Scheduling with release times and deadlines

Jan Elffers

Scheduling with release times and deadlines

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jan Elffers
born in Haarlem, the Netherlands

Algorithmics Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Scheduling with release times and deadlines

Author: Jan Elffers
Student id: 4014340
Email: elffersj@gmail.com

Abstract

We study the single machine version of the task scheduling problem with release times and deadlines. This problem is too simple to be of practical importance in itself, but it is also used as a relaxation in algorithms for the Job Shop scheduling problem, which is a more practical task scheduling problem. We study exact algorithms for solving the single machine problem. We propose a new lower bound for the single machine problem, which we call the half-preemptive lower bound, and analyze its practical performance when used in a branch and bound algorithm. We also study the theoretical hardness of the single machine problem with a fixed set of task lengths. For the set $\{1,P\}$, the problem is known to be solvable in polynomial time. We present an argument that the problem with two non-unit task lengths is NP-complete.

Contents

Chapter 1

Introduction

Task scheduling is a research area in Computer Science (CS) and Operations Research (OR). Task scheduling has numerous applications. In e.g. smart grids, consumers have jobs that have some kind of temporal flexibility that can be exploited by the scheduler to decrease the load on the network. In a production process, task scheduling also occurs; there could also be dependencies between tasks. Over the past 50 years, many different formalisms for scheduling problems have been studied. Numerous books covering many different contexts and solution methods have been published; see, for example, the book by Pinedo (2008).

The subject of this thesis is perhaps one of the simplest possible scheduling problems: non-preemptive task scheduling with release times and deadlines on one machine. In our problem, one is given a set of tasks. Tasks have a release time r, a deadline d and a processing time p; each task has to be scheduled for a length of p units somewhere in its *availability interval* $[r,d]$. Only one task can execute at any time (which corresponds to the fact that all tasks must be executed on one machine), and task execution is non-preemptive, i.e. tasks cannot be interrupted. The goal in our problem is to schedule the tasks such that no job finishes execution after its deadline; if this is impossible, we want (informally) to find a schedule with no job finishing execution long after its deadline. This scheduling problem is intractable in theory (Pinedo, 2008); however, branch and bound algorithms have proved to work well in various applications.

Although this problem is too simple to be directly useful in practice, it is still relevant for a number of more practical problems. First, the single-machine scheduling problem occurs as a subproblem in the Shifting Bottleneck method, a popular solution method for the Job Shop problem proposed by Adams et al. (1988). The Job Shop problem is a more general scheduling problem. Second, new solution methods for the single-machine problem could potentially be generalized to the case in which a bounded number of tasks (possibly more than one) can execute concurrently at any time, which is a problem similar to the Resource Constrained Project Scheduling Problem.

Algorithms for scheduling with both release times and deadlines are completely different from algorithms for the case with a global release time or deadline. Few results are known about the problem we consider; one positive result is that there is a polynomial-time algorithm for the case of identical task lengths (Simons, 1978).

In this thesis, we pursue two research directions. The first direction is to determine the computational complexity of the scheduling problem with task lengths restricted to a fixed set. Simons' algorithm for identical task lengths can be generalized to task lengths 1 and p together, but nothing is known for other sets of task lengths. The second direction is to gain a better understanding of the existing branch and bound algorithms for the scheduling problem (without restrictions on the task lengths). We experimentally compare two existing branch and bound algorithms; we continue with only one algorithm (Carlier's algorithm (Carlier, 1982); see also Section 2.3.2) because it clearly outperforms the other. We then compare an improved lower bound for use within the branch and bound algorithm and also find classes of instances that are hard to solve for this algorithm.

1.1 Contributions

The contributions of this thesis are the following. In our literature survey (Chapter 2) we give an overview of algorithms for single machine scheduling, and in the experimental analysis (Chapter 4) we analyze branch and bound algorithms. As a part of this we compared our implementation to an existing implementation and we found out that we missed some details that cause a great speedup. We also have theoretical contributions, presented in Chapter 3:

- A practical algorithm for the $\{1,p\}$ scheduling problem (Section 3.2);
- A new lower bound and upper bound for scheduling with release times and deadlines, both based on a polynomial-time algorithm for scheduling with two task lengths $\{1,p\}$.
- An argument why the $\{p,q\}$ scheduling problem is NP-complete for any pair of non-unit task lengths.

1.2 Outline

This thesis is structured as follows. In Chapter 2 we present existing algorithms for scheduling with release times and deadlines. We present both branch and bound algorithms for the general (unrestricted) problem and polynomial-time algorithms for the restricted problem with identical task lengths. In Chapter 3 we present our own contributions to the single machine scheduling problem. In Chapter 4 we do an experimental analysis of branch and bound algorithms: we compare the new lower bound to the preemptive lower bound and find classes of hard instances and we test whether some modifications to the branch and bound algorithm improve performance.

Chapter 2

Literature survey

2.1 Introduction

The problem we consider is *non-preemptive single-machine scheduling with release times and deadlines*. In this problem, we are given a set of tasks that have a release time, a deadline and a length, and the goal is to schedule the tasks on one machine such that each task starts execution after its release time and completes execution before its deadline. Each task executes for an amount of time equal to its length; tasks in our problem are *non-preemptive*, which means that execution of a task cannot be interrupted. The machine can execute only one task at a time. Because it may not be possible to finish all jobs before their deadlines, jobs are allowed to complete after their deadline, in which case they are called *late*.

Formally, a task is specified as a tuple (r, d, p), where r is the release time, d the deadline, and p the processing time of the task. A problem instance consists of a set of tasks $\{(r_i, d_i, p_i) \mid i = 1, \ldots, n\}$. We define a feasible schedule as follows:

Definition 2.1 (Feasible schedule)**.** *Let $\{(r_i, d_i, p_i) \mid i = 1, \ldots, n\}$ be an instance of the single machine scheduling problem with release times and deadlines. A feasible schedule is a function $T : \{1, 2, \ldots, n\} \to \mathbb{R}$ assigning start times to the tasks, such that $T(i) \geq r_i$ for each task, and the execution intervals $\{[T(i), T(i) + p_i) \mid i = 1, \ldots, n\}$ do not overlap.*

Feasible schedules form the set of possible solutions to the scheduling problem. Two feasible schedules are displayed in Figure 2.1. Our goal is to find a feasible schedule without jobs that complete much later than their deadlines. Formally, the lateness L_i of a task i with respect to a schedule T is defined as the difference between the task's completion time and its deadline: $L_i = T(i) + p_i - d_i$. Late jobs have $L_i > 0$. For a feasible schedule $T = T(1), T(2), \ldots, T(n)$, the maximum lateness of all tasks is defined as $L_{\max} = \max_{i=1,\ldots,n} L_i$. The problem we consider is the optimization problem to find a feasible schedule that minimizes this maximum lateness. In the three-field notation for scheduling problems (Graham et al., 1979), this problem is denoted $1|r_i|L_{\max}$. The problem of minimizing $L_{\max}$ is known to be strongly NP-hard (Pinedo, 2008).

Note that tasks have negative lateness if they complete before their deadlines, and it may be possible to find a feasible schedule with $L_{\max} < 0$. Although it does not seem to be useful to finish tasks long before their deadlines, the objective function

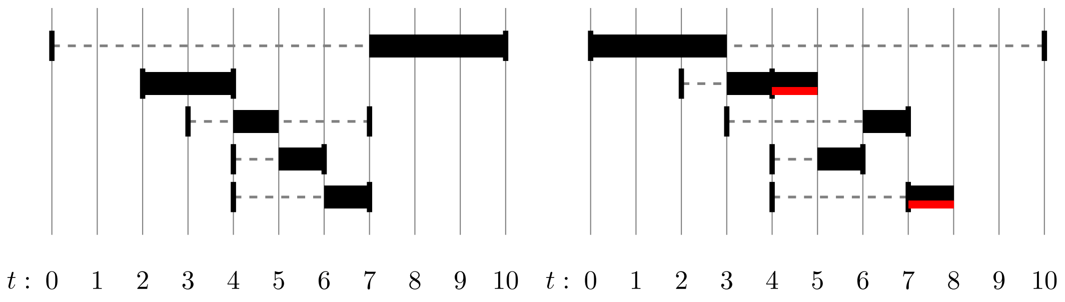


Figure 2.1: Two feasible schedules for the same problem. The thick vertical bars denote release time and deadline; the rectangles denote jobs scheduled at a time interval. The schedule on the left has no late jobs; in the schedule on the right, there are two jobs (marked red) that are late by one unit of time.

$L_{\max}$ (which can be negative) is most often used in the literature for this problem. The more intuitive definition of lateness of a task i, $\max(0, L_i)$, is known as the tardiness T_i. Minimizing $L_{\max}$ or the maximum tardiness $\max(0, L_{\max})$ is a matter of choice: one can always stop the search when a schedule without late jobs is found or one can continue to find a schedule minimizing $L_{\max}$.

This chapter is structured as follows. In Section 2.2, we present a greedy algorithm known as Schrage's heuristic which forms the basis of many of the more complicated algorithms we present later. In Section 2.3, we present two branch and bound algorithms: the one by McMahon and Florian (1975) and the one by Carlier (1982). In Section 2.4, we present algorithms for the restricted case with identical task lengths. The reason that we first discuss branch and bound algorithms is that the algorithms for the case of identical task lengths are much more involved. Finally, in Section 2.5, we present an application of our scheduling problem: the Shifting Bottleneck method for the Job Shop scheduling problem.

2.2 Schrage's heuristic

A greedy algorithm forms the basis of many more complicated algorithms we present later. The dispatching rule used in this greedy algorithm is known as Jackson's rule, the Earliest Due Date (EDD) rule, or Schrage's heuristic. What is most interesting about this rule is not that it works exceptionally well in practice, but that its behaviour can be analyzed mathematically and the kind of "mistakes" it makes can be recovered by a simple procedure. Correcting these mistakes in general requires branching; very roughly, this is what the branch and bound algorithms presented in Section 2.3 do.

The greedy algorithm constructs the schedule forwards in time; iteratively, jobs are added to the back of the schedule at the earliest possible starting time. To describe the dispatching rule used by the greedy algorithm, we say that a task i is *ready* at time t if $r_i \leq t$ and i is not yet scheduled. The rule can then be formulated as follows:

> Schedule the job with the *earliest deadline* of all tasks that are ready at the completion time of the last scheduled task (if no job is ready, wait until the first release time t' of all not-yet-scheduled tasks, and from all jobs with

release time t', schedule the one with the earliest deadline). If multiple jobs have the earliest deadline, any job can be chosen.

The pseudocode for the greedy algorithm is given in Algorithm 1.

Algorithm 1: Schrage's heuristic

```
   Data: $n$ tasks $(r_i, d_i, p_i)$, $i = 1, \ldots, n$.
   Result: schedule $T = T(1), T(2), \ldots, T(n)$.
 1 $TODO \leftarrow \{1, \ldots, n\}$
 2 $t \leftarrow \min_{i=1,\ldots,n} r_i$
 3 for $iteration \leftarrow 1$ to $n$ do
 4     $READY \leftarrow \{i \in TODO \mid r_i \leq t\}$
 5     if $|READY| > 0$ then
 6         $i \leftarrow \arg\min\{d_i \mid i \in READY\}$
 7     else
 8         $t \leftarrow \min\{r_i \mid i \in TODO\}$
 9         $i \leftarrow \arg\min\{d_i \mid r_i = t\}$
10     end
11     $T(i) \leftarrow t$
12     $TODO \leftarrow TODO \setminus \{i\}$
13     $t \leftarrow t + p_i$
14 end
```

This algorithm is optimal if all tasks have length 1, or if release times and deadlines are similarly ordered,[1] but not in general. For example, in Figure 2.1, Schrage's algorithm produces the schedule on the right with two late jobs, while the schedule on the left has no late jobs.[2] The mistake made by the greedy algorithm is that it schedules the job of length 3 at time $t = 0$ which delays the other four jobs too much. More precisely, if we schedule the first job at time $t = 0$, we can start processing the other four jobs no earlier than time $t = 3$, and we can finish the last of these no earlier than time $3 + (2 + 1 + 1 + 1) = 8$, one time unit later than the last deadline of the four jobs (time $t = 7$). This is suboptimal because we could start with job 2 at time $t = 2$ and finish all four jobs at time $t = 7$.

More generally, we have the following lower bound on $L_{\max}$, for any job set J:

$$L_{\max} \geq \min_{i \in J} r_i + \sum_{i \in J} p_i - \max_{i \in J} d_i$$

As it turns out, the greedy algorithm can be mathematically analyzed and a performance guarantee can be given. In order to state the mathematical lemma, we first need to introduce the following notation. For an instance of the scheduling problem $\{(r_i, d_i, p_i) \mid i = 1, \ldots, n\}$, denote by $T = T(1), T(2), \ldots, T(n)$ the schedule generated by Schrage's heuristic, and by $\sigma(1), \sigma(2), \ldots, \sigma(n)$ the sequence of job numbers in the

[1]The release times and deadlines of a set of tasks are *similarly ordered* if the tasks can be ordered such that $r_1 \leq r_2 \leq \ldots \leq r_n$ and $d_1 \leq d_2 \leq \ldots \leq d_n$.

[2]There is a tie between jobs 3 and 5 and time 6 during the execution of Schrage's algorithm, but in both cases the job scheduled last is late.

order in which they are scheduled (so $T(\sigma(1)) < T(\sigma(2)) < \ldots < T(\sigma(n))$. Denote by $\sigma[l,\ldots,r]$ the consecutive subsequence of σ $\{\sigma(i) \mid i = l,\ldots,r\}$. A *chain* of T is a maximal sequence of consecutive jobs without idle time between two successive jobs. We can now state the lemma.

Lemma 2.2 (Performance guarantee for Schrage's heuristic (Potts, 1980; Carlier, 1982)). *Let $\sigma(i)$ be a job with maximum lateness in T (if there is a tie, choose any job with maximum lateness). Let $\sigma(h)$ be the head of the chain of which job $\sigma(i)$ is part. One of the following is true:*

1. *For $J = \sigma[h,\ldots,i]$,*
$$L_{\max} = \min_{j\in J} r_j + \sum_{j\in J} p_j - \max_{j\in J} d_j$$

2. *There exists a job $c = \sigma(j)$ preceding $\sigma(i)$ in its chain such that, for $J = \sigma[j+1,\ldots,i]$,*
$$L_{\max} < p_c + \min_{j\in J} r_j + \sum_{j\in J} p_j - \max_{j\in J} d_j$$

Proof. Because immediately before $\sigma(h)$ no job is scheduled, the greedy algorithm did not have any pending jobs with release times less than $r_{\sigma(h)}$ after job $\sigma(h-1)$ was scheduled (or at the beginning of the algorithm). Therefore, $r_{\sigma(k)} \geq r_{\sigma(h)}$ for $k \geq h$.

If all jobs preceding $\sigma(i)$ in its chain have lower deadlines, the greedy algorithm is optimal. Formally, the execution of $J = \sigma[h,\ldots,i]$ can start no earlier than time $r_{\sigma(h)} = \min_{j\in J} r_j$ and the execution takes at least $\sum_{j\in J} p_j$ units of time, so the job of J scheduled last has lateness at least $\min_{j\in J} r_j + \sum_{j\in J} p_j - \max_{j\in J} d_j$. Because $\sigma[h,\ldots,i]$ are in the same chain and the job scheduled last has the highest deadline, we get

$$L_{\max} = L_{\sigma(i)} = \min_{j\in J} r_j + \sum_{j\in J} p_j - d_{\sigma(i)} = \min_{j\in J} r_j + \sum_{j\in J} p_j - \max_{j\in J} d_j$$

so we have the first case, and the greedy algorithm is optimal, because it meets the lowerbound for job set J.

If some jobs preceding $\sigma(i)$ in its chain have higher deadlines, we may have done work on jobs with later deadlines during the time that we should have made progress with completing all jobs before deadline $d_{\sigma(i)}$. Note, however, that the greedy algorithm would schedule such a less urgent job only when all not-yet-scheduled jobs with later deadlines were not yet available, so the mistake made is that, although no urgent job was available at the time that the job is started, it keeps running for some time when the more urgent jobs become available. Formally, let $\sigma(j)$ be the last job scheduled before $\sigma(i)$ with a later deadline than $\sigma(i)$. At the time $\sigma(j)$ was scheduled, none of the jobs in $\sigma[j+1,\ldots,i]$ was ready, because their deadlines are lower and the greedy algorithm would instead have scheduled one of them. So the set of jobs $J = \sigma[j+1,\ldots,i]$ could have started at time $\min_{k\in J} r_k$ but were instead started only at time $T(\sigma(j)) + p_{\sigma(j)}$, which is less than $p_{\sigma(j)}$ time units later. So we have the second case for this set J and $c = \sigma(j)$.

□

In the second case, job c is called the *interference job*, because it delays the execution of jobs with earlier deadlines. In the schedule on the right in Figure 2.1, the job of length 3 is the interference job. A corollary of this lemma is that the lateness of a schedule produced by the greedy algorithm is always less than $\max_{i=1,\ldots,n} p_i$ units away from the optimal lateness.

2.3 Branch and bound algorithms

Greedy algorithms run quickly and often produce a reasonably good schedule (in the case of Schrage's heuristic, the greedy algorithm also has a performance guarantee), but the schedules produced by these algorithms usually are not optimal. To find a schedule minimizing the maximum lateness $L_{\max}$, we can use branch and bound algorithms. In these algorithms, a search tree is explored, where each path in the search tree corresponds to a sequence of decisions made that further constrain the problem instance. Examples of such decisions are adding a precedence constraint between jobs, or tightening a job's availability interval. For each node in the search tree, a lower bound and an upper bound on the maximum lateness of the modified problem associated with this node are calculated. In our case, the upper bound is simply a feasible (but not necessarily optimal) schedule's maximum lateness. The use of these bounds is that nodes with lower bounds exceeding the least upper bound (in our case, the maximum lateness of the best schedule found so far) can be *pruned*, because we will not find a better solution by further exploring these nodes.

A general purpose approach for planning and scheduling problems is to construct a search tree of partial schedules, with branching corresponding to adding a job to the back of the schedule each time. One of the first branch and bound algorithms for scheduling with release times and deadlines (the algorithm by Baker and Su (1974)) indeed works this way; however, state-of-the-art algorithms use a different branching scheme, based on tightening availability intervals. Nodes in the search tree of these branch and bound algorithms represent a modified scheduling problem in which some availability intervals of the tasks are tightened. We present two algorithms following this approach: the one by McMahon and Florian (1975) and the one by Carlier (1982). The difference between the two branch and bound algorithms lies in their branching schemes. Both calculate lower and upper bounds the same way. The lower bound used is the maximum lateness of an optimal schedule for the *preemptive* version of the scheduling problem. For the preemptive version of the problem, Schrage's heuristic is optimal if we add only one time unit of a job per iteration. This can also be implemented to run in $O(n \log n)$ time regardless of the task lengths. The upper bound used is the maximum lateness of the schedule produced by Schrage's heuristic.

2.3.1 The algorithm of McMahon and Florian (1975)

The branching scheme of McMahon and Florian (1975) is based on a somewhat simpler analysis of Schrage's heuristic than the analysis given in Lemma 2.2. Similarly, we consider a job $\sigma(i)$ with maximum lateness within the greedy schedule $T(1), T(2), \ldots, T(n)$. Let $\sigma(h)$ be the head of the chain of $\sigma(i)$. If all jobs in $\sigma[h, \ldots, i-1]$ in its chain have earlier deadlines, this schedule is optimal; otherwise, some job in $\sigma[h, \ldots, i-1]$ that has a later deadline than job $\sigma(i)$ should be delayed so that $\sigma(i)$ finishes earlier. Indeed, if all jobs in $\sigma[h, \ldots, i-1]$ with later deadlines than $\sigma(i)$ remain scheduled before $\sigma(i)$, then either $\sigma(i)$ or a job with earlier deadline has to finish at time $T(\sigma(i)) + p_{\sigma(i)}$ or later, which does not improve the maximum lateness.

