, or a planned task is canceled. Moreover, even if the number of tasks remains the same, their properties, such as priority or duration, may change. All these changes result in reconsideration of the precomputed schedule.

In this thesis, we concentrate on the problems of scheduling intervals on single and multiple machines and scheduling of idle intervals in mixed-criticality real-time systems, subject to insertion and deletion of intervals. Our goal is to analyze the complexity of these problems and find dynamic algorithms that efficiently maintain the interval set in the presence of real-time events.

1.1 Background

A basic scheduling problem consists of a single machine and a set of *jobs*. Each job J_i is characterized by a *release time* r_i, a *deadline* d_i and a *length* ℓ_i. A *schedule* is a function σ that for each job specifies the time at which the machine starts processing the job. The machine completes processing a job J_i after ℓ_i units of time. A schedule is called *feasible* if no two jobs are processed at the same time and none of the jobs starts before its release time. A schedule is said to *meet the deadlines* every job is completed before its deadline. The goal is to find a feasible schedule such that it meets the deadlines and maximal completion time of all jobs is minimal possible.

The basic scheduling problem was shown to be NP-complete through the reduction from 3-partition problem [73]. However, with specific constraints on release times, lengths or deadlines of the jobs the problem becomes solvable in polynomial-time. If all jobs are released at the same time or their deadlines are equal, the algorithm in [67] finds the solution in $O(n \log n)$ time. If all jobs have unit-time length, then $O(n)$ time is sufficient to find a solution [47]. A more general algorithm for the jobs whose processing times are equal takes $O(n \log n)$ time [50].

In the situation when more than one machine is available, the problem of scheduling jobs with different processing length is NP-hard even if there are only two machines and all the jobs have equal deadlines and release times [73]. Therefore polynomial algorithms exist only for problems with jobs of unit-time or equal length. The problem of scheduling n jobs of equal length on m machines was shown to be solvable in $O(n^3 \log \log n)$ time [89], the result which was later improved to $O(mn^2)$ [90]. If all jobs have unit-time length, then an algorithm in [40] solves the problem in $O(n \log n)$ time simply by sorting jobs in non-decreasing order of their deadlines.

To cope with NP-hardness of scheduling problems, significant effort has been put into the development of approximation algorithms. An approximation algorithm finds a solution that is guaranteed to be at most ρ time worse than an optimal solution, where $\rho \geq 1$. Both the time complexity and the worst-case ratio bound ρ are considered when measuring the successfulness of an algorithm.

Many of the approximation algorithms for the single machine scheduling problem are based on the *Earliest Deadline First* (EDF) rule [67]: the jobs are scheduled in order of nondecreasing deadline. First to analyze EDF rule for the case when release times of jobs are arbitrary were Kise et al. [69]. They performed a thorough analysis of six approximation algorithms and showed that each of these algorithms has the worst-case ratio bound of 2. Later, Potts [84] proposed an $O(n^2 \log n)$ algorithm with the worst-case ratio bound of $3/2$. Another algorithm, with a better complexity of $O(n \log n)$ and the same ratio, was proposed by Nowicki and Smutnicki [83]. An improvement to the worst-case ratio bound at the cost of running time was obtained by Hall and Shmoys [61]. Their algorithm has the worst-case ratio bound of $4/3$ and complexity of $O(n^2 \log n)$.

In the case of multiple machines, one of the initial papers on the analysis of approximation algorithms is due to Graham [56]. He showed that if the jobs are scheduled in order of their availability, then the maximal completion time is at most $2 - \frac{1}{m}$ worse than of the optimal schedule. This scheduling approach is attractive because it does not require any computational power and can be applied to a wider range of scheduling problems. In his other work [57], Graham studied the *Longest Processing Time* (LPT) order. He showed that if jobs are sorted in LPT order, then the approximation ratio improves to $4/3 - 1/(3m)$. Moreover, in each of these publications Graham showed that the ratio bound is tight.

A completely different approach, called *multifit* (MF) was studied by Coffman et al. [29]. The MF algorithm applies binary search to find a minimal completion time C' such that all jobs are completed by C' on m machines. With every iteration, the algorithm attempts to fit the jobs on the machines. The jobs are processed in the order of non-increasing lengths. A job is placed on the first machine that has enough time before C' to complete the job. Depending on the success or failure of the attempt, the algorithm decreases or increases the value C'. Then the iteration is repeated. The time complexity of the algorithm is $O(n \log n + k \cdot n \log m)$ and the ratio is $1.22 + 2^{-k}$, where k is the number of iterations performed by the algorithm. The bound was improved by Friesen [49] from 1.22 to 1.2. Subsequently, Yue [103] improves the bound to $13/11$.

Hochbaum and Shmoys [64] further exploited the duality between scheduling on m machines and bin-packing problems. They considered each machine as a bin of capacity d, and for any $\epsilon > 1$ they define an approximation algorithm with the worst-case ratio bound of ϵ, which runs in time $O((n/\epsilon)^{1/\epsilon^2})$. They also designed special algorithms for $\epsilon = 1/5 + 2^{-k}$ and $\epsilon = 1/6 + 2^{-k}$, which have running times $O(n(k + \log n))$ and $O(n(km^4 + \log n))$.

Another way of solving the problem is to allow *preemption* of jobs. In a preemptive scheduling, execution of a job may be interrupted at any time and resumed at the same time on another machine or at a later time on any machine. Liu and Layland [77] studied EDF preempted scheduling of jobs on a single machine. According to EDF, a job J_i is preempted at time t by a newly released job J_k if J_k has a smaller deadline. They proved that there exists a feasible schedule that meets all deadlines if and only each job is completed before its deadline in the EDF schedule. Their proof is based on the following idea. Let S be a feasible schedule that meets the deadlines. Let J_1, J_2 be two jobs such that J_1 has a smaller deadline. If J_1 is scheduled after J in the schedule S, then we can exchange the execution time of jobs and still have a feasible schedule that meets the deadlines.

The problem of preemptive scheduling on multiple machines was solved by reduction to a network flow problem [see 19, ch. 5.1]. The complexity of the algorithm is $O(n^3)$. More efficient algorithms exist for the case when all jobs are released at the same time. Sahni [86] described an algorithm with $O(n \log nm)$ time complexity. Later, Baptiste [7] provided an algorithm with better complexity of $O(n \log n)$. If the problem is further simplified by disregarding deadlines, then the following algorithm due to McNaughton [79] solves the problem in $O(n)$ time.

First, compute $D = max\{max\{p_j\}, \sum \frac{p_j}{m}\}$. In other words, D is either the longest job or the average machine load. Then start placing the jobs on the machines in arbitrary order. Once the length of a schedule on a machine becomes greater than D, split the last job J_i into two parts of length $\ell_i - t$ and t and place the first part into the schedule of the next machine. Since there are mD units of processing time and $D - t \geq \ell_i - t$ for any t, all jobs are scheduled, and no job is simultaneously processed on two machines.

The problems described above are only a small part of the scheduling theory. There are roughly 900 different specifications of the basic scheduling problem [20]. The machines can be *identical*, *uniform* or *unrelated*. The number of machines can be fixed or arbitrary. For each job there might be a specific order of machines the job must visit. Moreover, in some cases a job may visit the same machine more than once. There might be precedence constraints that impose restrictions on the scheduling order, in the sense that some jobs must be completed before others. Finally, there are several optimization criteria, such as minimizing number of overdue jobs or minimizing maximal overdue time. We refer the reader to the following books for details. The book by Brucker [19] provides an elaborate formulation of scheduling problems and algorithms. The book by Leung [74] covers the advances in scheduling theory in different disciplines. A recent paper by Potts and Strusevich [85] gives a historical overview of the key developments in the theory of scheduling in the past fifty years.

1.2 Aplications

There are numerous applications of the scheduling theory to real-life problems, which range from the satellite photography to the organization of data on a hard drive. We provide examples of some problems that can be formulated in terms of jobs and machines.

Crew Scheduling. A general crew scheduling problem is formulated as follows [12, 81]. A set of tasks and a set of crews are given. Each task is associated with a starting time and a finishing time. All crews are identical and can perform any of the tasks. The goal is to minimize the number of crews needed for completion of the tasks. Clearly, looking at the crews as identical parallel machines, we have a variant of the scheduling problem.

The crew scheduling problem appears in the area of mass transportation. Bus [95, 102, 101], train [21, 22] and airline [36, 37] companies have timetables of their fleet. Each vehicle in the fleet must be operated by a crew. For a bus, a crew consists of only one member - a driver. For a plane, a crew consists of pilots and flight attendants. The goal of a company is to use a minimal number of crews to operate the planned timetable.

Airport Operations. There are scheduling problems at every stage of aircraft turn-around at an airport. The first problem appears when aircrafts are on their way to the airport. There are hundreds of arriving planes, but only dozens of gates. The scheduler must assign aircrafts to gates, taking into account walking distance for transfer, timetables of departing planes, compatibility between aircrafts and gates [78, 42, 60]. Then, while aircrafts are still in the air, the traffic control tower must assign the landing time and the runaway for each of the aircrafts [18, 44].

For safety reasons, there must be minimum time separation between landing of two aircrafts.

While on the ground, extremely tight flight schedules leave very little time for services that must be performed. After disembarkation of passengers, the interior of an aircraft must be cleaned [96]. Cleaning time depends on both the size of the aircraft and the size of the cleaning team. Thus we have a model where jobs are airplanes, the machines are cleaning teams, and the processing time of each job is different on different machines. At the same time, a team of engineers must carry out the inspection of the aircraft. An engineer must have a license for a corresponding type of an aircraft, but an engineer can have more that two licenses. Considering teams of engineers as machines, the scheduling problem of aircraft inspections is studied in [41].

Unexpected flight delays may be caused by improper scheduling of refueling tankers. A fuel tanker must fill an aircraft before its departure. The problem of minimizing refueling trucks required for aircraft servicing is studied in [5].

Other ground services are airline catering [63] and baggage handling [8, 28]. For catering, a delivery team, consisting of a driver and a loader, must load the food at the production unit and unload it into the aircraft. A delivery team member must have the skills for a specific type of aircraft. Similarly to refueling problem, the machines are delivery teams and the jobs are catering of flights. For baggage handling, there are piers where baggage is accumulated before being loaded into an aircraft. Each pier is collecting baggage from only one flight to avoid mishandling of luggage. Thus for each flight there is a job of baggage processing for the period of check-in. The machines are piers, where baggage is collected.

Finally, aircrafts are ready to depart. The problem of take-off order at London Heathrow airport is studied in a series of papers by Atkin et al. [2, 3, 4]. In their model, aircraft can be regarded as jobs with a target starting time. The length of a job depends on the size of an aircraft. For safety reasons, there is a minimum separation between two jobs, which depends on the weight categories of the two aircrafts. The re-ordering of jobs is restricted. The goal is to reduce delay with as little reordering as possible while obeying separation rules and ensuring that as many aircraft depart within their target time.

Classroom Assignment. Consider a university. It has a set of rooms and a set of classes. The rooms are continuously available and are used by lecturers to give classes. Each class has fixed starting time and finishing time. No two classes can meet in the same room at the same day and time. The goal is to find an assignment of classes to rooms that uses minimal number of rooms [46, 25].

A similar problem is an assignment of rooms for exams [34]. The main difference is that a room may be assigned to more than one exam, since each student works on his/her own questions. Moreover, there is a constraint that a student cannot take two exams in a period of several days.

Movie Screening. In a film industry, a movie generates most of its revenue during the first days of its release. After a week or two the interest to the movie, as well as the number of selling tickets, begin to drop. However, a movie theater cannot stop showing the movie and replace it with another one because there is a minimum period for which a movie must be screened continuously immediately after its release. Thus, a movie theater has a problem of choosing a set of films that will be on the theater's screens with the objective to maximize revenue [35]. In terms of scheduling, there is a set of weighted jobs and a set of machines. The goal is to schedule jobs to maximize the total weight of completed jobs.

Delivery Guarantee. A delivery guarantee is the commitment of a company to deliver the product within a specified period, or give a discount on the price. For example, a pizza company may guarantee 30-minutes delivery or give $3 discount. The profit-maximization model of delivery guarantee strategy was studies by Chatterjee et al. [26]. In their model, finding a schedule with a minimal number of overdue jobs is one of the factors that influences company's profit.

Bandwidth Trading. The backbone of the Internet is a physical infrastructure of cables and routers, which connects countries and continents. The backbone is maintained by about a dozen of network operators, called Tier 1 providers. Tier 1 providers sell the bandwidth of their network to the Internet Service Providers (ISPs), who in their turn sell the bandwidth to end-users. The contract between Tier 1 provider and an ISP specifies the bandwidth size and the period for which the bandwidth is used.

Bhatia et al. [13] studied the algorithmic aspects of bandwidth trading. In their model, they view the bandwidth contracts as a set of machines, and the reservation requests as jobs. There is a revenue associated with each job, and a cost is associated with each machine. Their goal is to either schedule all the jobs and minimize the incurred cost or maximize the revenue by using a fixed number of machines. The bandwidth allocation problem was also studied by Chen et al. [27].

Assignment problem. The problem comes from the area of combinatorial optimization. The general problem is formulated in terms of *agents* and *tasks*. A task can be assigned to any agent for some cost. The cost of assignment varies according to the pair agent-task. The constraints are that all tasks must be completed, and only one agent can perform a task. The goal is to minimize the total cost of the assignments.

Shmoys and Tardos [88] studied the general assignment problem as the scheduling of jobs on parallel machines with costs. It is easy to see the connection: an agent is a machine, and a task is a job. They designed a polynomial-time algorithm that given a budget C and amount of available time units T_i for which each machine is available returns a schedule of cost at most C where each machine is used for at most $2T$ time units, or proves that no such feasible schedule exists.

Interval Graphs. There exists a connection between scheduling theory and graph theory. Namely, some problems of scheduling jobs with fixed start and end times can be formulated in terms of an interval graph problem. A job has *fixed* start and end times if its length equals its dealing minus its release time [1]. Naturally, such job can be represented as an interval. An *interval graph* is a graph whose vertices can be represented as intervals such that two intervals intersect if and only if two corresponding vertices are connected by an edge [55, ch. 8]. Several linear time algorithms exist that verify whether a graph can be represented as a set of intervals [32, 33, 71].

Some of the problems on interval graphs can be formulated in scheduling terms. One problem is *k-coloring* of an interval graph. In this problem, we ask whether the graph vertices can be colored by k colors such that no two vertices of the same color are connected by an edge. The problem is equivalent to the problem of scheduling intervals on k machines. Indeed, a machine represents a color, and since two intersecting intervals cannot be allocated on the same machine, no two connected vertices are assigned the same color. The k-coloring problem was studied in [23, 16, 45]. Another problem is *maximal independent set*, which asks for a maximal set of pairwise disconnected vertices. In scheduling terms, the maximal independent set problem is to find a maximal possible set of intervals that can be completed on a single machine. Gupta et al. [58] solves this problem with a $O(n \log n)$ algorithm that processes intervals, representing a given graph.

1.3 Summary of Results

In the following, we present the topics and results obtained in each chapter of the thesis. The reader is referred to the relevant chapter for the formal definitions and proofs.

Chapter 2. Preliminaries

In this chapter, we introduce basic definitions, facts and algorithms for the standard scheduling problems. The terminology and concepts here are used throughout the thesis. The chapter is divided into three sections.

The first section concerns scheduling of intervals. An *interval* is a job such that its length equal to the difference between its deadline and its release time. We describe standard static algorithms that optimally schedule a set of intervals on single and multiple machines, and prove their time complexity.

The second section concerns real-time scheduling of tasks. A *task* is a generator of an infinite sequence of jobs. It generates jobs of the same length, but of different deadlines and release times. Naturally, only one job of a task can be active in the system. We give an overview of the main algorithms that schedule jobs, generated by a set of tasks.

The third section concerns amortized analysis. An *amortized analysis* is a powerful technique that one may use to find a tight upper bound on the time taken by a sequence of operations. Its power comes from the observation that only a small percentage of operations in a sequence show worst-case behavior. We describe the application of amortized analysis for the *splay tree* data structure, and show that a sequence of m splay tree operations takes $O(m \log n)$ time in the worst case [91].

Chapter 3. Dynamic Scheduling of Monotonic Intervals on Single Machine

In this chapter, we focus on the *dynamic* problem of monotonic interval scheduling on a single machine. An interval set is monotonic if no interval properly contains another interval. We design two data structures that minimize total running time of an arbitrary sequence of the following operations:

- insert(i), which adds an interval i into the interval set,

- remove(i), which deletes an interval i from the interval set,
- query(i), which returns true if and only if i belongs to the optimal set returned by the greedy algorithm.

The greedy algorithm (Algorithm 2.1) is the standard algorithm that solves the static variant of the problem. The algorithm starts by sorting intervals in the order of their finishing times. Once the sorting is complete, it marks the interval with the smallest finishing time as the member of an optimal set. Then, it visits intervals in this order and marks an interval that does not intersect with the last marked interval as the member of an optimal set. After all intervals have been visited, the algorithm returns the marked vertices as the *greedy optimal set* (Definition 2.1.5). The time complexity of the algorithm is $O(n \log n)$. It is known that a greedy optimal set is unique and it has maximal size.

In Section 3.1 we describe how to recognize monotonic interval sets.

In Section 3.2 we describe and analyze complexity of the operation right_compatible. The operation returns the next interval in the greedy optimal set. The complexity of the operation on a monotonic interval set is $\Theta(\log n)$ (Theorem 3.2.1).

In Section 3.3 we describe and analyze time complexity of three possible extensions of the greedy algorithm for the dynamic environment. The first approach is to keep intervals sorted in a binary search tree and apply the greedy algorithm for each query operation. This approach has poor performance on sequences with many query operations.

The second approach is to use an additional binary search tree as the container for the greedy optimal set. With a data structure that support splitting and merging of the trees, the greedy optimal set can be updated incrementally after each update of the interval set. However, we show how to construct a sequence of m insert operations with total running time of either $O(mn)$ or $O(mn \log n)$, depending on how we build the tree for the greedy optimal set.

Finally, the third approach is to use right_compatible(i) operation, which allows us to skip some intervals. With this operation, the construction of the greedy optimal set takes $\Theta(k \log n)$ time, where k is the size of the set. However, we describe a sequence of insert operations where every insertion results in a new greedy optimal set that contains half of the interval set. Thus, the total time taken by such sequence is in $\Theta(mn \log n)$.

On the contrary, our data structures have significantly lower upper bound. We prove that in our data structures the total running time of any sequence of m

operations is $O(m \log n)$. We apply amortized analysis to achieve this bound.

In section 3.4 we describe our first data structure. Let I be a set of interval. Let $\prec_{\mathsf{f}}$ denote the order of intervals by their finishing time. Given two intervals i and r, we call r *the right compatible interval* and denote is by $\mathsf{rc}(i)$ if for any $x \in I$ such that $i \prec_{\mathsf{f}} x \prec_{\mathsf{f}} r$ the intervals i and x overlap (Definition 2.1.1). We define *compatibility forest* (CF) as follows:

Definition 3.4.1 *A* compatibility forest *is a graph $\mathcal{F}(I) = (V, E)$ where $V = I$ and $(i, j) \in E$ if $j = \mathsf{rc}(i)$*

Essentially, the forest $\mathcal{F}(I)$ connects nodes by in accordance with the greedy algorithm: for any node i in the forest $\mathcal{F}(I)$, if the greedy algorithm starts at i, then the algorithm 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 $\prec_{\mathsf{f}}$-least interval is the greedy optimal set.

We describe a data structure that maintains compatibility forest of an interval set. Using amortized analysis, we prove the upper bound on the time complexity of the operations:

Theorem 3.4.4 *The algorithms* insertCF *and* removeCF *take $O(\log^2 n)$ amortised time. The algorithm* queryCF *take $O(\log n)$ amortised time.*

Furthermore, we prove that the bound of $O(\log^2 n)$ is sharp:

Theorem 3.4.5 *In* CF *data structure there exists a sequence of k update operations with $\Theta(k \log^2 n)$ total running time.*

In Section 3.5 we improve the upper bound of the update operations by designing a new data structure, which we call *linearised tree.* All operations of this data structure take amortized $O(\log n)$ time.

Call two intervals i and j *equivalent*, and denote it as $i \sim j$, if and only if $\mathsf{rc}(i) = \mathsf{rc}(j)$. We arrange intervals that are equivalent in a path using the $\prec_{\mathsf{f}}$-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 tuple $(I; E = E_{\sim} \cup E_c)$, where

- $(i, j) \in E_{\sim}$ 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.

The advantage of a linearised tree over a compatibility forest is that the linearised tree is a binary tree. Therefore, when we insert or remove an interval, we need to redirect at most two existing edges in the linearised tree. This allows to improve the amortized bound of operations. However, there is a disadvantage: a path in a linearised tree may contain intersecting intervals. Indeed, if two intervals are equivalent, then they intersect. We discuss this disadvantage in Section 3.6.

Using splay tree in the implementation of the linearised tree, we achieve the following result:

Theorem 3.5.8 *The* queryLT*,* insertLT *and* removeLT *operations solve the dynamic monotonic interval scheduling problem in $O(\log n)$ amortized time, where n is the size of the set I of intervals.*

Having two data structures of similar complexity, we empirically compare them in Section 3.7. In our experiments, we measure the total and the average running time of a sequence of m operations on initially empty interval set. The sequence consists of n insert operations, rn remove operations and qn query operation, where n is a linearly increasing number and r and q are fixed parameters of the experiment.

The experimental result verifies our theoretical analysis and shows that in a random environment CF performs as fast as LT to within a constant factor, despite the worst $(\log^2 n)$ time upper bound. Considering that CF is relatively easy to implement, CF can find its practical applications.

The results of this chapter have been published in [52, 54].

Chapter 4. Dynamic Interval Scheduling on Multiple Machines

In this chapter, we continue our study on dynamic interval scheduling. We consider the case of scheduling on multiple machines. Call two intervals *compatible* if they do not intersect. Given a set of intervals I, the static problem asks to find a partition of I such that the number of subsets in the partition is minimal possible and every subset contains pairwise compatible intervals. In the dynamic problem, the input is an arbitrary sequence of the update operations: insert(i) operation adds i into the interval set, and remove(i) deletes i from the set. As in the previous chapter, the goal is to find a data structure that minimizes the total running time of the sequence.

We design a data structure whose complexity of operations is $O(d + \log n)$, where d is the maximal number of pairwise intersecting intervals. Our data structure is based on the following observation: if two machines in a schedule have intersecting periods of idleness, then there exists a schedule where these idle periods are united. In Sections 4.1 and 4.2 we study the concepts of *idle intervals* and *nested scheduling*. In Section 4.3 we describe our data structure. Finally, in Section 4.4 we prove that the bound is tight.

Definition 2.1.9 *A function $\sigma : I \to \{1, \dots, k\}$ from a set of intervals I to a set of k natural numbers is called a* scheduling function *if any two distinct intervals $a, b \in I$ do not intersect whenever $\sigma(a) = \sigma(b)$.*

The *size* of a scheduling function is the cardinality of its codomain. For an interval a the $\sigma(a)$ is called *the schedule number.* A *schedule* S_i is a set of intervals with the same schedule number. The set of idle intervals of a schedule $S_i = \{a_i, \dots, a_z\}$ is

$$\text{Idle}(S_i) = \bigcup_{i=1}^{z-1} \{\, [f(a_i), s(a_{i+1})] \,\} \cup \{\, [-\infty, s(a_1)] \,\} \cup \{\, [f(a_z), \infty] \,\}$$

Now suppose we are inserting an interval a into a set I. We prove a theorem that describes the conditions when the size of a scheduling function does not change:

Theorem 4.1.4 *Let σ be a scheduling function for I. There exists an scheduling function of the same size for $I \cup \{a\}$ if and only if there exists a set of idle intervals $\mathcal{C} = \{c_1, \dots, c_z\}$ such that*

- *c_1 starts before a,*
- *c_z finishes after a,*
- *c_i starts before c_{i+1},*
- *$c_i \cap c_j \neq \emptyset$ if and only if $j = i + 1$.*

In the proof of the theorem, we provide a method that changes scheduling function without changing its size. The idea of the method is the following. Let

a, b be two idle intervals that intersect. Let $a_1, \dots, a_k$ and $b_1, \dots, b_\ell$ be all the intervals that start after a and b in the corresponding schedules. We move intervals $a_1, \dots, a_k$ to the schedule of b, and move intervals $b_1 \dots, b_\ell$ to the schedule of a. Since idle intervals a and b intersect, the new schedules consist of compatible intervals. See Figure 4.3, page 77 as an example of the theorem application.

From the theorem we imply that it undesirable to have sequences of intersecting idle interval. Therefore, in Section 4.2 we define the concept of *nested scheduling.*

Definition 4.2.1 *A set J of intervals is* nested *if there exists one interval that covers any other interval and for all $b_1, b_2 \in J$, it is either that b_1 covers b_2, or b_2 covers b_1, or b_1, b_2 are compatible.*

Definition 4.2.3 *We say that σ is a* nested scheduling function*, if its set of corresponding idle intervals is nested.*

We prove the optimality of nested scheduling. A nested scheduling function is optimal in a sense that there does not exist a scheduling function of smaller size. We use the known fact that the minimal possible size of a scheduling function is equal to the maximal number of pairwise intersecting intervals in the interval set (Lemma 2.1.11).

Theorem 4.2.4 *If σ is a nested scheduling function, then σ is optimal.*

Furthermore, we prove that any interval set possesses a nested scheduling. Given a nested scheduling of an interval set I we show how to construct a nested scheduling for a set $I \cup \{a\}$, where a is a new interval. Note that insertion of any interval breaks the nestedness of the function. In Section 4.2.1 we consider all possible case and establish the following theorem:

Theorem 4.2.6 *For any set of intervals I there exists a scheduling function σ such that its set of corresponding idle intervals is nested.*

In Section 4.3, taking into account the strong properties of nested scheduling, we design our data structure. Observe that any nested set of intervals defines a tree under set-theoretic inclusion $\subseteq$. Our data structure maintains nested scheduling as a tree of idle intervals, where the root is an interval of infinite length $[-\infty, \infty]$.

We use *interval tree* data structure to represent the nested tree of idle intervals (see Figure 4.16). Briefly, each internal node in an interval tree is associated with a number, and stores all intervals that contain this number. Traversing the tree, we can find the intervals that intersect some point p. Therefore, when inserting a new interval a, we can find idle intervals that intersect the endpoints of a. We show that a new interval a results in changes of $O(d)$ idle intervals. In the interval tree, these changes can be done in one traversal. Thus, we prove the following:

Theorem 4.3.2 *The data structure based on interval tree maintains the optimal scheduling and supports insertions and deletions in $O(d + \log n)$ worst-case time.*

Finally, in Section 4.4, we prove that the bound of $O(d + \log n)$ is sharp for any data structure that maintains a nested scheduling.

Theorem 4.4.4 *An update operation in a data structure representing a nested tree takes at least $\Omega(\log n + d)$ time.*

The results of this chapter have been published in [53].

Chapter 5. Dynamic Slack Reclamation from Multiple Processors

In this chapter, we define and analyze the problem of multiprocessor slack reclamation. The problem is a generalization of the slack reclamation problem from a single processor, which appears in the context of Elastic Mixed-Criticality (E-MC) model of real-time systems [97, 98].

Informally, the E-MC model consists of high-criticality and low-criticality tasks. Every task produces an infinite sequence of periodic jobs, which are executed by the processors. Each jobs has release time, deadline and execution requirement. The execution requirements of the jobs and their periodicity determine the utilization of the processors. We need to schedule jobs on the processors such that the total utilization is maximal possible, and the processors are not overloaded.

At run-time, a high-criticality task may take less time than it was provisioned by the scheduler, generating *slack* in the system. The slack is undesirable idleness of the processor. To increase processor utilization, a low-criticality task may release an early job, but only if there is enough slack in the system. The slack reclamation problem asks whether a processor has enough slack to complete an early job before its deadline.

In Section 5.2 we formally describe the E-MC model.

In multi-processor systems, it might be possible that each processor has some amount of slack, but none of the processors has enough slack for execution of an early job. However, similarly to the idea of nested scheduling (Theorem 4.1.4), it might be possible to reschedule the tasks such that the distributed slack is collected on one processor.

In Section 5.3 we propose the idea of *swap schedule*, which allows reclamation of slack from multiple processors. Let $\mathcal{C}_1, \ldots, \mathcal{C}_m$ be m processors of the system. A scheduling function σ partitions the tasks into m disjoint sets $\Gamma_1, \ldots, \Gamma_m$ such that the sum of tasks utilizations in every set does not exceed 1. Jobs generated by the tasks are executed in the order of earliest deadline.

At run-time, if the processor $\mathcal{C}_x$ has slack in the period $[a, b]$ and the processor $\mathcal{C}_y$ has slack in the period $[b, c]$, we add a *swap point* $(b, \mathcal{C}_x, \mathcal{C}_y)$ in the swap schedule. There is a constraint: both processors must not have an unfinished job. Later, when we reach the time b, we change the scheduling function σ such that the tasks executing on the processor $\mathcal{C}_x$ will be executing on the processor $\mathcal{C}_y$, and vise-versa (Figure 5.2, page 102). Thus, we combine slack from both processors on the processor $\mathcal{C}_x$.