Formally, the set of jobs $G_{\sigma(i)} = \{\sigma(k) \mid h \leq k < i, d_{\sigma(k)} > d_{\sigma(i)}\}$ are the candidate jobs to be delayed. This set of jobs is called the *generating set* of job $\sigma(i)$. The branching rule of McMahon and Florian creates a new branch for each of these candidate jobs $\sigma(k) \in G_{\sigma(i)}$ by delaying the release time of the candidate job: $r_{\sigma(k)} := r_{\sigma(i)}$.

McMahon and Florian conducted experiments with instances distributed as follows. The release times are sampled uniformly at random from $[0,\ldots,r_{\max}]$, the deadlines from $[0,\ldots,d_{\max}]$, and the processing times from $[1,\ldots,p_{\max}]$. Release times, deadlines and processing times are all sampled mutually independently. Note that because the signed maximum lateness is minimized, the difference between release times and deadlines does not matter: the algorithm does not stop when a $L_{\max} = 0$ schedule has been found. The authors were able to solve all instances with up to 50 jobs in under one second, but they do not seem tho have run experiments with larger instances.

There also exists another branching scheme, based on a more complicated analysis of the cause of job i having this lateness (Carlier, 1982). This will be explained next.

2.3.2 The algorithm of Carlier (1982)

Carlier's branching scheme is more directly related to Lemma 2.2 than the one by McMahon and Florian. Again, let $\sigma(i)$ be a job with maximum lateness within the greedy schedule $T(1), T(2), \ldots, T(n)$. Assume there are jobs with later deadlines preceding $\sigma(i)$ in its chain. As in Lemma 2.2, let c be the last job with later deadline preceding $\sigma(i)$. Instead of making a branch for every candidate job that could be scheduled after $\sigma(i)$, Carlier's branching scheme only tightens the availability interval of the *interference job* c. The new observation is that c has to be scheduled either *before* or *after* all jobs in J. This is because if we schedule c in between of jobs in J, the job of J scheduled last will have a completion time of at least $\min_{j\in J} r_j + \sum_{j\in J} p_j + p_c$, which means that the current schedule is better (because, by Lemma 2.2, $L_{\max} < \min_{j\in J} r_j + \sum_{j\in J} p_j + p_c$ for the current schedule). Therefore, we create two branches, corresponding to the interference job c being scheduled before or after J. For the branch with job c to be scheduled *before* J, we bring forward the deadline of c to $d_c := d_{\sigma(i)} - \sum_{j\in J} p_j$. In any feasible schedule with c being placed *before* J, the last job in J has deadline $d \leq d_{\sigma(i)}$ and finishes at least $\sum_{j\in J} p_j$ units after c, so it will have lateness not less than the new value of d_c; this implies that the new deadline of d_c does not change the objective function value for the schedules in which c is indeed placed before J. For the branch with c to be scheduled *after* J, we delay the release time of c to $r_c := \min_{j\in J} r_j + \sum_{j\in J} p_j$, which clearly is a lower bound on the release time in a feasible schedule with c placed after J.

The new deadline and release time can sometimes be improved upon: if c is placed before J, then for all $i \in J$, $d_i - \sum_{j\in J: d_j \leq d_i} p_j$ is an upper bound on the deadline for task c. Similarly, if c is placed after all jobs in J, $r_i + \sum_{j\in J: r_j \geq r_i} p_j$ is a lower bound on the release time for task c, for all $i \in J$.

Although these modified release times and deadlines "suggest" that job c must be placed before or after J, they do not represent a hard constraint. Also, in subsequent branches availability intervals of other jobs can be tightened which may decrease the effect of previously tightened availability intervals. Because of this, the greedy algorithm may violate some precedence constraints that the branching steps represent. Alternatively one could add the precedence constraints explicitly and use a modified greedy algorithm, preemptive greedy algorithm and conflict analysis. We implemented a version remembering the precedence constraints explicitly, but the performance did not improve.

Carlier conducted experiments with instances distributed in the same way as in the experiments of McMahon and Florian (see Section 2.3.1). He was able to solve almost all instances with a number of jobs up to 1000, although he does not report the run times of the algorithm. We compare the two branch and bound algorithms as a part of the experimental analysis in Chapter 4.

2.4 Polynomial time algorithms for the equal-processing-times case

If all processing times are equal, the scheduling problem can be solved in a smarter way. All known algorithms reduce the optimization problem "minimize the maximum lateness" to a sequence of decision problems of the form "decide whether there exists a feasible schedule with maximum lateness $\leq \Delta t$", using binary search on the maximum lateness. Because a schedule with maximum lateness $\leq \Delta t$ exists if and only if a feasible schedule without late jobs exists in the modified problem with Δt added to each task's deadline, the decision version amounts to solving the problem: given a set of jobs, does there exist a feasible schedule without late jobs?

The first polynomial time algorithm for the equal-processing-times case was presented by Simons (1978) with run time $O(n^2 \log n)$. Although it may be the most direct and intuitive approach, the algorithm seems to be quite involved. This opinion is consistent with the literature, because all the papers that cite this paper we found only mention that this case is solvable. A few years later, Carlier (1981) proposed a different solution based on dynamic programming, which is much easier to understand. In the same year, Garey, Johnson, Simons, and Tarjan (1981) published a third algorithm for the same problem. This algorithm is based on finding so-called "forbidden regions". It is easy to understand, although the proof of correctness is not that simple.

Finally, Dürr and Hurand (2011) presented a formulation of the scheduling problem as a Simple Temporal Problem (STP; a formalism for scheduling problems introduced by Dechter et al. (1991)). There also is a more general dynamic programming algorithm of Baptiste (2000) that directly solves the optimization problem for a large class of objective functions, including $L_{\max}$, the weighted number of late jobs $\sum w_i U_i$, and the weighted sum of completion times $\sum w_i C_i$.

We present here three algorithms for the decision version: we skip the algorithm by Simons (1978). We present the algorithm by Garey et al. (1981) in Section 2.4.1, the algorithm by Carlier (1981) in Section 2.4.2, and the Simple Temporal Problem formulation by Dürr and Hurand (2011) in Section 2.4.3.

2.4.1 The forbidden regions algorithm of Garey et al. (1981)

The branch and bound algorithms resolve the conflict extracted by Lemma 2.2 by adjusting release times and deadlines of the jobs. This results in multiple branches that have to be explored separately. In the equal-processing-times case, we can adjust the scheduling problem in a different way such that we do not need to branch: we can make the greedy algorithm "somewhat less greedy" at some point in time by marking an interval as a *forbidden region*.

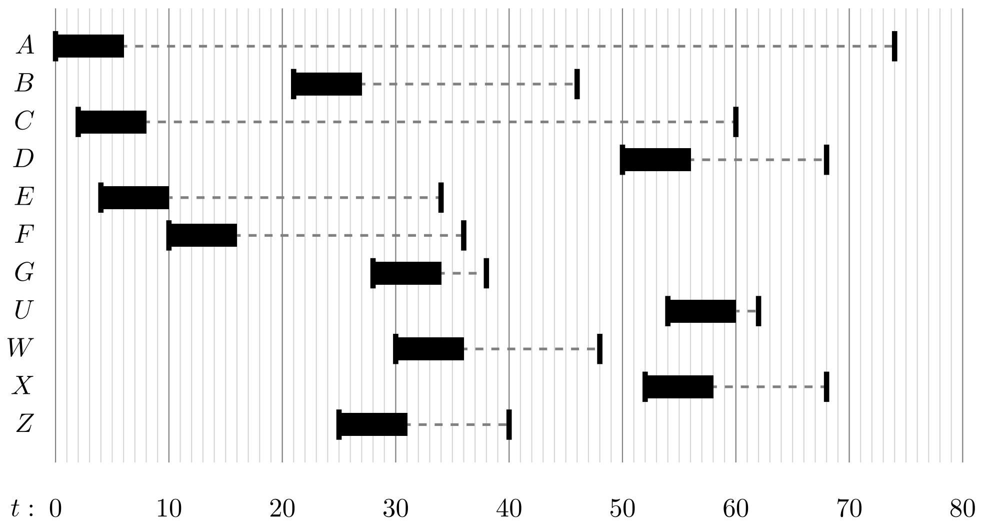


Figure 2.2: The example scheduling problem used in (Garey et al., 1981).

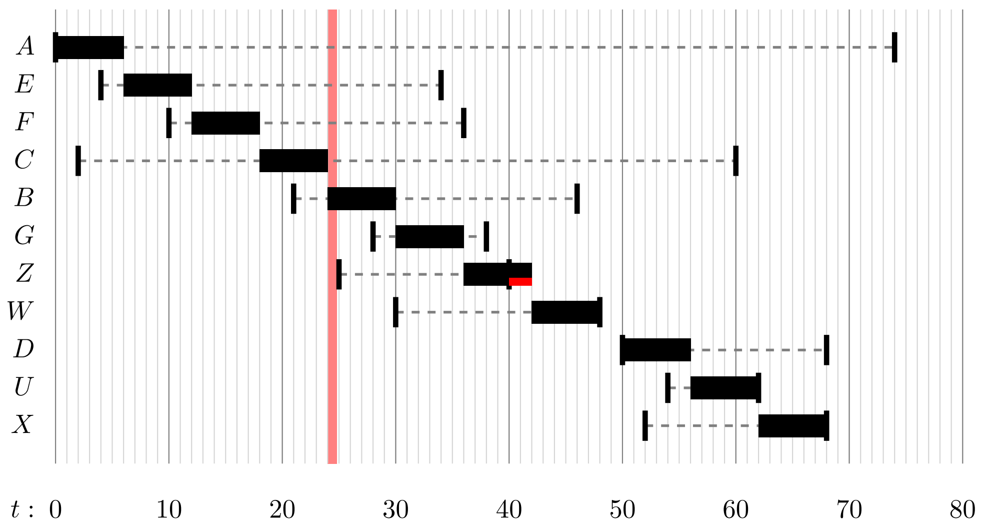


Figure 2.3: The first run of the greedy algorithm. In this run, Job Z is late and the forbidden region $[24, 25)$ (marked) is discovered.

We will now describe the improved conflict analysis. Suppose we have an instance of n tasks $\{(r_i, d_i, p_i) \mid i = 1, \dots, n\}$, where all job lengths are equal: $p_i = p$. Using the same notation as in Lemma 2.2, let $T(1), T(2), \dots, T(n)$ denote the sequence of start times produced by Schrage's heuristic, let $\sigma(1), \sigma(2), \dots, \sigma(n)$ denote the sequence of jobs in the order they are scheduled, and let $\sigma(i)$ be the first job in the sequence

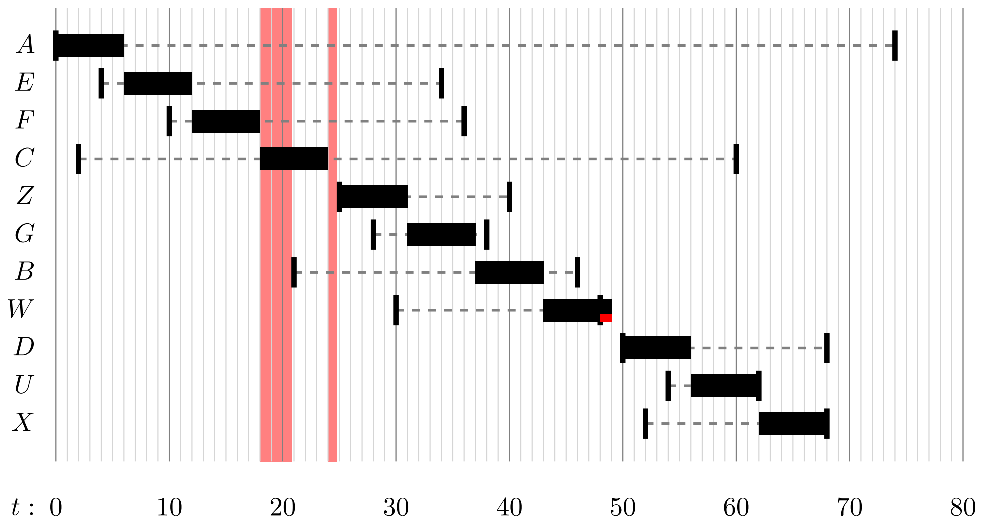


Figure 2.4: The second run of the greedy algorithm. In this run, job W is late and the forbidden region $[18, 21)$ is discovered.

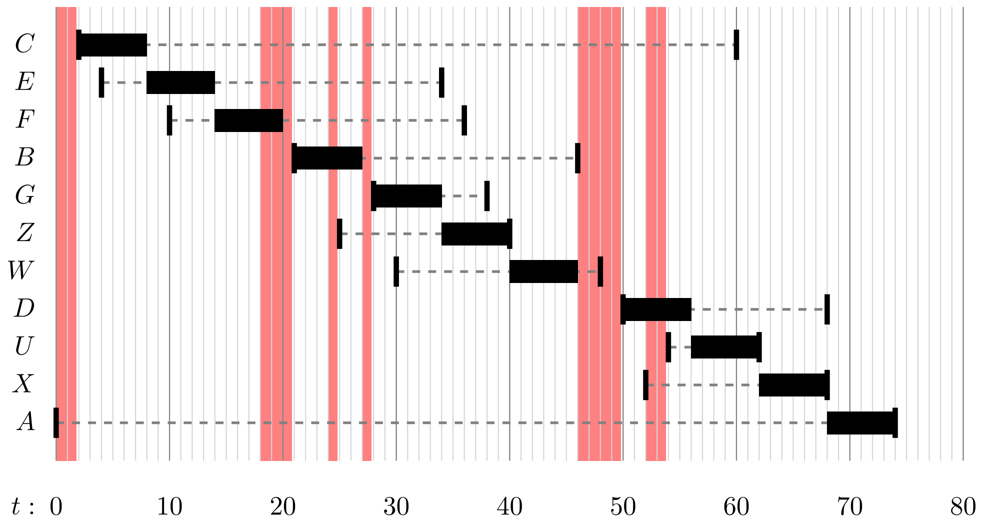


Figure 2.5: The final run of the greedy algorithm. All forbidden regions found are marked.

that is late. If all jobs preceding $\sigma(i)$ in its chain have lower deadlines, no feasible schedule without late jobs exists; otherwise, let $c = \sigma(j)$ be the last job before $\sigma(i)$ with a higher deadline. In a schedule without late jobs, the set of jobs $\sigma[j+1,\ldots,i]$ should start earlier. The crucial observation is that *no job should start at time $T(c)$*. More precisely, no job should start in the interval $[T(c), \min_{k\in\sigma[j+1,\ldots,i]} r_k)$.

We can then run a modified greedy algorithm that differs from the original one only in that it skips the start times in $[T(c), \min_{k\in\sigma[j+1,\ldots,i]} r_k)$. This interval is called a *forbidden region*. Formally, forbidden regions are defined as follows.

Definition 2.3 (Forbidden region)**.** *Let $S = \{(r_i, d_i, p_i) \mid i = 1,\ldots,n\}$ be an instance of* *scheduling with release times and deadlines where all job lengths are equal: $p_i = p$. A* forbidden region *is a time interval $[t,t')$ during which no task can start in any feasible schedule for S.*

Algorithm 2: The modification to Schrage's heuristic that respects forbidden regions.

```
Data: n tasks $(r_i, d_i, p_i)$, $i = 1, \dots, n$; set of forbidden regions
      $\{[t_j, t'_j) \mid j = 1, \dots, k\}$.
Result: schedule $T = T(1), T(2), \dots, T(n)$.
1  $TODO = \{1, \dots, n\}$
2  $t \leftarrow \min_{i=1,\dots,n} r_i$
3  for $iteration \leftarrow 1$ to $n$ do
4      $t \leftarrow \max(t, \min\{r_i \mid i \in TODO\})$
5      Increase $t$ until $t$ is not in a forbidden region $[t_j, t'_j)$ anymore.
6      $S \leftarrow \{i \in TODO \mid r_i \leq t\}$
7      $i \leftarrow \arg\min\{d_i \mid i \in S\}$    /* choose ready task with the earliest
       deadline */
8      $T(i) \leftarrow t$
9      $TODO \leftarrow TODO \setminus \{i\}$
10     $t \leftarrow t + p_i$
11 end
```

The pseudocode of the modified greedy algorithm subject to forbidden regions is given in Algorithm 2. So far, we argued that the first conflict can be represented by introducing a forbidden region. For this approach to work we need to be able to do something similar in the subsequent runs of the modified greedy algorithm. It turns out that this is indeed possible: the modified greedy algorithm that already incorporates several forbidden regions allows for the same analysis. This is true because the modified greedy algorithm still maximizes the throughput, over all possible schedules that respect the forbidden regions found so far. The formal proof of this can be found in Lemma 3 of the original paper by (Garey et al., 1981). The conflict analysis of the modified greedy algorithm is similar, although the set of jobs J that is delayed too much may no longer be scheduled without idle time in between, because of forbidden regions the modified greedy algorithm already knows of. Again, after scheduling a late job for the first time, we go backwards in time until we either find a job with a later deadline that could potentially be postponed, or we determine that the jobs processed so far start execution at the earliest possible time (subject to the forbidden regions found so far). In the first case, we discover a new forbidden region; in the second case, we have detected infeasibility. The pseudocode of this algorithm is given in Algorithm 3. It is not necessary to reschedule the jobs before the interference job, but this gives a longer pseudocode. The execution of the algorithm on a sample problem is displayed in Figures 2.2– 2.5.

Algorithm 3 runs in time polynomial in only n: the first iteration, the greedy algorithm makes chains of jobs starting at some job's release time, so the set of possible starting times is $S_{start} = \{r_i + k \cdot p \mid i = 1, \dots, n; k = 0, \dots, n-1\}$. Forbidden regions always are of the form $[t, r_{\min})$, where $t \in S$ and $r_{\min}$ is a job release time. The modified greedy algorithm also forms chains, starting at job release times and forbidden regions end times. Because the forbidden regions end at job release times, the starting times are still in S. A new forbidden region $[t, r_{\min})$ always invalidates at least one element of S_{start}, so at most n^2 iterations are needed, so the total run time is $O(n^3 \log n)$, because the greedy algorithm can be implemented to run in $O(n \log n)$ time.

It is also possible to find forbidden regions systematically, backwards in time. This is explained in the original paper (Garey et al., 1981). This approach can be implemented straightforwardly in $O(n^2 \log n)$ time and, with a very complicated optimization step (also explained in the original paper), in $O(n \log n)$ time.

Algorithm 3: The forbidden regions algorithm for equal processing times.

```
Data: n tasks $(r_i, d_i, p_i = p)$ with the same length $p_i = p$, $i = 1, \ldots, n$
Result: schedule $T = T(1), T(2), \ldots, T(n)$ or "impossible".
$REGIONS \leftarrow \emptyset$
while true do
    $T(1), T(2), \ldots, T(n) \leftarrow$ SchrageModified($\{(r_i, d_i, p_i) \mid i = 1, \ldots, n\}, REGIONS$)
    if $L_{\max} = 0$ then
        return $T(1), T(2), \ldots, T(n)$
    else
        Let $\sigma(i)$ be the ID of the job scheduled in position $i$, for $i = 1, \ldots, n$
        $i \leftarrow \min\{j \mid L_{\sigma(j)} > 0\}$
        $r_{\min} \leftarrow r_{\sigma(i)}$
        foundregion ← FALSE
        for $j \leftarrow i - 1$ to 1 do
            if $T(\sigma(j)) \leq r_{\min} - p$ then
                return impossible
            end
            if $T(\sigma(j)) < r_{\min}$ then
                $REGIONS \leftarrow REGIONS \cup [T(\sigma(j)), r_{\min})$
                foundregion ← TRUE
                break
            else
                $r_{\min} \leftarrow \min(r_{\min}, r_{\sigma(j)})$
            end
        end
        if foundregion = FALSE then
            return impossible
        end
    end
end
```

2.4.2 The dynamic programming algorithm of Carlier (1981)

Carlier (1981) proposed a dynamic programming algorithm that steps forwards in time. For every deadline t, it computes an optimal partial schedule (a schedule of a subset of the jobs) that can be completed before time t so that the tasks that could have been completed by time t, but now have to be scheduled in $[t, \infty)$, induce the minimal possible load on $[t, \infty)$. The only candidate subsets of jobs for time t are those in which all jobs with deadlines before time t are part of the subset. These schedules are called t-active partial schedules. Formally, they are defined as follows.

Definition 2.4 (t-active partial schedule)**.** *Suppose we have an instance of n tasks $\{(r_i, d_i, p_i) \mid i = 1, \dots, n\}$. Consider a feasible partial schedule $T : S \to \mathbb{R}$ that assigns starting times to a subset of the tasks $S \subseteq \{1, \dots, n\}$ such that none of the jobs in S is late. T is called t-active if*

- *The partial schedule completes no later than time t: $T(i) + p \leq t$ for all $i \in S$.*
- *All jobs with deadlines less than t ($d_i < t$) are scheduled by T.*

Although jobs with deadlines $d < t + p$ clearly also should be included in the partial schedule, it does not seem to be necessary to detect this already at time t. For a t-active partial schedule, we can consider the *load vector*, which captures exactly the load the partial schedule induces on $[t, \infty)$.

Definition 2.5 (Load vector for t-active partial schedules)**.** *Suppose we have an instance of n tasks $\{(r_i, d_i, p_i) \mid i = 1, \dots, n\}$. For a t-active partial schedule that schedules job set S, we define the load vector as the multi set of the "times remaining" for all tasks that have release time $< t$:*

$$\text{load}(S, t) = \{d_i - t \mid i \notin S, r_i < t\}$$

Two example t-active partial schedules with the corresponding load vectors are displayed in Figures 2.6 and 2.7. Load vectors with fewer elements, or with larger elements in it (which means the tasks remain available for longer) are the most flexible. Formally, we can define a partial order $\prec$ on t-active partial schedules as follows.

Definition 2.6 (Dominance criterion for t-active partial schedules)**.** *A t-active partial schedule scheduling job set S dominates another t-active partial schedule scheduling job set S′ if $|\text{load}(S, t)| \leq |\text{load}(S', t)|$ and, if the elements of $\text{load}(S, t)$ and $\text{load}(S', t)$ in increasing order are $t_1 \leq t_2 \leq \ldots \leq t_n$ and $t'_1 \leq t'_2 \leq \ldots \leq t'_{n'}$, $t_i \geq t'_i$ for $i = 1, 2, \dots, n$.*

The approach followed so far is fairly standard. We can calculate the set of t-active partial schedules for each time offset t using dynamic programming: each t-active partial schedule is either one of the $(t-1)$-active partial schedules, or one of the $(t-p)$-active partial schedules with a new job added to the back. The problem with this approach is that, even after using the dominance criterion for pruning, it is not clear how many candidate subsets of jobs remain; there could be exponentially many subsets, which might make the algorithm inefficient. However, as it turns out, one subset of jobs induces a load vector that dominates all others.

It can be proven that either the best $(t-p)$-active partial schedule plus a new job is the best t-active partial schedule, or no job can end at time t and the best $(t-1)$-active partial schedule is also the best t-active partial schedule. If both possibilities do not result in a t-active partial schedule, the instance is unsolvable. What is surprising is that extending the candidate of time $t-p$, if this leads to a t-active partial schedule, is always better than using the candidate of time $t-1$.

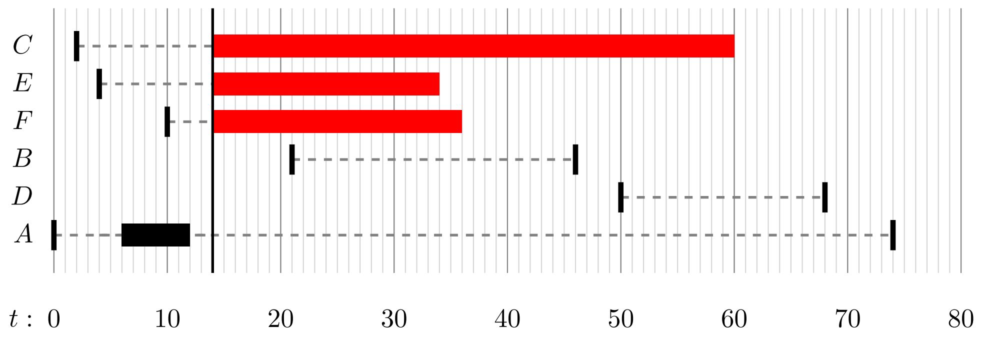


Figure 2.6: A t-active schedule for $t = 14$. The red bars indicate pending jobs.

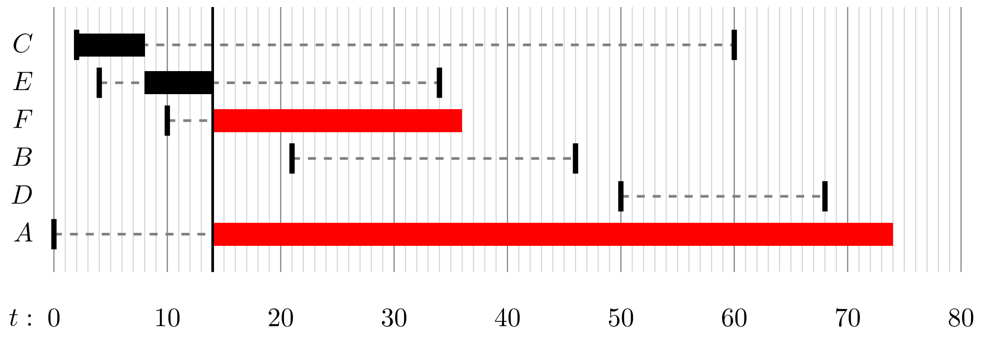


Figure 2.7: Another t-active schedule for $t = 14$. The load induced by this second schedule on the remaining time is lower.

The construction of the best t-active partial schedule is related to forbidden regions as follows: if the candidate from time $t - p$ cannot be extended, then t must be a forbidden *end* time. So, this algorithm finds forbidden regions for the "reversed" scheduling problem with release times and deadlines swapped.

2.4.3 The Simple Temporal Problem formulation

Finally, we present a formulation of the equal-processing-times feasibility problem as a Simple Temporal Problem (STP), a formalism to encode simple scheduling problems introduced by Dechter et al. (1991). An STP instance is a linear program in which all constraints are binary constraints. An STP instance consists of a set of real variables $X_1, X_2, \ldots, X_n$ and a set of *difference constraints* of the form $X_j - X_i \leq w_{i,j}$. The question is to decide whether there exists a solution $x_1, x_2, \ldots, x_n$ that satisfies all constraints. The solution of the STP involves a graph representation: a weighted directed graph that represents the constraints. In terms of this graph, the STP has a solution if and only if the graph does not contain a negative cycle. If no negative cycles exist, the solution can be found by calculating single-source shortest paths. The most well-known algorithm for these tasks is the Bellman-Ford algorithm (Bellman, 1958). A useful property is that, if the bounds $w_{i,j}$ in the constraints are all integers, then the Bellman-Ford algorithm also produces an integer solution.

We will now describe the formulation of our scheduling problem. It is possible to use a number of variables polynomial in n by a technique called discretization; however, we present the (simpler) formulation without discretization, which is polynomial in n and the total time T between the earliest release time and latest deadline. Assume all availability windows are in the range $[0,T]$. We create $T+1$ variables $S_0, S_1, \ldots, S_T$. S_i denotes the number of tasks with starting time in the interval $[0,i)$. The following constraints guarantee that the variables $S_0, S_1, \ldots, S_T$ represent start times of n tasks that execute in $[0,T]$ and do not overlap:

$$S_0 = 0 \tag{2.1}$$

$$S_{t+1} \geq S_t \qquad (t = 0, \ldots, T-1) \tag{2.2}$$

$$S_{t+p} \leq S_t + 1 \qquad (t = 0, \ldots, T-p) \tag{2.3}$$

$$S_T = n \tag{2.4}$$

It remains to find constraints that guarantee that the start times encoded by the variables $S_0, S_1, \ldots, S_T$ can be matched with the n tasks.

As it turns out, from a formulation as a bipartite matching problem of this subproblem we can derive the necessary constraints. Suppose the n starting times are $t_1, t_2, \ldots, t_n$. We construct a bipartite graph $G_{match} = (J \cup T, E)$ with the set of tasks (jobs) $J = \{1, 2, \ldots, n\}$ on one side and the set of start times $T = \{t_1, t_2, \ldots, t_n\}$ on the other side. An edge from task i to start time t_j exists if $r_i \leq t_j \leq d_i - p$. We want to derive a criterion for a perfect matching from tasks to start times to exist in this graph. Hall's marriage theorem for the existence of perfect matchings in a bipartite graph can be used here. Formally, the theorem is as follows.

Theorem 2.7 (Hall's marriage theorem)**.** *Let $G = (A \cup B, E)$ be a bipartite graph with bipartition (A,B). Define the neighbours $N(x)$ of a vertex x as $N(x) = \{y \mid \{x,y\} \in E\}$ and the neighbourhood $N(S)$ of a set of vertices S as $N(S) = \bigcup_{x \in S} N(x)$. A* perfect matching *from A into B exists if and only if $|N(S)| \geq |S|$ for all subsets $S \subseteq A$.*

In our case, the neighborhood of each vertex in J is a "segment" (a consecutive subsequence) of T if we order T in increasing order: $t_1 \leq t_2 \leq \ldots \leq t_n$. For such graphs, only $O(n^2)$ constraints are necessary to encode the existence of a perfect matching: only constraints with $N(S)$ being a segment of $[t_1, t_2, \ldots, t_n]$ are needed. The following theorem states this observation formally.

Theorem 2.8 (Hall's marriage theorem for "segment" neighborhoods)**.** *Let $G = (A \cup B, E)$ be a bipartite graph with bipartition (A,B). Denote the vertices of B by $b_1, b_2, \ldots, b_n$. Assume that the neighborhood of each vertex $x \in A$ is a "segment" of B, that is, $N(x) = \{b_i, b_{i+1}, \ldots, b_j\}$. A* perfect matching *from A into B exists if and only if, for each "segment" $S = \{b_i, b_{i+1}, \ldots, b_j\}$ of B, the number of vertices of A that* must *be matched to a vertex inside S does not exceed $|S|$: $|\{x \in A \mid N(x) \subseteq S\}| \leq |S|$.*

In the case of our graph G_{match}, Theorem 2.8 says that the jobs can be matched with the starting times if for each segment of starting times $t_i, t_{i+1}, \ldots, t_j$, the number of jobs that *must* be matched to one of $t_i, t_{i+1}, \ldots, t_j$, does not exceed $j - i + 1$. These constraints can be encoded as difference constraints: $S_R - S_L$ represent the number of start times in $[L,R)$, and task i *must* start in $[L,R)$ if $[r_i, d_i - p] \subseteq [L,R)$, so:

$$S_R - S_L \geq |\{i \mid [r_i, d_i - p] \subseteq [L,R)\}| \quad (\text{for all } 0 \leq L \leq R \leq T) \tag{2.5}$$

These constraints together represent the scheduling problem as a system of difference constraints. Such systems can be decided quickly by running a shortest path algorithm on the graph representation. In this graph representation vertices correspond to the variables $S_0, S_1, \ldots, S_T$ and edges correspond to the difference constraints: for each difference constraint $S_j - S_i \leq w_{i,j}$ an edge from vertex S_i to vertex S_j with weight $w_{i,j}$ is created. For constraints on one variable instead of two one can create an auxiliary variable representing a "zero" value and adding difference constraints between the variables and this auxiliary variable.

The correctness of this approach and running time guarantee follow from results on the Simple Temporal Problem; this approach however does not give much additional insight in the problem.

2.4.4 Extension to the $\{1, p\}$ case

The constraint system formulation can be extended to allow tasks of length 1 together with the tasks of length p in the model. By Theorem 2.8, a set of unit length jobs with integer release times and deadlines $S_1 = \{(r_i, d_i) \mid i = 1, \ldots, n\}$ can be scheduled without lateness if and only if, for all time intervals $[L, R]$,

$$|\{i \mid L \leq r_i \leq d_i \leq R\}| \leq R - L$$

This can be incorporated in the constraint system formulation of Section 2.4.3 by modeling that the jobs of length p should leave enough empty space within each time window $[L, R]$. However, the formula for the space used by length-p jobs in window $[L, R]$ is not simply equal to $p \cdot (S_{R-p+1} - S_L)$ because some length-p jobs may overlap partially with $[L, R]$. Sgall (2012) observed that we can nevertheless use $p \cdot (S_{R-p+1} - S_L)$ for the occupied space by length-p jobs in $[L, R]$. The added constraints are therefore

$$S_{R-p+1} - S_L \leq \lfloor \frac{R - L - \{i \mid L \leq r_i \leq d_i \leq R\}|}{p} \rfloor \tag{2.6}$$

This formulation cannot be easily generalized to two task lengths larger than one, because the "sufficiency" of the constraints for the feasibility of the schedule with respect to the length-1 jobs only holds because length-1 jobs can be put in all remaining time units (which is not true for larger jobs: for example, if we have two non-consecutive gaps of one time unit, we cannot add a length-2 job to the schedule).

2.4.5 Summary

We have presented three polynomial time algorithms for the equal-processing-times case. The forbidden regions algorithm and Carlier's algorithm are similar, because Carlier's algorithm uses the notion of forbidden regions implicitly in the construction of the "best partial schedule" for each time offset.

The algorithm based on the Simple Temporal Problem (STP) formulation allows for various ways to determine consistency. Forbidden regions can also be discovered by propagation of the STP constraints so, in principle, the reasoning of the forbidden regions algorithm can be simulated by combining STP inequalities. If we use the Bellman-Ford algorithm to solve the STP, forbidden regions are not used. Instead, we initially schedule all jobs immediately after each other starting at the earliest possible release time, and resolve conflicts by increasing the lower bounds on the starting time of the k'th earliest starting length-p task, for each $k = 1,\ldots,n$, until a feasible solution is found or inconsistency is detected.

A very interesting question which we have not been able to answer is: does there exist a generalization of forbidden regions for the $\{1, p\}$ case?

2.5 Application: Job Shop Scheduling

Although the single machine scheduling problem in itself might be too simple to be useful in practice, the problem also occurs as a relaxation of more practical scheduling problems. Solving a relaxation optimally gives a lower bound on the optimal objective function value for the original problem (this is also true if we compute a lower bound on the optimal objective function value for the relaxation). The problem we discuss here is the Job Shop Scheduling problem. The Job Shop problem is a multi-machine scheduling problem in which, for every task, the machine on which it has to be scheduled is fixed. Tasks have precedence constraints between them but they do not have release times and deadlines. A precedence constraint is a directed constraint that specifies that one task must finish before another task can start. The objective function we consider is the *makespan* of a schedule, that is, the difference between the last completion time and the earliest starting time among all jobs in a schedule. The goal is to find a schedule satisfying the precedence constraints with minimum makespan. Formally, a Job Shop problem is defined as follows.

Definition 2.9 (Job Shop Scheduling problem)**.** *An instance of the Job Shop Scheduling problem consists of a set of machines $\{1,2,\ldots,M\}$, a set of jobs $J_1,J_2,\ldots,J_N$ with processing times $p_1,p_2,\ldots,p_N$ and a fixed machine on which they have to be scheduled $m_1,m_2,\ldots,m_N$, and a set of precedence constraints between jobs. The goal is to find a schedule for the jobs satisfying the precedence constraints with minimum makespan.*

There exists a restriction in which the precedence constraints partition the jobs into a set of chains. For this restriction, high quality benchmark are available.[3] We will also use this version in the experimental evaluation in Section 4.5.2. This version is defined as follows.

Definition 2.10 (Chains Job Shop Scheduling problem)**.** *An instance of the Chains Job Shop Scheduling problem consists of a set of machines $\{1,2,\ldots,M\}$ and a set of chains of jobs $S_1,S_2,\ldots,S_N$ (the problem dimensions are written $N \times M$ in the literature). Each chain S_i has length M and consists of an ordered sequence of jobs $J_{i,1},J_{i,2},\ldots,J_{i,M}$. These jobs have a processing time $p_{i,j}$ and a machine $m_{i,j}$ on which they have to be executed. There are precedence constraints $J_{i,1} \to J_{i,2} \to \ldots \to J_{i,M}$ for $i = 1,\ldots,N$. Each chain visits each machine exactly once: $\{m_{i,1},m_{i,2},\ldots,m_{i,M}\} = \{1,2,\ldots,M\}$ for all $i = 1,\ldots,N$. The goal is to find a schedule for the jobs satisfying the chain precedence constraints with minimum makespan.*

[3]See, for example, the benchmark set of Taillard at `http://www.emn.fr/z-auto/clahlou/mdl/Benchmarks.html`.

Job Shop problems are computationally hard to solve optimally. For example, there exist 20×15 benchmarks for which the problem of finding an optimal solution has been unsolved for 20 years.[4]

Exact algorithms for the Job Shop problem have to prove that no schedule exists with a makespan less than the best schedule found. As explained in Section 2.3, in branch and bound algorithms one establishes that no better schedule exists when the lower bounds of all nodes in the queue are not less than the current optimal solution value. Branch and bound algorithms are also used for solving the Job Shop problem exactly. One branch and bound algorithm (Applegate and Cook, 1991) adds precedence constraints in the branching step. The lower bound computed at each node is the relaxation of the Job Shop problem (with added precedence constraints) to a single machine scheduling problem.

The one-machine relaxation for the Job Shop problem is as follows. Denote by j the ID of the machine which the relaxation corresponds to. We can define a single machine problem for machine j with heads and tails. For each job J_i that has to be scheduled on machine j, we create a job with length p_i and a head and a tail. The head and the tail length are derived from the precedence constraints: they represent the maximum length of a chain of jobs that must precede and succeed job i in any schedule, respectively. Head and tail lengths can be computed as longest paths in a directed acyclic graph with edges representing precedence constraints (for the details, see the paper by Adams et al. (1988)). The lower bound is then equal to the minimum makespan schedule of this single machine problem with heads and tails. The heads and tails can be executed in parallel, but the "body" parts must be scheduled sequentially.

The problem with heads and tails is similar to our original problem with release times and deadlines. The lengths of heads directly correspond to release times, and longer tails correspond to earlier deadlines. Formally, if a task's head, body and tail length are a_i, d_i and q_i, respectively, then the problem instance $\{(a_i, d_i, q_i) \mid i = 1, \ldots, n\}$ is equivalent to the following instance of the problem with release times and deadlines:

$$\{(r = a_i, d = -q_i, p = d_i) \mid i = 1, \ldots, n\}$$

Therefore, the single machine scheduling problem with release times and deadlines indeed occurs as a relaxation of the Job Shop Scheduling problem. Next, we discuss the Shifting Bottleneck method, an approximation algorithm for the Job Shop problem based on this single machine relaxation.

2.5.1 The Shifting Bottleneck method

The Shifting Bottleneck method (Adams et al., 1988) is an approximation algorithm for Job Shop Scheduling. A high-level description is as follows. The algorithm iteratively fixes the ordering of jobs for a single machine, which it identified as the *bottleneck* in this iteration. The identification of the bottleneck happens by solving the single machine relaxation of the original problem as defined above. The "bottleneck value" of each machine on which the ordering of jobs is not yet fixed is defined as the minimum makespan of the associated single machine scheduling problem. Each iteration, we fix the order of chains on a machine with highest bottleneck value.

[4] Taillard's benchmark set contains such instances. See `http://optimizizer.com/TA.php`.

The head and tail length are computed by calculating longest paths in a directed acyclic graph, as explained above. When the order of jobs on a machine is fixed, precedence constraints are added to this graph so that heads and tails become longer.

The high-level pseudocode of this method is shown in Algorithm 4. It is also possible to re-optimize already fixed machines after each iteration, but we have not studied this part in detail.