In Section 5.4 we define the `k-SLACK_RECLAMATION` problem. The input to the problem is: a scheduling function σ; a set of periods of slack; a low-criticality task X that can release an early job. An algorithm, solving this problem, outputs YES, if there exists a feasible swap schedule such that the amount of slack reclaimed from k processors is greater or equal to the length of X. Otherwise, it outputs NO. We prove the following theorem:

Theorem 5.4.1 *The problem* `k-SLACK_RECLAMATION` *is NP-complete even if* $k = 2$.

Chapter 2

Preliminaries

2.1 Interval Scheduling

Consider the following scheduling problem that arises in every university. You have a lecture room, and many professors request to use the room for their lectures. Every lecture has a specific starting time s and finishing time f. The goal is to satisfy as many requests as possible with the natural constraint that no pair of professors give a lecture in the same room at the same moment of time.

More formally, there are n request, where each request i specifies the starting time $s(i) \in \mathbb{R}$ and the finishing time $f(i) \in \mathbb{R}$. Every request i have its starting time $s(i)$ strictly less than its finishing time $f(i)$. Without loss of generality we assume that the starting times and the finishing times of the requests are distinct. If it is not the case, we apply the following normalization procedure. Let i and j be two requests, ϵ be a small positive real number. Assume $f(i) \leq f(j)$. If the requests i and j are identical, that is their starting and finishing times are the same, we set $s(j) = s(j) + \epsilon$ and $f(j) = f(j) + \epsilon$. If only their finishing times are equal, we only add ϵ to $f(j)$. Finally, if $f(i) = s(j)$, that is these requests share only one moment of time, we add ϵ to $s(j)$.

It is convenient to look at the requests as *intervals* of time. Each request i corresponds to an interval $[s(i), f(i)]$. From the interval point of view, we refer to $s(i)$ as the *left endpoint* of i and to $f(i)$ as the *right endpoint* of i. Recall that an interval $[x, y]$ is a set of real numbers z such that $x \leq z \leq y$. Therefore, we can apply the usual set-theoretic operations - union, intersection, subset - to the intervals.

We define two orderings of the intervals. The first ordering, denoted by $\prec_{\mathsf{f}}$,

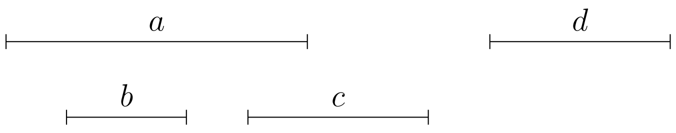


Figure 2.1: Example of relationships between the intervals: b and c are compatible, a and c overlap, b is nested in a. The interval d is the right compatible interval of a and c. The interval c is the left compatible interval of d.

sorts intervals by their finishing time from left to right. That is, an interval a is less than interval b in order $\prec_{\mathsf{f}}$ if $f(a) < f(b)$. The seconds ordering, denoted by $\prec_{\mathsf{s}}$, sorts intervals by their starting time, increasingly. That is, an interval a is less than interval b in order $\prec_{\mathsf{s}}$ if $s(a) < s(b)$. The order is strict since we assume the endpoints of all intervals are distinct. Throughout our work, by the $\prec_\gamma$-*least* interval, the $\prec_\gamma$-*greatest* interval, the $\prec_\gamma$-*next* interval, the $\prec_\gamma$-*previous* interval, we mean the least, greatest, next and previous interval with respect to $\prec_\gamma$, where $\gamma \in \{\mathsf{f}, \mathsf{s}\}$. When the order of intervals is clear from the context, we omit $\prec_\gamma$ prefix.

We say that two intervals are *compatible* if their intersection is an empty set. Otherwise, the intervals *overlap*. We say that an interval a *covers* an interval b if $s(a) < s(b)$ and $f(a) > f(b)$, or equivalently $b \supset a$. In this case, b is *nested* in a. As an example, consider Figure 2.1. The intervals a and c overlap, the intervals b and c are compatible, and the interval a covers the interval b.

Given an interval $i \in I$ we distinguish the *right compatible* interval and *the left* compatible interval:

Definition 2.1.1. *[Right Compatible Interval] An interval $r \in I$ is called* the right compatible interval *and denoted by $\mathsf{rc}(i)$ if for any $x \in I$ such that $i \prec_{\mathsf{f}} x \prec_{\mathsf{f}} r$ the intervals i and x overlap.*

Definition 2.1.2 (Left Compatible Interval)**.** *An interval $\ell \in I$ is called* the left compatible interval *and denoted by $\mathsf{lc}(i)$ if for any $x \in I$ such that $\ell \prec_{\mathsf{f}} x \prec_{\mathsf{f}} i$ the intervals i and x are incompatible.*

Note that if an interval r is the right compatible interval of an interval i, it is not alway the case that i is the left compatible interval of r. Consider intervals a, c and d on the example on Figure 2.1. The right compatible interval of a is d, but the left compatible interval of d is c, not a.

When we talk about sets of intervals, we say that a set of intervals I is *compatible* if all intervals in I are pairwise compatible. We say that a set of intervals I is *nested* if it has no overlapping intervals and there exists an interval that covers any other interval. Figure 2.2 gives an example of such sets.

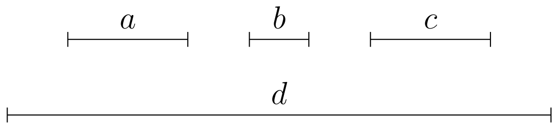


Figure 2.2: The set $\{a, b, c\}$ is a compatible set, while set $\{a, b, c, d\}$ is a nested set.

In the next two sections we describe two problems of interval scheduling. Given the set of intervals I, the first problem asks for a compatible subset of I that has maximal size. The second problem asks for a partition of I into the smallest number of subsets, where each subset is a compatible set of intervals.

2.1.1 Scheduling on Single Machine

Definition 2.1.3 (Interval Scheduling Problem)**.** *Given a set of intervals I, find a subset J such that J is a compatible set and it has maximal possible size.*

Let us discuss an instance of the problem on Figure 2.3. The interval set on this figure consists of 9 intervals. To find a compatible subset of maximal size, we start with a trivial subset J_1 containing only one interval a. We see that a overlap with the intervals b and c, but it is compatible with the interval g. Therefore we define another set $J_2 = \{a, g\}$. The intervals in J_2 overlap with intervals b, c, e, d, f, but they are compatible with intervals h and i. Since h and i overlap, we can add only one of them into J_2. We choose the interval i and define $J_3 = \{a, g, i\}$. At this step, each of the intervals not in J_3 overlap with at least one interval in J_3. However, does J_3 have a maximal size? It appears, that no, we can remove g and add intervals d and f. That is, we define the set $J_4 = \{a, d, f, i\}$. The set J_4 is a compatible set and has maximal size.

Definition 2.1.4. *A compatible subset of maximal size is called* optimal.

Note that there might be several optimal subsets. In the example on Figure 2.3 the set $J_5 = \{c, d, f, i\}$ is also optimal.

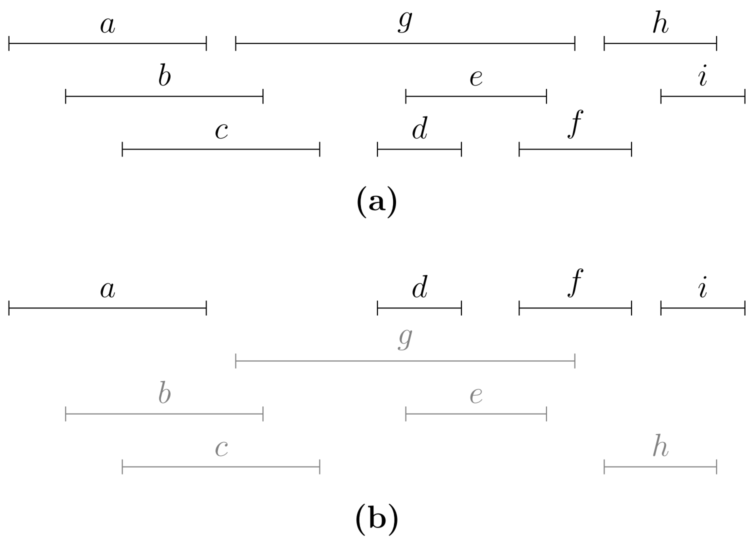


Figure 2.3: (a) Instance of an interval scheduling problem. (b) The set $\{a, d, f, i\}$ is a solution to the instance.

We describe the *greedy scheduling algorithm* [70] that finds an optimal subset. The main idea of the algorithm is that with each iteration we choose a compatible interval with minimal finishing time. This idea is quite natural. Imagine a real line starting at 0 and going into infinity. Each interval takes a portion of it. Therefore when we choose interval with minimal finishing time, we ensure that we take as little space of the real line as possible. Hence there is more space for the coming intervals.

The algorithm starts by sorting intervals in $\prec_{\mathsf{f}}$ order. Let $a_1, \ldots, a_n$ denote the intervals in this order. The first interval a_1 is placed into the set R, which is empty initially. Then algorithm iterates through the intervals $a_2, \ldots, a_n$. For each interval a_i, if a_i is compatible with the $\prec_{\mathsf{f}}$-greatest interval in R, then a_i is added into R. Otherwise a_i is dropped. When iteration finishes, the algorithm returns the set R. The pseudo-code of the algorithm is presented as Algorithm 2.1.

Algorithm 2.1 GreedySchedulingAlgorithm

```
1: Let $a_1, \dots, a_k$ denote intervals in $\prec_{\mathsf{f}}$-order.
2: $R \leftarrow \{a_1\}$
3: $last \leftarrow a_1$
4: for $i \leftarrow 2, \dots, n$ do
5:     if $a_i$ is compatible with $last$ then
6:         $R \leftarrow R \cup \{a_i\}$
7:         $last \leftarrow a_i$
8: end for
9: return $R$
```

Note that every interval i_k added by the algorithm is the right compatible interval of the previously added interval. Thus we give a formal definition to the set R returned by the greedy algorithm:

Definition 2.1.5 (Greedy Optimal Set)**.** *For a collection of intervals I, the subset of intervals $J = \{i_1, \dots, i_k\}$ is called greedy optimal set if i_1 is the $\prec_{\mathsf{f}}$-least interval in I and $i_{j+1} = \mathsf{rc}(i_j)$ for every $1 \leq j < k$.*

We prove the know results that the greedy optimal set is unique and optimal.

Theorem 2.1.6. *The greedy optimal set is unique.*

Proof. For contradiction, assume that there exist two different greedy optimal sets R and R'. Let i_k and i'_k be the first different intervals in these sets. The endpoints of these intervals are different by our assumption of distinctness. Therefore, one of the intervals is $\prec_{\mathsf{f}}$-smaller than the other. Let $i_k \prec_{\mathsf{f}} i'_k$. We know that $i_{k-1} = i'_{k-1}$. But then i'_k is not the right compatible interval of i'_{k-1}, since i_k is also compatible with i'_{k-1} and $i_k \prec_{\mathsf{f}} i'_k$. We reached a contradiction. □

Theorem 2.1.7. *The greedy optimal set consists of pairwise compatible intervals and has the maximal possible size.*

Proof. Let $R = \{a_1, \dots, a_k\}$ be the greedy optimal set. Take two intervals a_i and a_j such that $a_i \prec_{\mathsf{f}} a_j$. If $j = i + 1$ then $a_j = \mathsf{rc}(a_i)$. Therefore the intervals are compatible. Otherwise, there exists intervals $a_{i+1}, \dots, a_{i+t}$ for some $t \geq 1$. Each of this intervals is the right compatible interval of the previous interval. Therefore a_{i+t} is compatible with a_i. Since $a_j = \mathsf{rc}(a_{i+t})$, the intervals a_i and a_j are compatible.

For the second statement, assume that there exist a compatible set $O = \{b_1, \ldots, b_m\}$ of greater size. Consider the intervals a_1 and b_1. Since the interval a_1 the smallest interval, we have that $f(a_1) \leq f(b_1)$, when equality is achieved if $a_1 = b_1$. Now consider the intervals a_2 and b_2. Since $a_2 = \mathsf{rc}(a_1)$, we know that a_2 has the smallest finishing time among all the intervals that are compatible with a_1. Taking into account that $f(a_1) \geq f(b_1)$, we imply that $f(a_2) \leq f(b_2)$, that is the interval a_2 finishes at most as late as the interval b_2. Continuing this process, we eventually achieve the intervals a_k and b_k. The same inequality holds for them as well, namely $f(a_k) \leq f(b_k)$.

By our assumption, the set O has at least one more interval b_{k+1}. This interval is compatible with b_k. Since $f(a_k) \leq f(b_k)$, the intervals a_k and b_{k+1} are compatible. Therefore there exists the right compatible interval for a_k. However, it contradicts with the fact that Algorithm 2.1 stops when the last added interval has no right compatible interval. Thus our initial assumption that there exists a set of greater size if false. □

We provide a quick analysis of the algorithm's running time. The algorithm starts by sorting intervals, which takes $O(n \log n)$ time. Then, at every iteration, the algorithm calls right_compatible subroutine. On the sorted set of intervals the subroutine takes $O(\log n)$ time. Note that in the worst-case the subroutine is called at most once for every interval. Thus the overall running time is $O(n \log n)$ worst-case.

2.1.2 Scheduling on Multiple Machines

In the previous section we discussed an interval scheduling problem on a single machine. The problem concerns a situation where only one machine is given, and the goal is to complete as many jobs as possible. In this section, we discuss the problem of interval scheduling on multiple machines. The difference is that the number of machines is unlimited, but every machine has a utilization price. Once we use a machine to process a job, we must pay the price. The goal is to process all jobs and minimize the number of machines used for job processing. Formally, the problem is stated as follows:

Definition 2.1.8 (Multimachine Interval Scheduling Problem). *Given a set of intervals I find a partition of I into the sets $S_1, \ldots, S_k$ such that each subset contains pairwise compatible intervals and the number of subsets is minimal possible.*

We define functions that partition an interval set into compatible subsets as *scheduling functions*:

Definition 2.1.9 (Scheduling Function). *A function $\sigma : I \to \{1, \ldots, k\}$ from a set of intervals I to a set of k natural numbers is called a* scheduling function *if any two distinct intervals $a, b \in I$ are compatible whenever $\sigma(a) = \sigma(b)$.*

The number k is called the *size* of the scheduling function. For an interval $a \in I$, the number $\sigma(a)$ is called the *schedule number*. A set of intervals with the same schedule number i is called a *schedule* and denoted by S_i:

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

As an illustration for the problem, consider the set of intervals I in Figure 2.4(a). With a little vertical rearrangement of the intervals, we find the partition of I into three compatible subsets. The partition is shown in Figure 2.4(b), where each row corresponds to a subset of compatible intervals. The natural question is how do we know that we cannot partition I into a smaller number of compatible subsets? In this example, it is easy to see that intervals d, e, c share a common point. The following definition gives a name to the maximal number of pairwise overlapping intervals:

Definition 2.1.10 (Depth of an Interval Set). *Let I be an interval set. The* depth *of I, denoted by $d(I)$, is the maximal number of pairwise overlapping intervals.*

As discussed above, the interval set on Figure 2.4 has depth 3. It is not hard to see that we cannot find a partition with less than three subsets of compatible intervals. Otherwise, two of these intervals would be in the same subset, violating its compatibility. We make this observation formal in the following lemma:

Lemma 2.1.11. *The number of subsets in any partition of an interval set I is at least $d(I)$ if each subset contains pairwise compatible intervals.*

Proof. For contradiction, assume that $S_1, \ldots, S_k$ is a partition of an interval set I such that $k < d(I)$ and each set S_i contains pairwise compatible intervals. By the definition of the depth of I, there exists pairwise overlapping intervals $a_1, \ldots, a_k, \ldots, a_{d(I)}$. By pigeonhole principle, the first k intervals $a_1, \ldots, a_k$ must be in different subsets $S_1, \ldots, S_k$. Without loss of generality, assume that $a_i \in S_i$ for $1 \leq i \leq k$. Now consider the interval a_{k+1}. It must belong to some subset S_j. However, S_j contains an interval a_j. The intervals a_{k+1} and a_j share a commont point, which contradicts with the assumption of compatibility of the set S_j. □

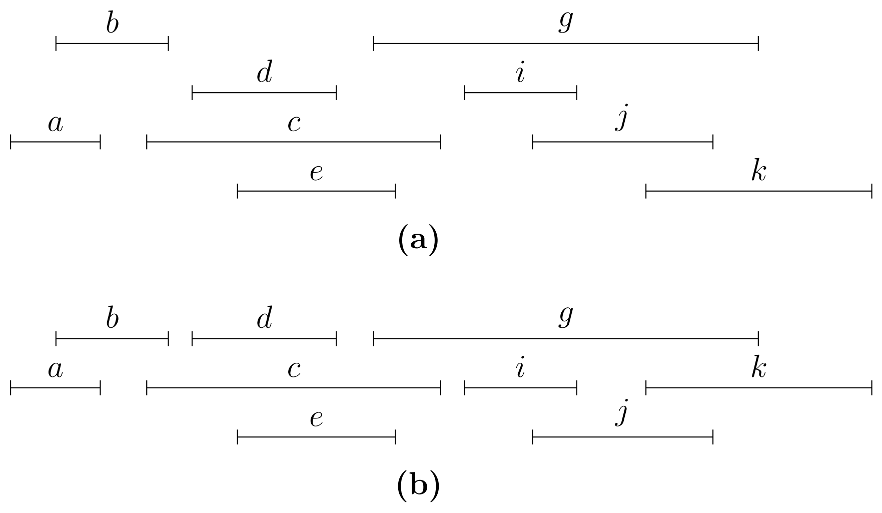


Figure 2.4: (a) An instance of a Multimachine Interval Scheduling Problem with 9 intervals. (b) A solution of the instance where all intervals are partitioned into three subsets: $S_1 = \{b, d, g\}, S_2 = \{a, c, i, k\}, S_3 = \{e, j\}$.

Lemma 2.1.11 allows us to verify an optimality of a scheduling function: a scheduling function is optimal if its size equals to the depth of the interval set. We use this observation to define an optimal scheduling function:

Definition 2.1.12 (Optimal Scheduling Function)**.** *A scheduling function* $\sigma : I \to \{1, \ldots, k\}$ *is* optimal *if* $k = d(I)$*.*

Moreover, the lemma suggests us an idea of an algorithm: we need to take care of pairwise overlapping intervals. Having these observations in mind, we describe a simple algorithm that returns an optimal scheduling function.

We start by sorting intervals in the $\prec_{\mathsf{s}}$-order. Let $a_1, \ldots, a_n$ denote the intervals in this order. Then, we take the first intervals a_1 and add this interval into the first subset S_1. The subset has been empty, so there is no need to check whether a_1 intersects with intervals in S_1. Next, we take the second interval a_2. Our goal is to use as few subsets as possible. Therefore we check whether a_2 intersects with a_1. If it does we place a_2 into the second subset S_2. Otherwise we add a_2 into S_1. We continuing this process. At some moment of time we have placed $a_1, \ldots, a_{k-1}$ intervals into $S_1, \ldots, S_m$ subsets. Let $last(S_j)$ denote the last interval added into S_j. Now we choose a subset for the interval a_k. For each S_j, we compare a_k with $last(S_j)$. If a_k is compatible with some interval $last(S_j)$, we add a_k into the subset S_j, and continue to the next interval. Otherwise, we add a_k

Algorithm 2.2 SimpleMultimachineScheduling

```
1: Sort the interval set $I$ by the starting time of the intervals
2: Initialise $S_1, S_2$ as empty sets
3: $k \leftarrow 1$
4: for $i \leftarrow 1, \dots, n$ do
5:     for $j \leftarrow 1, \dots, k+1$ do
6:         if $S_j$ is empty or $last(S_j)$ is compatible with $a_k$ then
7:             $S_j \leftarrow S_j \cup \{a_i\}$
8:             break for
9:     end for
10:    if $a_i \in S_{k+1}$ then
11:        $k \leftarrow k + 1$
12:        $S_{k+1} \leftarrow \emptyset$
13: end for
14: return $S_1, \dots, S_k$
```

into new subset S_{j+1}. After we added last interval a_n into a subset, the algorithm terminates. The returned subsets define the scheduling function for the interval set I. The pseudo-code of this algorithm is presented as Algorithm 2.2.

In the next lemma, we state that Algorithm 2.2 is correct and optimal.

Lemma 2.1.13. *Let I be an interval set, $S_1, \dots, S_k$ be the sets returned by Algorithm 2.2. The following statements hold:*

- $d(I) = k$
- $I = S_1 \cup \ldots \cup S_k$
- *S_i is compatible for every $1 \leq i \leq k$*

Proof. First we prove that the depth of the interval set I equals to the number of sets k returned by the algorithm. We consider two cases. The first case is when k never increased, that is, the line 11 of the algorithm is never executed. In this case, each interval a_i is compatible with the previous interval a_{i-1}. Therefore all intervals are pairwise compatible, and the depth of the interval set is 1. Thus k equals to $d(I)$.

In the second case, we suppose that k is increased at least once. Let us consider each time when k is increased. From the line 10 we see that a_i has been added into S_{k+1}. It can only happen when a_i is incompatible with $last(S_j)$ for every $1 \leq j \leq k$. Therefore, the starting point $s(a_i)$ belongs to every interval $a_i, last(S_1), \ldots, last(S_k)$. Together these intervals form a set of $k+1$ pairwise overlapping intervals. Thus, when we increase k by 1, we know that the depth of the interval set is $k+1$.

The second statement of the lemma follows easily from the observation that after each iteration of the first for-loop interval a_i belongs to one of the sets $S_1, \ldots, S_k$.

For the third statement of the lemma let us consider the moment right before we add the interval a_i into the set S_j. If S_j is empty, then $S_j \cup \{a_i\}$ is a compatible set. If S_j has size 1, then the only interval in S_j is $last(S_j)$. Since we add a_i into S_j only when a_i is compatible with $last(S_j)$, the set $S_j \cup \{a_i\}$ is a compatible set. Finally, if S_j has size greater than 1, take any interval $b \in S_j$ that is not $last(S_j)$. By construction of S_j, b is compatible with $last(S_j)$. Since a_i is compatible with $last(S_j)$ we have $f(b) < s(last(S_j)) < s(a_i)$. Therefore b and a_i are compatible intervals. Thus $S_j \cup \{a_i\}$ is a compatible set. $\square$

While Algorithm 2.2 compares only minimal number of intervals, this number can be as high as $O(n^2)$. As an example, consider a set of intervals $\{a_1, \ldots, a_n\}$ such that $s(a_1) < \ldots < s(a_n) < f(a_1) < \ldots < f(a_n)$, which is shown on Figure 2.5. In this set every interval intersects with every other interval. Therefore at the ith iteration the algorithm compares interval a_i with $a_{i-1}, \ldots, a_1$. Thus the worst-case complexity of the algorithm is $O(n^2)$.

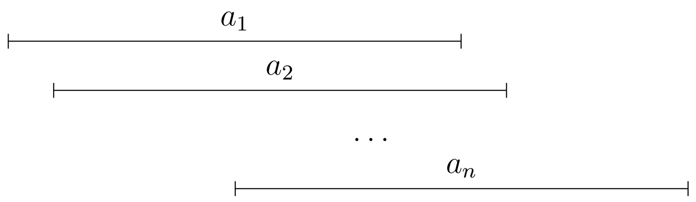


Figure 2.5: A set of pairwise intersecting intervals

The weakest point of the previous algorithm is that it does not take into account finishing times of the last intervals in the sets $S_1, \ldots, S_k$. Indeed, at the ith iteration, instead of looking for a set S_j such that a_i and $last(S_j)$ are compatible, with a little bit of extra information we can report that at the moment $s(a_i)$ none of the last intervals has finished. This idea was introduced by Gupta et al. [59]. We describe his algorithm that finds an optimal scheduling function in $O(n \log n)$ worst-case time.

Algorithm 2.3 OptimalMultimachineScheduling

```
1: $p_1, \ldots, p_{2n} \leftarrow$ sort $I$ in order of intervals' starting times
2: $k \leftarrow 1$
3: $Free \leftarrow \{S_1\}$
4: for $i \leftarrow 1, \ldots, 2n$ do
5:     if $p_i$ is the starting time of an interval then
6:         if $Free$ is empty then
7:             $k \leftarrow k + 1$
8:             $Free \leftarrow \{S_k\}$
9:         $S \leftarrow$ a set from $Free$
10:        $S \leftarrow S \cup \{I(p_i)\}$
11:        $Free \leftarrow Free \setminus \{S\}$
12:    else
13:        $S \leftarrow$ a set containing $I(p_i)$
14:        $Free \leftarrow Free \cup \{S\}$
15: end for
16: return $S_1, \ldots, S_k$
```

Let $p_1, p_2, \ldots, p_{2n}$ be the endpoints of all intervals in I sorted in increasing order. Note that p_i can be the starting time or the finishing time of an interval. We denote by $I(p_i)$ an interval that starts or finishes at p_i. Let t be a moment of time. We call a set S_j *free* at the moment t if the last interval in S_j finishes before t.

Initially, there is only one free set S_1. We take the first endpoint p_1. Since p_1 is the starting time of some interval, we put the interval $I(p_1)$ in the set S_1. Suppose, the endpoint p_2 is also a starting time of some interval. The set S_1 is not free at the moment p_2. Therefore we create a new set S_2 and put $I(p_2)$ into this set. Now suppose that p_3 is the finishing time of some interval. Since only two interval has started, the endpoint p_3 is either a finishing time of either $I(p_1)$ or $I(p_2)$. Therefore we mark the set that contains $I(p_3)$ as free. The process continues. For every endpoint p_i we consider two cases. If p_i is a starting time, we put $I(p_i)$ into a free set, creating a new one if there are no free sets. Otherwise, we mark the set that contains $I(p_i)$ as free. The process stops after we have considered all of the endpoints. The pseudo-code of the algorithm is presented as Algorithm 2.3.

We briefly analyze the complexity of the algorithm. There are $O(n)$ iterations of the for-loop, where each iteration takes constant time, and there is a sorting operation on the endpoints of the intervals, which takes $O(n \log n)$ time. Therefore, the overall complexity of the algorithm is $O(n \log n)$ worst-case. To show correctness and optimality of the algorithm, we prove the following lemma:

Lemma 2.1.14. *Algorithm* OptimalMultimachineScheduling *returns an optimal partition of an interval set.*

Proof. First, we prove that each of the k returned sets contains pairwise compatible intervals. For contradiction, assume that the set S_j contains two overlapping intervals a and b. Without loss of generality, assume that $s(a) < s(b) < f(a)$. Consider the iteration of the algorithm when p_i is the starting time of b. By our assumption, the algorithm has added b into S_j, meaning that S_j belongs to the set $Free$. However, S_j was deleted from $Free$ when the algorithm processed $s(a)$. Moreover, since the algorithm has not processes $f(a)$, S_j has not been added to the set $Free$. Thus, the assumption that S_j is not a compatible set is false. For the optimality of the algorithm, we prove that the algorithm increases k by one only if there are $k+1$ pairwise overlapping intervals. Let p be an endpoint during an iteration of the algorithm when k is increased. It means that the line 7 of the algorithm is executed. Therefore p is the starting time of an interval, and none of the k sets is free. We observe two things. The first thing is that p does not belong to any of the already scheduled intervals. Therefore $last(S_1), \ldots, last(S_k)$ and $I(p)$ are all distinct intervals. The second thing is that for every $1 \leq j \leq k$ the interval $last(S_j)$ has not finished. Therefore all interval $last(S_1), \ldots, last(S_k), I(p)$ share a common point p. Thus the the algorithm increases k only if the depth of the interval set is $k+1$. □

The Algorithm 2.3 is simple and efficient. Moreover, Gupta et al. [59] showed that the $O(n \log n)$ bound is minimal possible for any algorithm that solve interval scheduling problem for multiple machines.

2.2 Real-Time Scheduling

Imagine a modern vehicle. It has many hidden sensors that monitor engine, brakes, wheels, and other parts of the vehicle. There is a central processor, which is responsible for controlling engine, brakes, steering and other units based on the information it receives from the sensors. For the engine, the processor adjusts the amount of injected fuel to ensure the minimal possible fuel consumption. For the brakes, it calculates the brake pressure on each wheel to ensure stability of the vehicle. For the steering, the processor monitors whether the car starts sliding at a turn on an icy road. Each of these tasks requires computational power of the processor. However, the safety of a driver depends not only on the correctness of a computation, but also on the time at which the results are produced. Even a small delay in computation of the wheel angle may result in serious injuries. Therefore accurate management of the processor's computational power is integral to real-time system design.

A real-time system Γ, also called a *task set* or a *task system*, is a finite collection of *tasks* $T_1, \ldots, T_n$, where each task generates an infinite sequence of *jobs*. Each task T_i is characterized by a four-tuple (a_i, e_i, d_i, p_i), where

- the *offset* a_i is the moment of time at which the first job of task is ready to be executed by the processor;
- the *execution requirement* e_i is the upper limit on the amount of time units needed to complete execution of the task's job;
- the *relative deadline* d_i is the length of a window in which the job must receive e_i units of processor's time, that is if a job arrives at time t the processor must finish its execution at time $t + d_i$;
- the *period* p_i is the number of time units between the arrival times of successive jobs generated by the task.