Algorithm 4: The shifting bottleneck method for the Job Shop Scheduling problem

```
Data: Number of chains $N$; number of machines $M$; $N \times M$ problem instance
      $S = \{(m_{i,j}, p_{i,j}) \mid i = 1,\dots,N, j = 1,\dots,M\}$
Result: A possibly suboptimal schedule
1  $TODO = \{1,\dots,M\}$
2  order$[j] \leftarrow \emptyset$ for $j = 1,\dots,M$
3  for $iteration \leftarrow 1$ to $M$ do
4      tmporder$[j] \leftarrow \emptyset$ for $j = 1,\dots,M$
5      bottleneck$[j] \leftarrow \infty$ for $j = 1,\dots,M$
6      foreach $j \in TODO$ do
7          $\{(a_i, d_i, q_i) \mid i = 1,\dots,N\} \leftarrow$ RELAXATION$(j, S, \bigcup_{j' \notin TODO}$ order$[j'])$
8          tmporder$[j] \leftarrow$ SOLVERELAXATION$(\{(a_i, d_i, q_i) \mid i = 1,\dots,N\})$
9          bottleneck$[j] \leftarrow$ MAKESPAN(tmporder$[j], \{(a_i, d_i, q_i) \mid i = 1,\dots,N\})$
10     end
11     $j_{fix} \leftarrow \arg\max\{$bottleneck$[j] \mid j \in TODO\}$
12     order$[j_{fix}] \leftarrow$ tmporder$[j_{fix}]$
13     $TODO \leftarrow TODO \setminus \{j_{fix}\}$
14 end
15 return $\bigcup_{j=1,\dots,M}$ order$[j]$
```

Chapter 3

Own contributions

3.1 Introduction

In this chapter, we present our own contributions for the single machine scheduling problem. This chapter is divided into two parts. In the first part, we present a practical algorithm for the scheduling problem with task lengths $\{1, p\}$ that is based on the Bellman-Ford algorithm for the STP formulation presented in Sections 2.4.3 and 2.4.4. We also present a new lower bound and upper bound for the unrestricted single machine scheduling problem that work by solving a $\{1, p\}$ scheduling problem. The practical algorithm for $\{1, p\}$ is presented in Section 3.2; the new lower bound and upper bound are presented in Section 3.3.

In the second part, we consider the generalization of the $\{1, p\}$ problem to other pairs of integers $\{p, q\}$ and in general fixed sets of processing times. Not much is known about the complexity of this problem: all polynomial time solvability results are "dominated by" the solvability of $\{1, p\}$. Sgall (2012) states that the complexity of the single machine problem for other bounded size sets of task lengths is unknown. The only negative results we are aware of were presented by Simons and Warmuth (1989), but the results are for the multi-machine variant of the scheduling problem. One of their results is the NP-completeness of the multi-machine variant with task lengths $\{1, p\}$ if the number of machines and p are both part of the input. We started by trying to solve special cases and generalizing the algorithms that work in those cases. The results of this part are presented in Section 3.4. We did not get very far in this direction. Because of this, we expected that the problem might be NP-complete. We think that we indeed found a reduction from an NP-complete problem to the scheduling problem for all $p > q > 1$. The proposed proof is presented in Section 3.5.

3.2 A practical algorithm for the $\{1, p\}$ case

Looking at the execution of the Bellman-Ford algorithm on the STP formulation (see Section 2.4.3), we can devise a practical algorithm for the $\{1, p\}$-problem. The version of the Bellman-Ford algorithm we use initially sets each S_t at its highest possible value and executes updates of the form $S_j \leq S_i + w_{i,j}$, This corresponds to each length-p job being scheduled as early as possible initially. During the execution it maintains an upper bound on S_t for $t = 0, \dots, T$ and the constraints are used to tighten the upper bounds. The update $S_t \leq k$ corresponds to the fact “the $(k+1)$’th earliest starting length-p job should start no earlier than time t”. Therefore we can maintain lower bounds on the k’th earliest starting length-p job for $k = 1,\dots,n$. Instead of iterating over all constraints in Equations 2.5 and 2.6, we schedule the length-p jobs greedily (subject to the lower bounds on the k’th earliest starting length-p job found so far), with the length-1 jobs scheduled in the open spaces left by the length-p jobs. When a length-p job is scheduled after its deadline (or a length-1 job remains pending until after its deadline) the conflict is analyzed and a lower bound is tightened (or infeasibility is detected). Next, we discuss these two cases in more detail. We use the notation $T(1),\dots,T(n)$ for the starting times of the n length-p jobs, and $\sigma(1),\dots,\sigma(n)$ for the order in which the length-p jobs are scheduled.

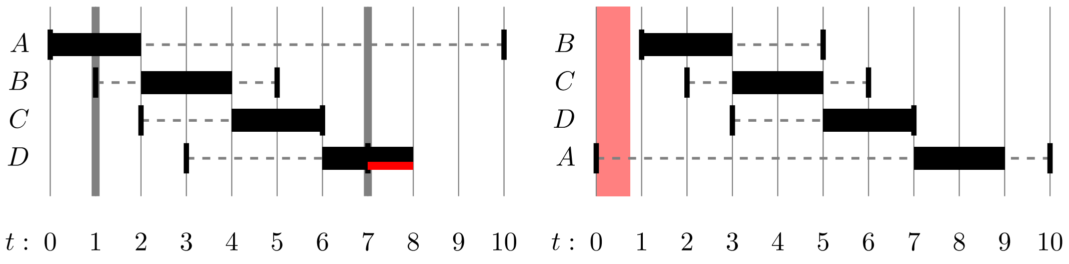


Figure 3.1: Visualization of resolving a late length-P job. The first figure shows the result of running the greedy algorithm. The thick vertical bars at $t = 1$ and $t = 7$ denote the time interval in which more jobs need to be scheduled than what is currently done; this is resolved by delaying the first job (slot) to time $t = 1$. In the second figure, the red rectangle at $[0,1]$ denotes the new lowerbound and the result of running the greedy algorithm subject to this new lowerbound.

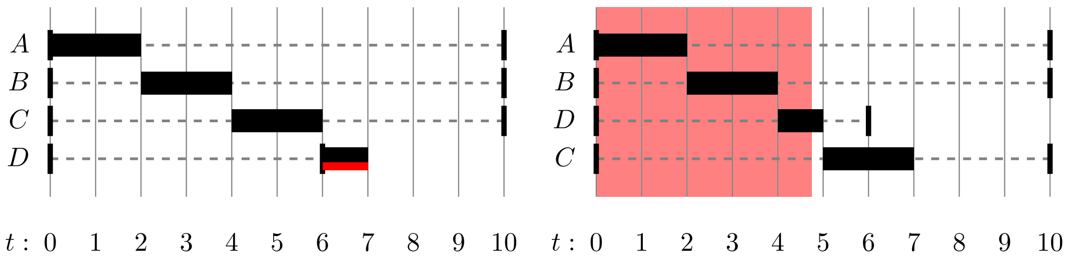


Figure 3.2: Visualization of resolving a late length-1 job. The first figure shows the result of running the greedy algorithm. There are three length-P jobs preceding the late length-1 job, so we update the lower bound of the third length-P job to $t_{at} - P + 1 = 6 - 2 + 1 = 5$ (t_{at} denotes the time the conflict was discovered, which happened after the third length-P job was scheduled at time 6). In the second figure, the red rectangle at $[0,5]$ denotes the new lowerbound for the job at the third position in a feasible schedule and the result of running the greedy algorithm subject to this new lowerbound.

In the first case, a length-p job is scheduled after its deadline. Let $\sigma(i)$ be the first late length-p job. Similarly to the STP algorithm, we want to find a constraint of Equation 2.5 that is violated. We go back in time to jobs $\sigma(i-1),\sigma(i-2),\dots,\sigma(1)$ until we encounter a job $\sigma(j)$ ending before its successors could have started: $T(\sigma(j)) + p \leq r_{\min}$, where $r_{\min} = \min_{k \in \sigma[j+1,\ldots,i]} r_k$. The number of jobs starting in $[r_{\min}, d_{\sigma(i)} - p]$ is one too few. Because the greedy algorithm schedules the job in the k'th position as early as possible (subject to the lower bounds found so far) for all $k = 1, \ldots, n$, the only way to fix the lack of tasks starting in $[r_{\min}, d_{\sigma(i)} - p]$ is to shift the slot of job $\sigma(j)$ to time $t = r_{\min}$. Therefore the the lower bound of position j is increased to $r_{\min}$. If we do not find such a job $\sigma(j)$, then by the EDD rule of the greedy algorithm all jobs $\sigma(1), \ldots, \sigma(i-1)$ have earlier deadlines than $\sigma(i)$ and there are too few slots starting in $[\min_{k \in \sigma[1,\ldots,i]} r_k, d_{\sigma(i)} - p]$, and this cannot be fixed because the slots already pushed to starting times after $t = d_{\sigma(i)} - p$ cannot be used anymore. This operation corresponds to relaxing the constraint of Equation 2.5 for $[L, R) = [\min_{k \in \sigma[1,\ldots,i]} r_k, d_{\sigma(i)} - p + 1)$. The operation is visualized in Figure 3.1.

In the other case, a length-1 job remains in the queue until after the greedy algorithm passes its deadline d. Assuming that the length-1 jobs together have a feasible schedule (this can be determined easily at the start of the algorithm by a greedy algorithm), the problem must be that length-p jobs leave too few space to this job and some other length-1 jobs. Because each length-p slot is scheduled as early as possible subject to the lower bounds, the only solution is to shift the last slot past the deadline d by at least one time unit. Note that we do not have to determine the set of length-1 jobs with corresponding $[r_{\min}, d]$ for which the length-p jobs leave too few space, because we will shift the last length-p job to some time after $d_{\max}$ anyway. If $\sigma(i)$ denotes the last scheduled length-p job, we adjust the lower bound of the i'th starting time to $t - p + 1$, where t denotes the offset at the time the greedy algorithm passed the deadline d. How to find the constraint corresponding to this operation is explained by Sgall (2012). This operation is visualized in Figure 3.2.

The pseudocode is given in Algorithm 5. In the pseudocode, we use zero-based indices everywhere (the lowerbound for the first job's starting time is lowerbound[0]).

Algorithm 5: Practical implementation of a solver for the STP formulation.

```
Data: Length-1 tasks $S_1 = \{(r_i^1, d_i^1) \mid i = 0, \dots, n_1 - 1\}$; length-$P$ tasks
      $S_P = \{(r_i^P, d_i^P) \mid i = 0, \dots, n_P - 1\}$.
Result: A schedule of all tasks, or INFEASIBLE
if no feasible schedule for $S_1$ exists then return INFEASIBLE
$TODO_1 = \{0, \dots, n_1 - 1\}$; $TODO_P = \{0, \dots, n_P - 1\}$
lowerbound$[i] \leftarrow -\infty$ for $i = 0, \dots, n_P - 1$
$t_{at} \leftarrow -\infty$
schedule $\leftarrow$ empty list
while $|TODO_1| + |TODO_P| > 0$ do
    if $TODO_P = \emptyset$ then $t_P \leftarrow \infty$
    else $t_P \leftarrow \min(\text{lowerbound}[n_P - |TODO_P|], \min_{i \in TODO_P} r_i^P)$
    if $TODO_1 = \emptyset$ then $t_1 \leftarrow \infty$
    else $t_1 \leftarrow \min_{i \in TODO_1} r_i^1$
    $t_{at} \leftarrow \max(t_{at}, \min(t_1, t_P))$
    if $TODO_1 \neq \emptyset$ and $\min_{i \in TODO_1} d_i^1 \leq t_{at}$ then
        lowerbound$[n_P - |TODO_P| - 1] \leftarrow t_{at} - P + 1$
        Backtrack until before the last added length-$p$ job
    else
        if $t_P \leq t_{at}$ then
            $i \leftarrow \arg\min\{d_i^P \mid i \in TODO_P, r_i^P \leq t_{at}\}$
            if $t_{at} + P > d_i^P$ then
                $r_{\min} \leftarrow r_i^P$; #seen $\leftarrow 0$
                foreach $P$-entry $J_P(t, j)$ in schedule in reverse order do
                    if $t < r_{\min}$ then
                        lowerbound$[n_P - |TODO_P| - 1 - \#\text{seen}] \leftarrow r_{\min}$
                        Backtrack until before the $P$-job with ID $j$
                    else
                        $r_{\min} \leftarrow \min(r_{\min}, d_j^P)$; increment #seen
                    end
                end
                if #seen $= n_P - |TODO_P|$ then return INFEASIBLE
            else
                schedule $\leftarrow$ concatenate(schedule, $J_P(t = t_{at}, i = i)$)
                $TODO_P \leftarrow TODO_P \setminus \{i\}$
                $t_{at} \leftarrow t_{at} + P$
            end
        else
            $i \leftarrow \arg\min\{d_i^1 \mid i \in TODO_1, r_i^1 \leq t_{at}\}$
            schedule $\leftarrow$ concatenate(schedule, $J_1(t = t_{at}, i = i)$)
            $TODO_1 \leftarrow TODO_1 \setminus \{i\}$
            $t_{at} \leftarrow t_{at} + 1$
        end
    end
end
return schedule
```

3.2.1 Optimizations

We actually implemented only one optimization. This optimization processes the length-1 tasks more efficiently in the case we have many tasks with the same availability interval $[r, d]$. In our experiments, the instances feeded to the $\{1, p\}$-algorithm often have unit-length jobs with the same availability interval (because the $\{1, p\}$-algorithm is called through the half-preemptive lower bound function, which cuts one task into pieces of with the same availability intervals; see Section 3.3).

We can think of K unit length tasks with the same availability interval as a *preemptive* job of length K of which we schedule parts over time. The precise "amount of job" scheduled by the modified Algorithm 5 at time t_{at} (if we decide that we schedule a length-1 job), is the minimum of the total number of units remaining of the job, the next time a job of length P becomes available (or we reach the lowerbound currently delaying the next length-P job) (this is variable t_P in the algorithm), and the first release time of another preemptive job that is not ready yet. As a formula, the number of units taken equals:

$$\min\{\text{unitsremaining}(i), t_P - t_{at}, \min\{r_i^1 \mid r_i^1 > t_{at}\} - t_{at}\}$$

In our instances there may also be many length-p tasks with the same availability interval. However, the optimizations required to handle these length-p tasks as a big task of which we can take multiple pieces of length p at once seem to be much harder to implement; we did not implement this step.

3.3 A new lower bound and upper bound

In this section, we present an application of the $\{1, p\}$ version for solving the general case of scheduling with release times and deadlines. Recall that in Carlier's branch and bound algorithm (see Section 2.3; this is the algorithm of choice to solve the unrestricted single machine scheduling problem), at each node we compute a lower bound on the objective function value $L_{\max}$ of any schedule that will be produced in this subtree. The lower bound Carlier proposed is the preemptive relaxation of the scheduling problem in this subtree. Using the $\{1, p\}$ algorithm, we can calculate an improved lower bound, based on a half-preemptive relaxation. For a fixed integer P, we can relax each non-preemptive task (r_i, d_i, p_i) into $\lfloor p_i/P \rfloor$ non-preemptive blocks of length P, and a preemptive part of length $p_i \mod P$. This way of splitting a task is visualized in Figure 3.3. The resulting scheduling problem can be solved with the $\{1, p\}$ algorithm and the calculated $L_{\max}$ value is a valid lower bound. It is not clear which values of P give the best lower bounds.

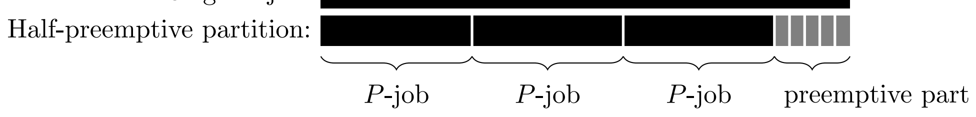


Figure 3.3: The splitting of a task into non-preemptive parts of the same length and a remaining (preemptive) part.

We have also attempted to "round" the solution from this half-preemptive relaxation back to a non-preemptive schedule for the original problem. The general strategy we followed is to extract smaller availability intervals from the relaxation (if possible) and then run Schrage's greedy algorithm on the resulting instance. The length-P tasks' position in an optimal half-preemptive schedule can directly be used as input for Schrage's algorithm. All other tasks however may have parts of it spread over time. We do not know whether something can still be done in this case. We did try a few simple ideas such as restricting the $[r,d]$-interval to the time between the earliest start time and latest finish time of the pieces, but the results actually were worse than just using the original availability intervals for these tasks. Therefore, we create a modified instance in which the length-P jobs have a fixed position. Note that Schrage's algorithm does not respect the fixed positions completely (it may be that some other task starts just before the start of the fixed interval) but because of the performance guarantee (see Section 2.2), the fixed positions are more or less respected.

Formally, let $S = \{(r_i, d_i, p_i) \mid i = 1, \ldots, n\}$ be the original scheduling problem. Let $\Delta t = L_{\max}^{halfpreemptive}$ be the optimal maximum lateness of the halfpreemptive relaxation. For all i with $p_i = P$, let $T(i)$ be the starting time in an optimal schedule of task i. This is well-defined for these i because they are represented by only one block. The modified instance is defined as

$$\{(r_i, d_i + \Delta t, p_i) \mid i = 1, \ldots, n; p_i \neq P\} \cup \{(T(i), T(i) + P, P) \mid i = 1, \ldots, n; p_i = P\}$$

Observe that the optimal halfpreemptive schedule fits within these availability intervals. Schrage's algorithm may introduce additional lateness but it will be less than $\max\{p_i \mid i = 1, \ldots, n; p_i \neq P\}$. This is because the length-P job cannot become the interference job anymore because their availability intervals do not contain any other intervals. An interesting theoretical improvement over Schrage's algorithm is that the above rounding procedure with $P = \max\{p_i \mid i = 1, \ldots, n\}$ gives a schedule with lateness error at most $P - 2$, one unit less than Schrage's algorithm.

In an attempt to make this approach more useful in the case almost all task lengths are different, we tried to round up task lengths in order to be able to represent all tasks with lengths in $[P', P]$ as big non-preemptive blocks. This however did not result in improved performance of the branch and bound algorithm.

3.4 Attempts to find improvements for the case with few different task lengths

The original motivation to consider the problem with a fixed set of task lengths was that the bin packing problem can be reduced in a very simple way to the single machine scheduling problem, and the decision version of bin packing is NP-complete with arbitrary item sizes, but it can be solved by dynamic programming if there is a bounded number of item sizes. In the bin packing problem, we are given n items with sizes $a_1, a_2, \ldots, a_n$, and a number of bins with capacities $c_1, c_2, \ldots, c_{\#\text{bins}}$. The task is to pack the items in the bins such that each bin does not contain items with total size exceeding the capacity of the bin. The reduction given by Pinedo (2008) from bin packing to the single machine scheduling problem represents the bins as time intervals separated by "separator" jobs that have only one possible starting time; for each item, a job with task length equal to the item's size is added with an availability window ranging over all bins. It is clear how feasible schedules for this instance correspond to valid packings of the bins.

For a bounded number K of item sizes, the bin packing problem is known to be solvable in polynomial time using a dynamic programming algorithm. This algorithm calculates all sets of items that fit in the first i bins, iteratively for $i = 1, \ldots, \#\text{bins}$. Because two items of the same size are indistinguishable, there are only $O(n^K)$ distinct sets of items, and this calculation will take at most $O(n^{2K})$ time per bin, which is polynomial time for constant K.

An algorithm for our scheduling problem with a set of task lengths P must therefore be able to solve the bin packing problem with item sizes from P. In the literature search for bin packing algorithms with a bounded number of item sizes, we found two useful results. First, for mutually divisible sets of item sizes, the problem can be solved greedily using the First Fit Decreasing algorithm (Coffman et al., 1987). This algorithm sorts the items in descending order of item size and then puts each item (in this order) in the first bin (the one with the lowest index) that has enough capacity left. So in the case of divisible item sizes, dynamic programming is not necessary. We also found a rather complicated mathematical paper by Goemans and Rothvoß (2013) that solves the special case with identical bin capacities very efficiently. In the case all bin capacities are equal to a constant C, the input now consists of only K integers: the K frequencies for each item size. The algorithm proposed by the authors then works polynomial in the *number of digits* of the input (so polynomial in $\log(n)$). The algorithm involves complex mathematics from convex optimization; we have not studied this algorithm in detail. The authors also discuss our scheduling problem as an application: this scheduling problem can also be solved in time polynomial in $\log(n)$; the catch however is that all jobs of the same length also need to have the same release time and deadline (so in this restriction there are only K different release times and deadlines, too).