We denote the ith job of a task T_j by T_j^i. A job has three parameters: $a(T_j^i)$ is the *arrival time* of the job which specifies when the job is available for execution; $e(T_j^i)$ is the execution requirement of the job which is equal to the execution requirement of the task e_j; $d(T_j^i)$ is the deadline of the job with the interpretation that the job must receive e_j units of processor time in the interval $[a(T_j^i), d(T_j^i)]$.

Given a collection of jobs $\mathcal{J}$ we define a m-processor schedule S as the function from the Cartesian product of real numbers and the collection of jobs to the set of numbers from 0 to m: $S : \mathbb{R} \times \mathcal{J} \to \{0, \ldots, m\}$. If at time t a job J is being processed at the processor k, then $S(J, t) = k$, otherwise $S(J, t)$ equals zero. We call a schedule *feasible* if every job J in the collection receive $e(J)$ units of processor time between its arrival time and its deadline. A job is called *active* at time t if it has been released before t, its deadline is greater than t, and the amount of time given to the job by the processors is less than its execution requirement. Note that the number of active jobs may be more than the number of available processors.

A schedule is called *preemptive* if there exists a job whose execution on a processor is interrupted and continued at a later time. A common reason for an interruption of a job is the arrival of a new more urgent job. Note that preemptive scheduling is strictly more powerful than nonpreemptive scheduling. For an example consider a collection of two preemptive jobs $J = (0, 4, 8)$ and $I = (2, 3, 6)$. Since job J is active at time 0, we start executing it on the processor. The second job arrives at time 2. We have two choices: either continue execution of J, or preempt J and start execution of I. In the first case, J finishes at time 4, and I misses its deadline at time 6. In the second case, both jobs finish without missing their deadlines. The example is shown on Figure 2.6.

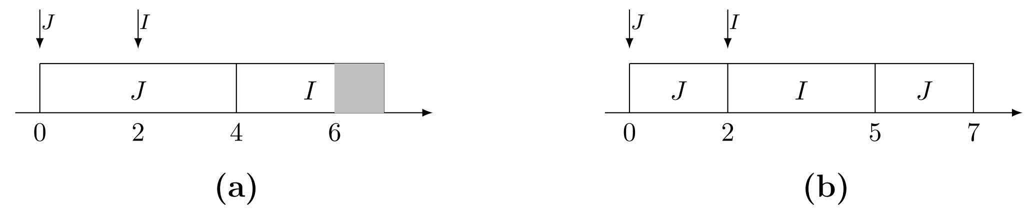


Figure 2.6: Scheduling of two jobs $J = (0, 4, 10)$ and $I = (2, 3, 6)$. (a) J is not preempted when I arrives, and I misses its deadline. (b) feasible schedule if J is preempted.

The *utilization* $U(T_i)$ of a task is defined as a ratio of its execution requirement e_i to the period p_i: $U(T_i) \overset{\text{def}}{=} \frac{e_i}{p_i}$. In other words, utilization of a task is the portion of time the processor must devote to execute all jobs of the task. The utilization of a real-time system Γ is the sum of the utilizations of all tasks in this system:

$$U(\Gamma) \overset{\text{def}}{=} \sum_{i=1}^{n} U(T_i)$$

A task $T = (a, e, d, p)$ can be either *periodic* [77] or *sporadic* [82]. If the task is periodic then the arrival time of two successive jobs T^j and T^{j+1} are separated by exactly p units of time. Thus we can calculate the arrival time of each job of the task: the first job arrives at time a, the second job arrives at time $a + p$, ..., the jth job arrives at time $a+(j-1)p$. If the task is sporadic, the next job T^{j+i} arrives at an unknown time t that is at least p units of time greater then the arrival time of the previous job T^j. A real-time system consisting of periodic (sporadic) tasks is called periodic (sporadic).

In this thesis, we restrict out attention to the preemptive scheduling of periodic task systems.

Whenever one investigates properties of a task system, the *feasibility* problem is considered. For a periodic task system, the feasibility problem asks whether there exists a feasible schedule for the collection of jobs generated by the task systems. For a sporadic task system, the problem becomes more complex because a sporadic task system is legally permitted to generate infinitely many distinct collections of jobs. Thus, the feasibility problem for a sporadic task system asks whether there exists a feasible schedule for *every* collection of jobs generated by the task system.

Leung and Merrill showed that the feasibility problem for an arbitrary periodic task system is NP-hard:

Theorem 2.2.1 (Leung and Merrill [76])**.** *The problem of deciding whether an arbitrary periodic task system is feasible on one processor is NP-hard.*

As a corollary to the theorem, the feasibility problem on m processor is NP-hard. Moreover, it follows from their proof that for a task system Γ with $U(\Gamma) < c$, where c is an arbitrary small positive constant, the problem remains NP-hard. Therefore different restrictions on the task models are considered.

A periodic task system is called *synchronous* if the first jobs of all tasks arrive simultaneously at some moment of time. Otherwise, a task system is called *asynchronous*. If a system is synchronous, then the offset of all task is assumed to be zero, and it is disregarded from the parameters of the tasks. The synchronous property is important for consideration since it provides us with a finite representation of the infinite job sequence generated by the tasks. Indeed, let L be the least common multiple of tasks' periods: $L \stackrel{\text{def}}{=} LCM(p_1, \dots, p_n)$. Let $\mathcal{J}_k$ be the set of jobs such that their arrival times lie in the period from kL to $(k+1)L$: $\mathcal{J}_k = \{J \mid kL \leq a(J) < (k+1)L\}$. Then the infinite sequence of jobs equals to the union $\mathcal{J}_0 \cup \mathcal{J}_1 \cup \dots$ of these sets. It is not hard to see that the there exists a bijective function from a set $\mathcal{J}_i$ to the set $\mathcal{J}_0$. Thus we only need to search for a feasible schedule for the set $\mathcal{J}_0$, which exists if and only if there exists a feasible schedule for the infinite set $\mathcal{J}$.

Note that only periodic task systems are synchronous since the periodicity constraint forces the jobs arrive synchronously. In a sporadic task system, even if all the first jobs are released at the same time, a task may postpone releasing a job thus breaking synchronization.

Other restrictions on a task model that are studied in the literature concern relationship between deadlines and periods of the tasks. The task system (periodic or sporadic) is said to be:

- *implicit-deadline* if every task of the system has its deadline equal to its period: $d_i = p_i$ for every $1 \leq i \leq n$;
- *constrained-deadline* if every task of the system has its deadline no larger then its period: $d_i \leq p_i$ for every $1 \leq i \leq n$;

Until recently the complexity of feasibility problem for synchronous task system remained open. In 2010, Eisenbrand and Rothvoß proved that the problem of testing whether an `EDF` algorithm (see section 2.2.1 below) produces a feasible uniprocessor schedule for a synchronous periodic task system is coNP-hard, even if deadlines of the tasks are constrained. Since feasibility of a synchronous task system implies existence of an `EDF` schedule [77, 39], their result implies coNP-hardness of the feasibility problem for such task systems. For synchronous constrained-deadline systems on multiple processors, Bonifaci et al. obtained the same result in the same year.

However, the feasibility problem is tractable if the task system is periodic implicit-deadline. The result for one processor was obtained by Liu and Layland [77] and for m processors by Horn [65]:

Theorem 2.2.2 (Liu and Layland [77]). *A periodic implicit-deadline task system Γ is feasible on one processor if and only if total task utilization is at most 1: $U(\Gamma) \leq 1$.*

Theorem 2.2.3 (Horn [65]). *There exists a feasible m processor schedule for the jobs generated by a periodic implicit-deadline task system Γ if and only if $U(\Gamma) \leq m$.*

Thus the complexity of feasibility problem for periodic task systems on one and m processors is well understood. Figure 2.7 shows the hierarchy of periodic task systems with different combinations of restrictions. The most general class consists of asynchronous task systems without any restrictions on the offsets and deadlines of the tasks. The most narrow class consists of implicit-deadline synchronous task systems.

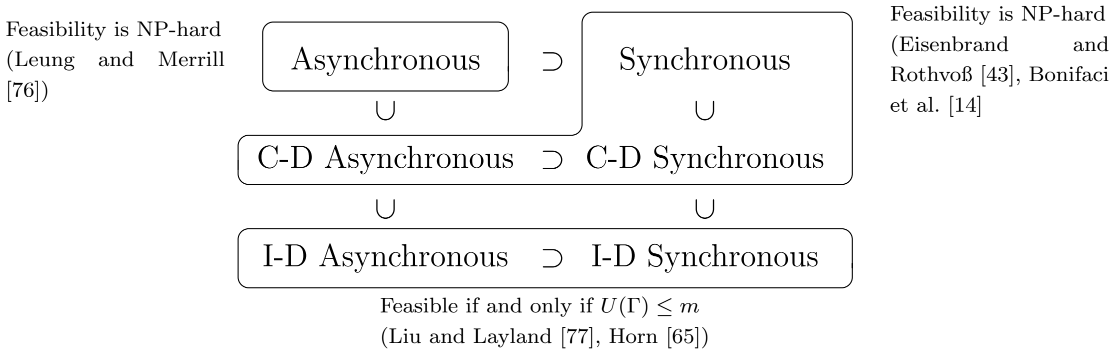

Feasibility is NP-hard (Leung and Merrill [76])
Asynchronous
$\supset$
Synchronous
Feasibility is NP-hard (Eisenbrand and Rothvoß [43], Bonifaci et al. [14]
$\cup$
$\cup$
C-D Asynchronous
$\supset$
C-D Synchronous
$\cup$
$\cup$
I-D Asynchronous
$\supset$
I-D Synchronous
Feasible if and only if $U(\Gamma) \leq m$
(Liu and Layland [77], Horn [65])

Figure 2.7: Complexity of feasibility problem for different types of task systems. Abbreviations C-D and I-D represent constrained- and implicit-deadline systems.

The feasibility problem for sporadic task systems is solved in the uniprocessor case. Baruah et al. [10] showed that the problem is reducible to the feasibility problem for synchronous task system:

Theorem 2.2.4 (Baruah et al. [10])**.** *Let Γ be a sporadic task system of consisting of n tasks $T_1 = (e_1, d_1, p_1), \ldots, T_n = (e_n, d_n, p_n)$. Let Γ' be a corresponding synchronous task system consisting of tasks $T_1 = (e_1, d_1, p_1), \ldots, T_n = (e_n, d_n, p_n)$. The task system Γ is feasible on one processor if and only if Γ' is feasible on one processor.*

Combining Theorem 2.2.4 with the result for periodic task systems, we have the following:

Theorem 2.2.5. *An implicit-deadline sporadic task system is feasible on one processor if and only if its utilization is at most 1.*

Theorem 2.2.6. *The uniprocessor feasibility problem of a sporadic task system with constrained or arbitrary deadlines is coNP-hard.*

In the multiprocessor case, neither the exact polynomial-time algorithms are known nor the complexity of the problem. To the best of our knowledge, the first algorithm for testing multiprocessor feasibility of an arbitrary and constrained-deadline sporadic task system is obtained by Bonifaci and Marchetti-Spaccamela [15]. The algorithm has $2^{2^{O(s)}}$ time complexity and $2^{O(s)}$ space complexity, where s is the input size of the task system.

2.2.1 Scheduling Algorithms for Real-Time Systems

It is easy to schedule jobs if at any time there are more available processors than active jobs. All we need to do is to choose a free processor and start execution of an active job on this processor. However, it is usually quite the reverse: the number of available processors is less than the number of active jobs. Therefore a scheduling algorithm must make two decisions: on which processor a job should be executed, and in what order execute the jobs. We will refer to the first problem as *allocation* problem and to the second problem as *priority* problem. Based on the decisions a scheduling algorithm makes at run-time it belongs to one class in each of the following two categories [24]:

- In allocation category, algorithms are classified with respect to the degree of interprocessor migration:
  - *No migration* or *partitioned.* All jobs of a task are executed on the same processor.
  - *Restricted migration* or *global.* Different jobs of a task can be executed on different processors. However, if a job is preempted, its execution must be resumed on the same processor.
  - *Unrestricted migration.* No restriction on job migration: different processor can execute the same job at different period of times. However, parallel execution is not allowed.
- In priority category, algorithms are classified with respect to the changes of the relative priorities of jobs:
  - *Static priorities.* All tasks are assigned different priorities, all jobs of the same task have the same priority. A job of a higher priority task always preempts a job of a lower priority task.
  - *Job-level dynamic priorities.* Priorities of tasks are not important, jobs receive priorities individually. However, if a job J_i is being executed when a job J_j is active, then J_i will be started only when J_i is finished.
  - *Unrestricted dynamic priorities.* No restrictions on the priorities of the jobs: relative priorities of two jobs may change at any time.

We provide a description of the three standard scheduling algorithms, one from each priority category: rate-monotonic (`RM`) that falls into static priority class, earliest deadline first (`EDF`) from the job-level dynamic priority class, and least-laxity first (`LLF`) that belongs to the class of unrestricted priorities. To focus on the details of the algorithms we describe them in the case of single processor. Then we describe important results in the case of multiple processors.

Rate-monotonic. The `RM` algorithm [77] is a static priority algorithm that assigns fixed priorities to the tasks based on their period: the longer the period, the smaller the priority. That is if two tasks A and B have periods 30 and 40, respectively, then every job of the task A has a higher priority then any job of the task B. Therefore for any natural numbers i, j a job A^i always preempts a job B^j. When two task have the same periods, the higher priority is assigned arbitrarily. Let us provide an example of a schedule computed by `RM` algorithm.

Let Γ be a synchronous implicit-deadlines system of three task A, B and C with preemption allowed. The task A, B, C has the execution requirements of 2, 2 and 4 and the period of 6, 8 and 12, respectively. Recall that in synchronous systems the first jobs of all tasks are released at time 0, and in implicit-deadline systems relative deadline of a task equals to its period. Note that 24 is the least common multiply of tasks' periods. Therefore it is sufficient to run the scheduler in the first major cycle from 0 to 24. The example is shown on Figure 2.8.

The scheduler compares priorities of active jobs whenever there is more than one active job. At time 0 there are three active jobs, one of each task. Task A has the smallest period. Therefore the job A^1 is executed first. The next job to be executed is B^1 at time 2, and C^1 at time 4. At time 6, while job C^1 is still active, the second job of task A arrives. Since A has higher priority than C, the job A^2 preempts C^1. For the same reason, the job B^2 occupies the processor at time 8 instead of C^1. Nevertheless, C^1 finishes before C^2 is released. The job A^3 is released at the same time as C^2. Therefore C^2 waits until A^3 is completed. Later, at times 16 and 18, C^2 is preempted by B^3 and A^4. Finally, the job C^2 is completed at time 22. Since none of the jobs misses its deadline in the period from 0 to 24, we conclude that the task system Γ is schedulable by `RM` algorithm.

Note that there are $n!$ different priority assignments of n tasks, and priority assignment based on the periods is only one the many others. In our example above we can assign priorities to tasks A, B, C such that C has higher priority than B, which has a higher priority that A. However, with this assignment, the job A^1 does not complete by its deadline. Therefore a natural question arises: what is

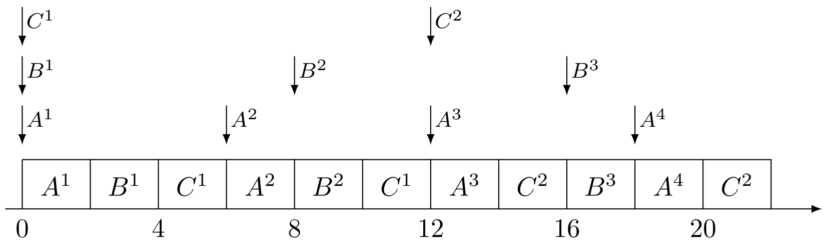

C^1
C^2
B^1
B^2
B^3
A^1
A^2
A^3
A^4
A^1 B^1 C^1 A^2 B^2 C^1 A^3 C^2 B^3 A^4 C^2
0
4
8
12
16
20

Figure 2.8: Rate-monotonic scheduling of three tasks $A(2, 6), B(2, 8)$ and $C(4, 12)$, where (x, y) denote the execution requirement and the period, respectively.

the optimal priority assignment? [77] proved that the rate monotonic assignment is optimal in the sense that if a task set is not schedulable by `RM` algorithm then it is not schedulable by any other static priority scheduling algorithm.

Theorem 2.2.7 (Liu and Layland [77]). *If a feasible priority assignment exists for some task set, then the rate monotonic priority assignment is feasible for that task set.*

In the same paper, Liu and Layland also identified and proved a sufficient test for the `RM` algorithm that is based on the overall utilization of the system

Theorem 2.2.8. *The task system* Γ *of* n *tasks is schedulable if* $U(\Gamma) \leq n(2^{1/n} - 1)$.

When the number of tasks is increasing, the value of the equation approaches 0.693. However, the test is sufficient, but not necessary, as there exist task sets with utilization greater than 0.7 that can be successfully scheduled by the `RM` algorithm. Lehoczky et al. [72] showed that in the average case large task sets with utilization as high as 0.9 do not miss deadlines if scheduled by `RM` algorithm.

Earliest-deadline first. The `EDF` algorithm is a job-level dynamic priority algorithm. It assigns priorities to the jobs at runtime based on the jobs' deadlines: if jobs A^i and B^j are active at time t, the `EDF` algorithm schedules the job with the earlier deadline first. In contrast to the rate monotonic priority assignment, jobs may have different relative priorities at different time moments. Consider the following example.

A synchronous implicit-deadline task system Γ consists of two tasks A, B with the execution requirement 2 and 4 and the periods 5 and 7, respectively. At time 0 the job A^1 has an earlier deadline than the job B^1, and therefore it is scheduled first. At time 5 the job A^2 is released. Since B^1 has an earlier deadline than A^2,

A^2 does not preempt B^1. For the same reason, B^2 does not preempt A^2 and A^3 does not preempt B^2. However, a preemption does happen at time 15 when the job A^4 is released: A^4 has the deadline at 20, while currently processing job B^3 has the deadline at 28. See the Figure 2.9.

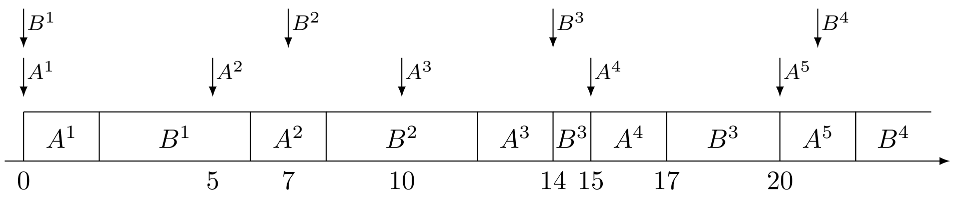


Figure 2.9: Rate-monotonic scheduling of three tasks $A = (2, 5)$ and $B = (4, 7)$.

Note that the task system above is not schedulable by the `RM` algorithm: the job B^1 misses its deadline, because in the period from 0 to 7 the task A takes 4 units of time of the processor. Thus, by the theorem 2.2.7, there is no feasible static priority assignment for this task system.

The `EDF` scheduling possesses the following strong property:

Theorem 2.2.9 (Liu and Layland [77]). *The set of tasks, when scheduled by* `EDF` *algorithm, meets all deadlines if and only if the total utilization of the task set does not exceed 1.*

Consequently, this property of `EDF` algorithm gives us the necessary and sufficient feasibility test of a task set. If the total utilization of the task set exceeds 1, then the task set is not feasible. Otherwise, we know that the task set can be scheduled by `EDF` algorithm and meet all deadlines.

Least Laxity First. The *laxity* of an active job at time t is the amount of time a job can wait for the execution and still meet the deadline. Formally, for a job J the laxity $L(J, t)$ at time t is the difference between job's relative deadline $d(J) - t$ and remaining computation time $c(J)$: $L(J, t) = d(J) - t - c(J)$. For example, a job J with the deadline at 6 and execution requirement of 4 at time 0 is 2. If the job is executed from 0 to 1, then its laxity remains the same. Otherwise, it decreases by 1.

The `LLF` algorithm assigns priorities to the jobs based on their laxities: the smaller the laxity, the higher the priority. The algorithm belongs to the class of unrestricted priorities algorithm. Indeed, while a job J is being executed, its laxity remains the same, but the laxity of other active jobs decreases. Therefore

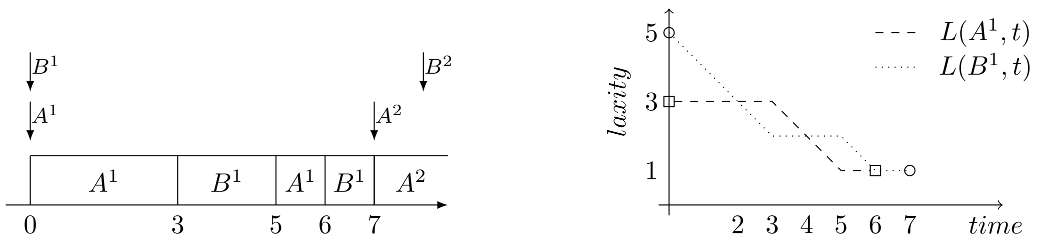


Figure 2.10: Least-laxity first scheduling of two tasks $A = (4, 7)$ and $B = (3, 8)$ and the graph showing changes in laxity value with time.

the priority of waiting jobs is increasing and may become higher than the priority of the job J. Let us give an example of this process.

Let Γ be a task system of two tasks A and B with execution requirements of 4 and 3 and the periods of 7 and 8, respectively. At time 0 the laxities of the jobs A^1 and B^1 are 3 and 5. Therefore A^1 is scheduled first. With time, the relative deadline of both jobs is decreasing. However, since A^1 is being executed, its remaining computation time is decreasing as well. Therefore, the laxity of A^1 does not change, while the laxity of B^1 is decreasing. after 3 units of time the laxity of B^1 becomes smaller then the laxity of A^1. Therefore B^1 preempts A^1. For the next two units of time, we have the opposite situation: the laxity of B^1 does not change while the laxity of A^1 is decreasing. At time 5, A^1 has higher priority and it preempts B^1. Thus the same jobs preempt each other at different moments of time. The example is shown on Figure 2.10.

As `EDF` algorithm, `LLF` algorithm produces a feasible schedule for a task set if and only if the total utilization of the task set does not exceed 1 [82]. However, `LLF` algorithm does more job preemptions than `EDF` algorithm. In a system where preemption results in additional memory management of the processor's cache, this property of `LLF` algorithm becomes a disadvantage.

Multiprocessor scheduling. Recall that in the case when several processors are available, an algorithm belongs to one the three classes: partitioned, restricted migration, full migration. The partitioned approach allows to reduce a multiprocessor scheduling problem to a set of single processor problems. Once we have chosen a set of tasks for a processor, we may apply, for example, `RM` or `EDF` algorithm. However, finding an optimal allocation of tasks to the processor is a bin-packing problem, which is NP-hard. Thus, one must apply a non-optimal heuristic to partition the tasks.

In the class of restricted migration algorithms, it also has been shown that the problem of deciding whether an arbitrary task set can be feasibly scheduled by a fixed- or dynamic-priority algorithm on M processors is NP-hard [75]. However, if the total utilization of a task set is less or equal to $\frac{M+1}{2}$, a restricted migration algorithm will produce a feasible schedule for the task set.

`EDF` algorithm, being optimal in the single processor case, was compared experimentally under partitioned and restricted migration schemes. Empirical results have show that `EDF`-based partitioning algorithms are superior to `EDF`-based global partitioning algorithms [6, 11].

The only algorithms that can achieve maximal possible utilization of M are based on the notion of *proportionate fairness (Pfair)* [9]. Such algorithms fall into the class of full migration and the class of unrestricted dynamic. The idea is that over an interval $[0, t)$ every task T receives the amount of time proportionate to its utilization. For an overview of Pfair scheduling and many of its extensions we refer the reader to Leung [74, ch. 31]. The disadvantage of proportionate schedule is that the number of preemptions and interprocessor migrations may be high.

2.3 Amortized Complexity

In this section we describe an *amortized analysis* [99, 66, 92], which is a powerful technique for finding tight upper bounds on total computation time taken by a sequence of operations. Note that we talk about a *sequence* of operations rather than a single operation. The core idea is that if an ith operation in a sequence is affected by the preceding operations then it might be the case that the time taken by the operation is less than the time the operation takes in the worst-case. Let us give an example that illustrates the concept of amortized analysis.

Let `inc` be an operation that increments a binary counter by one. Suppose that the cost of this operation is the number of bits in the counter that has been changed. It is easy to see that the time taken by `inc` in the worst-case is in $O(n)$, where n is the length of the counter. However, if an operation has changed n bits, the subsequent $2^{n-1} - 1$ operations will not change the nth bit in the counter. In other words, the first bit in the counter is changed by every operation. The second bit is changed by every second operation. The third bit is changed by every fourth operation. And so on. Therefore we calculate the cost of a sequence of m operations as the sum of times every bit in the counter is changed:

$$m + \left\lfloor \frac{m}{2} \right\rfloor + \left\lfloor \frac{m}{4} \right\rfloor + \ldots \leq 2m$$

Thus the time taken by any sequence of m `inc` operations is at most $2m$, and not $O(m \cdot \log m)$ as one might obtain by applying the worst-case analysis.

One of the approaches that formally defines the amortized running time of operations is based on the *potential* [99]. Let $\sigma_1, \ldots, \sigma_m$ be a sequence of operations that are applied to some data structure D. Let $\Phi(D)$ be a *potential function* that maps all possible states of the data structure to real numbers. Let the cost of ith operation be c_i. We define the amortized cost of i operation as the actual cost operation plus the difference in the potential of the data structure:

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The amortized cost of a sequence of m operations is:

$$\sum_{i=1}^{m} \hat{c_i} = \sum_{i=1}^{m} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m} c_i$$

Note that if the final potential is as large as the initial potential then the total amortized cost is greater or equal to the total actual cost.

$$\sum_{i=1}^{m} \hat{c_i} \geq \sum_{i=1}^{m} c_i$$

In this case the amortized cost of a sequence of operations is the upper bound on the actual cost, even if the amortized cost of some operations in the sequence is smaller that the actual cost of these operations.

Intuitively, when we overcharge ith operation, we save the difference $\hat{c_i} - c_i$ in the data structure. Later, when other operations take their worst-case time, we will use the saved time to average the total time of the sequence.

Let us apply this technique to the problem of computing the cost of m operations that increment a binary counter. We define the potential of the counter as the number of bits that are set to 1. Without loss of generality assume that the counter is set to 0 before the first operation the sequence. Then the initial potential $\Phi(D_0)$ is 0.

Now suppose that as the result of ith operation the kth bit in the counter is set to 1. Then the actual cost of the operation is k, because all the bits on the positions less than k has been set to 0. On the other hand, the potential of the

D_{i-1}	...	1	1	0	$\underline{\mathbf{0}}$	1	1	1	...
D_i	...	1	1	0	$\underline{\mathbf{1}}$	0	0	0	...

Figure 2.11: If a bit at the position k is changed then all bits at the positions less than k are changed too.

counter has decreased by $k-2$, because $k-1$ bits has been set to 0 and the bit at the position k has been set to 1. Figure 2.11 illustrates this difference. The amortized cost is therefore

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - (k-2) = 2$$

Thus the total amortized cost of m operations is $2m$. Moreover, since the potential of the counter in its final state is a positive number, the total amortized cost is the upper bound on the total actual cost. Furthermore, if the counter is not 0 initially, then the total actual cost is still in $O(m)$.

The amortized analysis has been used to construct efficient data structures such as dynamic array [30], disjoint sets [30], skew heap [94], Fibonacci heaps [48], self-adjusting top trees [100], and others. In the next section we describe *splay tree* data structure, which we use in Chapter 3 in our solution of dynamic interval scheduling problem.

2.3.1 Splay Tree Data Structure

A *splay tree* [93] is a self-adjusting binary search tree with the additional property that any element we access becomes the root of the tree. It supports standard operations such as insertions, deletions and searching of a node. In addition, a splay tree allows splitting a tree into two trees and joining two trees into a single tree, given that all elements in one tree are smaller than all elements in the other tree.