3.4.1 Generalizations of Carlier's dynamic programming algorithm

After the literature search we conclude that in the general case of non-divisibile item sizes, the dynamic programming algorithm is the only known exact solution, and an algorithm for the scheduling problem should incorporate it. We tried to modify the dynamic programming algorithm for identical task lengths of Carlier (1981) (see Section 2.4.2) to handle multiple task lengths. This algorithm calculates a representation of the subsets of tasks that can be scheduled in the time window $[r_{\min}, t]$ iteratively for $t = r_{\min}, r_{\min} + 1, \ldots, d_{\max}$ (where $r_{\min}$ and $d_{\max}$ denote the earliest release time and the last deadline of all tasks). For this approach to work efficiently, we need a compact representation of the set of subsets of tasks that can be scheduled in $[r_{\min}, t]$. Exponentially many subsets can be possible, but we need to remember only a set of subsets that always contains a candidate that extends to a feasible schedule, if a feasible schedule exists. If such subsets still consist of exponentially many candidates, there might exist a representation which describes the complete set of possibly optimal subsets of tasks without needing to store each such subset. The original algorithm for identical task lengths relies on the observation that only one subset of tasks has to be stored for each time offset t. The main question is whether something similar can be done for multiple task lengths.

The following lemma can be used to reduce the number of states the dynamic programming algorithm needs to remember.

Lemma 3.1. *If for two tasks (r_i, d_i, p_i) and (r_j, d_j, p_j) the release times and deadlines are similarly ordered ($r_i < r_j$, $d_i < d_j$) and the task lengths are identical ($p_i = p_j$), then any feasible schedule without lateness $T = T(1), T(2), \ldots, T(n)$ in which $T(i) > T(j)$ can be modified such that $T(i) < T(j)$ without introducing lateness.*

Proof. Because $T(i) > T(j)$ both tasks are scheduled in $[r_j, d_i]$, which is still the case if the tasks are swapped. If we swap the tasks, no lateness is introduced and all other tasks remain at their original time interval. □

If a set of equal-length tasks $\{i_1, i_2, \ldots, i_k\}$ have similarly ordered release times and deadlines ($p_{i_1} = p_{i_2} = \ldots = p_{i_k}$, $r_{i_1} \le r_{i_2} \le \ldots \le r_{i_k}$ and $d_{i_1} \le d_{i_2} \le \ldots \le d_{i_k}$), then by Lemma 3.1 for this set of tasks the subsets the dynamic programming algorithm needs to consider can be restricted to $\{\{i_1, i_2, \ldots, i_j\} \mid j = 0, \ldots, k\}$. Therefore, *if the $[r, d]$ intervals are similarly ordered for the tasks for each task length, the state space has size $O(n^K)$*, which is polynomial in n for constant K. More generally if we can partition the tasks in a bounded number W of sets of tasks with similarly ordered intervals for each task length, the state space has size $O(n^{K \cdot W})$ which is polynomial in n for constant K and W. However, this does not extend to an efficient algorithm for the general case, because there can be heavily nested availability intervals. For example, if $r_1 < r_2 < \ldots < r_n$ and $d_n < d_{n-1} < \ldots < d_1$, no two tasks have similarly ordered intervals so there are still 2^n valid states. So it is clear that a hardness proof, if it exists, must contain heavily nested availability intervals.

3.4.2 Special cases with heavily nested availability intervals

Next, we considered special cases with heavily nested availability intervals. The simplest case is that of perfectly nested intervals: $r_1 < r_2 < \ldots < r_n$ and $d_n < d_{n-1} < \ldots < d_1$. This case however can be solved by dynamic programming, because of the following observation:

Lemma 3.2. *Let $T = T(1), T(2), \ldots, T(n)$ be a feasible schedule without lateness. Assume tasks $1, 2, \ldots, i$ ($i < n$) are scheduled directly after each other in some order (the time between the start of the first task of $\{1, 2, \ldots, i\}$ and the end of the last one is exactly $\sum_{j=1,\ldots,i} p_j$). It is optimal to schedule task $i+1$ either immediately before or immediately after all tasks in $\{1, 2, \ldots, i\}$.*

Proof. Without loss of generality, we can assume that there is no idleness at all between the task scheduled first and the task scheduled last in T, because all availability intervals contain the current time interval occupied by job 1 (so moving jobs closer to job 1 never introduces lateness). We can move task $i+1$ towards the block of tasks $\{1, 2, \ldots, i\}$ by iteratively swapping it with all tasks in between the block $\{1, 2, \ldots, i\}$ and task $i+1$. After each swap, the swapped task j is still scheduled within its availability interval, because it is scheduled in the availability interval of task $i+1$, and $j > i+1$ so $[r_j, d_j]$ contains $[r_{i+1}, d_{i+1}]$. □

So, using dynamic programming we can calculate all offsets at which the set of tasks $\{1,2,\ldots,i\}$ can start, and solve the problem in polynomial time. Formally, the run time of this algorithm is as follows. Because we can always shift the schedule such that at least one task starts at its release time, the start times for the innermost task we need to consider are of the form $r_i \pm \sum_{j\in S} p_j$ for some task i and some subset of tasks S. So if we denote by $W = \sum_{i=1,\ldots,n} p_i$ the total length of all tasks, there are at most $n \cdot \min(W, n^K)$ start times to be considered. For a fixed starting time of task 1, the dynamic programming algorithm runs in time $O(W)$ or $O(n^K)$ per iteration (adding a job), which is done $n-1$ times. Therefore, the total run time is $O(\min(n^2 \cdot W^2, n^{2K+2}))$.

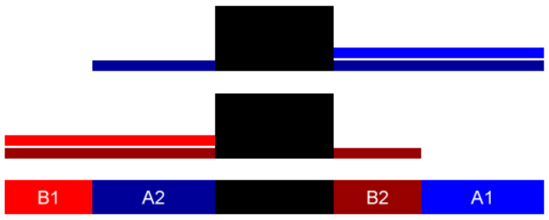


Figure 3.4: A counterexample for the dynamic programming algorithm with multiple stacks: the jobs with the largest availability intervals in both stacks (A_2 and B_2) must be placed closest to the center.

With this case being solvable, we considered the generalization to the case of *two* perfect nested sequences ("two stacks"). The straightforward generalization of the dynamic programming algorithm for one stack is to extend the block of jobs by placing one of the innermost jobs immediately to the front or to the back. This however does not work. Consider the instance in Figure 3.4. There are four tasks, A_1, A_2, B_1 and B_2, and another "separator" task occupying a fixed time block. The tasks A_1 and A_2 have length p_A, and the tasks B_1 and B_2 have length $p_B < p_A$. The availability intervals are chosen such that a feasible schedule must have an A-job and a B-job on both sides. It is then easy to see that the only valid schedule has the two "outermost" jobs (A_2 and B_2) closest to the center, and the two innermost jobs at the boundaries.

For the special case with all length-p_A tasks belonging to one stack and all length-p_B tasks belonging to the other stack, there actually exists a polynomial time solution because this case can be formulated as a Monotone Two Variables Per Inequality constraint system (a linear program in which all constraints are of the form $a \cdot X_i - b \cdot X_j \leq w$ with $a, b > 0$), and finding an integer solution to such systems can be done in pseudo-polynomial time (Hochbaum and Naor, 1994). However, this solution does not seem to generalize to more than two stacks.

3.5 NP-completeness of the scheduling problem with two non-unit task lengths

For a pair of positive integers p, q, we define the problem $\{p,q\}$-SCHEDULING as the following decision problem: given a set of tasks with integer release times and deadlines and *task lengths from the set* $\{p,q\}$, decide whether a non-preemptive schedule without late jobs (a schedule with $L_{\max} \leq 0$) exists. It is known that $\{1,p\}$-

SCHEDULING can be solved in polynomial time, for all integer task lengths p. In this section, we prove that all other cases are NP-complete. Formally, we have the following result.

Theorem 3.3. *For every two integers $p > q > 1$, $\{p,q\}$-SCHEDULING is NP-complete.*

We present a reduction from the Boolean Satisfiability (SAT) problem. The SAT problem is well-known to be NP-complete. A SAT instance consists of a set of boolean variables $x_1, x_2, \dots, x_n$, and a set of clauses $C_1, C_2, \dots, C_m$ over these variables. Each clause C_j is of the form $l_1 \vee l_2 \vee \dots \vee l_{|C_j|}$, where each literal l_k is a variable x_i or its negation $\neg x_i$. The problem is to decide whether there exists an assignment of truth values (TRUE or FALSE) to the variables such that each clause is satisfied, that is, that each clause contains a literal that evaluates to TRUE in this assignment.

The high-level idea is to build two large stacks of jobs that are centered around $t = 0$, one for length-p jobs and one for length-q jobs. We add jobs occupying fixed positions to the time before $t = 0$ that leave empty spaces of length $p + q$ in between; this restricts the combinations of p-jobs and q-jobs that can be scheduled before $t = 0$. In the time after $t = 0$, we add jobs with small availability intervals that impose additional restrictions on the set of jobs that can be scheduled there.

3.5.1 Presentation of the construction

We construct the corresponding instance of $\{p,q\}$-scheduling as follows. Before time $t = 0$, there are *#bins* bins of size $p + q$ that are separated by so-called “separator” jobs that have only one possible position in a feasible schedule. We use separators of size q and number the bins $i = 1, \dots, \#bins$ from $t = 0$ backwards in time. Bin i therefore has time window $[t_i^G, t_i^G + p + q]$, where $t_i^G = -i \cdot (p + q + q)$. With each bin we associate 4 jobs, two p-jobs and two q-jobs. These jobs are denoted p_i^{INNER}, p_i^{OUTER}, q_i^{INNER}, q_i^{OUTER}. The outer jobs have release time $r(p_i^{\text{OUTER}}) = r(q_i^{\text{OUTER}}) = t_i^G$ and the inner jobs have release time $r(p_i^{\text{INNER}}) = r(q_i^{\text{INNER}}) = t_i^G + 1$. The deadlines of these jobs are such that availability intervals for each task length form a perfect nested sequence, and the p-jobs are relatively urgent:

$$d(p_i^{\text{OUTER}}) \geq d(p_i^{\text{INNER}}) > 0 \qquad \text{for } i = 1, \dots, \#bins \tag{3.1}$$
$$d(q_i^{\text{OUTER}}) \geq d(q_i^{\text{INNER}}) > 0 \qquad \text{for } i = 1, \dots, \#bins \tag{3.2}$$
$$d(p_i^{\text{INNER}}) \geq d(p_{i-1}^{\text{OUTER}}) \qquad \text{for } i = 2, \dots, \#bins \tag{3.3}$$
$$d(q_i^{\text{INNER}}) \geq d(q_{i-1}^{\text{OUTER}}) \qquad \text{for } i = 2, \dots, \#bins \tag{3.4}$$
$$d(q_i^{\text{INNER}}) \geq d(p_i^{\text{OUTER}}) \qquad \text{for } i = 1, \dots, \#bins \tag{3.5}$$

In other words, if we draw a grid with the p-jobs in the top row and the q-jobs in the next row (ordered by increasing release time within a row), then going up or right increases urgency of the task. The purpose of these conditions 3.1–3.5 is the following.

Lemma 3.4. *For each $i = 1, \dots, \#bins$, it is optimal to place either $\{p_i^{\text{OUTER}}, q_i^{\text{INNER}}\}$ or $\{q_i^{\text{OUTER}}, p_i^{\text{INNER}}\}$ in bin i.*

Proof. We present an exchange argument that fills the bins as described in the lemma, in the order $i = 1, \dots, \#bins$. Formally, we prove by induction that it is optimal to fill bins $j = 1, \dots, i$ this way, for $i = 1, \dots, \#bins$. The base case $i = 0$ (for 0 bins) is clearly true; for the induction step ($i-1 \to i$), assume we have a feasible schedule with bins $j = 1, \dots, i-1$ filled as described in the lemma. First, suppose bin i does not contain a p-job. Because of Equations 3.3, 3.4, 3.5, p_i^{OUTER} is more urgent than all q-jobs in J, so we can exchange p_i^{OUTER} with a number of q-jobs currently scheduled in bin i. If we cannot simply add p_i^{OUTER} to the bin, the bin has $N_q \leq 1 + \lfloor p/q \rfloor$ length-q jobs; we choose $N_q - 1$ of these and swap them with job p_i^{OUTER}. This works, because the p-job can be placed at the end of the bin together with at most one length-q job, and the $N_q - 1$ jobs use space $q \cdot (N_q - 1) \leq q \cdot \lfloor p/q \rfloor \leq p$ so they fit in the original position of the p-job. This does not change the contents of gaps $1, \dots, i-1$, because these bins are already completely filled with jobs from their own bins. Therefore, it is optimal to place one p-job in bin i. We can then place q_i^{OUTER} in bin i if there is no q-job in bin i. It follows from the stack properties (Equations 3.1–3.4) that it is optimal to choose a p-job and a q-job belonging to bin i, and because the inner jobs have earlier deadlines than the outer jobs but cannot be placed together in the bin, it is optimal to choose exactly one inner job and one outer job. This completes the induction step. □

On the time space after $t = 0$, we add groups of jobs and separator jobs directly after each other. This means that we maintain a time offset, representing the space occupied so far. This offset equals the last deadline of the jobs in the last group (and $t = 0$ for the first group), except when the last group was a "delayed literal" group; then, the time offset is one less than the last deadline. The groups are added as follows.

Algorithm 6: Construction of the part after $t = 0$

```
1  Add $S$-groups followed by separators $n$ times;
2  **foreach** *clause* $C_j$ **do**
3      Add $2n$ times an "ordinary literal" group;
4      Add a CL-$p$-group followed by a separator;
5      **foreach** *literal* $l \in C_j$ **do**
6          Add "ordinary literal" groups for $l' = 1, \dots, l-1$;
7          Add a "special literal" group for literal $l$;
8          Add "delayed literal" groups for $l' = l+1, \dots, 2n$;
9          Add a CL-$p$-$q$-group followed by a separator;
10     **end**
11     Add $2n$ times an "ordinary literal" group;
12     Add a CL-$q$-group followed by a separator;
13 **end**
14 Add $T$-groups followed by separators $n$ times;
```

Lemma 3.5. *In a feasible schedule that is modified according to Lemma 3.4, each decision job that is scheduled after $t = 0$ is scheduled between the two separators between which the job's deadline falls.*

Proof. Let J_{chosen} be the set of decision jobs scheduled after $t = 0$; by Lemma 3.4, J_{chosen} contains exactly one p-job and one q-job from each bin. Let J_{dummy} be the set of all dummy jobs (two from each S-box, two from each T-box, and one from each literal-in-clause box). Observe that the space $[t_L, t_R]$ between two consecutive separators exceeds $\sum\{p_i \mid (r_i, d_i, p_i) \in J_{chosen} \cup J_{dummy}, t_L \leq d_i \leq t_R\}$ by at most one, because the only groups that occupy more space than their own need are S groups, T groups and "special literal" groups, and there is never more than one such group between two consecutive separators. Because all task lengths are ≥ 2, jobs with later deadlines cannot be added. □

Lemma 3.6. *It is optimal to schedule each decision job either before $t = 0$ in its own bin, or after $t = 0$, between the two separators between which the job's deadline falls.*

Proof. Follows from Lemma 3.4 and Lemma 3.5. □

The next step is to argue that the polarities of a variable are consistent. Variable x_i ($1 \leq i \leq n$) occurs in the i'th S-group, the i'th T-group, and at positions $2i-1$ and $2i$ in the list of literal groups for each clause C_j, $|C_j|$ times. The positive literal x_i is associated with the most urgent jobs: the first two jobs in the i'th S-group, the first two jobs in the i'th T-group, and the four jobs in group at position $2i$ in the list of groups for each clause. The negative literal $\neg x_i$ is associated with the other jobs of variable x_i: the last two jobs in the i'th S-group, the last two jobs in the i'th T-group, and the four jobs in group at position $2i+1$ in the list of groups for each clause. We say that a literal is assigned TRUE within a group if the outermost p-job corresponding to it is placed right, and we say that a literal is assigned FALSE within a group if the outermost q-job corresponding to it is placed right. The following lemma states that the variables are represented consistently.

Lemma 3.7. *For each variable x_i, either all positive occurences are assigned* TRUE *and all negative occurences are assigned* FALSE*, or all positive occurences are assigned* FALSE *and all negative occurences are assigned* TRUE.

Proof. By inspection, at least one of the outermost jobs in S_i must be placed right. Suppose we schedule the first outermost p-job right. Let $G_1^+, G_2^+, \ldots, G_k^+$ be the ordered list of groups inside the clause blocks associated with literal x_i, and let $G_1^-, G_2^-, \ldots, G_k^-$ be this ordered list for $\neg x_i$. Observe that $(S_i, G_1^+), (G_1^+, G_2^+), \ldots, (G_{k-1}^+, G_k^+), (G_k^+, T_i)$ are linked through the bins that have the p-jobs in one group and the q-jobs in the next. Because we chose the outermost p-job in the S_i group, we have to choose the innermost q-job in G_1^+, and therefore also the outermost p-job in G_1^+. Finally, we have the outermost p-job chosen in G_k^+, and therefore the innermost q-job must be chosen in T_i. If we choose the innermost q-job for x_i in group T_i, we must choose the outermost q-job for $\neg x_i$ in T_i. Then we proceed backwards through the G^--list, from G_k^- to G_1^-, and finally to S_i where the innermost p-job for $\neg x_i$ must be chosen.

Otherwise, the second outermost p-job of S_i must be scheduled right. The same argument works but now we go forward along G^- and backward along G^+. □

Finally we argue that, in a feasible schedule, all clauses are satisfied. Observe that in a clause C_j, the groups CL-p, CL-p-q[1], CL-p-q[2], ..., CL-p-q[$|C_j|$], CL-q are linked through bins having a p-job in one group and a q-job in another. If no literal satisfies C_j, then each CL-p-q group demands that at least one job in its group placed right is an outermost job, because the literal not satisfying the clause introduces an empty space, which delays the CL-p-q group. The CL-p group and the CL-q group also demand the outermost job placed right, which amounts to a total need of $|C_j| + 2$ outermost jobs placed right while there are only $|C_j| + 1$ bins, which is impossible.

If instead literal l satisfies C_j, then the “special group” for literal l does not introduce an empty space, and it does not require an outermost job placed right; so, the $|C_j| + 1$ bins can help the CL-p group, the CL-q group and the $|C_j| - 1$ CL-p-q groups of the other literals of C_j.

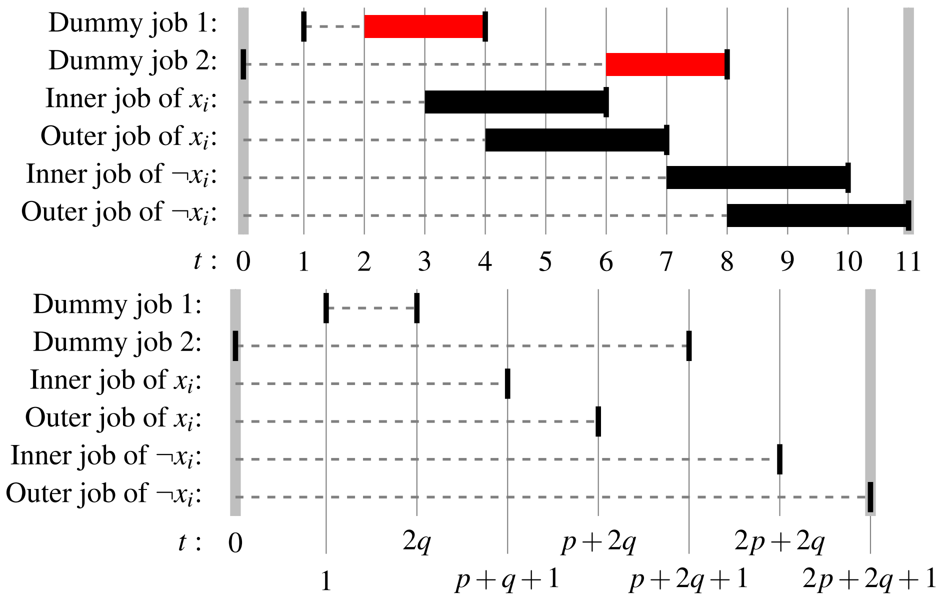


Figure 3.5: Layout of an S-group (for $p = 3, q = 2$ and the general case). Note that it is impossible to schedule both inner jobs, but both combinations of one inner job and one outer job are possible.

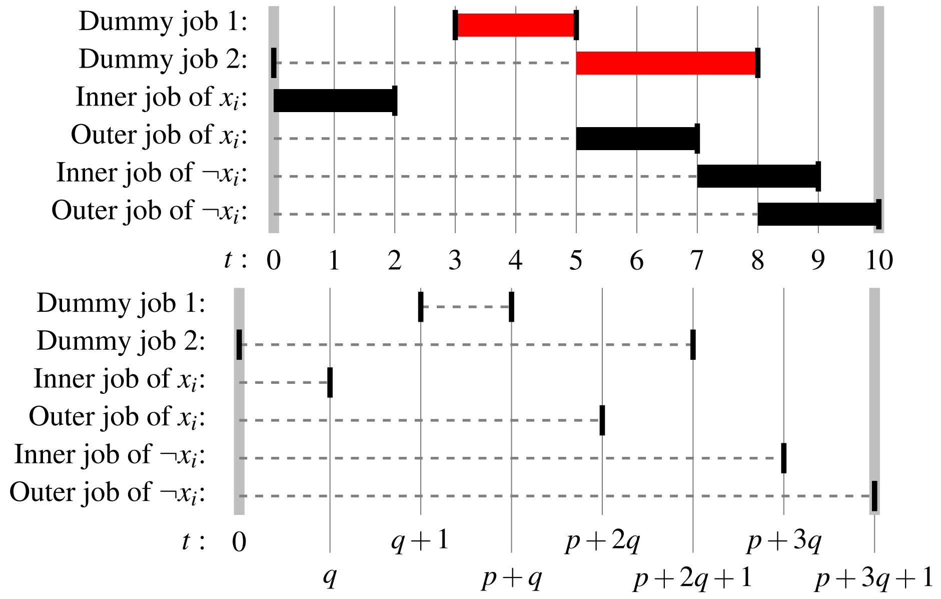


Figure 3.6: Layout of a T-group (for $p = 3, q = 2$ and the general case). Note that it is impossible to schedule both inner jobs, but both combinations of one inner job and one outer job are possible.

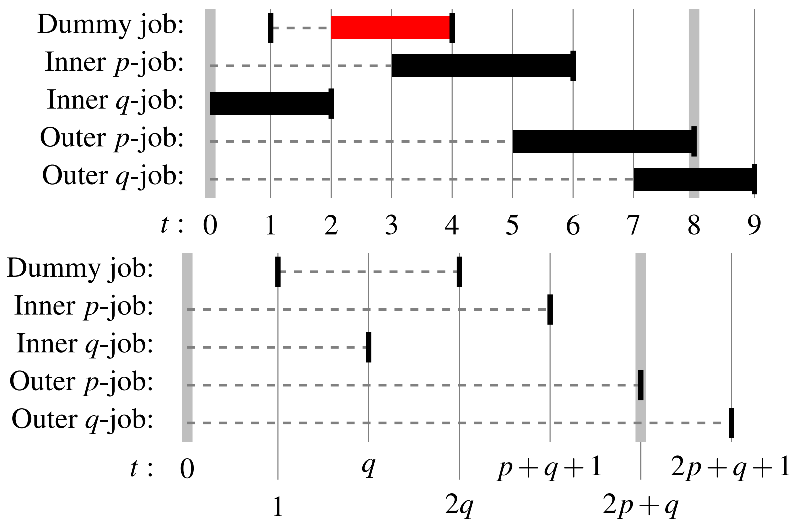


Figure 3.7: Layout of a “special literal” group (for $p = 3, q = 2$ and the general case). Note that it is impossible to schedule both inner jobs, but both combinations of one inner job and one outer job are possible. Note also that the group’s allocated space ends before the last deadline.

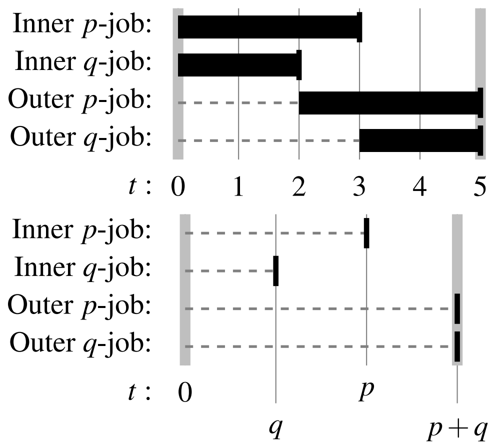


Figure 3.8: Layout of an “ordinary literal” group (for $p = 3, q = 2$ and the general case). Note that it is impossible to schedule both inner jobs, but both combinations of one inner job and one outer job are possible.

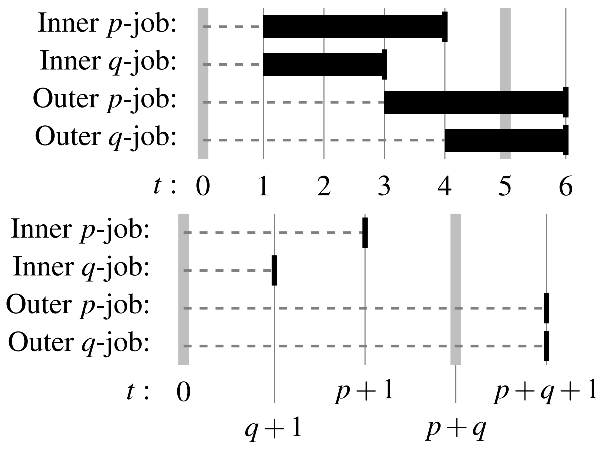


Figure 3.9: Layout of a "delayed literal" group (for $p = 3, q = 2$ and the general case). Note that it is impossible to schedule both inner jobs, but both combinations of one inner job and one outer job are possible. Note also that the group's allocated space ends before the last deadline.

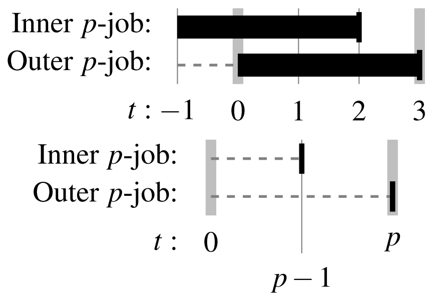


Figure 3.10: Layout of a CL-p-group (for $p = 3, q = 2$ and the general case). We must always choose the outermost job here (this group could be removed but then clauses of size 1 become a special case).

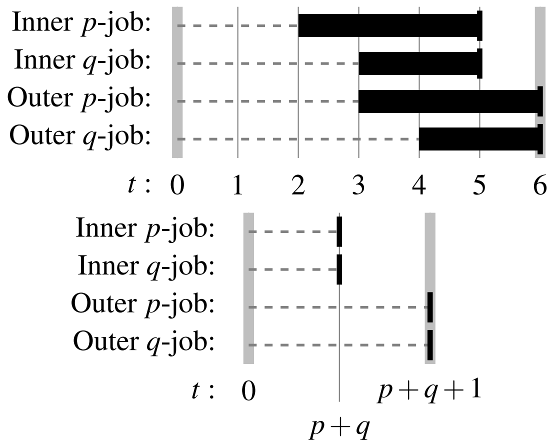


Figure 3.11: Layout of a CL-p-q-group (for $p = 3, q = 2$ and the general case). If the literals before this group are not delayed, we can choose both inner jobs here so that the outer jobs can increase flexibility elsewhere. If the literals before are delayed, then we must choose one outermost job here.

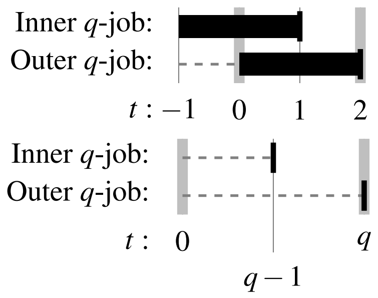


Figure 3.12: Layout of a CL-q-group (for $p = 3, q = 2$ and the general case). We must always choose the outermost job here (this group could be removed but then clauses of size 1 become a special case).

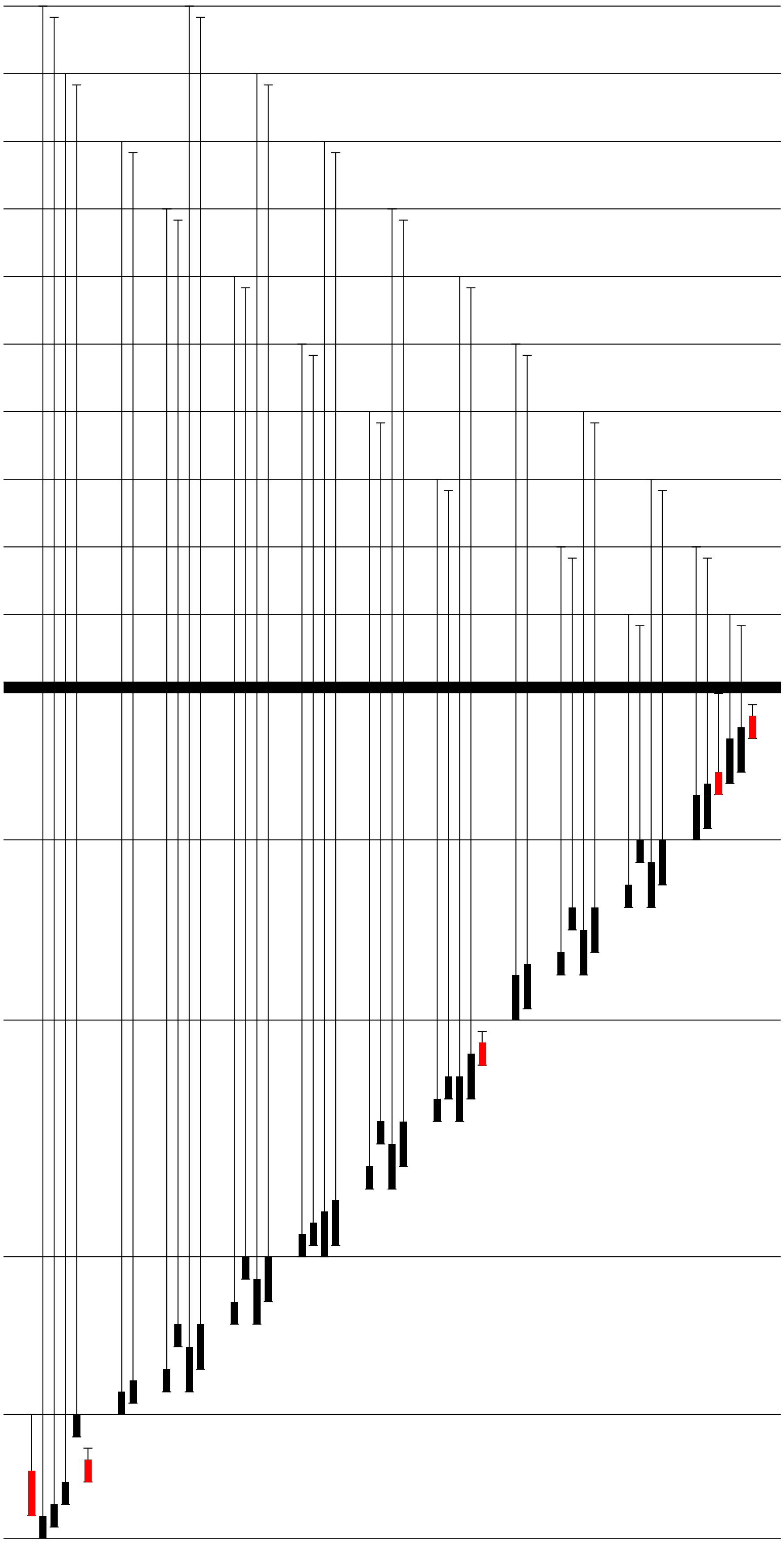


Figure 3.13: Instance for formula with 1 clause "(x_1)".

Chapter 4

Empirical evaluation of the branch-and-bound algorithm

4.1 Introduction and test setup

In this part we test the practical hardness of the single machine scheduling problem. We implemented both branching schemes presented in Section 2.3: Carlier's algorithm and McMahon and Florian's algorithm. We also implemented the original preemptive lower bound function and the half-preemptive lower bound presented in Section 3.3. We implemented the algorithm for the $\{1,p\}$ problem as described in Section 3.2, with the optimization that multiple unit length tasks with the same availability interval are treated as one preemptive job.

We implemented the branch and bound algorithm and our algorithm for $\{1,p\}$-scheduling in C++ and compiled the code with `g++` using the `-O3` optimization flag. We ran all tests on Ubuntu 12.04 using an Intel Core 2 Duo T9600 2.8 GHz CPU with 4GB memory. In all tests, we measure the run time of the solver and the number of nodes processed by the branch and bound algorithm. A node is processed if the algorithm actually calculates the greedy schedule with this node's tightened availability intervals; we do not count nodes that are never retrieved from the queue because they have a too high lower bound to possibly contain a better solution than what is already found.

This experimental analysis is structured as follows. In Section 4.2, we explain the benchmarks we used. In Section 4.3, we verify that our implementation is correct and that we did not miss implementation details that make our solver much slower than the branch and bound algorithms could be. Then, in Section 4.4 we test which parameters of our benchmark model generate the hardest instances, and we do the actual experimental evaluation in Section 4.5.

4.2 Benchmarks used

We have tested the branch and bound solver in two ways: as a standalone solver, and as a subroutine in a solver for the Job Shop problem.

For the Job Shop problem, we used a solver (Applegate and Cook, 1996) that executes the Shifting Bottleneck algorithm (see Section 2.5; the actual implementation uses a more complicated algorithm (with limited backtracking) than the basic version we presented). Numerous benchmarks sets for the Job Shop problem have been proposed; we used the benchmark set of Taillard (1993), containing 80 problems with sizes between 15×15 and 100×20.

For the benchmarks on which we tested the solver directly, we generalized the benchmarks used in the literature by introducing correlation between release times and deadlines. The benchmarks used in the literature, such as in Carlier's paper (Carlier, 1982) in which he proposed his branch and bound algorithm, use the uniform distribution on a given interval for release times, deadlines and processing times: all variables are sampled independently. This means that the model has two parameters: a range for the release times and deadlines $[0,T]$ and a range for the processing times $[1, p_{\max}]$.[1] All release times, deadlines and processing times are then sampled independently and uniformly at random from the corresponding ranges.

We generalize this distribution by introducing a parameter $\alpha \in [-1,1]$ to represent correlation between release times and deadlines. For $\alpha \geq 0$, we generate the release times and deadlines with the formulas

$$\begin{aligned} r &= T \cdot (|\alpha| \cdot X_{common} + (1-|\alpha|) \cdot X_r) \\ d &= T \cdot (|\alpha| \cdot X_{common} + (1-|\alpha|) \cdot X_d) \end{aligned}$$

where X_{common}, X_r, X_d are sampled from the uniform distribution on $[0,1]$ independently from each other. Similarly, for $\alpha < 0$ the formulas are

$$\begin{aligned} r &= T \cdot (|\alpha| \cdot X_{common} + (1-|\alpha|) \cdot X_r) \\ d &= T \cdot (|\alpha| \cdot (1 - X_{common}) + (1-|\alpha|) \cdot X_d) \end{aligned}$$

The correlation between r and d equals $\rho(\alpha) = 2\alpha^2/((2\alpha-1)^2+1)$ if $\alpha \geq 0$ and $\rho(\alpha) = -\rho(-\alpha)$ for $\alpha < 0$. So for $\alpha = -1, 0, 1$ we have $\rho(\alpha) = \alpha$, but this is not true for other α.

4.3 Verification of the implementation

Before doing the actual experimental analysis, we tested our branch and bound algorithm and also tested the lower bound and upper bound functions individually (this was especially necessary for our algorithm for the $\{1, p\}$ problem). Concretely, we first tested our implementation of existing ideas of Carlier with a trivial dynamic programming over subsets scheduling algorithm with run time $O^*(2^n)$ as the reference implementation. In the second step, we tested our algorithm for the $\{1, p\}$ problem using both the brute force implementation (for small instances) and the already verified branch and bound algorithm (for larger instances) as the reference implementation.

Next to this verification of correctness, we also did a verification of the solver's performance by comparing it against the single machine solver part of the Job Shop solver we use for our experiments on Job Shop problems. To our surprise, this solver solved some of our most difficult instances within seconds, but as it turns out the program contained a bug: the problem is that, in the branch that the interference job

[1] Because optimizing $L_{\max}$ remains the same problem if one delays all deadlines by the same amount of time, we can assume that release times and deadlines start from time 0. One could also use separate ranges $[0, r_{\max}]$ and $[0, d_{\max}]$, but the case $r_{\max} = d_{\max} = T$ seems to be general enough.

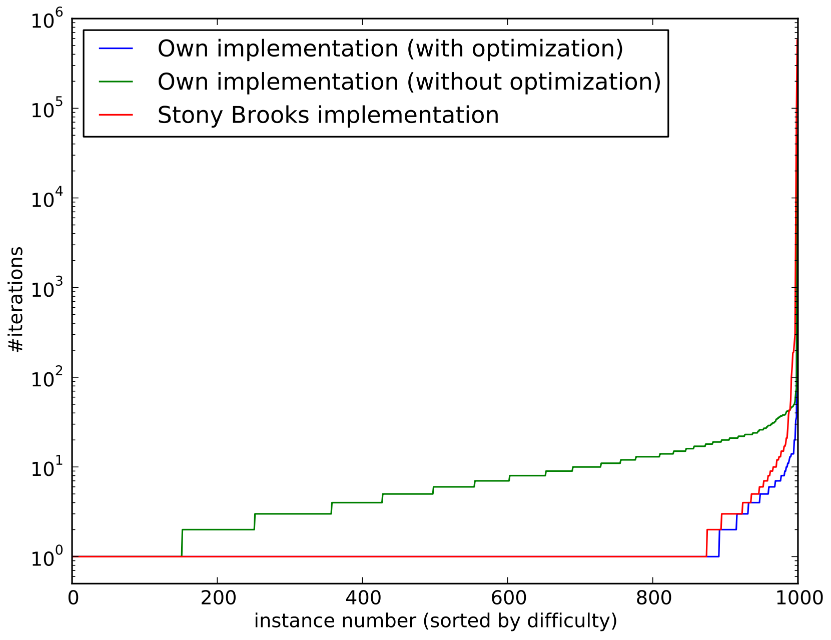
has to be placed after the set of delayed jobs J, the new release time is set to $\max_{i \in J} r_i + \sum_{i \in J} p_i$, which is incorrect; this should be $\min_{i \in J} r_i + \sum_{i \in J} p_i$.[2] Their implementation also contained one optimization that we did not implement: when a node is processed, they check whether Schrage's schedule modified by moving the interference job to the place immediately after the set of delayed jobs is better than the best schedule found so far. This was also presented in Carlier's original paper. We added this optimization to our solver. Other implementation details in which the two solvers differ are: instead of using a priority queue, their solver iterates over all elements of the queue to find the most urgent one in Schrage's algorithm, which means it takes $O(n^2)$ time worst case. As the number of jobs is less than 100 in most benchmarks, this does not make a great difference. It also does not recalculate the preemptive lower bound at each node, but instead computes only the lowerbound $\min_{j \in J} r_j + \sum_{j \in J} p_j - \max_{j \in J} d_j$ for the set of delayed jobs J and for the set J with the interference job added to it. This was also presented in Carlier's paper, but we think that computing the whole preemptive lower bound will not change much in performance because one has to compute the greedy schedule using Schrage's heuristic at each iteration, which also takes $O(n \log n)$ time.

We then compared the performance of both implementations. We compare three implementations: our implementation with and without the optimization to "try" the schedule with the interference job placed directly after the delayed jobs, and the implementation from the Job Shop solver. We first tested the configuration $\alpha = 0$ (no correlation between release times and deadlines), $N = 100$, $p_{\max} = 10$ and $T = N \cdot p_{\max}/2 = 500$. We tested 1000 instances. The number of iterations the solvers required are displayed in Figure 4.1. These instances are very simple for all solvers; in 99% of the cases, all solvers terminate within 100 iterations (and within 0.01 seconds in time). Still, we can see that the optimization makes our solver faster, and that the Stony Brook solver performs similarly to our solver on these instances. Because this distribution is a bit too simple to really compare performance of the solvers, we compared the solvers on a benchmark with $\alpha = -1$ (so release times and deadlines are negatively correlated). We use $p_{\max} = 10$, $N = 50$ and $T = N \cdot p_{\max}/2 = 250$ and a time limit of 10 seconds per instance. The results are visualized in Figure 4.2. For instances on which some solver was terminated at the time limit, we set the number of iterations to infinity (10^9). We see that both the number of iterations and the run time of the solvers are comparable, with our implementation doing slightly better in this case.