All operations are based on the specific operation *splay*. The splay operation, applied to a node x, moves the node to the top of the tree by several steps, where each step is one of the following types:

- *Zig:* if the parent of x is the root, rotate the edge that joins x and its parent.
- *Zig-zag:* if x is left child and its parent $p(x)$ is right child, or vice-versa, then

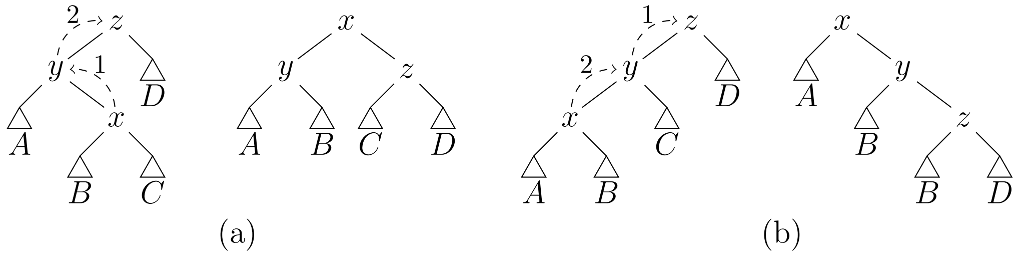


Figure 2.12: Splaying steps: zig-zag (a) and zig-zig (b)

rotate the edge between x and $p(x)$, and then rotate the edge between x and its new parent. See Figure 2.12 (a)

- *Zig-zig:* if x and its parent $p(x)$ are both left children or both right children, then rotate the edge that joins $p(x)$ and $p(p(x))$. Then rotate the edge that joins x and $p(x)$. See Figure 2.12 (b)

Splaying at node x takes time proportional to the depth of x. In the worst case, when x is the deepest node in a "lined" tree, where every node has only one child, the splaying takes time linear in the number of nodes. However, with every splaying step the subtrees of x are moved by up at least one level. Therefore, after accessing the deepest node in a "lined" splay tree, the height of the tree decreases, and future splaying of the deepest nodes take less time. Consider the example on Figure 2.13. in this example, the height of the tree is 9 initially. After performing splay operation on the node 1, which is at the bottom of the tree, the height of the tree reduces to 7. Furthermore, after performing splay operation on the node 2, the height of the tree reduces to 4.

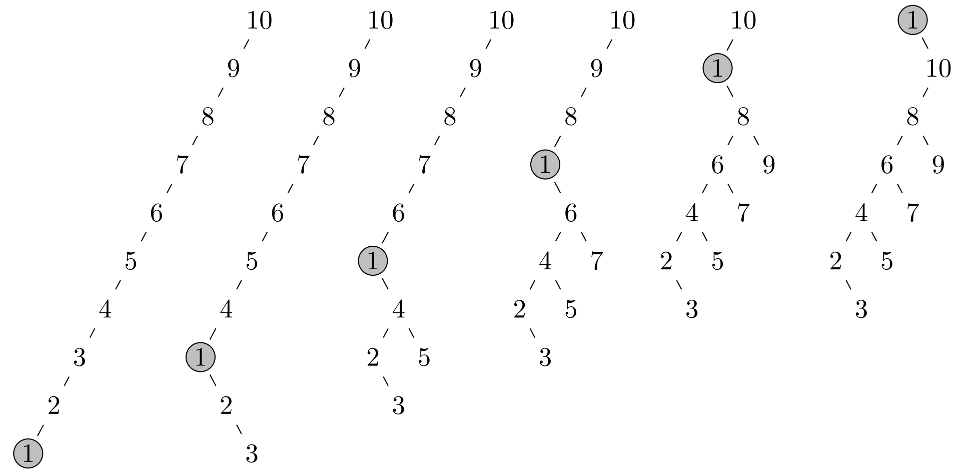
Sleator and Tarjan [93] used amortized analysis to prove the following theorem

Theorem 2.3.1 (Sleator and Tarjan [93]). *The total amortized cost of m splay operations on a tree with n nodes is $O((m+n)\log n + m)$*

In other words, the amortized cost of splay operation is $O(\log n)$. We briefly describe how to use amortized analysis to obtain this result.

Let the *size* $s(x)$ of a node x be the number of nodes in the subtree rooted at x divided by n, the total number of nodes in the tree. Let the *rank* $r(x)$ of a node x be $\log s(x)$. Then the potential function $\Phi(T)$ of the tree is the sum of the ranks of all its nodes.

Consider a splaying step: zig, zig-zag or zig-zig. Recall that the amortized cost of an operation is the sum of actual cost plus the difference in the potential


(a) Splaying node 1

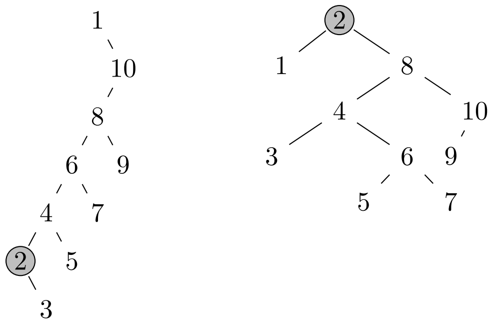

(b) Splaying node 2

Figure 2.13: Splay tree changes its shape and reduces its height after splay operations.

of the data structure. Let r^{i-1} and r^i denote the rank function just before and after an ith splaying step. It is not hard to see that the ranks of only three nodes are affected: the splaying node x, its parent y, and the parent of its parent z. Therefore the change in potential $\Delta\Phi$ is

$$\Delta\Phi = r^i(x) + r^i(y) + r^i(z) - r^{i-1}(x) - r^{i-1}(y) - r^{i-1}(z)$$

The actual cost of the zig splaying step is 1 rotation, and for the other two splaying steps is 2 rotations. Sleator and Tarjan [93] showed in the first case the amortized cost is at most $1 + 3(r^i(x) - r^{i-1}(x))$, and in the other two case is at most $3(r^i(x) - r^{i-1}(x))$.

Now we are ready to compute the total amortized cost of a splay operation as the sum of amortized costs of splaying steps:

$$\begin{aligned}
\hat{c}(splay) &= \hat{c}(step_1) + \hat{c}(step_2) + \ldots + \hat{c}(step_k) \\
&\leq 3(r^1(x) - r^0(x)) + 3(r^2(x) - r^1(x)) + \ldots + 1 + 3(r^k(x) - r^{k-1}(x)) \\
&\leq 1 + 3(r^k(x) - r^0(x))
\end{aligned}$$

Recall that the rank is a logarithmic function. Since the size of a root is 1 and minimal possible size of x is $\frac{1}{n}$, we have that the amortized cost of the splay operation is at most $3\log n + 1$. Also, note that maximal difference between $\Phi(T_m)$ and $\Phi(T_0)$, where T_m and T_0 is the states of the splay tree after and before a sequence of splay operations, is at most $\sum_{i=1}^{n}(\log 1 - \log(\frac{1}{n})) = n\log n$, because the maximal size of a node is 1 and minimal size of a node is $\frac{1}{n}$. Thus we have that the total amortized cost of a sequence of m splay operations is $O((m+n)\log n + m)$.

The splay operation is the subroutine of every other operation on a splay tree. To access an element, we splay the tree on the node containing this element or splay the last visited node if the element is not in the tree. To split a tree at the element x, we splay at the node containing x, and then delete the edge between x and its right child. To join two trees, we splay at the maximal element in the tree with smaller elements, and the create an edge between this element and the root of the other tree. The insertion and deletion of a node are done with operation join and split as subroutines. Since each operation does a constant number of edge additions and removals, all of this operations has the amortized cost equal to the amortized cost of the splay operation. Thus any sequence of search, insert, delete, join, and split operations have total amortized cost of $O((m+n)\log n + m)$.

Chapter 3

Dynamic Scheduling of Monotonic Intervals on Single Machine

In this chapter we study the *dynamic* problem of monotonic interval scheduling on a single machine. Recall that the static scheduling problem for a set of intervals I is to find a compatible subset of intervals of maximal size. In the dynamic problem, the input is an arbitrary sequence $o_1, \ldots, o_m$ of the following operations:

- insert(i) adds an interval i into the set I if $i \notin I$,
- remove(i) deletes an interval i from the set I if $i \in I$,
- query(i) returns true if and only if i belongs to the greedy optimal set (see Definition 2.1.5)

Our goal is to design an algorithm that minimizes the total running time of any sequence of these operations. We focus on the class of *monotonic interval sets*, which we define in Section 3.1.

Chapter 3 is organized as follows. In Section 3.1, we formally define *monotonic interval sets*. In Section 3.2, we describe algorithms for searching left and right compatible intervals of a given interval. In Section 3.3, we analyse two naive approaches to the dynamic problem of interval scheduling. In Sections 3.4 and 3.5, we provide two our data structures that solve the problem. In Section 3.6 we discuss an alternative operation report that prints out all intervals in the greedy optimal set. Finally, in Section 3.7, we empirically compare our data structures.

3.1 Monotonic Interval Set

Definition 3.1.1. *The set I of intervals is called* monotonic *if for any two intervals $x, y \in I$ neither $x \subset y$ nor $y \subset x$.*

In other words, in a monotonic interval set no interval covers another interval. An example of monotonic interval sets is a set of intervals of equal length with different starting and finishing times. Monotonic interval sets are connected with proper interval graphs. Recall that an interval graph is a 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. A proper interval graph can be converted to a monotonic interval set by a linear time algorithm [31, 38, 62].

Note that in a monotonic interval set the orders $\prec_{\mathsf{s}}$ and $\prec_{\mathsf{f}}$ are equal. Therefore if an interval i starts before an interval j then i finishes before j. The following lemma gives us a method to check monotonicity of an interval set.

Lemma 3.1.2. *Let $I = \{i_1, \dots, i_n\}$ be a set of intervals such that $i_1 \prec_{\mathsf{f}} \dots \prec_{\mathsf{f}} i_n$. The set I is monotonic if and only if for all $s \in \{1, \dots, n-1\}$ neither $i_{s+1} \subset i_s$ nor $i_s \subset i_{s+1}$.*

Proof. One direction follows directly from the definition of monotonic set. For the other direction, let I be a set of intervals $\{i_1, \dots, i_n\}$ ordered by the finishing times. Take the interval i_1 with the smaller finishing time. By assumption, i_1 does not contain i_2 and i_2 does not contain i_1. It implies that $s(i_1) < s(i_2)$. Now take the interval i_3. Since i_3 does not contain i_2 and i_2 does not contain i_3, we have that $s(i_2) < s(i_3)$. Therefore $s(i_1) < s(i_3)$, and, since $f(i_1) < f(i_3)$, we have i_1 does not contain i_3 and i_3 does not contain i_1. Continuing these reasoning we have that for any interval i_s neither i_1 contains i_s nor i_s contains i_1. Thus the set I is monotonic. $\square$

Lemma 3.1.3. *Let $I = \{i_1, \dots, i_n\}$ be a set of intervals ordered by the finishing times. There exists an algorithm that decides if I is monotonic in $O(n)$ time.*

Proof. By Lemma 3.1.2, to check whether I is monotonic, one goes through the intervals $i_1, \dots, i_n$ and checks whether i_{k+1} contains i_k where $1 \leq k < n$. This takes time $O(n)$. $\square$

We assume that monotonicity of the interval set is not violated after insertion of a new interval. Indeed, if we keep the intervals sorted by their finishing time, all we need to do after an insertion of i is to look at the next and the previous intervals of i. By Lemma 3.1.2 if i is not contained and does not contain these intervals the set remains monotonic.

3.2 Operations right_compatible and left_compatible

We keep intervals in a binary search tree in order of their starting times. Let us denote this tree by $T(I)$.

Monotonicity of the interval set I implies an important property of $T(I)$: if an interval $i \in T(I)$ is not compatible with an interval j, then the left subtree of i does not contain $\mathsf{rc}(j)$ and the right subtree of i does not contain $\mathsf{lc}(j)$. This allows us to define two efficient operations: right_compatible(j) and left_compatible(j). We do not need left_compatible(j) for the representation of the compatibility forest, but we will need this operation for our second data structure described in Section 3.5.

The algorithms for theses operations are essentially a standard search in a binary search tree. For the right_compatible operation, we begin by examining the root j of the tree. If $j \prec_{\mathsf{f}} i$ or j is not compatible with i, we search for $\mathsf{rc}(i)$ in the right subtree of j. If $j \succ_{\mathsf{f}} i$ and compatible with i, we remember j as a potential $\mathsf{rc}(i)$ and search for a smaller compatible interval in the left subtree. We terminate the search if j is a leaf and return the remembered interval or nil if no interval is remembered. The pseudo-code for this operation is presented in Algorithm 3.2.

Algorithm 3.1 right_compatible(i)

```
1: $r \leftarrow$ nil
2: $j \leftarrow$ the root in the interval tree $T(I)$.
3: while $j \neq$ nil do
4:     if $j \prec_{\mathsf{f}} i$ or $j$ overlaps $i$ then
5:         $j \leftarrow$ the right child of $j$
6:     else
7:         $r \leftarrow j$
8:         $j \leftarrow$ the left child of $j$
9: end while
10: return $r$
```

The algorithm for the left_compatible operation is similar, except that we replace "$\prec_{\mathsf{f}}$" with "$\succ_{\mathsf{f}}$" and go the left child of the current node instead of the right child. The pseudo-code for this operation is presented in Algorithm 3.2.

Algorithm 3.2 left_compatible(i)

```
1:  $\ell \leftarrow$ nil
2:  $j \leftarrow$ the root in the interval tree $T(I)$.
3:  while $j \neq$ nil do
4:      if $j \succ_{\mathsf{f}} i$ or $j$ overlaps $i$ then
5:          $j \leftarrow$ the left child of $j$
6:      else
7:          $\ell \leftarrow j$
8:          $j \leftarrow$ the right child of $j$
9:  end while
10: return $\ell$
```

The correctness of the operations is proved by the following lemma.

Theorem 3.2.1. *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. Indeed, suppose $i \prec_{\mathsf{f}} j$. As I is a monotonic set, $s(i) < s(j) < f(i) < f(j)$. Take any interval ℓ that finishes after i, but before j. By monotonicity of I, $s(i) < s(\ell) < s(j)$. Therefore $s(i) < s(\ell) < s(j) < f(i) < f(\ell) < f(j)$. Thus ℓ overlaps both i and j.

Proof. Both operations takes time $\Theta(\log n)$ as the algorithms visit nodes on only one path and the length of a path from a leaf to the root in a balanced binary search tree is $\Theta(\log n)$.

To prove the correctness of right_compatible, we use the following loop invariant:

If rc(i) $\in I$, *then the subtree rooted at j contains* rc(i) *or the remembered interval r is* rc(i).

Initially, j is the root of $T(I)$ and r is nil, so the invariant holds. Each iteration of the **while** loop executes either line 5 or lines 7-8 of Alg. 3.2. Suppose line 5 is executed. Then we have $j \prec_{\mathsf{f}} i$ or j overlaps i. If $j \prec_{\mathsf{f}} i$ then all intervals in the left subtree of j are less than i. If $j \succ_{\mathsf{f}} i$ 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 $j \succ_{\mathsf{f}} i$ 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 is in I 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 $r = \mathsf{nil}$.

To prove the correctness of left_compatible, we use the following loop invariant:

If $\mathsf{lc}(i) \in I$, then the subtree rooted at j contains $\mathsf{lc}(i)$ or the remembered interval ℓ is $\mathsf{lc}(i)$.

Similarly to the proof above, we go to the left subtree of j only when $\mathsf{lc}(i)$ is not in the right subtree of j. Moreover, when we go to the right subtree of j, we remember j as a candidate for the left compatible interval. In any case we preserve the loop invariant. Thus the algorithm is correct. □

3.3 Naive Dynamic Algorithms

The first naive dynamic algorithm is to store all intervals in a self-balancing search tree ordered by their finishing time and apply GreedySchedulingAlgorithm 2.1 at each query operation. Clearly, in a sequence of operations, where every odd operation inserts an interval and every even operation asks whether the last interval belongs to the greedy set, average running time per operation is $O(n + \log n)$.

A modified yet still naive dynamic algorithm is this. Store the greedy optimal set sorted in $\prec_{\mathsf{f}}$-order in a self-balancing binary search tree T that allows splitting a tree into two trees and joining two trees into a single tree. After each $\mathsf{insert}(i)$ or $\mathsf{remove}(i)$ operation split the tree T into two parts T_ℓ and T_r, which contains intervals that finishes before and after $s(i)$. All the intervals in T_ℓ remain in the greedy optimal set, but we remove some or all intervals from T_r. Let j_0 be the greatest interval in T_ℓ. Now start adding intervals $j_1, \ldots, j_k$ into T_ℓ as in the standard greedy algorithm. If at some moment of time j_i is an interval in T_r, then stop the greedy scheduling algorithm: we know that the algorithm will choose the intervals in T_r that are greater than j_i. Therefore we split T_r at j_i and merge the greater part with the modified T_ℓ. Otherwise the greedy scheduling algorithm stops when there are no intervals to consider.

The weak part of this algorithm is that an insertion or a deletion of an interval may change the greedy optimal set completely. Indeed, consider a compatible set of intervals $A = \{a_1, \ldots, a_k\}$. We construct a compatible set of intervals B such that every interval in B intersect with exactly two intervals of A;

$$B = \{b_i \mid s(b_i) \in a_i \text{ and } f(b_i) \in a_{i+1} \text{ and } f(b_i) < s(b_{i+1}) \text{ for all } 1 \leq i < k\}$$

Let $I = A \cap B$. It is easy to see that A is the greedy optimal set of I. We insert an interval x that overlap with a_1, but does not overlap with b_1. The greedy optimal subset of $I \cup \{x\}$ is $B \cup \{x\}$, which has no common intervals with A. See an example in Figure 3.1. Therefore the modified naive dynamic algorithm will iterate through all intervals in $I \cup \{x\}$. Moreover, we keep adding similar intervals at the beginning of the interval set, forcing updates of the greedy optimal set. Thus the average running time per operation is linear.

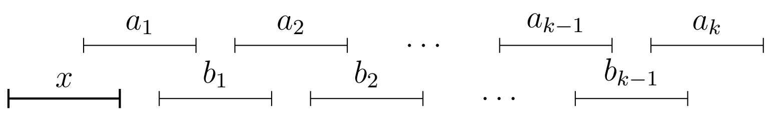


Figure 3.1: Addition of x creates completely new greedy optimal set

One may consider to use operation right_compatible to search for the interval in the greedy optimal set. In this case, updating greedy optimal set takes $O(k \log n)$ worst-case time, where k is the number of intervals inserted into the set. However, with greedy optimal set containing half of the intervals, the complexity of the operation becomes $O(n \log n)$.

3.4 Compatibility Forest Data Structure (CF)

Let I be a set of intervals. We define the *compatibility forest* as follows

Definition 3.4.1. *A* compatibility forest *is 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 *root* of a tree is a node without a parent. A *leaf* is a node with no children. Figure 3.2 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.

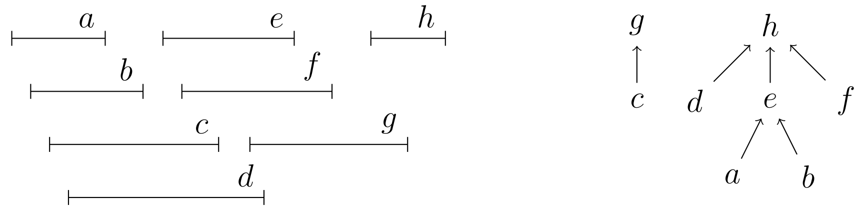


Figure 3.2: Example of a monotonic set of intervals and its compatibility forest. The compatibility forest contains two trees rooted at g and h, respectively.

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 $\prec_{\mathsf{f}}$-least interval is the greedy optimal set. Our first dynamic algorithm amounts to maintaining this path in the forest $\mathcal{F}(I)$.

The representation of the forest is developed from the dynamic tree data structure as in [91]. 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 has at most one incoming solid edge. A sequence of nodes $u_0, \ldots, u_k$ 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. An important subroutine that makes a path from a node v to the root solid is called *expose*. The operation *expose* 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. Hence, after exposing node v, all the edges on the path from v to the root become solid.

The solid paths of the compatibility forest are represented by a set of splay trees. The order of nodes in every splay tree is determined by $\prec_{\mathsf{f}}$. Since the solid paths are disjoint, a node u belongs to only one splay tree. We denote a splay tree of a node u by ST_u. The dashed edges of the compatibility forest are not represented explicitly, but computed when needed. To compute dashed edged we store all intervals in a self-balancing binary search tree in the $\prec_{\mathsf{f}}$ order from left to right. This tree, denoted by $T(I)$, supports the operations right_compatible and left_compatible that respectively return $\mathsf{rc}(i)$ and $\mathsf{lc}(i)$ of a given interval i.

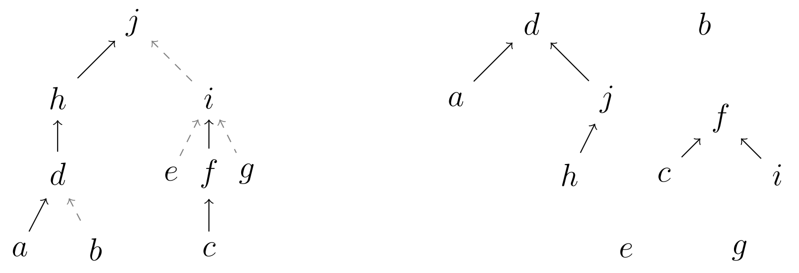


Figure 3.3: On the left, $\{a, d, h, j\}$ is a solid path in a compatibility forest. On the right, the same path $\{a, d, h, j\}$ is represented as a splay tree.

We denote this representation of the compatibility forest by CF. In the next sections we describe the algorithms of CF that solve the dynamic interval scheduling problem.

3.4.1 Update and query operations on the compatibility forest

We call the algorithms queryCF, insertCF and removeCF for the query, insertion, and removal operations, respectively. The expose operation is a subroutine of the update operations. The acronym CF indicates that these algorithms are based on the representation of the compatibility forest described above.

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

Algorithm 3.3 queryCF(i)

```
1: $m \leftarrow \mathsf{minimum}(T(I))$
2: if $\mathsf{find}(\mathrm{ST}_m, i) = i$ then
3:     return true
4: else
5:     return false
```

The operation expose*:* first, we find the maximum element j in the splay tree ST_i. Since i and j are in the same splay tree, they belongs to the same solid path. Then we search for the right compatible interval $i' = \mathsf{rc}(j)$. If i' does not exist, then j is the root in the compatibility forest. Therefore we stop the process. Otherwise (j, i') is a dashed edge in the compatibility forest. We split the splay tree at i' into trees $L(i')$ and $R(i')$ such that $i' \in R(i')$. This operation makes the solid edge that enters i' dashed. Finally we join ST_i with $R(i')$. Thus we make the edge (j, i') solid. We repeat the process taking i' as i.

Algorithm 3.4 expose(i)

```
1: $j \leftarrow \mathsf{maximum}(\mathrm{ST}_i)$
2: $i' \leftarrow \mathsf{right\_compatible}(j)$
3: while $i'$ is not nil do
4:     $\mathsf{splitR}(\mathrm{ST}_{i'}, i')$
5:     $\mathsf{join}(\mathrm{ST}_i, R(i'))$
6:     $j \leftarrow \mathsf{maximum}(\mathrm{ST}_{i'})$
7:     $i' \leftarrow \mathsf{right\_compatible}(j)$
8: end while
```

The operation insertCF*:* first, we add the interval i into the tree $T(I)$. Then we locate the next interval r of i. Since $f(i) < f(r)$, the interval i may be a substitute of r in the greedy optimal set of r. Indeed, let j be an interval such that the greedy algorithm choses r if started at j. Then i, if compatible with j, will be choose instead of r.

Therefore, if such r exists, we find the interval j before r in the splay tree ST_r. The edge (j, r) is solid. If the interval j is compatible with i, we split the tree ST_r at j such that $j \in L(j)$. The the edge (j, r) is now dashed. Then we add the new interval i into $L(j)$. This operation makes the edge (j, i) solid. We restore the longest path of the compatibility forest by exposing the least interval in $T(I)$. An example of the operation is in Figure 3.4.

Algorithm 3.5 insertCF(i)

```
1: insert($T(I), i$)
2: $r \leftarrow$ successor($T(I), i$)                ▷ Find the next interval of $i$
3: if $r \neq$ nil then
4:     $j \leftarrow$ predecessor($\mathrm{ST}_r, r$)      ▷ Find a solid edge $(j, r)$
5:     if $j \neq$ nil and $j$ is compatible with $i$ then
6:         splitL($\mathrm{ST_j}$, j)                ▷ Destroy the solid edge $(j, r)$
7:         insert($L(j), i$)
8: expose(mininum($T(I)$))
```

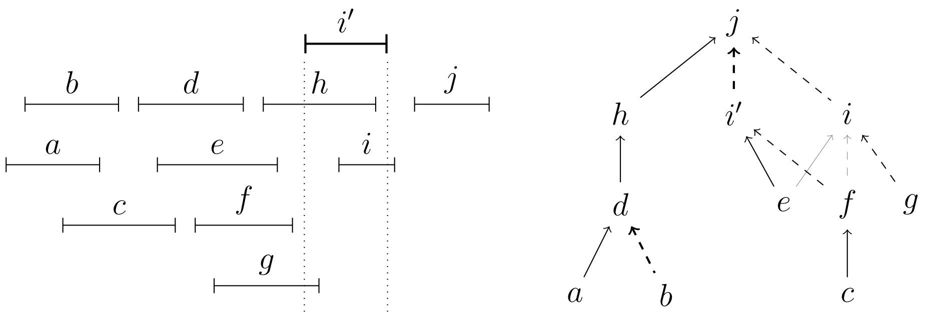


Figure 3.4: Adding i' in a set of intervals result in the changes of the compatibility forest. The edge (e, i') is new and solid. The operation does not make dashed edges that are not on the path from the least interval solid. For example, the edge (g, i) remains dashed.

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 compatibility forest by exposing the least interval in $T(I)$.

Algorithm 3.6 removeCF(i)

```
1: split($\mathrm{ST}_i, i$)          ▷ Delete all solid edges incident to $i$
2: remove($\mathrm{ST}_i, i$)         ▷ Destroy the splay tree of $i$
3: remove($T(I), i$)          ▷ Delete $i$ from the interval tree $T(I)$
4: expose(minimum($T(I)$))
```

3.4.2 Correctness and complexity 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. Also note that if i and j are subsequent intervals in a splay tree then (i, j) is a solid edge. The next lemma shows that (A1) and (A2) are invariants indeed and that the operations correctly solve the dynamic monotonic interval scheduling problem.

Lemma 3.4.2. *(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. 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 $\mathsf{size}(i)$ 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 3.4.3. *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, we consider the previous update operations.

A heavy dashed edge must have been converted from either a heavy solid edge or a light dashed edge. We first count the number of edges converted from heavy solid to heavy dashed. Such conversion can only occur during expose operation. By ($\star\star$) each expose converts at most $\log n$ light dashed edges to light solid edges. Therefore by ($\star$) it may convert at most $\log n$ heavy solid edges to heavy dashed edges.

We then count the number of stages that convert an edge from light dashed to heavy dashed. Such conversion can only occur during an update to $\mathcal{F}(I)$ by the insert or remove operations.

For $\mathsf{insertCF}(i)$, let j be the next interval of i in I. Note that by inserting i, some number of children of j in $\mathcal{F}(I)$ may be redirected to i. Let P_j be the path in $\mathcal{F}(I)$ from j to the root and P_i be the path from i to the root. This operation will cause the sizes of nodes on P_j to decrease and the sizes of nodes on P_i to increase.

For $\mathsf{removeCF}(i)$, let j be the next interval of i in I and let $\ell = \mathsf{rc}(i)$. After removing i, j becomes the right compatible interval of all children of i. Therefore all children of i are redirected to j. Let P_j be the path in $\mathcal{F}(I)$ from j to the root and P_ℓ be the path from ℓ to the root. This operation will cause the sizes of P_j to increase and the sizes of nodes on P_ℓ to decrease. Figure 3.5 illustrates these structural changes.

As discussed above, both update operations may cause the sizes of nodes in one path in $\mathcal{F}(I)$ to increase and the sizes of nodes in another path to decrease.

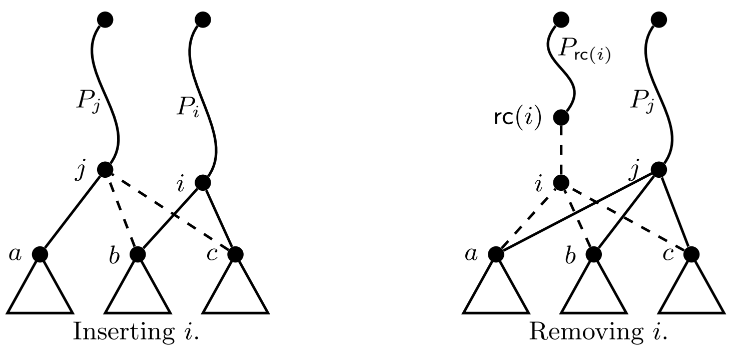


Figure 3.5: Redirections of edges in CF, where j is the next interval of i.

This may introduce new heavy dashed edges to $\mathcal{F}(I)$. Suppose the size of a path P in $\mathcal{F}(I)$ increases. Then some number of light dashed edges may become heavy. By ($\star\star$) there can only be at most $\log n$ such edges. Suppose the size of another path Q in $\mathcal{F}(I)$ decreases. Then every heavy edge (u, v) on Q may become light, which may result in some light edge (u', v) becoming heavy. By ($\star\star$) again, there can be at most $\log n$ such edges (u, v). Hence summarising the above, there can be $O(\log n)$ light dashed edges changing to heavy dashed edges during an update operation.

Hence, the total number of heavy dashed edges created after k update operations is $O(k \log n)$. □

Lemma 3.4.2 and Lemma 3.4.3 give us the following theorem:

Theorem 3.4.4. *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 the compatibility forest because insertion or removal of intervals may result in redirections of a linear number of edges. Indeed, let $b, a_0, a_1, \ldots, a_k$ be the intervals such that $\mathsf{rc}(a_i) = b$ for all $0 \leq i \leq k$. Take c such that $f(a_k) < s(c) < s(b)$ and insert it into the compatibility forest. Since b is the right compatible interval for all a_i, now c becomes the right compatible interval for all a_i. Every edge (a_i, b) must be deleted and every edge (a_i, c) must be created. 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.

Theorem 3.4.5 (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 $o_1, \ldots, o_k$ of k update operations. The first n operations in the sequence are the insert operations. We set n to be $2^m - 1$, where m is a positive natural number. The goal of these insertions is to create a full perfect binary tree[1] of height $\log n$ in the compatibility forest. The remaining $k - n$ operations are pairs of insertCF and removeCF operations. These update operations are chosen in such a way that every operations visits $\log n$ dashed edges.

We recursively describe a set of insert operations that construct a full perfect binary tree level by level. We denote by $i_{\ell,s}$ the sth interval on the level ℓ.

Basis: At the beginning, we insert an interval $i_{0,1}$ with arbitrary left and right endpoints. The interval becomes the root of the tree and fills the level 0 of the tree.

First step: We insert two intervals $i_{1,1}$ and $i_{1,2}$ such that:

- $i_{1,1}$ and $i_{1,2}$ overlap,
- $i_{1,1}$ finishes before $i_{1,2}$,
- $i_{1,2}$ finishes before $i_{0,1}$ starts.

The intervals $i_{1,1}$ and $i_{1,2}$ fill the level 1 of the tree.

Recursive step: Let ℓ be the level filled up on the previous step. We denote by j_s the sth interval we insert on the level $\ell + 1$, i.e. j_s is $i_{\ell+1,s}$. The first interval j_1 has an arbitrary starting time, but its finishing time is less that $s(i_{\ell,1})$. The second interval j_2 has starting time between $s(j_1)$ and $f(j_1)$ and has finishing time between $f(j_1)$ and $s(i_{\ell,1})$. In other words, the right compatible interval of j_1 and j_2 overlap is $i_{\ell,1}$. Hence j_1 and j_2 are the children of $i_{\ell,1}$ in the compatibility forest.

For the third interval j_3 we want it to be a child of $i_{\ell,2}$ in the compatibility forest. But at the same time, it must overlap the intervals j_1 and j_2. Therefore we choose the starting time of j_3 between $s(j_2)$ and $f(j_1)$ and the finishing time of j_3

[1] A binary tree is called *full perfect* if all the leaves are on the last level of the tree and every internal node has exactly two children.

between $s(i_{\ell,1})$ and $s(i_{\ell,2})$. For the forth interval j_4, we choose $s(j_4)$ to be between $s(j_3)$ and $f(j_1)$ and we choose $f(j_4)$ to be between $f(j_3)$ and $s(i_{\ell,2})$.

Suppose we have inserted t intervals. The starting time of the next interval j_{t+1} is between $s(j_t)$ and $f(j_1)$. The finishing time $f(j_{t+1})$ depends on the parity of t. If t is an even number then $f(j_{t+1})$ is between $s(i_{\ell,t/2})$ and $s(i_{\ell,1+t/2})$. If t is an odd number, then $f(j_{t+1})$ is between $f(j_t)$ and $s(i_{\ell,\lceil t/2\rceil})$. Once we inserted 2^ℓ intervals we move to the new level. Figure 3.6 shows an example of filling up the 2nd level of the binary tree.

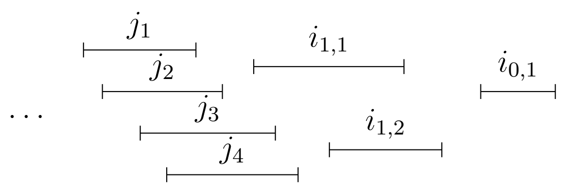


Figure 3.6: Compatibility forest for this set of intervals contains only one full perfect binary tree

After n operations the compatibility forest consists of exactly one full perfect binary tree. Recall that in our representation every node has at most one incoming solid edge. Therefore there is always a path from the root to a leaf that traverses only dashed edges. This observation leads to the description of the remaining $k-n$ operations in the sequence.

Let o_s be one of the $k-n$ remaining operations, that is $n < s \leq k$. Let T_{s-1} be the state of the tree before the operation o_s. Let h be the height of T_{s-1}. Let j_{s-1} be an interval in T_{s-1} such that a path from this interval to the root consists of dashed edges only.

If s is even then the o_s is an insertCF operation. The interval i inserted by this operation is such that $\mathsf{rc}(i) = j_{s-1}$. Note that i overlaps with all interval that finishes before j_{s-1}. In particular, if $i_{h,1} \prec_{\mathsf{f}} j_{s-1}$, the inserted interval i overlaps with $i_{h,1}$. Therefore i becomes the $\prec_{\mathsf{f}}$-least interval in the compatibility forest and the expose operation traverses $\log n$ dashed edges, creating the a greedy optimal set.

If s is odd then the o_s is a removeCF operation. The interval deleted by this operation is the interval inserted by the previous operation o_{s-1}. The expose operation traverses only one dashed edge, because we return to the greedy optimal set that was two operations before.

Thus, before every insert operation there always exists a path consisting of $\log n$

dashed edges. With a sufficiently large k, the total time taken by the operations in the sequence is $\Theta(k \log^2 n)$. $\square$

3.5 Linearised Tree Data Structure (LT)

The dynamic algorithm based on compatibility forest described above has different time complexity of update operations and query operation. In particular the update operations may take longer than the query operation. In this section we describe a new dynamic algorithm for solving the monotonic interval scheduling problem. Our goal is to improve the running time for the update operation. To achieve this we introduce a new data structure called the *linearised tree*. The dynamic algorithm based on this data structure performs all operations in amortised $O(\log n)$ time.

We say that intervals i and j are *equivalent*, written as $i \sim j$, if and only if $\mathsf{rc}(i) = \mathsf{rc}(j)$. The equivalence class of i is denoted by $[i]$. In other words, two intervals are in the same equivalence class if they are siblings in the compatibility forest. Note that if $i \sim j$ then i and j overlap.

We arrange all intervals of an equivalence class in a path using the $\prec_{\mathsf{f}}$-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. Figure 3.7 shows an example of a linearised tree.

$$\begin{tikzcd}
g & & h & \\
c \arrow[u] & d \arrow[ur] & e \arrow[u] & f \arrow[ul] \\
& a \arrow[ur] & & b \arrow[ul]
\end{tikzcd}
\qquad
\begin{tikzcd}
& g \arrow[rrrr] & & & & h \\
c \arrow[ur] & & d \arrow[r] & e \arrow[r] & f \arrow[ur] & \\
& a \arrow[r] & b \arrow[ur] & & &
\end{tikzcd}$$

Figure 3.7: Example of a compatibility forest (left), where $\{a, e, h\}$ is the greedy optimal set, and the corresponding linearised tree (right).

Formally, the *linearised tree* $\mathcal{L}(I)$ is a tuple $(I; E = E_\sim \cup E_c)$, 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.

The following lemmas follow easily from the definitions above.

Lemma 3.5.1. *Every node in a linearised tree $\mathcal{L}(I)$ has at most one equivalent child and at most one compatible child.*

Lemma 3.5.2. *The root of the linearised tree $\mathcal{L}(I)$ is the greatest interval in I.*

We represent the linearised tree of an interval set I as a set of dynamic trees. Each dynamic tree stores the intervals of some path in $\mathcal{L}(I)$. As in CF the dynamic trees are connected by dashed edges, which we compute when needed. An interval i belongs to only one dynamic tree, denoted by ST_i. Linking and cutting dynamic tree we update the edges in the $\mathcal{L}(I)$. In addition to the dynamic trees, we also store the intervals ordered from left to right by $\prec_{\mathsf{f}}$ in a self-balancing binary search tree $T(I)$. The tree $T(I)$ is used to compute previous and next intervals as well as left compatible and right compatible intervals of a given interval. We denote this representation of the linearised tree by LT.

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 provide more details below in the description of the query and update algorithms of LT.

3.5.1 The operation queryLT

Suppose we want to detect if an interval i is in the greedy optimal set. First, we check if i is the least interval. If yes, we are done. Otherwise we need to consider the path P from the least node m to the root in the linearised tree $\mathcal{L}(I)$. Since the nodes of the the path P may be stored in the different dynamic trees of LT, we call expose(m) to make sure that all the nodes of P are in one dynamic tree. Recall that in LT a interval in P not necessary belongs to the greedy optimal set. For example, in Figure 3.7, b belongs to P, but is not in the greedy optimal set. Therefore we look at the direct predecessor j of i in the path P. If j and i are compatible, we return true. Otherwise, we return false.

Algorithm 3.7 queryLT(i)

```
1:  $m \leftarrow$ 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 $i \neq$ find($\mathrm{ST}_m, i$) then       ▷ $i$ is not on the path from $m$ to the root
6:      return false
7:  $j \leftarrow$ predecessor($\mathrm{ST}_m, i$)            ▷ $(j, i)$ is an edge in LT
8:  if $i$ is compatible with $j$ then
9:      return true
10: else
11:     return false
```

Lemma 3.5.3. *The operation* queryLT(i) *returns* true *if and only if a given interval i belongs to the greedy optimal set of I.*

Proof. Let J be the greedy optimal set of I and m be the least element of I. Suppose the algorithm queryLT(i) outputs true. This can happen when (1) $i = m$. In this case i is the least element of I, hence i belongs to J; or (2) i is compatible with predecessor(ST_m, i). Note that for every interval x from ST_m there exists an interval y from the greedy set J such that $y \in [x]$. Consider such an interval $y \in J \cap [\text{predecessor}(\mathrm{ST}_m, i)]$. Since $y \in J$ and i is the next compatible interval of y, i belongs to the greedy optimal set J.

It is not hard to see by induction on the number of elements in J that $J \subseteq \mathrm{ST}_m$. Suppose the algorithm queryLT(i) outputs false. It happens in two cases. First, i is not in ST_m. Then $i \notin J$. Second, predecessor(ST_m, i) exists and is not compatible with i. Then i is not the least interval in $[i] \cap \mathrm{ST}_m$, but every element $x \in J$ is the least element in $[x] \cap \mathrm{ST}_m$. Hence $i \notin J$. □

3.5.2 The update operations insertLT and removeLT

To add an interval i into the linearised tree, we first insert i into $T(I)$. Then, 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.

Algorithm 3.8 insertLT(i)

```
1: insert($T(I), i$)
2: if $i$ is not the greatest interval in $I \cup \{i\}$ then        ▷ $i$ has a parent
3:     if lc(rc($i$)) $= i$ then                                    ▷ $(i, \mathsf{rc}(i)) \in E_c$
4:         link($i$, rc($i$))
5:     else                                                         ▷ $(i, \mathsf{next}(i)) \in E_\sim$
6:         link($i$, next($i$))
7: $j \leftarrow$ previous($i$)
8: if rc($j$) $=$ rc($i$) then                                      ▷ $(j, i) \in E_\sim$
9:     cut($j$) and link($j, i$)
10: $j \leftarrow$ lc($i$)
11: if rc($j$) $= i$ then                                           ▷ $(j, i) \in E_c$
12:     cut($j$) and link($j, i$)
```

To remove an interval i from the linearised tree, we first we delete i from $T(I)$. Then 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.

Algorithm 3.9 removeLT(i)

```
 1: if $i$ is not the root then
 2:     cut$(i)$
 3: $j \leftarrow$ previous$(i)$
 4: if rc$(j)$ = rc$(i)$ then                              ▷ $(j, i) \in E_\sim$
 5:     cut$(j)$
 6:     if $i$ = lc(rc$(i)$) then                          ▷ rc$(i)$ is a new parent of $j$
 7:         link$(j$, rc$(i))$
 8:     else if $i$ is not the the root then               ▷ next$(i)$ is a new parent of $j$
 9:         link$(j$, next$(i))$
10: $j \leftarrow$ lc$(i)$
11: if $i$ = rc$(j)$ then                                  ▷ $(j, i) \in E_c$
12:     cut$(j)$
13:     remove$(T(I), i)$
14:     $k \leftarrow$ next$(j)$
15:     if $j$ is not the root then
16:         if rc$(k)$ = rc$(j)$ then                      ▷ rc$(j) \neq i$ as we removed $i$ from $T(I)$.
17:             link$(j, k)$
18:         else
19:             link$(j$, rc$(j))$
20: else
21:     remove$(T(I), i)$
```

3.5.3 Correctness of the update operations

To prove correctness of the algorithms above, we state two claims about linearised trees. The first claim allows us to check if the given interval the greatest in its equivalent class. The second claim says that changes of the linearised tree after insertion or deletion of an interval i are local with respect to i. We abuse notation and write $(I; E)$ instead of $(I; E_c, E_\sim)$ and $(I'; E')$ instead of $(I'; E'_c, E'_\sim)$. Which edges are used will be clear from the context.

Claim 3.5.4. *An interval i is the greatest in $[i]$ if and only if* $\mathsf{lc}(\mathsf{rc}(i)) = i$.

Proof. Let i be the greatest interval in $[i]$. Then for any $j \in [i]$ we have that $j \prec_{\mathsf{f}} i$. Assume that $k = \mathsf{lc}(\mathsf{rc}(i)) \neq i$. Then $i \prec_{\mathsf{f}} k$ which is a contradiction.

For the other direction, assume that i is not the greatest interval in its equivalent class, that is there exists $j \in [i]$ such that $i \prec_{\mathsf{f}} j$. Clearly, j is compatible with $\mathsf{rc}(i)$. Therefore $\mathsf{lc}(\mathsf{rc}(i)) = j$, witch is a contradiction. □

Claim 3.5.5. *Let $\mathcal{L}(I) = (I, E)$ and $\mathcal{L}(I') = (I', E')$ be two linearised trees such that $I' = I \cup \{i\}$. Let j and k be intervals from the set I'. Then the following properties are satisfied:*

(1) if $(j, k) \notin E$ and $(j, k) \in E'$, then either $j = i$ or $k = i$.

(2) if $(j, k) \in E$ and $(j, k) \notin E'$, then $(j, i) \in E'$.

Proof. For the first property, we note that if two intervals from I are not connected by an edge in $\mathcal{L}(I)$ then they are not connected by an edge in a bigger linearised tree $\mathcal{L}(I')$. Hence either $j = i$ or $k = i$. For the second property, let $\ell \neq k$ be a parent of j in $\mathcal{L}(I')$. Because $(j, \ell) \notin E$ and $(j, \ell) \in E'$, the property (1) implies that either $j = i$ or $\ell = i$. Thus $\ell = i$. □

Lemma 3.5.6. *The operation* $\mathsf{insertLT}(i)$ *preserves linearised tree data structure.*

Proof. Consider intervals $j, k \in I'$, where $j \prec_{\mathsf{f}} k$ and $I' = I \cup \{i\}$. Let (I', E') be the resulting tree after the algorithm $\mathsf{insertLT}(i)$ is performed. We show that $(j, k) \in E'$ if and only if (j, k) is an edge in $\mathcal{L}(I')$.

($\rightarrow$) Suppose that $(j, k) \in E'$. We prove that (j, k) is an edge in $\mathcal{L}(I')$.

- Let $(j, k) \notin E$. Then the algorithm insertLT must have added (j, k) into E'. Any edge the algorithm adds is adjacent to i. First, we consider outgoing edges, that is, we consider the case when $j = i$. If the algorithm adds an edge from i to $\mathsf{rc}(i)$, then $i = \mathsf{lc}(\mathsf{rc}(i))$ (see lines 3-4 of the Algorithm 3.8). By Claim 3.5.4, i is the greatest interval in its equivalence class. If the algorithm adds an edge from i to the next interval k of i, then i is not the greatest interval in $[i]$ and $k \sim i$ (see lines 3-6). Second, we consider incoming edges, that is, we consider the case when $k = i$. If the algorithm adds an edge from $\mathsf{lc}(i)$ to i, then $j = \mathsf{lc}(\mathsf{rc}(j))$ (see lines 10-12). By Claim 3.5.4, j is the greatest interval in its equivalence class. If the algorithm adds an edge from the previous interval j of i to i, then $j \sim i$ (see lines 7-9). Note that any of the edges added by the algorithm is an edge in $\mathcal{L}(I')$. Hence (j, k) is an edge in $\mathcal{L}(I')$.

- Let $(j,k) \in E$. Assume that (j,k) is not an edge in $\mathcal{L}(I')$. By Claim 3.5.5 (j,i) is an edge in $\mathcal{L}(I')$. If j is the equivalent child of i, then j is the previous interval of i and $\mathsf{rc}(j) = \mathsf{rc}(i)$. If j is the compatible child of i, then $j = \mathsf{lc}(\mathsf{rc}(j))$. In both of these cases the algorithm deletes the edge from j (see lines 7-9 and 10-12 correspondently). Thus $(j,k) \notin E'$, which is a contradiction.

($\leftarrow$) Suppose that (j,k) is an edge in $\mathcal{L}(I')$. We prove that $(j,k) \in E'$.

- Let $(j,k) \in E$. Assume that $(j,k) \notin E'$. Then the algorithm $\mathsf{insertLT}$ must have deleted (j,k). There are two cases: j is the previous interval of i and $j \sim i$ (see lines 7-9), or $i = \mathsf{rc}(j)$ and j is the greatest interval in $[j]$ (see lines 10-12). In either case, j is a child of i in $\mathcal{L}(I')$, that is $k = i$, which is a contradiction to the assumption that $(j,k) \in E$.

- Let $(j,k) \notin E$. By Claim 3.5.5, either $j = i$ or $k = i$. Suppose $j = i$. If i is the compatible child of k in $\mathcal{L}(I')$, then $k = \mathsf{rc}(i)$ and, by Claim 3.5.4, $i = \mathsf{lc}(\mathsf{rc}(i))$. If i is the equivalent child of k, then k is the next interval i and $k \sim i$. The algorithm $\mathsf{insertLT}$ adds the edge (i,k) to E' in lines 3-6. Suppose, $k = i$. If j is the equivalent child of i, j is the previous interval of i and $j \sim i$. If j is the compatible child of i, then $j = \mathsf{lc}(\mathsf{rc}(j))$. The algorithm $\mathsf{insertLT}$ adds the edges (j,i) to E' in lines 7-12. In any case, the edge $(j,k) \in E'$.

□

Lemma 3.5.7. *The operation* $\mathsf{removeLT}(i)$ *preserves linearised tree data structure.*

Proof. Suppose $\mathcal{L}(I) = (I,E)$ is the linearised tree of a set I of intervals and $(I \setminus \{i\}, E')$ is the resulting tree after the algorithm $\mathsf{removeLT}(i)$ is performed. Consider intervals j and k in I, where $j \prec_{\mathsf{f}} k$. We want to show that $(j,k) \in E'$ if and only if (j,k) is an edge in $\mathcal{L}(I \setminus \{i\})$.

($\rightarrow$) Suppose $(j,k) \in E'$. We prove that (j,k) is an edge in $\mathcal{L}(I \setminus \{i\})$.

- Let $(j,k) \in E$. Assume that (j,k) is not an edge in $\mathcal{L}(I \setminus \{i\})$. By Lemma 3.5.5, either $j = i$ or $k = i$. If $j = i$, that is, i is a child of k in $\mathcal{L}(I)$, then the algorithm removes the edge (i,k) in line 2. Consider the case when $k = i$, that is, j is a child of i. If j is the equivalent child of i, the algorithm removes the edge (j,i) in lines 3-5. If j is the compatible child of

i, the algorithm removes (j, i) in lines 10-12. In either case $(j, k) \notin E'$, which is a contradiction.

- Let $(j, k) \notin E$. The algorithm removeLT must have added the edge (j, k). There are four possible cases. First, the algorithm adds an edge in line 7, that is, $k = \mathsf{rc}(i)$. Then j is the equivalent child of i and i is the greatest interval in $[i]$. After removing i, j is the greatest interval in $[j]$, so that j is the compatible child of k. Second, the algorithm adds an edge in line 9, that is, k is the next interval of i. Then $i \sim k$. Since j is the equivalent child of i, $j \sim k$. Third, the algorithm adds an edge in line 17. Then k is the next interval of j with respect to $I \setminus \{i\}$ and $j \sim k$. Finally, the algorithm adds an edge in line 19. Then j is the greatest interval in $[j]$ and $k = \mathsf{rc}(j)$ with respect to $I \setminus \{i\}$. In all these case the edge (j, k) is an edge in $\mathcal{L}(I \setminus \{i\})$.

$(\leftarrow)$ Suppose (j, k) is an edge in $\mathcal{L}(I \setminus \{i\})$. We prove that $(j, k) \in E'$.

- Let $(j, k) \in E$. Assume that $(j, k) \notin E'$. Then the algorithm removeLT must have deleted the edge (j, k). First, the algorithm removes an edge from i (see line 2). Second, it removes an edge from the equivalent child of i (see lines 3-58). Finally, it removes an edge from the compatible child of i (see lines 10-12). Thus the algorithms removes only edges, incident to i, but these edges are not in $\mathcal{L}(I \setminus \{i\})$, which is a contradiction.

- Let $(j, k) \notin E$. By Lemma 3.5.5 (j, i) is an edge in $\mathcal{L}(I)$. Suppose j is the equivalent child of i. The algorithm finds j in lines 3-5. If j is the compatible child of k in $\mathcal{L}(I \setminus \{i\})$, then i is the compatible child of k in $\mathcal{L}(I)$. If j is the equivalent child of k, then k is the next interval of i. The algorithm takes care of both cases in lines 6-9 and adds the edge (j, k) in line 9. Suppose j is the compatible child of i. The algorithm finds j in lines 10-11. If j is the equivalent child of k, then k is the next interval of j and $\mathsf{rc}(j) = \mathsf{rc}(k)$ with respect to $I \setminus \{i\}$. The algorithm adds the edge (j, k) in lines 15-17. If j is the compatible child of k, then the algorithm adds the edge in line 19. Thus, $(j, k) \in E'$.

□

Theorem 3.5.8. *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.*

Proof. Lemmas 3.5.3-3.5.7 prove the correctness of the operations. The complexity follows from the fact that every operation performs the constant number of the dynamic tree operations that have $O(\log n)$ amortised complexity. □

Note. The complexity type of the operation, amortised or worst-case, 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 [91]. 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.

3.6 Alternative query operation: report

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, at which we start the in-order traversal. Since ST_m equals to the greedy optimal set, the report operation performs optimally in CF.

Theorem 3.6.1. *The complexity of the* reportCF *operation is $\Theta(|J|)$, where J is the greedy optimal set.*

In the LT data structure, however, that path from the least interval to the root contains intervals that do not belong to the optimal set. Therefore, in-order traversal of ST_m may visit extra nodes. Namely, we need to filter out those nodes v in ST_m for which there exists a $u \in I$ such that $(u, v) \in E_{\sim}$. In the worst case ST_m contains all the intervals of I. Figure 3.8 shows an example of a linearised tree where all intervals belongs to the same path.

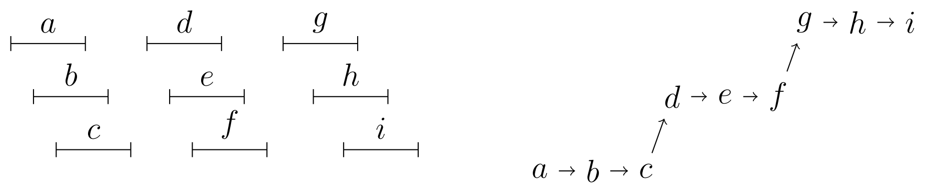


Figure 3.8: Example of a linearised tree where all interval belongs to the same path.

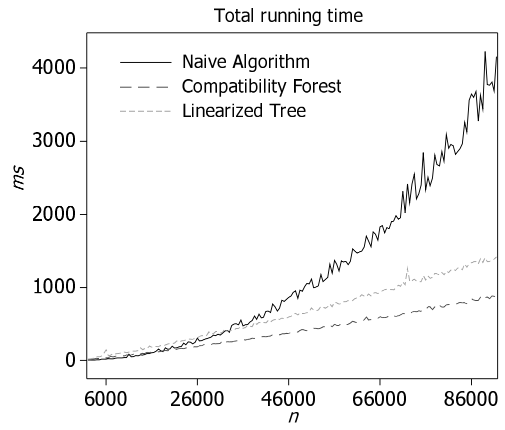
Theorem 3.6.2. *The complexity of the* reportLT *operation is $O(n)$, where n is the size of I.*

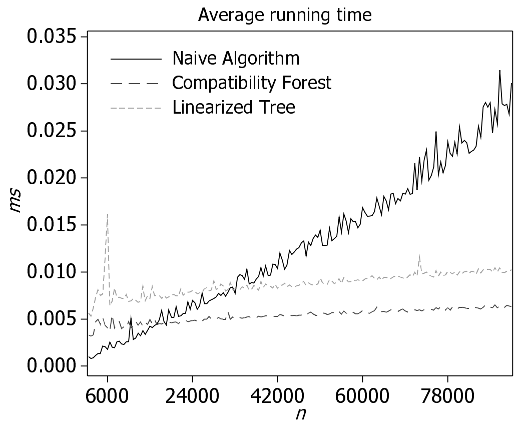
3.7 Experimental results

In this section we present an experimental comparison between three algorithms for solving monotonic case of the dynamic interval scheduling problem: the naive dynamic algorithm N described in Section 3.3, the algorithm CF based on the compatibility forest and the algorithm LT based on the linearised tree. We implemented these algorithms in Java. The algorithm N is based on the standard Java implementation of Red-Black tree, which we extended with left_compatible and right_compatible operations. We use the implementation of N in the algorithms CF and LT to store intervals and perform tree operations. In CF and LT we implemented bottom-up splay operation as described in [91]. We run the experiments on a laptop with *4GB of RAM* memory and *Intel Core 2 Duo 2130 Mhz, 3MB of L2 cache memory* processor.

In our experiments, we measure the total and the average running time of a sequence of m operations on initially empty interval set. The sequence consists of n insert operations, rn remove operations and qn query operation, where n is a linearly increasing number and r and q are fixed parameters of the experiment. We create a sequence of operations randomly while satisfying two conditions. First, whenever we invoke an insert operation of an interval i, we make sure that there is no interval i in the set. Second, whenever we invoke a remove operation of i, we make sure that i exists in the set. Thus every update operation calls for an actual change of the interval set.

To better understand the algorithms' performance, we defined the *sparsity* of an interval set I to be the upper bound on the ratio between the size of the greedy





n	N	CF	LT
66000	1823	597	928
67000	1745	590	968
68000	1802	612	972
69000	1904	610	965
70000	1931	610	1027

n	N	CF	LT
66000	0.01829	0.00599	0.00931
71000	0.02155	0.00598	0.00998
76000	0.02032	0.00606	0.00956
81000	0.02371	0.00625	0.00981
86000	0.02803	0.00651	0.01027

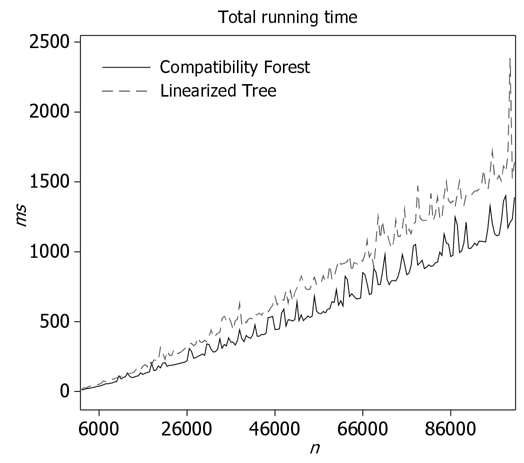
Figure 3.9: Sparsity is 0.1, $0.5n$ remove operations, $0.01n$ query operations, n insert operations.

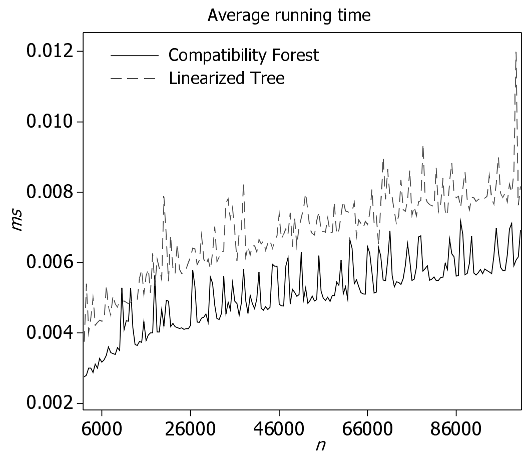
optimal set J and the size of I. The smaller the sparsity, the more intervals pairwise overlap. For example, if the sparsity is 1/2, we make sure by creating intervals of the length $2/n$ that at most every second interval can belong to J.