We conclude that the performance of the solvers is similar, both in terms of the number of iterations and in terms of runtime.

Next, we compared Carlier's and McMahon and Florian's branching schemes. We did two tests: $\alpha = 0$, $p_{\max} = 10$, $N = 35$, $T = N \cdot p_{\max}/2 = 175$ (1000 instances), and $\alpha = -1$, $p_{\max} = 10$, $N = 20$ and $T = N \cdot p_{\max}/2 = 100$ (250 instances). The results are given in Figure 4.3. We conclude that Carlier's branching scheme works better: while it is not much worse on the easy instances with $\alpha = 0$, its performance on those with $\alpha = -1$ is clearly much worse than that of Carlier's branching scheme. This is consistent with earlier results such as the comparison by Sadykov and Lazarev (2005). In the experiments that follow, we only use Carlier's scheme.

[2]The smallest example we found that was solved incorrectly is:
heads: $[3,1,2]$; bodies: $[2,3,3]$; tails: $[3,1,3]$. The solver reports that 12 is the optimal makespan, while the answer is 11, which is reached with the order of jobs [3, 1, 2].



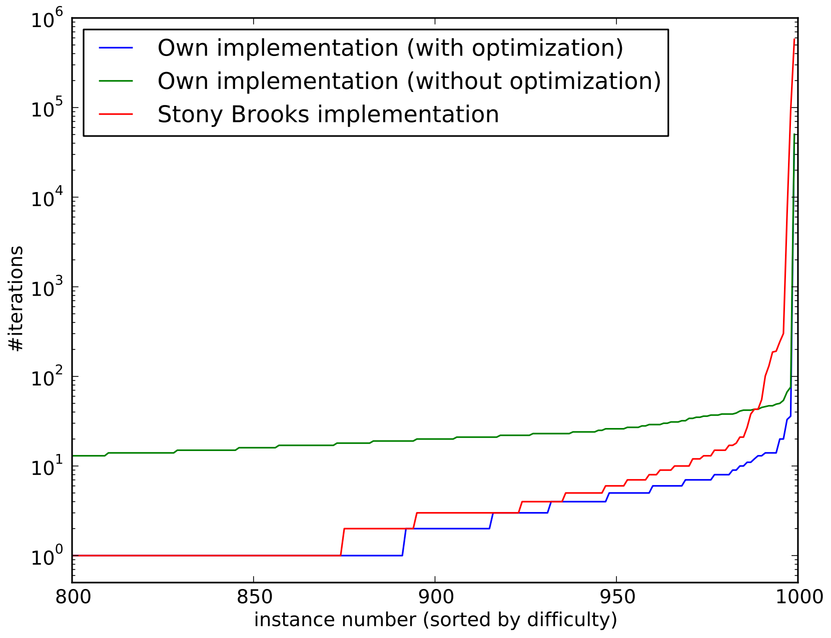
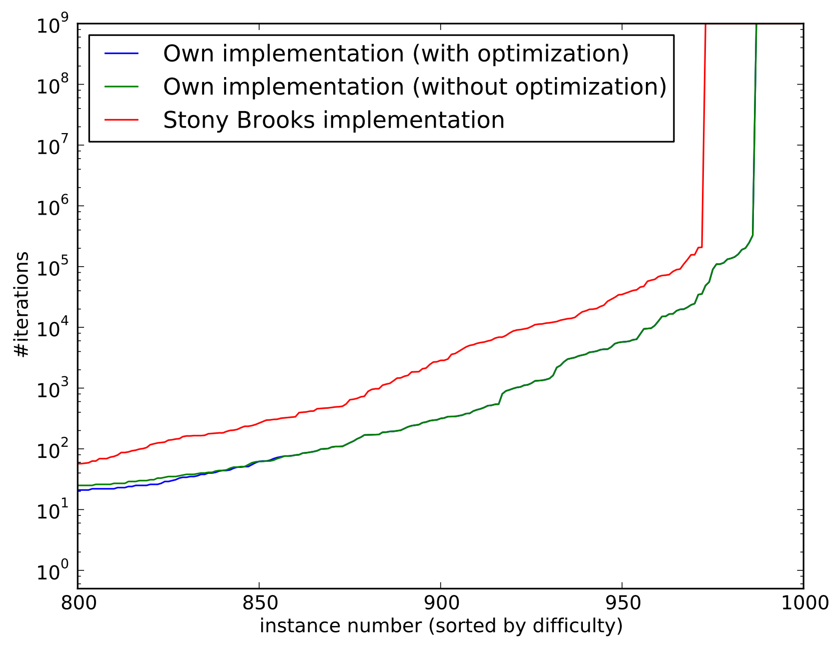


Figure 4.1: The number of iterations on 1000 instances with $N = 100$, $p_{\max} = 10$, $T = 500$ and $\alpha = 0$ (no correlation between release times and deadlines). For each solver, the sequence of number of iterations required is sorted. Above: all benchmarks; below: the hardest 20% for each solver.



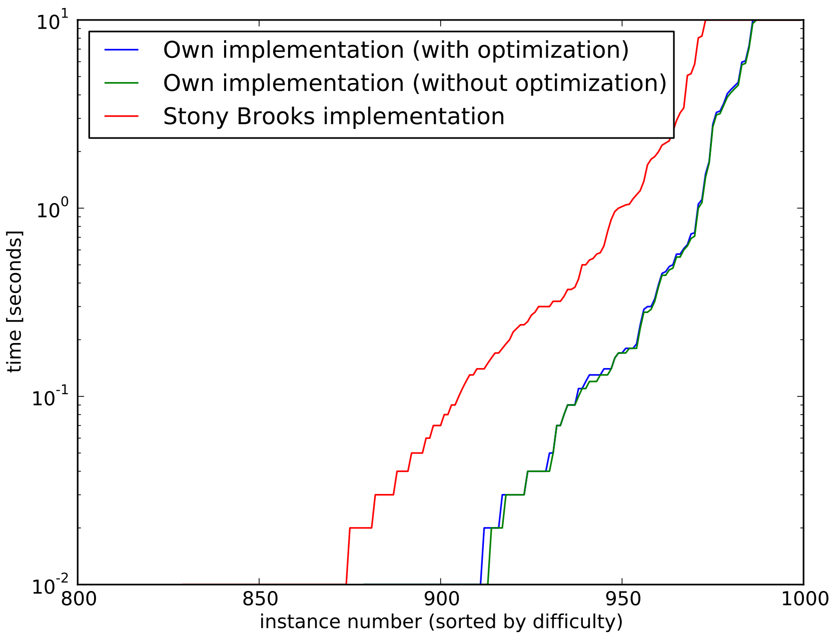
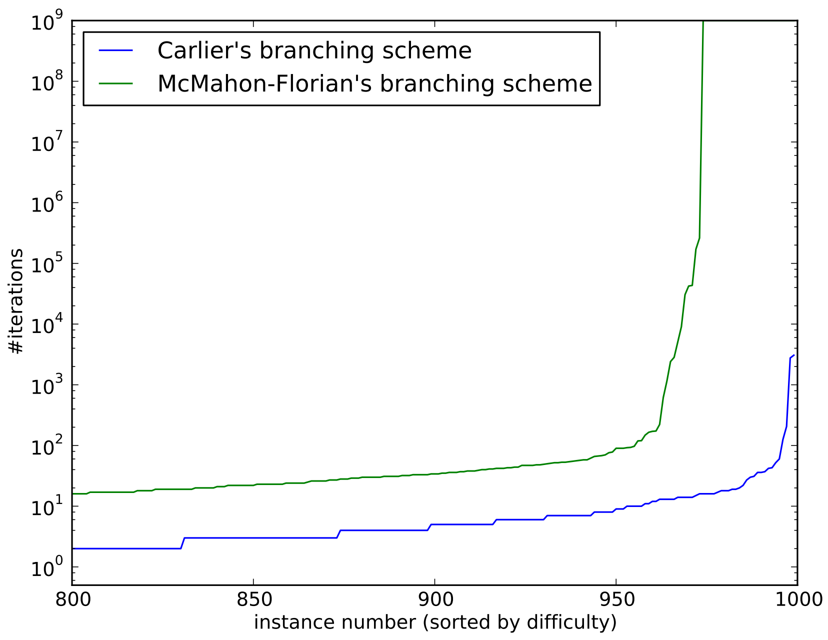


Figure 4.2: The number of iterations on the hardest 20% from 1000 instances with $N = 50$, $p_{\max} = 10$, $T = 250$ and $\alpha = -1$ (negative correlation between release times and deadlines).



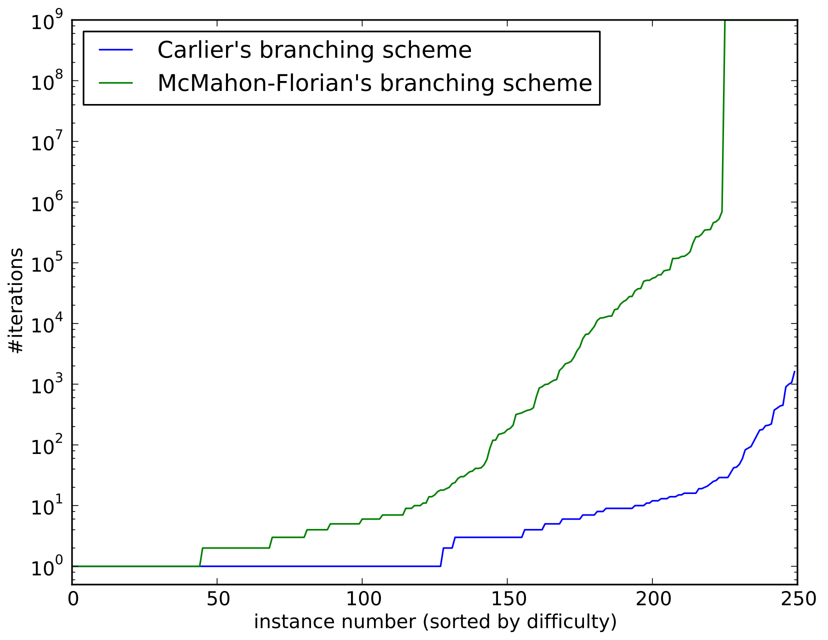


Figure 4.3: Above: the number of iterations on the hardest 20% from 1000 instances with $N = 35$, $p_{\max} = 10$, $T = 175$ and $\alpha = 0$ (no correlation between release times and deadlines); below: the number of instances with $N = 20$, $p_{\max} = 10$, $T = 100$ and $\alpha = -1$ (negative correlation between release times and deadlines).

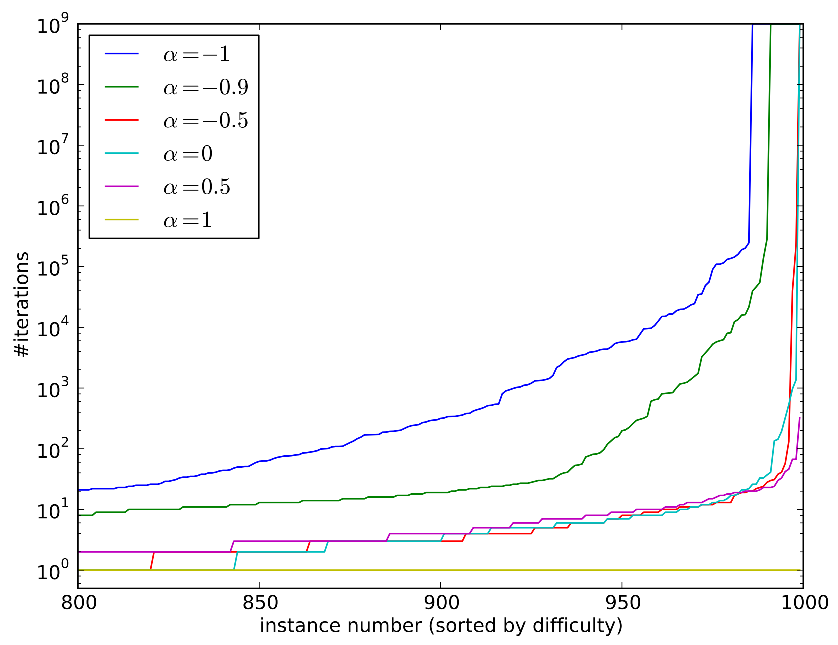


Figure 4.4: The number of iterations on the hardest 20% from 1000 instances with $N = 50$, $p_{\max} = 10$, $T = 250$ for various values of α.

4.4 The relation between hardness and the correlation between release times and deadlines

From the tests in Section 4.3, we saw that instances from the distribution with $\alpha = -1$ for $N = 50$ are much harder than instances from the distribution with $\alpha = 0$ for $N = 100$. We expect that negatively correlated instances are the hardest and values of α close to -1 give hard instances. We test this hypothesis for various other values of α. We use the distribution with $N = 50$, $p_{\max} = 10$ and $T = N \cdot p_{\max}/2 = 250$ for $\alpha \in \{-1, -0.5, 0, 0.5, 1\}$. We use a time limit of 10 seconds (a larger time limit is not needed to distinguish between the hardness of these values of α). The result is given in Figure 4.4 (again, if the time limit is reached, the number of iterations is set to 10^9). It is surprising that the $\alpha = -0.5$ instances are equally easy as the $\alpha = 0$ instances. We then tested $\alpha = -0.9$ and the result (also in Figure 4.4) indeed shows that the instances become harder when release times and deadlines become increasingly negatively correlated. We conclude that $\alpha = -1$ generates the hardest instances and other values close enough to -1 (such as -0.9) also give hard instances.

4.5 Evaluation of the new lower and upper bound

In this section we do the actual comparison between the original preemptive lower bound and greedy upperbound and the half-preemptive lower bound and upper bound (see Section 3.3). The half-preemptive lower bound procedure takes as input a block size P and divides each task with length p_i in a preemptive part of length $p_i \mod P$ and $\lfloor P/p_i \rfloor$ blocks. It then calls the $\{1,P\}$-solver to find a schedule with minimum $L_{\max}$ for the created problem; the $L_{\max}$ value of this schedule is a lower bound on the actual $L_{\max}$. The half-preemptive upper bound also takes as input a block size P and computes the same decomposition into blocks and a preemptive task as the half-preemptive lower bound. It also computes an optimal half-preemptive schedule and then rounds the schedule by fixing the positions of the tasks represented by exactly one block, and then feeding the instance to Schrage's algorithm. The bottleneck for these computations is obviously the $\{1,P\}$-solver. Because we need the same half-preemptive schedule for both the lower and the upper bound, we decided to compute the lower bound and upper bound for exactly the same values of P, remembering the optimal half-preemptive schedule after the lower bound is computed.

We have tried a number of combinations of values of P. We report only on the performance of two options that frequently worked well: $P = 1,\dots,p_{\max}$ and $P = \lceil p_{\max}/3 \rceil,\dots,p_{\max}$. Many other combinations of values of P are possible, but one of these two options often performed best in our tests. One could also choose the set of P dynamically, but we did not try this. Our experimental analysis consists of three parts. In Section 4.5.1, we analyze performance on benchmarks generated with the model defined in Section 4.2. In Section 4.5.2, we analyze performance of our solver integrated in an existing Jobshop solver (Applegate and Cook, 1996) that uses the Shifting Bottleneck algorithm (see Section 2.5). In Section 4.5.3, we analyze performance of our new lowerbound function as a standalone function on benchmarks generated with the model defined in Section 4.2.

4.5.1 Performance on randomly generated instances

We focus on two test distributions, $\alpha \in \{-1,0\}$. The $\alpha = -1$ distribution contains hard instances and the $\alpha = 0$ distribution contains easier instances. We always use $T = N \cdot p_{\max}/2$ and $p_{\max} = 10$. In all tests, we use a time limit of 60 seconds.

The results for $\alpha = -1$, and $p_{\max} = 10$ with 1000 instances are given in Figure 4.5 (for $N = 50$) and Figure 4.6 (for $N = 100$). The results show that on these instances, the new lower bound and upper bound perform much better on the most difficult instances. For $N = 100$, the preemptive+greedy version did not solve 91 out of 1000 instances (9%); the full half-preemptive lower bound version did not solve 5 out of 1000 instances (0.5%), and the half-preemptive lower bound version that tests only the $P \geq \lceil p_{\max}/3 \rceil$ failed to solve 10 instances (1.0%). So, on these instances, the improved lower bound and upper bound allow the branch and bound solver to solve about 8% more of the instances with $N = 100$ in one minute.

We think that the speedup is caused by the half-preemptive lower bound being able to prove optimality of the best solution found so far in more cases than the preemptive lower bound, but that the preemptive+greedy solver finds the optimal solution much earlier than when it can prove that this solution is optimal. To test this hypothesis, we precalculated the correct answer by the half-preemptive version (the version that computes the lower bound for all P), and then ran the preemptive+greedy solver, counting separately the number of iterations until the lower bound equals the answer and until the upper bound equals the answer.[3] The result is given in Figure 4.7. We see that indeed, in almost all instances the solver versions using the half-preemptive bounds can solve, the original algorithm also found the answer in under 100 iterations, but it cannot prove that this is optimal.

The case of uncorrelated release times and deadlines ($\alpha = 0$) can be solved efficiently with the original solver. This can be seen in Figure 4.4. We ran additional tests for larger instances: $N \in \{100, 1000, 10000\}$ (1000 instances for each N). In all runs we let $p_{\max} = 10$ and $T = N \cdot p_{\max}/2$. For $N = 50$, the solver terminates within 100 iterations in 99% of the cases; for $N = 100$, $N = 1000$ and $N = 10000$ the solver terminates within 100 iterations in 999 out of 1000 cases. Therefore, for this distribution the new lowerbound will not give an improvement.

4.5.2 Performance when integrated in a Jobshop solver

For this part, we integrated our one machine solver in the Shifting Bottleneck solver of Applegate and Cook (1996). We used the benchmark set by Taillard (1993). This set contains 80 benchmarks with sizes from 15×15 (15 chains of jobs on 15 machines) to 100×20 (100 chains of jobs on 20 machines). We do not use the backtracking option of the solver. It turns out that the new lower bound does not at all improve performance; instead, the solver becomes about 100 times slower. The results are displayed in Table 4.1 (for 15×15 instances) and Table 4.2 (for 20×15 instances). Clearly, the halfpreemptive bounds slow down the solver very much. An explanation for this is that the resulting single machine instances are as easy as those from the distribution with $\alpha = 0$ so that the halfpreemptive lower bound does not reduce the number of iterations, but it does cost much more computation time than the original preemptive lower bound.

	1	2	3	4	5	6	7	8	9	10
preemp.	0.06	0.05	0.06	0.05	0.05	0.06	0.04	0.07	0.05	0.04
all P	11.57	14.57	16.41	13.89	13.52	15.25	10.88	17.09	13.94	12.18
$P \geq p_{\max}/3$	5.89	7.32	8.11	6.90	6.86	7.73	5.40	8.69	7.27	5.95

Table 4.1: Results for Taillard's 15×15 instances.