The sparsity of I has an important influence on the algorithms N and CF. In the compatibility forest we conclude every update operation with the expose operation on the least interval in the set, which restores the missing edges between intervals from J. Therefore the smaller sparsity, the smaller chance of an update operation to affects the splay tree, representing set J. In the naive algorithm, the query operation may visit every interval from J. Therefore the smaller sparsity, the less maximal number of intervals the query operation may visit.

Experiment 1. The analysis of the algorithms shows that N updates the interval set faster than CF and LT, but queries the set slower. Therefore in the first experiment we measured the efficiency of the algorithms undergoing n insert, $0.5n$ remove and $0.01n$ query operations. The operations are shuffled as described above. We set the sparsity parameter to be 0.1. The result of the experiment is shown on the Figure 3.9.

The experiment shows that the difference of total running time between algo-





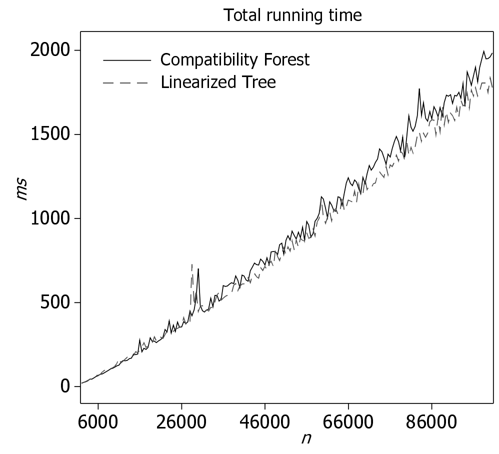
n	CF	LT
82000	901	1221
84000	975	1230
86000	966	1348
88000	995	1508
90000	1027	1412

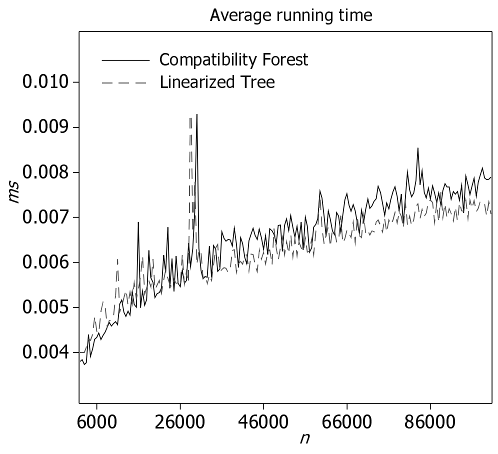
n	CF	LT
82000	0.00549	0.00744
84000	0.00580	0.00732
86000	0.00561	0.00783
88000	0.00565	0.00856
90000	0.00570	0.00784

Figure 3.10: Sparsity is 0.8, no remove operations, n query and insert operations.

rithms undergoing a sequence of operations with number of insertion less then 6000 is small, especially between CF and N. However, when we increase the number of insert operations, N performs much slower than two other algorithms. The average running time per operation of N is increasing similarly to a linear function, whereas the average running time per operation of CF and LT increases much slower. The experiment also shows that CF updates the interval set with low sparsity faster than LT.

Experiments 2 and 3. In the next two experiments we measure the performance of CF and LT undergoing a sequence of operations with the equal number of insert and query operations. We excluded N from the experiments because N performs too slowly when the number of query operations increases. The difference between the second and the third experiment is in the number of remove operations. Sequences in Experiment 2 do not contain remove operations. Sequences in Experiment 3 contain $0.5n$ remove operations. We set the sparsity parameter to be 0.8. Figure 3.10 shows the results of Experiment 2, Figure 3.11 shows the results of Experiment 3.





n	CF	LT
82000	1547	1425
84000	1688	1507
86000	1593	1587
88000	1660	1516
90000	1727	1572

n	CF	LT
82000	0.00754	0.00695
84000	0.00803	0.00717
86000	0.00740	0.00738
88000	0.00754	0.00689
90000	0.00767	0.00698

Figure 3.11: sparsity is 0.8, $0.5n$ remove operations, n query and insert operations.

The second experiment shows that if we do not allow remove operations, CF performs faster than LT. If we allow remove operations, CF performs slightly slower than LT. However, the results of Experiment 2 show that if the interval set is not sparse, CF inserts and removes intervals faster than LT.

Conclusion. The experimental result verifies our theoretical analysis and shows that both CF and LT runs significantly faster than the naive algorithm. Moreover, the results show that in a random environment CF performs as fast as LT to within a constant factor, despite the worst $(\log^2 n)$ time upper bound. Considering that CF is relatively easy to implement, CF can find its practical applications.

Chapter 4

Dynamic Interval Scheduling on Multiple Machines

In this chapter we continue our study on dynamic interval scheduling. We consider the case of multi-machine scheduling. Recall that the static problem for a set of intervals I is to find a partitioning function σ such that the number of subsets in the partition is minimal possible and subset is a compatible set of intervals. Such function is called an *optimal scheduling function*. In the dynamic problem, the input is an arbitrary sequence $o_1, \ldots, o_m$ of the following update operations:

- insert(i) adds an interval i into the set I if $i \notin I$,
- remove(i) deletes an interval i from the set I if $i \in I$,

Our goal is to design an algorithm that maintains an optimal scheduling function and minimizes total running time of any sequence of these operations.

Chapter 4 is organized as follows. In Section 4.1 we give definition of *idle intervals* and discuss why idle intervals are important. In Section 4.2 we define *nested scheduling functions*, prove that it is an optimal scheduling function and prove that there exists a nested scheduling function for any interval set. In Section 4.3 we describe two data structures that maintain nested scheduling function. The complexity of update operations in these data structures are $O(d \log n)$ and $O(d + \log n)$. Finally, in Section 4.4, we show that the bound $O(d + \log n)$ is tight for any data structure maintaining a nested tree.

4.1 Idle Intervals

Imagine a schedule S_i of the machine m_i. The schedule consists of compatible intervals $a_1, a_2, \ldots, a_z$, which are sorted by their starting time. According to the schedule, the machine processes the interval a_1 first, then it processes the interval a_2, then - interval a_3, and so on. The machine stops when it finishes processing the interval a_z. At first glance it seems that the machine is busy in the period from the start of a_1 to the end of a_z. However, the starting and finishing time of the intervals are fixed. Therefore there might be idle periods of time between subsequent intervals in the schedule. For an example, see Figure 4.1. We call idle periods of time in a schedule as *idle intervals*:

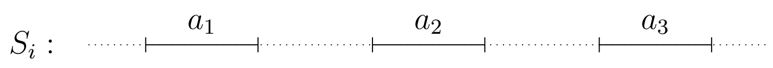


Figure 4.1: Schedule S_i consists of three intervals. The machine m_i is idle before $s(a_1)$, after $f(a_3)$ and in the periods of time from $f(a_1)$ to $s(a_2)$ and from $f(a_2)$ to $s(a_3)$.

Definition 4.1.1 (Idle Intervals). *Let $J = \{a_1, a_2, \ldots, a_m\}$ be a compatible set of closed intervals sorted in $\prec_{\mathsf{s}}$-order. Define the set of* idle intervals of J *as the following set*

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

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

Note that an idle interval can start at $-\infty$ or end at ∞. Naturally, such intervals represent a period of time when a machine is continuously available before it starts processing the first interval in its schedule or after it finishes processing the last interval.

Let $\sigma : I \to \{1, \ldots, k\}$ be a scheduling function. Recall the definition of the schedules $S_1, \ldots, S_k$ with respect to σ: $S_i = \{a \in I \mid \sigma(a) = i\}$. We define the set of idle intervals of a scheduling function as follows:

Definition 4.1.2. *The set of* idle intervals of σ *is*

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

Through the scheduling function σ we can also enumerate the set of idle intervals. Namely, the *schedule number* $\sigma(b)$ of an idle interval $b \in \text{Idle}(\sigma)$ is i if $b \in \text{Idle}(S_i)$. Note that the set $\text{Idle}(\sigma)$ includes an interval $(-\infty, \infty)$. This interval represents an extra machine that is always available to us, but not used for processing. For this infinite interval, we set $k+1$ to be its schedule number.

The idle interval set depends on the scheduling of intervals in I. Indeed, consider the set $I = \{a = (1,4), b = (2,5), c = (6,8)\}$. One possible scheduling function σ places interval a into the schedule S_1 and intervals b and c into the schedule S_2. Another function σ' places c and a into the schedule S_1 and b into the schedule S_2. In the set $Idle(\sigma)$, the interval that starts at 5 finishes at 6, while in the set $Idle(\sigma')$ the interval that starts at 5 finishes at ∞. The example is shown on Figure 4.2.

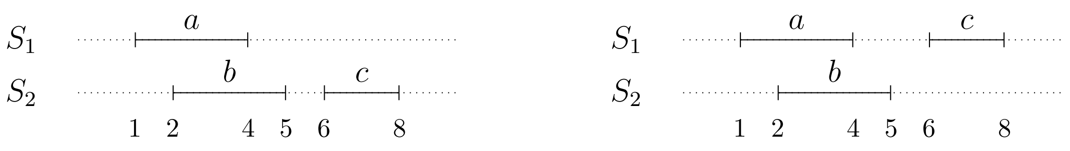


Figure 4.2: Different scheduling functions have different idle interval sets.

It is not hard to see that a depth of an idle interval set can be as big as the $|I|+1$. However, in the next lemma we prove that the depth of an idle interval set is at least $d(I)+1$

Lemma 4.1.3. *Let σ be a scheduling of I. We have $d(\text{Idle}(\sigma)) \geq d(I)+1$.*

Proof. First, observe that for any sets of intervals I and J, the following inequality holds true:

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

Now consider a scheduling function σ with schedules $S_1, \ldots, S_k$. Since the depth of each schedule S_i is 1, we have

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

Next, we calculate the depth of $Idle(\sigma)$. There are k schedules, and for each schedule S_i there is an idle interval $(-\infty, s(a_i^1))$, where a_i^1 is the $\prec_{\mathsf{s}}$-least intervals in S_i. Also, there is an idle interval $(-\infty, \infty)$ in $Idle(\sigma)$. Take a real number $x \in \mathbb{R}$ that is smaller than all the starting times of the intervals in I. In particular, x is smaller that the starting time of any a_i^1. Therefore x intersects with $k+1$ intervals in Idle(σ). Taking into account our previous inequality, we have

$$d(I) + 1 \leq k + 1 = d(\text{Idle}(\sigma))$$

$\square$

Having defined idle intervals and established connections between interval sets, scheduling functions and idle interval sets, we are ready to discuss insertions of new intervals. Let a be an interval we are inserting into the interval set I scheduled by the function σ. Since our goal is to maintain an optimal scheduling, we suppose that σ is optimal. It is easy to see that if there exists an idle interval b such that a is covered by b then we simply place a into the schedule of b. However, what if such idle interval does not exists? Does it imply that we must create a new schedule for a? Or can we change the scheduling function such that there exists an idle interval covering a? In the following theorem we prove that optimal scheduling functions for I and $I \cup \{a\}$ have the same size if and only if there exists a sequence of intersecting idle intervals whose total idle period covers a.

Theorem 4.1.4. *Let σ be a scheduling function for I. There exists an scheduling function of the same size for $I \cup \{a\}$ if and only if there exists a set of idle intervals $\mathcal{C} = \{c_1, \ldots, c_z\} \subset Idle(\sigma)$ such that*

- $c_i \prec_{\mathsf{s}} c_{i+1}$,
- $c_i \cap c_j \neq \emptyset$ *if and only if* $j = i + 1$,
- $s(c_1) < s(a)$ *and* $f(a) < f(c_z)$.

Before we prove the theorem, let us discuss an example in Figure 4.3. The figure shows two different scheduling σ_1 (left) and σ_2 (right) of a set of six intervals $I = \{b_1, b_2, b_3, b_1', b_2', b_3'\}$. Suppose that we are inserting an interval a. First, consider the set of idle intervals of the function σ_1. None of the idle intervals in $Idle(\sigma_1)$ covers the new interval a. However, there are three idle intervals $i_j = (f(b_j), s(b_j')), j \in \{1, 2, 3\}$ such that a is covered by $(s(i_1), f(i_3))$. Moreover, i_1 intersects with i_2 and i_2 intersects with i_3. The intersection of idle intervals implies that b_1' is compatible with b_2 and b_2' is compatible with b_3. Therefore we move the interval b_1' to the schedule S_2, the interval b'_2 to the schedule S_3, and the interval b'_3 to the schedule S_1. As a result we obtain the scheduling function σ_2. The function defines the idle interval $(f(b_1), s(b'_3))$ which covers the new interval a.

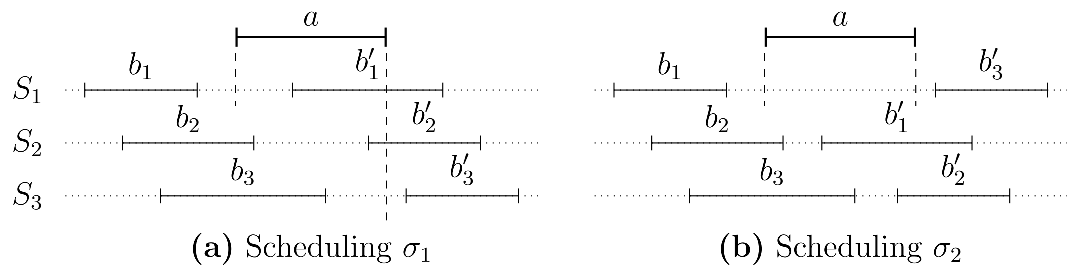


Figure 4.3: Inserting a new interval into an interval set, scheduled by two optimal functions. There is no idle interval in $Idle(\sigma_1)$ that covers a, while there is one in $Idle(\sigma_2)$.

Proof. For one direction, we use a contrapositive argument.

Assume that $d(I \cup \{a\}) = d(I) + 1$. This means that there exists a point $q \in a$ such that q intersects with $d(I)$ intervals in I. Therefore for any scheduling function, q does not belong to any of the idle intervals. Furthermore, for any set of intervals $\mathcal{C}$ the point q does not belong to $s(c_1), f(c_z)$. Thus there does not exists a set $\mathcal{C}$ such that subsequent idle intervals intersect, that is the condition $c_i \cap c_{i+1} \neq \emptyset$ for every $1 \leq i < z$ is false.

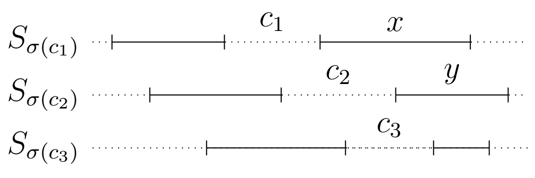
For the other direction, we use a proof by induction on the size of $\mathcal{C}$.

In the base case the set $\mathcal{C}$ consist of only one interval c_1. By definition of $\mathcal{C}$, $s(c_1) < s(a)$ and $f(a) < f(c_1)$. In other words, a is covered by c. Thus a scheduling function σ' for $I \cup \{a\}$ maps all intervals in I to the same numbers as σ and maps the new interval a to the number $\sigma(c_1)$.

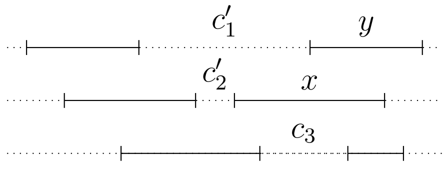
Now suppose that the theorem holds for sets of sizes up to z, and let $\mathcal{C} = \{c_1, c_2, \ldots, c_{z+1}\}$ be a set of idle intervals satisfying conditions of the theorem. We function σ' of the same size such that there exists a set of idle intervals $\mathcal{C}' = \{c'_1, c_3, c_4, \ldots, c_{z+1}\}$ where $c'_1 = c_1 \cup c_2$.

Our construction is by exchanging intervals in the schedules $S_{\sigma(c_1)}$ and $S_{\sigma(c_2)}$. Consider the point $p = f(c_1)$. In the schedule $S_{\sigma(c_1)}$ none of the intervals contain the point p. Moreover, since intersection of c_1 and c_2 is not empty, none of the intervals in $S_{\sigma(c_2)}$ contain p either. Therefore all intervals in the schedule $S_{\sigma(c_1)}$ that start after p are compatible with all intervals in the schedule $S_{\sigma(c_2)}$ that finish before p. Furthermore, the other intervals of the schedules - the intervals before p in $S_{\sigma(c_1)}$ and the intervals after p in $S_{\sigma(c_2)}$ - are pairwise compatible as well. Thus the following function σ' is a scheduling function:

$$\sigma'(b) = \begin{cases} \sigma(c_2) & \text{if } \sigma(b) = \sigma(c_1) \text{ and } a \succ_{\mathsf{s}} c_1, \\ \sigma(c_1) & \text{if } \sigma(b) = \sigma(c_2) \text{ and } a \succ_{\mathsf{s}} c_2, \\ \sigma(b) & \text{otherwise.} \end{cases}$$



(a) Scheduling σ



(a) Scheduling σ'

Figure 4.4: Two idle intervals c_1 and c_2 intersect in the set $Idle(\sigma)$. After rescheduling, $c'_1 = c_1 \cup c_2$ and $c'_2 = c_1 \cap c_2$.

The size of σ' equals to the size of σ.

Now consider the new set of idle intervals Idle(σ'). It is not hard to see that the only two idle intervals has changed: c_1 is now c'_1 and c_2 is now c'_2. The idle interval c'_1 starts at $s(c_1)$ and finishes at $f(c_2)$, and the idle interval c'_2 starts at $s(c_2)$ and finishes at $f(c_1)$. Since other idle intervals has not changed, we have the set $\mathcal{C}' = \{c'_1, c_3, \ldots, c_{z+1}\}$ of the size z satisfying conditions of the theorem. Thus, by induction hypothesis, there exists a scheduling function of the same size for the set $I \cup \{a\}$. $\square$

From the proof of the Theorem 4.1.4 we obtain two important things. First, the depth of an interval set increases after insertion only if there is no set of idle intervals whose union is a superset of the inserted interval. Second, we can decrease the size of a set of subsequently intersecting idle intervals by changing the schedule. Moreover, the intersecting intervals becomes nested after rescheduling. For example, in the Figure 4.4 the idle interval c'_2 is a subset of c'_1. In the next sections we employ these ideas into the construction of an efficient data structure that maintains an optimal scheduling of an interval set.

4.2 Nested Scheduling

In this section we give definitions of *nested interval sets*, *nested tree* and *nested scheduling*. Then we prove that there exists a nested scheduling function for any interval set. We describe how to restore nestedness of a scheduling after insertion and deletion of an interval.

Definition 4.2.1. *A set J of intervals is* nested *if there exists one interval that covers any other interval and for all $b_1, b_2 \in J$, it is either that b_1 covers b_2 or b_2 covers b_1 or b_1, b_2 are compatible.*

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

Lemma 4.2.2. *For any nested set J of intervals, $d(J) = 1 + h$, where h is the height of the nested tree* $\text{Nest}(J)$.

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

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

In the next definition we connect nested interval sets with scheduling functions.

Definition 4.2.3. *Let σ be a scheduling function of the set of intervals I. We say that σ is* nested schedule *if the set* $\text{Idle}(\sigma)$ *of idle intervals is nested.*

An example of a nested scheduling is presented in Figure 4.5.

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

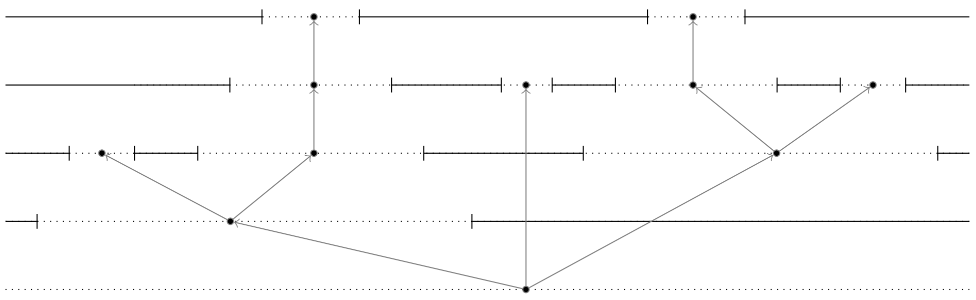


Figure 4.5: Example of a nested scheduling and corresponding nested tree

Theorem 4.2.4. *If $\sigma : I \to \{1, \ldots, k\}$ is a nested scheduling function, then the depth Idle(I) equals the depth I plus one: $d(\mathrm{Idle}(\sigma)) = d(I) + 1$. In particular, every nested schedule is optimal.*

Proof. Let $\sigma : I \to \{1, \ldots, k\}$ be a nested scheduling function for I. We will show that

$$d(I) \leq d(\mathrm{Idle}(\sigma)) - 1 \leq d(I)$$

The inequality on the left hand side follows from Lemma 4.1.3. For the right hand side, recall that by Lemma 4.2.2, we have $d(\mathrm{Idle}(\sigma)) - 1 = h$, where h is the height of the nested tree Nest(Idle(σ)). Therefore, it is sufficient to prove that $h \leq d(I)$.

Take a maximal path $b_0, b_1, \ldots, b_h$ in Nest(Idle(σ)) such that $f(b_1) \neq \infty$. Such path does exists since in every schedule there is an idle interval that starts at $-\infty$ and finishes at the $\prec_{\mathsf{s}}$-least interval of that schedule. For each $i \in \{1, \ldots, h\}$ let S_i be a schedule such that $b_i \in \mathrm{Idle}(S_i)$. We show that in every schedule S_i there exists an interval $a_i \in S_i$ such that $f(b_1)$ intersects with a_i.

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

A natural question is whether the schedule constructed by either *Algorithm* 2.2 or *Algorithm* 2.3 (described in Section 2.1.2) is nested. The next simple example gives a negative answer to this question. Indeed, consider the set I of intervals presented in Figure 4.6. Both algorithms yield the same scheduling, which is not nested:

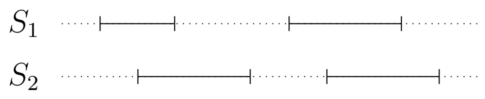


Figure 4.6: Neither *Algorithm* 2.2 nor *Algorithm* 2.3 produces a nested schedule.

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

4.2.1 Extending Nestedness after Insertion

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

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

- Let $L \subset \text{Idle}(\sigma)$ be the set of all the idle intervals that contain $s(a)$, but do not cover a. The set L, as $\text{Idle}(\sigma)$ is nested, is a sequence of embedded intervals $x_1 \supset \cdots \supset x_\ell$, where $\ell \geq 1$. Note that L can be the empty set.

- Let $R \subset \text{Idle}(\sigma)$ be the set of all the idle intervals that contain $f(a)$, but do not cover a. The set R, as above, is a sequence of embedded intervals $y_1 \supset \cdots \supset y_r$, where $r \geq 1$. Again, R can be empty as well.

- Let z be the shortest interval in $\text{Idle}(\sigma)$ that covers a. Such an interval exists since $(-\infty, \infty) \in \text{Idle}(\sigma)$. To simplify the presentation, we set $x_0 = y_0 = z$. Note that $x_0 \supset x_1$ and $y_0 \supset y_1$.

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

Case 1: L and R are empty sets

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

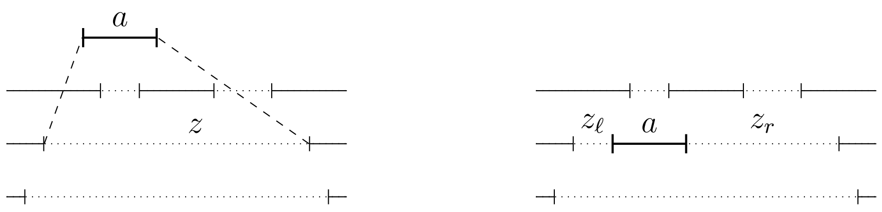


Figure 4.7: Rescheduling when a does not overlap with idle intervals

After insertion of a, the idle interval z is split into two intervals $z_\ell = [s(z), s(a)]$ and $z_r = [f(a), f(z)]$. Consider an arbitrary idle interval u in $Idle(\sigma)$. If u and z are compatible then u is compatible with both z_r and z_ℓ. If $z \subset u$ then the intervals z_ℓ and z_r are now covered by u. If $z \supset u$ then u is either covered by z_ℓ or z_r, or u does not intersect with the new idle intervals. Thus the resulting set of idle intervals is nested. See an example in Figure 4.7.

Case 2: L is not empty, but $R = \emptyset$

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

We schedule interval a for the machine $\sigma(x_\ell)$. We move all the jobs d of the machine $\sigma(x_\ell)$ such that $d \succ x_\ell$ to machine $\sigma(x_{\ell-1})$. In the schedule $S_{\sigma(x_{\ell-1})}$ there are other jobs that start after $x_{\ell-1}$. To avoid collisions, we move these jobs to the machine $\sigma(x_{\ell-2})$. We continue this on until we reach the jobs scheduled for the machine $\sigma(z)$. Finally, we move the jobs d from the machine $\sigma(z)$ such that $d \succ z$ to the machine $\sigma(x_\ell)$. Note that if $f(z) = +\infty$ there is simply no jobs on the machine $\sigma(z)$ to reschedule. Example of this process is on Figure 4.8. Formally, we define the new scheduling function σ_1 as follows and claim that σ_1 is nested:

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

Claim A. The scheduling σ_1 defined is nested.

Proof. The set of idle intervals Idle(σ_1) consists of the new interval $[f(a), f(z)]$ together with all the idle intervals of σ where the idle intervals x_ℓ, $x_{\ell-1}$, …, x_1,

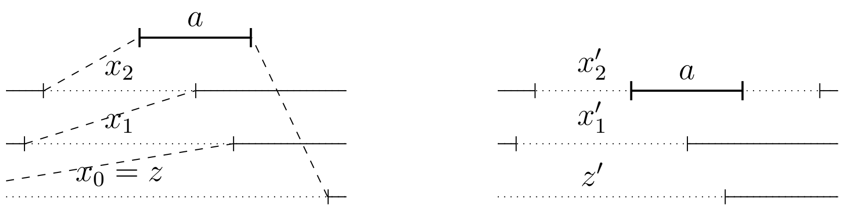


Figure 4.8: Rescheduling when idle intervals overlapping a contains only $s(a)$

and z are changed to the following new idle intervals $[s(x_\ell), s(a)]$, $[s(x_{\ell-1}), f(x_\ell)]$, …, $[s(x_1), f(x_2)]$, and $[s(z), f(x_1)]$, respectively. We denote the set of changed intervals by L'.

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

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

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

This case is symmetric to the previous case: we need to reorganize the intervals, but we reorganize the intervals with respect to the finishing time of the new interval a. First, we set $\sigma(a)$ equal to $\sigma(z)$. The new idle interval $(f(a), f(z))$ now intersects

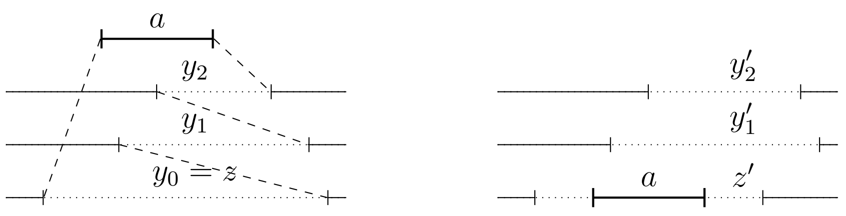


Figure 4.9: Rescheduling when idle intervals overlapping a contains only $f(a)$

with idle intervals $y_1, \ldots, y_r$. Therefore for every $1 \le i < r$ we move intervals from the machine $\sigma(y_i)$ that start after y_i to the next machine $\sigma(y_{i+1})$. To avoid collision on the last machine $\sigma(y_r)$ we move intervals from this machine that start after y_r to the machine $\sigma(z)$. An example with two intervals y_1 and y_2 is shown on Figure 4.9. Formally, we define a new scheduling function σ_2 as follows:

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

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

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

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

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

Lemma 4.2.5. *The scheduling function σ_2 is nested.*

Proof. Let K be the set of intervals in Idle(σ_2) that begins at or after $s(x_0)$ and ends at or before $f(x_0)$. In other words

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

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

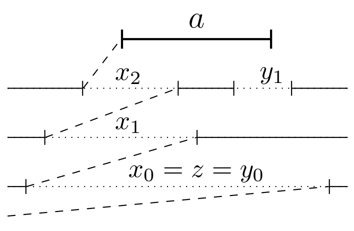


Figure 4.10: The first step of rescheduling: defining σ_1.

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

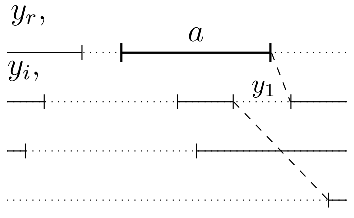


Figure 4.11: The second step of rescheduling: defining σ_2.

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

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

- Case 1: q contains $s(a)$. Then, similarly to p, the start of q is $s(x_j)$ for some $0 \le j \le \ell$. If $x_j \prec x_i$ then q covers p. Otherwise, p covers q.

- Case 2: q does not contain $s(a)$. Let r be a real number such that $r \in q \cap p$. If such r does not exists, then p and q are compatible. Otherwise $r \in [s(x_i), s(a))$ or $r \in (s(a), f(x_{i+1})]$. Since $\text{Idle}(\sigma)$ is a nested set, we have that either $q \subset x_i$ or $q \subset x_{i+1}$. Hence $q \subset p$.

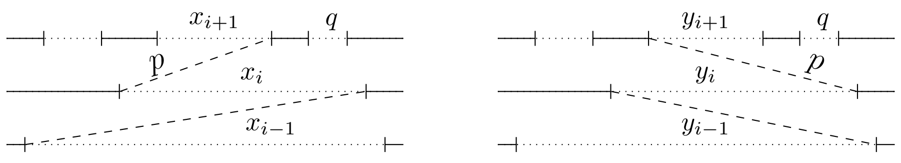


Figure 4.12: Nestedness is preserved

Now suppose p contains $f(a)$. Then the end of p is $f(y_i)$ for some $0 \leq i \leq r$. Moreover, the start of p is $f(a)$, if $i = r$, and $f(y_{i+1})$, otherwise. Similarly to the previous case, if q contains $f(a)$ then, depending on the end of q, one of the intervals covers the other. If q does not contain $f(a)$, there are two cases:

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

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

Theorem 4.2.6. *For any set of closed intervals I there exists a scheduling function σ such that* $\text{Idle}(\sigma)$ *is a nested set.*

Proof. We prove by induction on the size $|I|$ of I. When $|I| = 1$ it is clear that $\text{Idle}(I)$ is nested. The inductive step follows directly from the construction of σ_2 and Lemma 4.2.5. □

4.2.2 Restoring Nestedness after Deletion

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

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

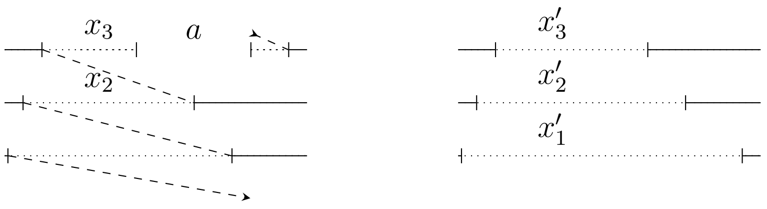


Figure 4.13: Rotation of the schedules preserves the nestedness

Case 1: L and R are empty sets

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

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

In this case, after deletion of the interval a, the idle interval $[s(x_\ell), f(y_r)]$ intersects with $x_{\ell-1}$. Therefore we move all the jobs d of the machine $\sigma(x_\ell)$ such that $d \succ x_\ell$ to the machine $\sigma(x_1)$. Then we move all the jobs d of the machine $\sigma(x_{\ell-1})$ such that $d \succ x_{\ell-1}$ to the machine $\sigma(x_\ell)$. We repeat this process on every machine $\sigma(x_i)$. We stop after we moved the jobs of the machine $\sigma(x_1)$ to the machine $\sigma(x_2)$. Denote the new scheduling function by σ_3. An example of the rescheduling is shown in Figure 4.13.

Claim B. The scheduling σ_3 defined is nested.

Proof. Let u and v be two idle intervals in Idle(σ_1). Similarly to the case in the previous subsection, if u and v are both old or both new idle intervals, then they are either compatible or one is contained in the other. So suppose u is a new idle interval, and v is old. Let t be a real number such that $t \in u \cap v$.

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

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

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

□

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

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

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

We reschedule the intervals in two passes. We start as in the Case 2. Namely, for every machine $\sigma(x_i)$ we move the intervals $d \succ \sigma(x_i)$ to the machine $\sigma(x_{i+1})$ if $1 \leq i < \ell$, and to the machine $\sigma(x_1)$, if $i = \ell$. Denote the resulted schedule by σ_1. Note that all the idle intervals in R except y_r are preserved. The interval y_r is changed to $y'_r = [s(x_\ell), f(y_r)]$. This interval now intersects with all other intervals in R. To restore the nestedness, we move all the intervals $d \succ y_i$ of the machine $\sigma_1(y_i)$ to the machine $\sigma_1(y_{i-1})$ if $1 < i \leq r$, and to the machine $\sigma(y'_r)$ if $i = 1$. Denote the final schedule by σ_4. We give an example of the rescheduling in Figure 4.14, leaving the formal description of σ_4 to the reader.

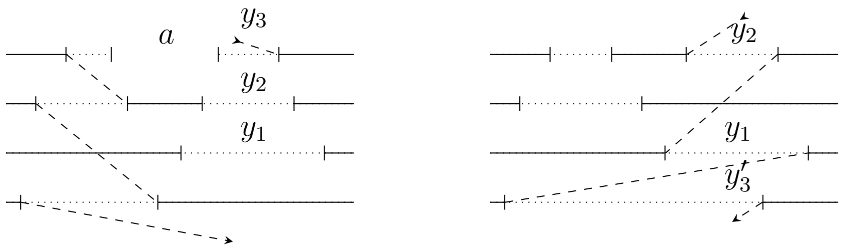


Figure 4.14: Rescheduling after deletion of an interval a.

Lemma 4.2.7. *The scheduling σ_4 is nested*

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

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

- $u = [s(x_1), f(y_1)]$. In this case, u is the longest interval in K. Therefore u covers v.

- u is one of the changed intervals in L. That is, $u = [s(x_i), f(x_{i_1})]$ for some $1 < i \leq \ell$. For contradiction, assume that u and v intersect. Then v contains either $s(x_i)$ or $f(x_{i-1})$. Moreover, since $v \notin L$, $s(a) \notin v$. In other words, v starts and finishes before or after the point $s(a)$. Therefore, v intersects with either x_i or x_{i-1}, which contradicts with the fact that $\text{Idle}(\sigma)$ is nested.

- u is one of the changed intervals in R. That is, $u = [s(y_{i-1}), f(y_i)]$ for some $1 < i \leq r$. Similarly, assume for contradiction that u and v intersect. Then v contains $s(f_i)$ and the endpoint $f(v)$ is between $f(a)$ and $s(y_i)$, or v contains $f(y_{i-1})$ and the start point is between $f(a)$ and $f(y_{i-1})$. Either case contradicts with the nestedness of $\text{Idle}(\sigma)$.

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

4.3 Data Structures for Nested Scheduling

While various data structures [68, 91] can be used for maintaining a set of nested intervals, they are not optimal in maintaining the nested schedule. Recall that a nested schedule depends on the set of interval I. Therefore when we insert or delete an interval from I, we need to update $O(d)$ intervals in a nested schedule.

In this section, we describe two data structures for nested scheduling. In both data structure we maintain the nested tree of σ. We assume that every endpoint is linked to two corresponding intervals, real and idle. For an example, if a schedule contains intervals $[3,6]$ and $[8,9]$ subsequently, we have a direct access from the potion 6 to the real interval $[3,6]$ and to the idle interval $[6,8]$.

In the first data structure, we maintain nested scheduling as a set of a self-balancing tree. We show that update operations in this data structure require $O(d \cdot \log n)$ time. In the second data structure, we maintain nested scheduling in an interval tree. We show that the complexity of update operations improves to $O(d + \log n)$.

4.3.1 Straightforward Implementation

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

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

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

For the `shortenLeft` and `shortenRight` operations, after shortening an idle interval, we need to update the children of the updated interval. Let v be an idle interval and p be new value. We split the children of v into two sets A and B. The set A contains idle intervals in $Sub(v)$ whose finishing time is less that p, and the set B contains idle intervals whose starting time is greater that p. We assume that

p does not intersect any of the children of v. Therefore $A \cup B = Sub(v)$. If the interval has been shortened at the left, we add intervals in A to the parent of v and set B as the children of the update intervals. If the interval has been shortened to the right, A becomes the children of v and B is merged with the children of v's parent. Shortening operation takes $O(\log n)$ time. An example in Figure 4.15 shows the result of applying `shortenRight` to the interval x_i.

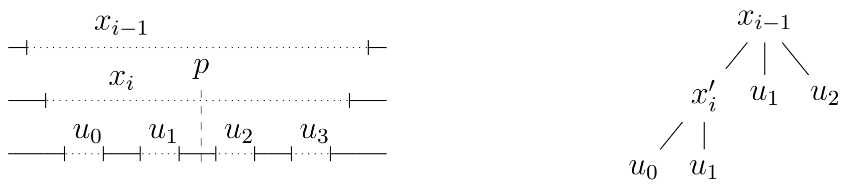


Figure 4.15: Changes in the nested tree after applying `shortenRight(`x_i, p`)`.

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

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

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

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

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

Theorem 4.3.1. *The data structure described above maintains the optimal scheduling and supports insertions and deletions in $O(d \cdot \log n)$ worst-case time.*

Proof. When we update or delete an interval, we change idle intervals that intersect with at most two points. It takes $O(d \log n)$ time to find these idle intervals. Once found, we change endpoints of every idle interval. To change endpoint of an idle interval in T takes $O(\log n)$ time, since we need to split and join children of the interval and its parent. We change endpoints of at most $2d$ intervals. Thus in total an update operation take $O(d \log n)$ time. The correctness of the operations follows from Lemma 4.2.5 and Lemma 4.2.7 □

4.3.2 Optimal Data Structure

In this section we describe how maintain a nested schedule and perform update operations in $O(d + \log n)$ worst-case time.

We store idle intervals in an *interval tree* [80]. An interval tree is a leaf-oriented binary search tree where leaves store endpoints of the intervals in increasing order. Intervals themselves are stored in the internal nodes as follows. For each internal node v the set $I(v)$ consists of intervals that contain the *split point* of v and are covered by the *range* of v. The split point of v, denoted by $split(v)$, is a number such that the leaves of the left subtree of v store endpoints smaller than $split(v)$, and the leaves of the right subtree of v store endpoints greater than $split(v)$. The range of v, denoted by $range(v)$, is defined recursively as follows. The range of the root is $(-\infty, \infty)$. For a node v, where $range(v) = (l, r]$, the range of the left child of v is $(l, split(v)]$, and the range of the right child of v is $(split(v), r]$. An example of an interval tree is shown in Figure 4.16.

We represent each set $I(v)$ as a linked list. The intervals in $I(v)$ are stored in order of their left endpoints. Since the set is nested, every interval in a list covers all the subsequent intervals in the list. To search for all intervals intersecting a given point p do the following. Start at the root and visit the nodes $v_0, \ldots, v_k$, where v_{i+1} is the right child of v_i if $p > split(v_i)$, and the left child otherwise. At every node v_i, scan $I(v_i)$ and report all intervals containing p. Note that p

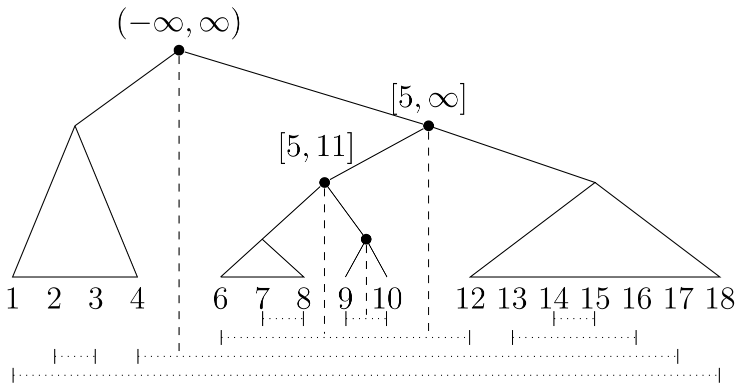


Figure 4.16: Nested set of intervals represented by interval tree data structure.

intersects with at most d intervals and the length of the path is $O(\log n)$. Thus the search takes $O(d + \log n)$ time.

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

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

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

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

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

The deletion of an interval a is similar to insertion. First we delete the intervals x_ℓ and y_r and the endpoints $s(a)$ and $f(a)$ from the interval tree. Then we traverse the path defined by $s(a)$. Recall that $x_{\ell-1}, \dots, x_1$ are the idle intervals that intersect with $s(a)$. We change the starting time of all of them. Suppose v is a node such that $x_i \in I(v)$. Clearly, x_{i+1} is in the $range(v)$. For every interval x_i we encountered we set $s(x_i)$ to be $s(x_{i+1})$. We move the changed intervals x'_i to the node v such that $split(v) \in x'_i \subset range(v)$. Note that v is on the path we are traversing. Similarly, we traverse the path defined by $f(a)$, update and move intervals y_i. Finally, we add a new idle interval $[s(x_1), f(y_1)]$ into the interval tree.

Theorem 4.3.2. *The data structure described above maintains the optimal scheduling and supports insertions and deletions in $O(d + \log n)$ worst-case time.*

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

The optimality of scheduling follows from Lemma 4.2.5 and Lemma 4.2.7. □

4.4 Tight Complexity Bound for Nested Trees

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

Theorem 4.4.1 (Shamos and Hoey 87). *$\Omega(n \log n)$ is a lower bound on the time required to determine if n intervals on a line are pairwise disjoint.*

Theorem 4.4.1implies that we cannot construct a nested tree of n intervals in less than $\Omega(n \log n)$ time:

Lemma 4.4.2. *$\Omega(\log n)$ is a tight bound on the time required to update a data structure that maintains a nested tree.*

Proof. For contradiction, assume that there is a data structure with a complexity $f(n) \in o(\log n)$. We create a nested tree of n intervals using this data structure. If the height of the tree is 1, then the intervals do not intersect. However, the time taken is $n \cdot f(n) \in o(n \log n)$, which contradicts Theorem 4.4.1. □

Next, we prove that nested scheduling function is unique.

Lemma 4.4.3. *If σ and τ are nested scheduling functions then* $\text{Idle}(\sigma) = \text{Idle}(\tau)$.

Proof. For contradiction, assume that there exist two nested scheduling functions σ and τ such that $\text{Idle}(\sigma) \neq \text{Idle}(\tau)$. Then there exist two idle intervals $a_0 \in \text{Idle}(\sigma)$ and $b_0 \in \text{Idle}(\tau)$ such that they have the same non-infinite starting time, but different finishing times, i.e. $s(a_0) = s(b_0) \neq -\infty$ and $f(a_0) \neq f(b_0)$. Without loss of generality, suppose that $f(a_0) < f(b_0)$. Now we take an interval b_1 from $\text{Idle}(\tau)$ that finishes at $f(a_0)$. If its starting time is less than $s(b_0)$ then intervals b_0 and b_1 overlap, which contradicts the nestedness of τ. Otherwise, we continue to $\text{Idle}(\sigma)$ and take an interval a_1 that starts at $s(b_1)$. If $f(a_1) > f(a_0)$ then a_1 and a_0 overlap and it is a contradiction. Otherwise, we continue in the same manner to $\text{Idle}(\tau)$. Since I is finite, this process eventually stops and one of the scheduling functions appears to be not nested. □

Theorem 4.4.4. *An update operation in a data structure representing a nested tree takes at least $\Omega(\log n + d)$ time.*

Proof. Let I be an interval set and $\text{Nest}(I)$ be the nested tree of I. By Lemma 4.4.3, $\text{Nest}(I)$ is unique. Let $v_0 v_1 \ldots v_d$ be longest path in $\text{Nest}(I)$. Now consider an interval a, which starts in the middle of v_d and finishes after the end of v_1. Clearly, $s(a)$ intersects with exactly d idle intervals. Therefore the trees $\text{Nest}(I)$ and $\text{Nest}(I \cup a)$ differ in $\Omega(d)$ nodes. Taking into account Lemma 4.4.2, an update operation of a nested tree requires $\Omega(\log n + d)$ time. □

Chapter 5

Dynamic Slack Reclamation from Multiple Processors

In this chapter, we study the problem of dynamic slack reclamation in the Elastic Mixed-Criticality (E-MC) task model for multiple processors. *Slack* is an idle period of time on a processor. The E-MC model defines two types of tasks: high-criticality and low-criticality. The high-criticality tasks may generate slack on the processors at run-time. The low-criticality tasks may reclaim the generated slack. Reclamation of dynamic slack aims to decrease the total amount slack in the system by efficient scheduling of tasks.

We consider a generalization of the slack reclamation problem introduced in [97, 98]. The basic version of the problem asks whether a processor has slack of length ℓ before time d. A generalized version of the problem asks whether k processors has total slack of length ℓ before time d. We call the generalized version the `k-SLACK_RECLAMATION` problem. The problem is motivated by the observation that if none of the processors has enough slack there might exist another scheduling with the enough amount of slack on one processor.

We prove that the `k-SLACK_RECLAMATION` problem is NP-complete.

5.1 Introduction

Many embedded real-time systems consist of tasks with different levels of importance. The levels of importance are usually called *criticality* levels: the more critical a task is, the more severe consequences of the task failure are. For example, there are five criticality levels in the avionic software: catastrophic, hazardous, major, minor and no effect. Execution of all high-criticality tasks in a timely manner is vital for the system and must not be terminated. On the other hand, a low-criticality task may be terminated at runtime without jeopardizing the integrity of the system.

For safety reasons, most critical tasks are scheduled on the basis of very strict and pessimistic assumptions. However, such assumptions rarely occur in real life, and it is not uncommon that a high-criticality task take less processor time than it was provisioned. The discrepancies between estimated and actual use of computational resources lead to inefficient use of the processors. One of the area in the field of mixed-criticality (MC) systems studies models that aim at the efficient use of computational power.

Su and Zhu [97] studied the Elastic Mixed-criticality model, where they suggested the idea of *early releases*. In this model, the low-critical tasks are given periods such that high-criticality tasks can be scheduled in the worst-case scenario. These periods may be undesirably long. However, the low-criticality tasks are also given *early release points*. These points specify the moments of time when a task can release an *early job*. If at run-time the high-criticality tasks have generated enough slack for an early job, the job is released. As the result, the slack is consumed by early jobs, while the high-criticality are guaranteed to be executed on time.

Later, Hang Su et al continued their work in [98] by considering multicore systems. Indeed, sometimes the amount of slack on a processor may not be large enough to host an early released instance of its low-criticality task. However, another low-criticality task from a different processor could be executed. Task migrations between cores, or *global early-release* of low-criticality tasks, allow more efficient slack reclamation.

The chapter is structured as follows. In Section 5.2 we formally describe the E-MC model. In Section 5.3 we describe how to update the schedule when we reclaim slack from multiple processors. Finally, in Section 5.4 we formally describe the `k-SLACK_RECLAMATION` problem and prove its NP-completeness.

5.2 Elastic Mixed-Criticality Model

The *Elastic Mixed-Criticality* (E-MC) model was proposed by Su and Zhu [97]. In this model the set of n tasks $\Gamma = \{T_1, T_2, \ldots, T_n\}$ is given. Each task T is characterized by the following parameters:

- criticality level $\zeta(T)$,
- the period $p(T)$,
- the worst-case execution time (WCET) $c(T, \ell)$ for each criticality level $\ell \geq \zeta(T)$ at which the task can be executed.

All tasks are synchronous and implicit-deadline.

We consider the systems with two different criticality levels: high and low, denoted by ζ^{high} and ζ^{low}, respectively. We say that T is a high-criticality task if its criticality level is ζ^{high}, and a low-criticality task if its criticality level is ζ^{low}. Hence a low-criticality task has only one WCET, denoted by $c(T)$, and a high-criticality task has two WCET, denoted by $c^{high}(T)$ and $c^{low}(T)$.

In addition to standard parameters, a low-criticality task in the E-MC model is associated with a set of possible *early-release points*, which are denoted by $p_1(T), \ldots, p_k(T)$. While the period $p(T)$ of a task specifies the required frequency of its execution, its early-release points reflect the *desirable* frequency. For example, consider a task of sending data from a camera sensor of a remote-controlled vehicle to the control panel. Each execution of the task results in a picture frame on the control panel. The task must be executed with non-zero frequency in order to provide an operator with a picture that is clear enough to control the vehicle. However, increasing frequency will result in more detailed picture and a more accurate control of the vehicle.

The utilization of a task is defined with respect to its criticality level. For a high-criticality task, the low-level and high-level utilizations are, respectively, $u^{low}(T) = \frac{c^{low}(T)}{p(T)}$ and $u^{high}(T) = \frac{c^{high}(T)}{p(T)}$. For a low-criticality task the desired and minimum utilizations are defined as $u_i(T) = \frac{c(T)}{p_i(T)}$ and $u^{min}(T) = \frac{c(T)}{p(T)}$. Similarly, the high-level and low-level total utilizations of all high-criticality tasks are defined as

$$U(H,H) = \sum_{\zeta(T)=\zeta^{high}} u^{high}(T) \quad \text{and}$$
$$U(H,L) = \sum_{\zeta(T)=\zeta^{high}} u^{low}(T)$$

For low-criticality tasks, which require minimum execution rate, the minimal total utilization is defined as

$$U(L,min) = \sum_{\zeta(T)=\zeta^{low}} u^{min}(T)$$

A set of E-MC tasks is said to be *E-MC schedulable* if the high-level execution requirements of high-criticality tasks and minimum service requirements of low-criticality tasks can be guaranteed in the worst case scenario. Thus, a set of E-MC tasks is schedulable on one processor by `EDF` if the sum of high-level utilization of high-criticality tasks and the minimum utilization of low-criticality tasks is at most 1:

Lemma 5.2.1 (Su and Zhu [97])**.** *A set of E-MC tasks is E-MC schedulable under `EDF` if $U(H,H) + U(L,min) \leq 1$.*

In the case of multiple processors, there is a question on the degree of task migration allowed: no migration, task-level and job-level (see Section 2.2.1). Su et al. [98] proposed and evaluated `P-EDF` scheduling where no migration of tasks is allowed. That is, they partition the set of tasks into m set $\Gamma_1, \ldots, \Gamma_m$. The tasks from the set Γ_i are executed on the processor i. Similarly to the task utilizations on one processor, the high-level and low-level utilization of tasks on ith processor is defined as follows:

$$\begin{aligned}
U_i(H,H) &= \sum_{\zeta(T)=\zeta^{high} \wedge T \in \Gamma_i} u^{high}(T) \\
U_i(H,L) &= \sum_{\zeta(T)=\zeta^{high} \wedge T \in \Gamma_i} u^{low}(T) \\
U_i(L,min) &= \sum_{\zeta(T)=\zeta^{low} \wedge T \in \Gamma_i} u^{min}(T)
\end{aligned}$$

Theorem 5.2.2 ([98])**.** *A set of E-MC tasks with given partition $\Gamma_1, \ldots, \Gamma_m$ is E-MC schedulable by `P-EDF` on m-processor system if $U_i(H,H) + U_i(L,min) \leq 1$ for every $1 \leq i \leq m$.*

The problem of finding a partition of tasks such that the equation in the Theorem 5.2.2 is satisfied is equivalent to the bin-packing problem [51] and therefore is NP-complete. Hence the E-MC schedulability of tasks on multiple processors was evaluated by various partitioning heuristics Su et al. [98].

5.2.1 Slack Generation and Reclamation

In the E-MC model, the schedulability of a task set is tested with the assumption that high-criticality tasks are executed for their high-level WCET. However, at

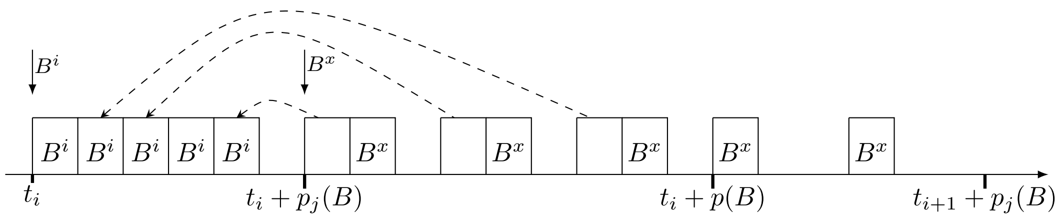


Figure 5.1: In the period from $t_i+p_j(B)$ to $t_i+p(B)$ the processor has no dedicated time for the task B since the job B^i has been completed by the beginning of the period. Therefore execution of the extra job B^x increases utilization of the processor in this period.

run-time a high-criticality task may take less time than it has been provisioned, thus generating *idle time* or *slack* on a processor. Moreover, if the total utilization of a processor is less than 1, then there are periods of time with no active tasks, and thus the processor is idle at these moments of time.

To reduce the amount of slack on a processor, an *extra* job of a low-criticality task may be released at one of the early release points, specified for the task. An extra job requires computational power of a processor but increases its utilization. Let us describe this observation in greater details.

Let B be a low-criticality task with k early released points. Assume that no extra job of B has been released until some moment of time $t_i = i \cdot p(B)$. Then in the period $[t_i, t_{i+1}]$ the task utilizes u^{min} of processor's time. In other words, a fraction $\frac{c(B)}{p(B)}$ of every 1 unit of time is dedicated for the execution of B. Now suppose that the job B^i has been completed by the early-release point $t_i + p_j(B)$ and we release an extra job B^x. The deadline of B^x is $t_i + p_j(B) + p(B)$. The extra job requires $p(B) \cdot u^{min}(B)$ units of time in the period from $t_i + p_j(B)$ to $t_{i+1}+p_j(B)$. However, since the job B^i has already been completed, no processor's time is dedicated for the task B in the period $[t_i+p_j(B), t_i+p(B)]$. Thus an extra job B^x increases utilization of the processor by $(p(B) - p_j(B)) \cdot u^{min}(B)$ in this period. Moreover, this is the exact amount of slack we need to release an extra job without overloading the processor [17]. See Figure 5.1 for graphical interpretation.

Su et al. [98] proposed the method for slack reclamation on one processor. In the next section we describe and analyze the problem of slack reclamation from several processors.

5.3 Slack Reclamation from Multiple Processors

In this section we study the idea of slack reclamation from several processors. The intuition behind the idea is that while none of the processors has enough slack for execution of an extra job, the *combined* slack on several processors may meet the extra job's demand.

Consider the following example. At some moment of time we have four jobs scheduled on two processors such that the first processor is idle in the period $[t_1, t_2]$ and the second processor is idle in the period $[t_2, t_3]$, where $t_1 < t_2 < t_3$. Denote the job that starts at t_2 by A^i and the job that starts at t_3 by B^j. Suppose that there is an extra job E^x of a low-criticality task E with WCET $c(E) = t_3 - t_1$ that can be released at time t_2. Since $c(E) > t_2 - t_1$ and $c(E) > t_3 - t_2$, none of the processors has enough slack for E^x given the current schedule. However, if we swap schedules of the processors at time t_2, there will be enough slack for E^x. Figure 5.2 pictures this example.

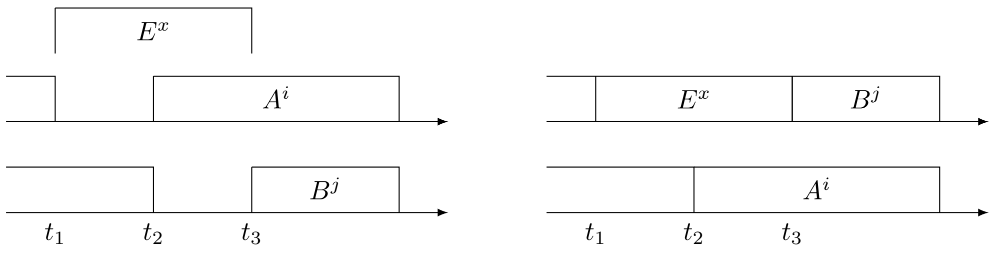


Figure 5.2: After swapping the schedules at time t^2 there is enough slack on one processor to allocate an extra job E^x.

We introduce the necessary notations. Let $\mathcal{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ be a set of m unit-speed identical processors. We define a mapping $\pi : \Gamma \to C$ that partitions the task set into m disjoint sets $\Gamma_1, \dots, \Gamma_m$ which satisfy the conditions of the Theorem 5.2.2.

Initially, the tasks in the set Γ_i are executed on the processor $\mathcal{C}_i$. However, at run-time we change task-to-core mapping. These changes are reflected by the bijective function $\sigma : \Gamma \to \Gamma$, which is called the *proxy function.* The allocation of a task T is determined by the composition of the partitioning function and the proxy function. We write $\sigma\pi(T)$ for $(\sigma \circ \pi)(T)$.

The proxy function is changed by the *swap operation*, which takes two processors $\mathcal{C}_i$ and $\mathcal{C}_j$ as an input. After the swap operation the tasks that has been executing on the processor $\mathcal{C}_i$ is executed on the processor $\mathcal{C}_j$, and vice-versa. That is, we set $\sigma(\mathcal{C}_i) = \mathcal{C}_y$ and $\sigma(\mathcal{C}_j) = \mathcal{C}_x$, where $\mathcal{C}_x = \sigma(\mathcal{C}_i)$ and $\mathcal{C}_y = \sigma(\mathcal{C}_j)$. The operation is denoted by $\mathsf{swap}(\mathcal{C}_i, \mathcal{C}_j)$.

The swap operations are scheduled in advance. Otherwise the combined slack would not have an advantaged over the slack on any of the processors. Indeed, the combined slack after a swap operations is equal to the amount of slack in the period (t, t_s) on one processor plus the amount of slack in the period (t_s, ∞) on the other processor, where t is the current moment of time and t_s is the moment of swap operation. Therefore if we call swap immediately, that is if $t_s = t$, there is no increase in the available slack on any of the processors.