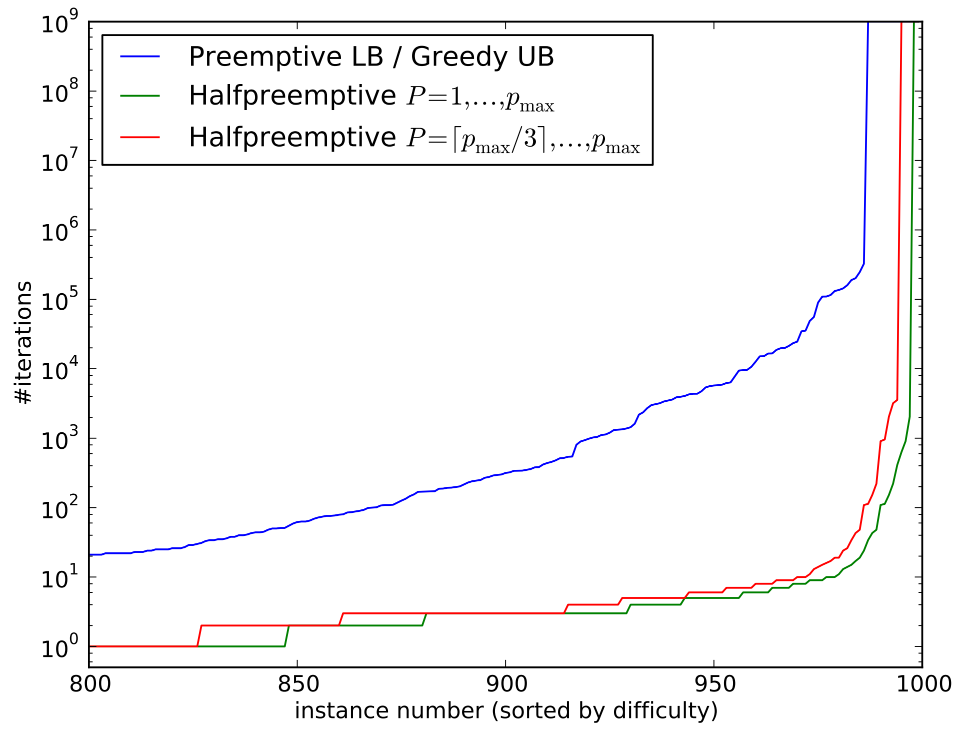
	1	2	3	4	5	6	7	8	9	10
preemp.	0.09	0.08	0.07	0.08	0.09	0.08	0.09	0.09	0.09	0.09
all P	33.92	28.99	22.32	25.45	28.67	31.10	28.64	28.94	29.56	25.36
$P \geq p_{\max}/3$	15.79	14.19	10.43	11.96	13.92	14.51	13.28	13.63	14.35	12.17

Table 4.2: Results for Taillard's 20×15 instances.

[3]The lowerbound equals the answer when all nodes in the queue have a lowerbound greater than or equal to the answer.

4.5.3 Runtime analysis of the lowerbound function

Finally we analyze the runtime of the half-preemptive lower bound on random instances from the distribution with $\alpha = 0$. We use as parameters for the distribution $p_{\max} = 10$ and $T = N \cdot p_{\max}/2$; we benchmark the half-preemptive lowerbound with $P = p_{\max}/2$ for $N = 100, 200, 300, \ldots, 1000$ with 1000 instances for each N. One run of the lowerbound means calculating the maximum lateness of a halfpreemptive schedule, so we use binary search on the maximum lateness and repeatedly call the $\{1, P\}$-solver. Before the computation starts, we call Schrage's algorithm for the initial guess of $L_{\max}$ (the instance to Schrage's algorithm consists of blocks of length P and preemptive parts) so that the interval on which binary search must be done has size at most $p_{\max}$. The results are displayed in Figure 4.8. More than half of the instances are very easy and are solved very quickly; however for the average runtime and for the 75th percentile of the instances, the run time of the solver seems to be in the order of $O(n^2)$.



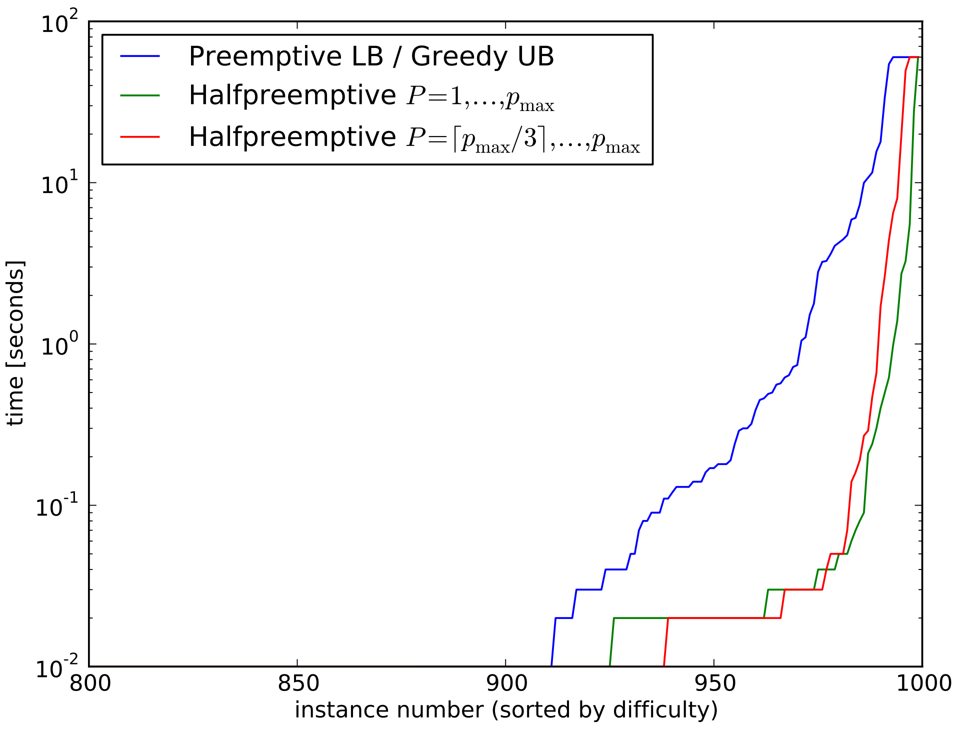
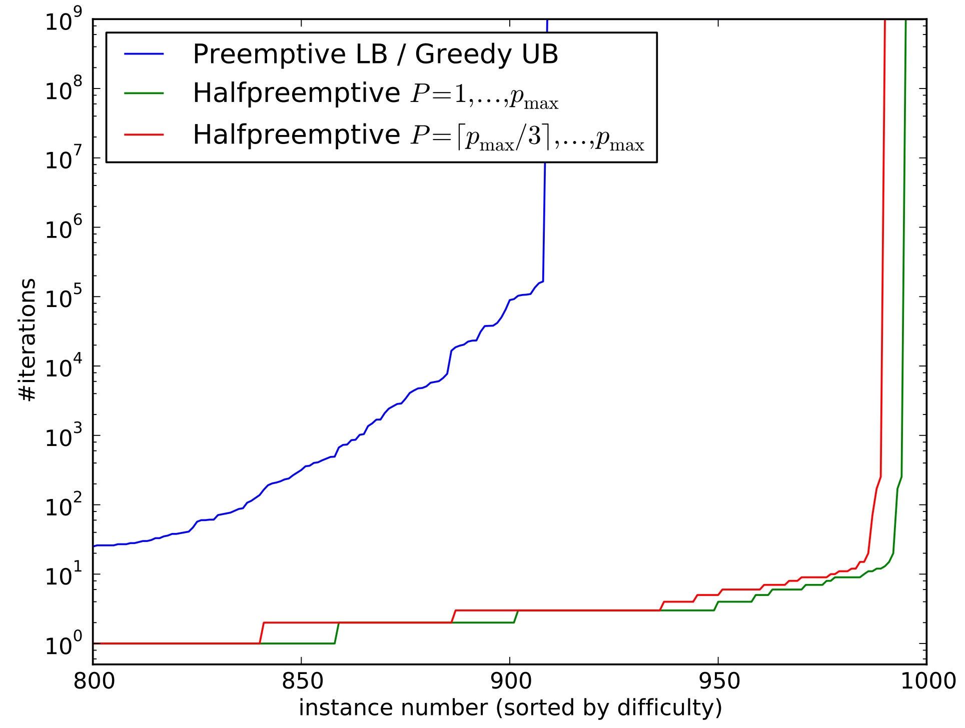


Figure 4.5: The number of iterations and the runtime on the hardest 20% from 1000 instances with $\alpha = -1$, $N = 50$, $p_{\max} = 10$ and $T = 250$



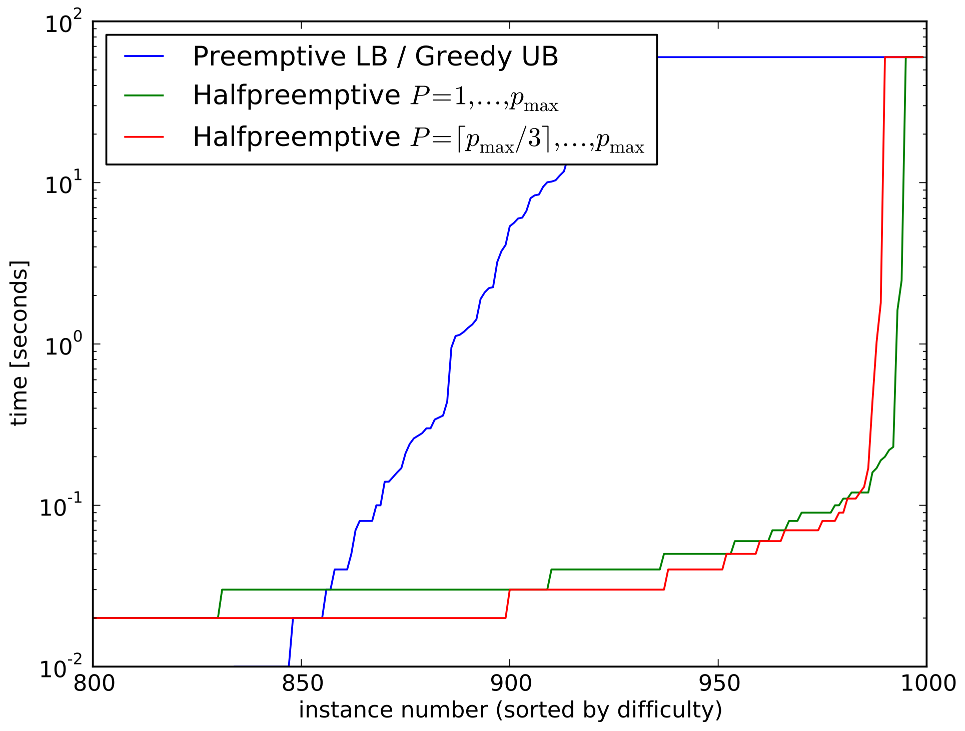


Figure 4.6: The number of iterations and the runtime on the hardest 20% from 1000 instances with $\alpha = -1$, $N = 100$, $p_{\max} = 10$ and $T = 500$

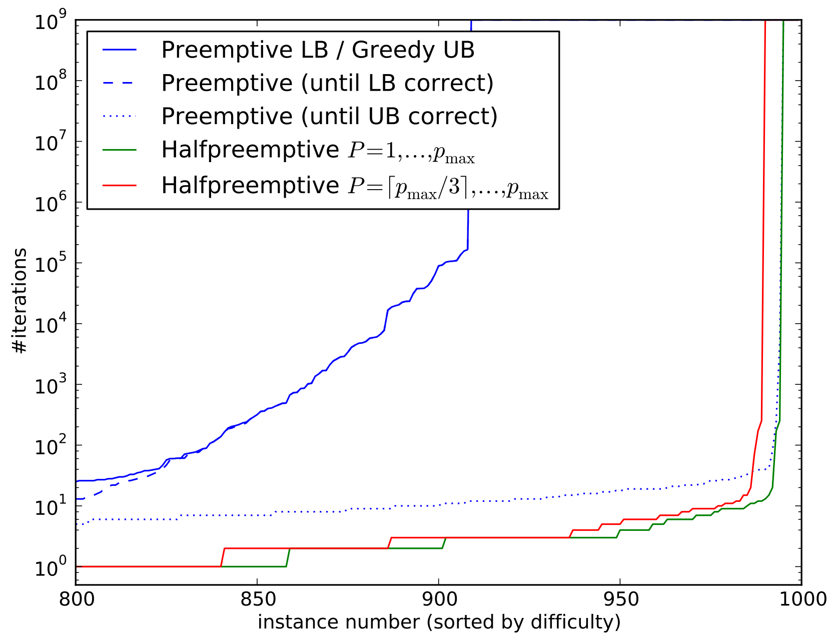


Figure 4.7: The number of iterations until the lower bound and upper bound equal the optimal answer (for the preemptive+greedy version). Note that, for each graph, the numbers of iterations are sorted, so the numbers of iterations of all graphs for one specific instance number do not correspond to the same instance (this is also why the run time of the preemptive solver is not the maximum of the time until the lower bound is correct and the time until the upper bound is correct).

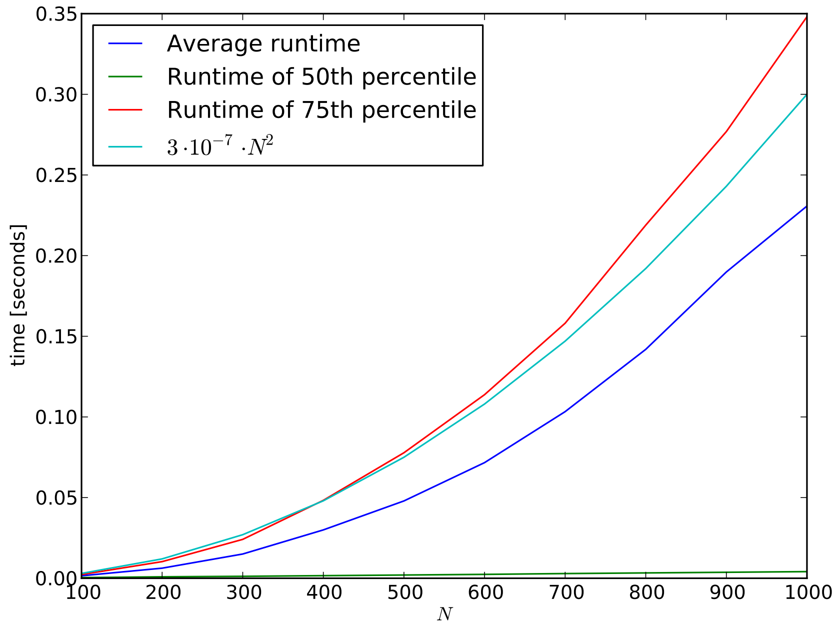


Figure 4.8: Run time of the lowerbound function as a standalone function on benchmarks with $\alpha = 0$, $p_{\max} = 10$, $T = N \cdot p_{\max}/2$.

Chapter 5

Conclusion

In this thesis, we have studied the single machine problem with release times and deadlines. We have surveyed existing algorithms in Chapter 2. We studied polynomial time algorithms for the equal processing times case and its generalization to the case with task lengths from the set $\{1, P\}$, for some integer P. The latter case can also be viewed as the problem in which all non-preemptive tasks have the same length, and all other tasks are preemptive. We then studied branch and bound algorithms for the unrestricted case. These branch and bound algorithms modify the availability intervals of the tasks in the branching step.

We have presented a practical algorithm for the $\{1, P\}$ problem in Section 3.2, and we have presented a new lower bound (the half-preemptive lower bound) and a corresponding upper bound in Section 3.3. We have done an experimental analysis of the branch and bound algorithm and the new lower bound in Chapter 4. The conclusions of this experiment are that the original branch and bound algorithm usually can find the optimal solution, but with certain instances, the preemptive lower bound is not able to prove optimality. This happens with instances with negatively correlated release times and deadlines. On instances without this property, the original branch and bound algorithm terminates within 100 iterations in 99% of the cases, and the new lower bound only slows down these iteration, which leads to much worse performance.

We have also presented an NP-completeness result for the task scheduling problem with two non-unit task lengths by a reduction from the Satisfiability (SAT) problem. This is presented in Section 3.5. We understand that the formulation must still be improved and that it is too difficult to determine whether the current construction is correct. We did implement this reduction and test it on small formulas; for example, the branch and bound algorithm is able to prove that the formula $x_1 \land \neg x_1$ is unsatisfiable.

The original motivation to study this single machine scheduling problem was to learn new techniques for dealing with the combination of release times and deadlines in a non-trivial way: techniques that are more problem specific than greedy algorithms and general-purpose heuristic search algorithms. These techniques could then be applied to practical scheduling problems, such as the problem with a resource profile instead of only one machine as the available capacity over time.

The forbidden regions algorithm for identical task lengths fits this description very well: it solves a problem optimally that greedy algorithms cannot solve. However, the practical applicability of the identical task lengths model is fairly limited. The same holds for the generalization to task lengths $\{1,P\}$. The only use is that we can compute a half-preemptive schedule, which should have less preemptions than the schedule produced by the preemptive version of the EDD rule. We described a procedure to round the $\{1,P\}$-schedule back to a non-preemptive schedule, but this procedure only improves upon the greedy algorithm if many tasks have the same length.

The branch and bound algorithm of Carlier does not seem to be very different from solution techniques for more general task scheduling problems such as Precedence Constraint Posting.

To summarize, we have not been able to apply the algorithms we studied to cause improved performance on practical problems, but we did learn new interesting techniques by studying exact algorithms for the case with both release times and deadlines.

Bibliography

Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):pp. 391–401, 1988. ISSN 00251909. URL `http://www.jstor.org/stable/2632051`.

David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991. doi: 10.1287/ijoc.3.2.149. URL `http://dx.doi.org/10.1287/ijoc.3.2.149`.

David Applegate and William Cook. Jobshop solver software. `http://www.cs.stonybrook.edu/~algorith/implement/jobshop/implement.shtml`, 1996.

Kenneth R. Baker and Zaw-Sing Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics Quarterly*, 21(1):171–176, 1974. ISSN 1931-9193. doi: 10.1002/nav.3800210112. URL `http://dx.doi.org/10.1002/nav.3800210112`.

Philippe Baptiste. Scheduling equal-length jobs on identical parallel machines. *Discrete Applied Mathematics*, 103(13):21 – 32, 2000. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/S0166-218X(99)00238-3. URL `http://www.sciencedirect.com/science/article/pii/S0166218X99002383`.

R. E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

Jacques Carlier. Problemes d'ordonnancement à durées égales. *QUESTIIO*, 5(4):219–228, 1981.

Jacques Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42 – 47, 1982. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/S0377-2217(82)80007-6. URL `http://www.sciencedirect.com/science/article/pii/S0377221782800076`.

E.G Coffman, Jr., M.R Garey, and D.S Johnson. Bin packing with divisible item sizes. *Journal of Complexity*, 3(4):406 – 428, 1987. ISSN 0885-064X. doi: http://dx.doi.org/10.1016/0885-064X(87)90009-4. URL `http://www.sciencedirect.com/science/article/pii/0885064X87900094`.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

Christoph Dürr and Mathilde Hurand. Finding total unimodularity in optimization problems solved by linear programs. *Algorithmica*, 59(2):256–268, February 2011. ISSN 0178-4617. doi: 10.1007/s00453-009-9310-7. URL `http://dx.doi.org/10.1007/s00453-009-9310-7`.

M. R. Garey, David S. Johnson, Barbara B. Simons, and Robert Endre Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM J. Comput.*, 10 (2):256–269, 1981.

Michel X. Goemans and Thomas Rothvoß. Polynomiality for bin packing with a constant number of item types. *CoRR*, abs/1307.5108, 2013.

R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979. doi: http://dx.doi.org/10.1016/S0167-5060(08)70356-X. URL `http://www.sciencedirect.com/science/article/pii/S016750600870356X`.

Dorit S. Hochbaum and Joseph Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM J. Comput.*, 23(6):1179–1192, 1994.

Graham McMahon and Michael Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):pp. 475–482, 1975. ISSN 0030364X. URL `http://www.jstor.org/stable/169697`.

Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. ISBN 0387789340, 9780387789347.

C. N. Potts. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):pp. 1436–1441, 1980. ISSN 0030364X. URL `http://www.jstor.org/stable/170101`.

Ruslan Sadykov and Alexander Lazarev. Experimental comparison of branch-and-bound algorithms for the $1|r_j|L_{\max}$ problem. In *MAPSP*, 2005. URL `http://www.math.u-bordeaux1.fr/~rsadykov/papers/SadykovLazarev_MAPSP05.pdf`.

Jiri Sgall. Open problems in throughput scheduling. In *ESA*, pages 2–11, 2012.

Barbara Simons. A fast algorithm for single processor scheduling. In *FOCS*, pages 246–252, 1978.

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

Éric Taillard. Jobshop benchmarks. `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html`, 1993.