We define a *swap schedule* as a sequence of *swap points* $(t, \mathcal{C}_i, \mathcal{C}_j)$, where t is a natural number and $\mathcal{C}_i, \mathcal{C}_j$ are two different processors. Each swap point $(t, \mathcal{C}_i, \mathcal{C}_j)$ schedules an invocation of the swap operation with parameters $\mathcal{C}_i$ and $\mathcal{C}_j$ at the time t. A swap schedule is denoted by $\mathcal{S}$.

We place a restriction on the swap operations: tasks on a processor $\mathcal{C}_i$ cannot be swapped if an active job of one of these tasks has been preempted. In other words, we do not allow job-level migration. We call a swap operation *feasible* if it does not violate this restriction. We call a swap scheduled *feasible* if all swap operations in the schedule are feasible.

Static and Dynamic Slack. We distinguish two types of slack: *static* and *dynamic*. The *static* slack is the period of time when there are no active jobs. The *dynamic* slack is the period of time dedicated for the execution of job that has already been completed. The main difference is that static slack depends on the schedule and can be precomputed offline, while dynamic slack is generated at runtime and cannot be predicted.

A period of slack δ is characterized by three parameters: the starting time $\delta.a$, the length $\delta.\ell$ and the deadline $\delta.d$. The starting time and the deadline specify the window in which the slack can be used. The length specifies the amount of time units available for execution of a job. We denote a period of slack as a triple $(\delta.a, \delta.\ell, \delta.d)$.

As an example, consider the following set of two tasks: a high-criticality task A and a low-criticality task B. Worst-case execution times of the tasks are $c^{high}(A) = 3$, $c^{low}(A) = 2$ and $c(B) = 4$. The periods of tasks are $p(A) = 8$ and $p(B) = 10$. The task B has early release point $p_1(B) = 6$. We schedule the task set by `EDF` algorithm on one processor. In the worst case, the first jobs of the tasks are completed at time 7. Therefore there is a period of static slack $\delta = (7, 1, 8)$. Now suppose that the job A^1 used 2 units of time instead of 3. Thus, at runtime we have a period $\gamma = (2, 1, 8)$ of dynamic slack. Note that the deadline of γ equals to the deadline of A^1. See Figure 5.3 (a).

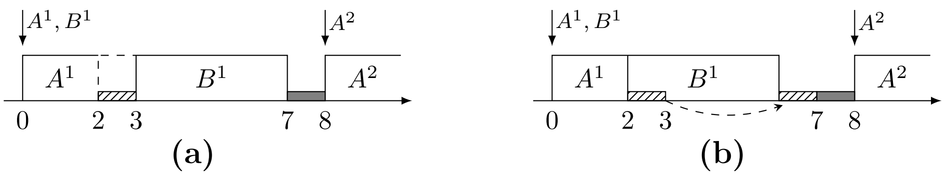


Figure 5.3: (a) In the worst case, A^1 and B^1 are completed by 7. At runtime, A^1 is completed at 2. Thus $(2, 1, 8)$ is a period of dynamic slack and $(7, 1, 8)$ is a period of static slack. (b) a period (2, 1, 8) of dynamic slack is pushed forward.

The important property of dynamic slack is that it can be claimed not only by extra jobs, but also by already active jobs. A period γ of dynamic slack is *pushed forward* by an active job B^i if the deadline of B^i is greater than the deadline of γ [104]. While we say that γ is pushed forward, the slack is in fact consumed by the job B^i. However, because B^j is now being executed earlier, this job produces a new slack. As a result the dynamic slack γ is transformed into another period of dynamic slack and is preserved for later jobs. The amount of preserved slack is equal to the length of γ of B^i, whichever is minimal. Let us describe an example.

In the Figure 5.3, we have a period of dynamic slack $\gamma = (2, 1, 8)$ generated at the moment 2. We also have an active job B^1. Since $\gamma.d \leq B^1.d$, the job claims the slack. As the results, we have a new period of dynamic slack $\gamma' = (3, 1, 10)$. Note that the deadline has changed. That is because γ' correspond to the job B^1.

5.4 Problem k-SLACK_RECLAMATION

In this section we formally define the problem k-SLACK_RECLAMATION as follows:

INPUT: a set of multicritical tasks allocated on m processors and scheduled by EDF algorithm; a collection of periods of dynamic and static slack; a low-criticality task X that is ready to be early released.

OUTPUT: YES, if there exists a feasible swap schedule such that the amount of slack reclaimed from k processors is greater or equal to the length of X. Otherwise, output NO.

While the `k-SLACK_RECLAMATION` problem is somewhat similar to the problem of finding nested scheduling of an interval set, which we described and studied in Chapter 4, we show that the problem of reclaiming slack from several processors is NP-complete.

Theorem 5.4.1. *The problem* `k-SLACK_RECLAMATION` *is NP-complete even if* $k = 2$.

Proof. We prove the theorem by reduction of the `PARTITION` problem to the `k-SLACK_RECLAMATION` problem. Recall that the `PARTITION` problem asks whether it is possible to divide a given multiset of positive integers into two subsets such that the sum of the numbers in one set equals to the sum of the numbers in the other set.

Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a multiset of positive integers. Let $\mathbb{B} = \Sigma_{i=1}^{n} a_i$. We construct a multi critical task system Γ from the set $\mathcal{A}$ as follows.

First, for each number a_i we create low-criticality tasks C_i with the execution time equal to a_i, with no early release points and with the period of $3\mathbb{B}$. Second, we create two high-criticality tasks A_1 and A_2 with maximal execution time of $2\mathbb{B}$, minimal execution time of $\frac{3}{2}\mathbb{B}$, and with the period of $3\mathbb{B}$. Then we create a s high-criticality task B with minimal and maximal execution time of $\mathbb{B}$, and with the period of $3\mathbb{B}$. Finally, we create two low-criticality tasks X and Y with the execution time of $1\mathbb{B}$, with the early release point $\frac{3}{2}\mathbb{B}$, and with that period of $2\mathbb{B}$.

Next, we partition the tasks for the execution on three processors $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$. We define the partitioning function $\pi : \Gamma \to \mathcal{C}$ as follows:

$$\pi(T) = \begin{cases} \mathcal{C}_1 & \text{if } T \text{ is } A_1 \text{ or } B \text{ for some i} \\ \mathcal{C}_2 & \text{if } T \text{ is } A_2 \text{ or } C_i \text{ for some i} \\ \mathcal{C}_3 & \text{if } T \text{ is } X \text{ or } Y \end{cases}$$

In other words, the tasks X and Y are executed by the processor $\mathcal{C}_3$, the tasks A and B are executed by the processor $\mathcal{C}_1$, and the tasks A_2 and C_i are executed by the processor $\mathcal{C}_2$ for every i. It is easy to see that utilization of each processor is 1.

Each of the tasks has a purpose. The tasks A_1 and A_2 are *generators*. They generate the dynamic slack of length $\frac{1}{2}\mathbb{B}$ on the processors. The task B is the *loader*. It increases the utilization of the processor to maximal possible value. The tasks X and Y are *consumers*. They consume the slack available for reclamation.

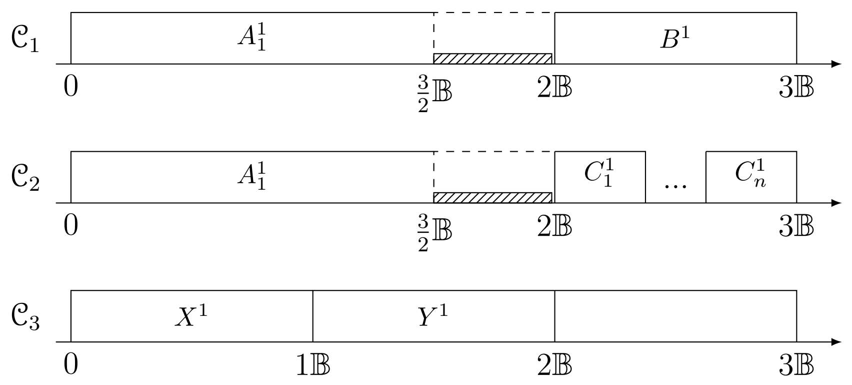


Figure 5.4: The total available slack generated by high-criticality tasks A_1 and A_2 is equal to $\mathbb{B}$ at time $\frac{3}{2}\mathbb{B}$.

The tasks $C_1, \dots, C_n$ are *verifiers*. They verify the solution to the corresponding instance of `PARTITION` problem.

Now consider the following situation. The first job of each task is released at time 0. The `EDF` algorithm schedules the jobs as shown in Figure 5.4. At run-time, the high-criticality jobs A_1^1 and A_2^1 take their minimal execution time to complete and, therefore, generate two periods of dynamic slack of length $\frac{1}{2}\mathbb{B}$. Moreover, at the time $\frac{3}{2}\mathbb{B}$ when the A^1 jobs has finished, the extra job of the task X is ready to be released. The extra job requires slack of length $1\mathbb{B}$. Therefore we ask whether it is possible to reclaim slack of length $1\mathbb{B}$ from the processors $\mathcal{C}_1$ and $\mathcal{C}_2$.

Assume that there exists an algorithm that outputs YES for the instance of the problem we described above. In other words, there exists a swap schedule with the processors $\mathcal{C}_1$ and $\mathcal{C}_2$ such that total slack on one of the processors is at least $1\mathbb{B}$. We argue that the sum of jobs executed on the processor $\mathcal{C}_1$ is equal to $\frac{\mathbb{B}}{2}$. There are two cases to consider.

Case 1. The extra job X^e is being executed on the processor $\mathcal{C}_1$. Since $\mathcal{C}_1$ has only $\frac{1}{2}\mathbb{B}$ of slack, at some moment of time t_1 we swap the schedules of the processors $\mathcal{C}_1$ and $\mathcal{C}_2$. If $t_1 < 2\mathbb{B}$, the job X^e has not used all slack on the processor $\mathcal{C}_1$, and the unused slack is now on the processor $\mathcal{C}_2$. Therefore we swap the schedules again at some moment of time $t_2 > t_1$. As the result, some of the jobs of the processor $\mathcal{C}_2$ are executed on the processor $\mathcal{C}_1$ in the period $[t_1, t_2]$. See Figure 5.5.

We claim that the sum of the lengths of the jobs that has been moved from $\mathcal{C}_2$ to $\mathcal{C}_1$ is $\frac{1}{2}\mathbb{B}$. First, recall that we do not allow swapping when there is an unfinished job on a processor. Second, $t_2 - t_1$ is equal to $\mathbb{B}$. Otherwise, the amount of collected slack would not have been enough. Finally, the amount of

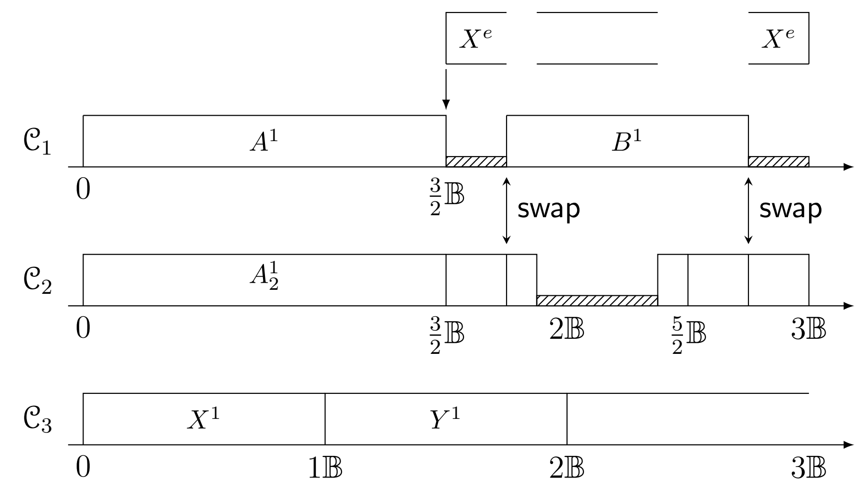


Figure 5.5: A swap schedule to reclaim two periods of dynamic slack on the processors $\mathcal{C}_1$ and $\mathcal{C}_2$ for the execution of the extra job X^e

slack in the period $[t_1, t_2]$ is equal to $\frac{1}{2}\mathbb{B}$. Since a job starts in the period if and only if it finishes in the period, the sum of the length of the jobs is equal to $\frac{1}{2}\mathbb{B}$.

Case 2. The extra job X^e is being executed on the processor $\mathcal{C}_2$.

Construct the same task system with the only difference: task B is mapped to the processor $\mathcal{C}_2$, and tasks $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are mapped to the processor $\mathcal{C}_1$. The rest of the proof is similar to the Case 1.

In both cases, if there exists a feasible swap schedule, the elements of the set $\mathcal{A}$ can be partitioned in two sets of equal total sum.

For the other direction, assume that there does not exist a partition of $\mathcal{A}$ into the sets of equal total sum. Take any moment of time t_1 such that $\frac{3}{2}\mathbb{B} < t_1 \leq 2\mathbb{B}$. Since the partition does not exist, there is no set of jobs such that the sum of their length plus the slack of $\frac{1}{2}\mathbb{B}$ equals to $t_1 + \mathbb{B}$. Therefore a possible swap can happen only after $t_1 + \mathbb{B}$. However, in this case we reclaim less that $1\mathbb{B}$ os slack. Hence there does not exist a feasible swap schedule with total reclaimed slack of $1\mathbb{B}$.

Thus the PARTITION problem is reducible to the 2-SLACK_RECLAMATION problem. □

Chapter 6

Conclusion

Scheduling as a general model of tasks and resources distribution is an active area of research. There are many unsolved problems, which are of interest in the industry. In this thesis, we have studied dynamic algorithms for three related scheduling problems.

First, we considered the scheduling problem on a single machine. We designed two data structures, Compatibility Forest and Linearised Tree, both of which allow efficient insertion, removal and query operations. Our approach to solving the problem is significantly better than a naive approach in terms of complexity as it is based on amortised data structures. Moreover, we experimentally compared the performance of the algorithms. The experiments showed that our data structures perform similarly to each other, and outperform the modified naive algorithm.

Second, we looked into the scheduling problem on multiple machines. We applied a novel approach – rather than representing periods of time when machines are working, we considered periods of time when machines are idle. This approach allowed us to define and study nested scheduling. We proved that a nested scheduling exists for any set of tasks. We designed a data structure that maintains nested scheduling. Moreover, we showed that the data structure is efficient by proving the tight complexity bounds on the insert and remove operations.

Finally, we studied the scheduling problem in Elastic Mixed-Criticality model of real time systems. We extended the reclamation problem of processor idle time to the case of multiple processors. We proved that the problem in NP-complete.

Several directions for further research remain open. The first direction is to relax monotonic restriction on an interval set. With an additional information in a binary search tree it is possible to implement right- and left-compatible operations.

However, the question is how to efficiently update paths in the Compatibility Forest: in the worst-case scenario, the algorithm may require too much time. Then, it would be interesting to look into experimental evaluation of different approaches.

The second direction is to develop an additional operation - *substitution*. This operation changes starting and/or finishing time of an existing task. A possible approach is to consider equivalence classes of intervals based on their compatibility. For example, if substitution does not change the equivalence class of an interval, it might be possible that such substitution takes constant time.

The third direction is to implement data structure for nested scheduling. This will give valuable insights on how nested scheduling performs in practice.

Finally, the other direction is to study approximation of the `k-SLACK_RECLAMATION` problem. The NP-completness of the problem follows from the `PARTITION` problem. This suggests that a simple heuristic, such as ordering of tasks by their length, or decreasing, might output good results. Also, it is interesting to experimentally compare multiprocessor slack reclamation with other models of real-time systems.

Overall, there are still many open doors and questions to answer related to scheduling. Hopefully, this thesis will take its niche in the area, and would be useful for future researches.

Bibliography

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

[2] Atkin, J. A., Burke, E. K., Greenwood, J. S., and Reeson, D. (2008a). A metaheuristic approach to aircraft departure scheduling at london heathrow airport. In *Computer-aided Systems in Public Transport*, pages 235–252. Springer.

[3] Atkin, J. A., Burke, E. K., Greenwood, J. S., and Reeson, D. (2008b). On-line decision support for take-off runway scheduling with uncertain taxi times at london heathrow airport. *Journal of Scheduling*, 11(5):323–346.

[4] Atkin, J. A., Burke, E. K., Greenwood, J. S., and Reeson, D. (2009). An examination of take-off scheduling constraints at london heathrow airport. *Public Transport*, 1(3):169–187.

[5] Babić, O. (1987). Optimization of refuelling truck fleets at an airport. *Transportation Research Part B: Methodological*, 21(6):479–487.

[6] Baker, T. P. (2005). A comparison of global and partitioned edf schedulability tests for multiprocessors. In *In International Conf. on Real-Time and Network Systems*. Citeseer.

[7] Baptiste, P. (2000). Preemptive scheduling of identical machines.

[8] Barth, T. and Pisinger, D. (2012). Scheduling of outbound luggage handling at airports. In *Operations Research Proceedings 2011*, pages 251–256. Springer.

[9] Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625.

[10] Baruah, S. K., Mok, A. K., and Rosier, L. E. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE.

[11] Bastoni, A., Brandenburg, B. B., and Anderson, J. H. (2010). An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14–24. IEEE.

[12] Beasley, J. and Cao, B. (1998). A dynamic programming based algorithm for the crew scheduling problem. *Computers & Operations Research*, 25(7):567–582.

[13] Bhatia, R., Chuzhoy, J., Freund, A., and Naor, J. S. (2007). Algorithmic aspects of bandwidth trading. *ACM Transactions on Algorithms (TALG)*, 3(1):10.

[14] Bonifaci, V., Chan, H., Marchetti-Spaccamela, A., and Megow, N. (2010). Algorithms and complexity for periodic real-time scheduling. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1350–1359.

[15] Bonifaci, V. and Marchetti-Spaccamela, A. (2012). Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 63(4):763–780.

[16] Bouzina, K. I. and Emmons, H. (1996). Interval scheduling on identical machines. *Journal of Global Optimization*, 9(3-4):379–393.

[17] Brandt, S. A., Banachowski, S., Lin, C., and Bisson, T. (2003). Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 396–407. IEEE.

[18] Brinton, C. R. (1992). An implicit enumeration algorithm for arrival aircraft. In *Digital Avionics Systems Conference, 1992. Proceedings., IEEE/AIAA 11th*, pages 268–274. IEEE.

[19] Brucker, P. (2007). *Scheduling algorithms*, volume 3. Springer.

[20] Brucker, P. and Knust, S. (2009). Complexity results for scheduling problems. http://www.informatik.uni-osnabrueck.de/knust/class/.

[21] Caprara, A., Fischetti, M., Toth, P., Vigo, D., and Guida, P. L. (1997). Algorithms for railway crew management. *Mathematical programming*, 79(1-3):125–141.

[22] Caprara, A., Toth, P., Vigo, D., and Fischetti, M. (1998). Modeling and solving the crew rostering problem. *Operations research*, 46(6):820–830.

[23] Carlisle, M. C. and Lloyd, E. L. (1995). On the k-coloring of intervals. *Discrete Applied Mathematics*, 59(3):225–235.

[24] Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., and Baruah, S. (2004). A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on scheduling algorithms, methods, and models*, pages 30–1.

[25] Carter, M. W. and Tovey, C. A. (1992). When is the classroom assignment problem hard? *Operations Research*, 40(1-supplement-1):S28–S39.

[26] Chatterjee, S., Slotnick, S. A., and Sobel, M. J. (2002). Delivery guarantees and the interdependence of marketing and operations. *Production and Operations Management*, 11(3):393–410.

[27] Chen, B., Hassin, R., and Tzur, M. (2002). Allocation of bandwidth and storage. *IIE Transactions*, 34(5):501–507.

[28] Chew, K. L. (1991). Cyclic schedule for apron services. *Journal of the Operational Research Society*, pages 1061–1069.

[29] Coffman, Jr, E. G., Garey, M. R., and Johnson, D. S. (1978). An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17.

[30] Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

[31] Corneil, D. G. (2004). A simple 3-sweep lbfs algorithm for the recognition of unit interval graphs. *Discrete Applied Mathematics*, 138(3):371–379.

[32] Corneil, D. G., Olariu, S., and Stewart, L. (1998). The ultimate interval graph recognition algorithm? In *SODA*, volume 98, pages 175–180.

[33] Corneil, D. G., Olariu, S., and Stewart, L. (2009). The lbfs structure and recognition of interval graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1905–1953.

[34] Dammak, A., Elloumi, A., and Kamoun, H. (2006). Classroom assignment for exam timetabling. *Advances in Engineering Software*, 37(10):659–666.

[35] Dawande, M., Drobouchevitch, I., Rajapakshe, T., and Sriskandarajah, C. (2010). Analysis of revenue maximization under two movie-screening policies. *Production and Operations Management*, 19(1):111–124.

[36] Dawid, H., König, J., and Strauss, C. (2001). An enhanced rostering model for airline crews. *Computers & Operations Research*, 28(7):671–688.

[37] Day, P. R. and Ryan, D. M. (1997). Flight attendant rostering for short-haul airline operations. *Operations research*, 45(5):649–661.

[38] Deng, X., Hell, P., and Huang, J. (1996). Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM Journal on Computing*, 25(2):390–403.

[39] Dertouzos, M. L. (1974). Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813.

[40] Dessouky, M. I., Lageweg, B. J., Lenstra, J. K., and Velde, S. (1990). Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44(3):115–123.

[41] Dijkstra, M. C., Kroon, L. G., van Nunen, J. A., and Salomon, M. (1991). A dss for capacity planning of aircraft maintenance personnel. *International Journal of Production Economics*, 23(1):69–78.

[42] Ding, H., Lim, A., Rodrigues, B., and Zhu, Y. (2005). The over-constrained airport gate assignment problem. *Computers & Operations Research*, 32(7):1867–1880.

[43] Eisenbrand, F. and Rothvoß, T. (2010). Edf-schedulability of synchronous periodic task systems is conp-hard. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1029–1034. Society for Industrial and Applied Mathematics.

[44] Ernst, A. T., Krishnamoorthy, M., and Storer, R. H. (1999). Heuristic and exact algorithms for scheduling aircraft landings. *Networks*, 34(3):229–241.

[45] Faigle, U. and Nawijn, W. M. (1995). Note on scheduling intervals on-line. *Discrete Applied Mathematics*, 58(1):13–17.

[46] Fizzano, P. and Swanson, S. (2000). Scheduling classes on a college campus. *Computational optimization and applications*, 16(3):279–294.

[47] Frederickson, G. N. (1983). Scheduling unit-time tasks with integer release times and deadlines. *Information Processing Letters*, 16(4):171–173.

[48] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

[49] Friesen, D. K. (1984). Tighter bounds for the multifit processor scheduling algorithm. *SIAM Journal on Computing*, 13(1):170–181.

[50] Garey, M., Johnson, D. S., Simons, B. B., and Tarjan, R. E. (1981). Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269.

[51] Garey, M. R. and Johnson, D. S. (1979). Computers and intractability: a guide to np-completeness.

[52] Gavruskin, A., Khoussainov, B., Kokho, M., and Liu, J. (2013). Dynamising interval scheduling: The monotonic case. In *Combinatorial Algorithms*, pages 178–191. Springer.

[53] Gavruskin, A., Khoussainov, B., Kokho, M., and Liu, J. (2014). Dynamic interval scheduling for multiple machines. In *Algorithms and Computation*, pages 235–246. Springer.

[54] Gavruskin, A., Khoussainov, B., Kokho, M., and Liu, J. (2015). Dynamic algorithms for monotonic interval scheduling problem. *Theoretical Computer Science*, 562:227–242.

[55] Golumbic, M. C. (2004). *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier.

[56] Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

[57] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2):416–429.

[58] Gupta, U. I., Lee, D.-T., and Leung, J.-T. (1982). Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467.

[59] Gupta, U. I., Lee, D.-T., and Leung, J. Y.-T. (1979). An optimal solution for the channel-assignment problem.

[60] Hagdorn-van der Meijden, L. and Kroon, L. G. (1990). *Pitfalls in obtaining solutions for an aircraft to gate assignment problem.* Erasmus Universiteit, Rotterdam School of Management.

[61] Hall, L. A. and Shmoys, D. B. (1992). Jackson's rule for single-machine scheduling: making a good heuristic better. *Mathematics of Operations Research*, 17(1):22–35.

[62] Heggernes, P., Meister, D., and Papadopoulos, C. (2009). A new representation of proper interval graphs with an application to clique-width. *Electronic Notes in Discrete Mathematics*, 32:27–34.

[63] Ho, S. C. and Leung, J. M. (2010). Solving a manpower scheduling problem for airline catering using metaheuristics. *European Journal of Operational Research*, 202(3):903–921.

[64] Hochbaum, D. S. and Shmoys, D. B. (1987). Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162.

[65] Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185.

[66] Huddleston, S. and Mehlhorn, K. (1982). A new data structure for representing sorted lists. *Acta informatica*, 17(2):157–184.

[67] Jackson, J. R. (1955). Scheduling a production line to minimize maximum tardiness. Technical report, DTIC Document.

[68] Kaplan, H., Molad, E., and Tarjan, R. E. (2003). Dynamic rectangular intersection with priorities. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 639–648. ACM.

[69] Kise, H., Ibaraki, T., and Mine, H. (1979). Performance analysis of six approximation algorithms for the one-machine maximum lateness scheduling problem with ready times. *Journal of the Operations Research Society of Japan*, 22(3):205–224.

[70] Kleinberg, J. and Tardos, É. (2006). *Algorithm design*. Pearson Education India.

[71] Korte, N. and Möhring, R. H. (1989). An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal on Computing*, 18(1):68–81.

[72] Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE.

[73] Lenstra, J. K., Kan, A. R., and Brucker, P. (1977). Complexity of machine scheduling problems. *Annals of discrete mathematics*, 1:343–362.

[74] Leung, J. Y. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press.

[75] Leung, J. Y.-T. (1989). A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1-4):209–219.

[76] Leung, J. Y.-T. and Merrill, M. (1980). A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118.

[77] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61.

[78] Mangoubi, R. and Mathaisel, D. F. (1985). Optimizing gate assignments at airport terminals. *Transportation Science*, 19(2):173–188.

[79] McNaughton, R. (1959). Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12.

[80] Mehlhorn, K. (1984). *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer.

[81] Mingozzi, A., Boschetti, M., Ricciardelli, S., and Bianco, L. (1999). A set partitioning approach to the crew scheduling problem. *Operations Research*, 47(6):873–888.

[82] Mok, A. K. (1983). Fundamental design problems of distributed systems for the hard-real-time environment.

[83] Nowicki, E. and Smutnicki, C. (1994). An approximation algorithm for a single-machine scheduling problem with release times and delivery times. *Discrete Applied Mathematics*, 48(1):69–79.

[84] Potts, C. (1980). Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):1436–1441.

[85] Potts, C. N. and Strusevich, V. A. (2009). Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society*, pages S41–S68.

[86] Sahni, S. (1979). Preemptive scheduling with due dates. *Operations Research*, 27(5):925–934.

[87] Shamos, M. I. and Hoey, D. (1976). Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215. IEEE.

[88] Shmoys, D. B. and Tardos, É. (1993). An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1-3):461–474.

[89] Simons, B. (1983). Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM J. Comput.*, 12(2):294–299.

[90] Simons, B. B. and Warmuth, M. K. (1989). A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM J. Comput.*, 18(4):690–710.

[91] Sleator, D. and Tarjan, R. (1983). A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391.

[92] Sleator, D. D. and Tarjan, R. E. (1985a). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208.

[93] Sleator, D. D. and Tarjan, R. E. (1985b). Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686.

[94] Sleator, D. D. and Tarjan, R. E. (1986). Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69.

[95] Smith, B. M. and Wren, A. (1988). A bus crew scheduling system using a set covering formulation. *Transportation Research Part A: General*, 22(2):97–108.

[96] Stern, H. I. and Hersh, M. (1980). Scheduling aircraft cleaning crews. *Transportation Science*, 14(3):277–291.

[97] Su, H. and Zhu, D. (2013). An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 147–152. EDA Consortium.

[98] Su, H., Zhu, D., and Mossé, D. (2013). Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *RTCSA*, pages 352–357.

[99] Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318.

[100] Tarjan, R. E. and Werneck, R. F. (2005). Self-adjusting top trees. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 813–822. Society for Industrial and Applied Mathematics.

[101] Wren, A., Fores, S., Kwan, A., Kwan, R., Parker, M., and Proll, L. (2003). A flexible system for scheduling drivers. *Journal of Scheduling*, 6(5):437–455.

[102] Wren, A. and Rousseau, J.-M. (1995). *Bus driver schedulingan overview.* Springer.

[103] Yue, M. (1990). On the exact upper bound for the multifit processor scheduling algorithm. *Annals of Operations Research*, 24(1):233–259.

[104] Zhu, D. and Aydin, H. (2009). Reliability-aware energy management for periodic real-time tasks. *Computers, IEEE Transactions on*, 58(10):1382–1397.