On-line scheduling and bin packing

On-line scheduling and bin packing

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 8 mei 2002
te klokke 14.15 uur

door

Rob van Stee

geboren te Vlissingen
in 1973

Promotiecommissie

Promotores:	Prof.dr.ir. J.A. La Poutré	(Technische Universiteit Eindhoven / CWI)
	Prof.dr. J.N. Kok	
Referent:	Prof.dr. K. Pruhs	(University of Pittsburgh)
Overige leden:	Prof.dr. J. van Leeuwen	(Universiteit Utrecht)
	Prof.dr. G. Rozenberg	
	Prof.dr. H.A.G. Wijshoff	
	Prof.dr. G. Woeginger	(Universiteit Twente)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). IPA dissertation series 2002-09.

Work carried out at Centre for Mathematics and Computer Science (CWI), Amsterdam, and Leiden University.

Online scheduling and bin packing
Rob van Stee
ISBN 90-77017-60-7

Preface

The research for this thesis was performed at Leiden University from August 1996 until August 1997, and at the Centre for Mathematics and Computer Science (CWI) in Amsterdam from September 1997 until August 2001. It was supervised by Han La Poutré from the CWI and Joost Kok from Leiden University.

I am grateful to the many people for the help and support they have offered me during this time, and I cannot list them all here.

I had a very enjoyable time at the CWI in the group SEN4, and I am grateful to Floortje, Sander, Enrico, David, Michiel, Pieter-Jan, Dimitri, Koye, Erich, Bill, Jeroen and Simona for this. Looking back on this time it seems even better to me now than it did then, but maybe it is often the case that you do not enjoy good things enough while they last. It was great.

My coauthors, Han La Poutré, Leah Epstein, Steve Seiden, Yossi Azar, Eric Bach, Joan Boyar, Lene Monrad Favrholdt, Tao Jiang, Kim Skak Larsen, and Guo-Hui Lin, helped during research and in presenting the work.

I am also grateful to my family and friends for their interest and support - even if some of them did keep asking when I would finally get my Master's degree! Dit is geen scriptie…

I would especially like to thank my parents for all they have done for me and still do. Although I feel that I can never thank them enough, I can certainly try. You always stand behind me.

Rob van Stee
Freiburg, 2002.

Contents

Chapter 1

Introduction

This thesis is concerned with various (scheduling) problems in which the information about the problem arrives in parts, and decisions have to be made while the information is still incomplete. In classical (scheduling) problems it is assumed that all information about the problem is known in advance. However, this assumption is not always realistic: in many cases, information arrives incrementally. In such cases, an *on-line algorithm* is needed, which can make decisions without knowing the complete input.

A natural and important example of a problem with incomplete information is *paging*, the problem of maintaining a small cache of fast memory in a computer system. In this problem, a controller has to decide which page to eject from the cache when a program requests a page that is not currently in the cache. If future requests are known, this is solved optimally by ejecting the page which will be requested the last among all pages in the cache. However, in real-life applications the sequence of future requests will not be known, and the decision has to be made in some other way. This problem has received a lot of attention over the years [2, 11, 14, 24, 50, 70].

In section 1.1, we discuss on-line algorithms, and we give an overview of the problems treated in this thesis in section 1.2. In section 1.3 we discuss two extensions of competitive analysis: allowing the use of randomization, and giving the on-line algorithm more resources. Finally, we give the outline of the thesis in section 1.4.

1.1 On-line algorithms

Probabilistic analysis One way to approach on-line problems is to make (probabilistic) assumptions about the unknown information and use those assumptions to define heuristics for the problem. These heuristics can then be studied using an average-case analysis, which considers the average performance over all possible inputs. However, it can be difficult to make

good assumptions about the future inputs. Also, heuristics based on average-case analysis could perform very badly in some extreme cases.

Worst-case analysis Another way to study these problems is to focus on the worst-case behaviour of algorithms. Since the cost for a particular instance may be arbitrarily high, the behaviour of an algorithm should be compared to other algorithms to get meaningful results. In particular, it is important to know how much worse an algorithm performs relative to an *optimal* algorithm, in other words, how much worse it is than the optimal solution for any given problem instance. This kind of analysis is known as *competitive analysis*, which was introduced by Sleator and Tarjan [70]. It involves comparing the performance of an on-line algorithm to the performance of an off-line algorithm that knows the entire problem instance in advance. We do not impose limits on the computational complexity of either the off-line or the on-line algorithm. Therefore the off-line algorithm can always generate an optimal solution.

This type of analysis can be viewed as a game between two players, an on-line algorithm and an *adversary* that both generates the problem instance and serves it as an off-line algorithm. The adversary tries to maximize its performance relative to the on-line algorithm.

Many problems have been studied using competitive analysis. Apart from the paging problem, these include a variety of scheduling problems, bin packing, routing and admission control on a network [35, 13, 4, 41, 3, 36, 6].

We will apply both of the above methods in this thesis, but the main focus will be on worst-case analysis. We now introduce the competitive ratio.

The competitive ratio We consider both algorithms that seek to minimize a cost and algorithms that seek to maximize a benefit. We denote the cost or benefit of an algorithm $\mathcal{A}$ on an input σ by $\mathcal{A}(\sigma)$. An optimal off-line algorithm is denoted by OPT. (There can be more than one optimal off-line algorithm for a given problem.)

We compare an on-line algorithm $\mathcal{A}$ to OPT using the *competitive ratio* [70], which for algorithms that try to minimize a certain cost is defined as follows:

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\mathcal{A}(\sigma)}{\mathrm{OPT}(\sigma)}$$

where the supremum is taken over all possible inputs. For algorithms that try to maximize a certain benefit, we define the competitive ratio as

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\mathrm{OPT}(\sigma)}{\mathcal{A}(\sigma)}.$$

In both cases, the best on-line algorithm for a problem is the one that has the lowest possible competitive ratio, and this ratio is at least 1.

The competitive ratio of a problem is defined as $\inf_{\mathcal{A}} \mathcal{R}(\mathcal{A})$.

The competitive ratio is clearly a worst-case measure, and by determining the competitive ratio of a certain problem, one can determine the benefit of knowing the entire problem instance in advance. An advantage of such a comparison is that if one can prove that an algorithm has a competitive ratio of $\mathcal{R}$, then any other algorithm can do at most a factor of $\mathcal{R}$ better on any input.

1.2 Problems considered in this thesis

We will now highlight the problem areas studied in this thesis.

On-line scheduling In this problem, n jobs with different processing requirements are to be scheduled on m parallel identical machines. Jobs arrive over time and each job has to be assigned to one of the machines and run there continuously until it is completed. Each machine can only run one job at a time. The on-line algorithm only becomes aware of a job when it arrives.

We also consider the case where the jobs arrive one by one (in a list) and each job arrives only after the previous one has been assigned.

The input is a job sequence $\sigma = \{J_1, \ldots, J_n\}$. Each job J_i arrives at its *release time* r_i and needs to be run for w_i time on one of the machines (w_i is the *size* or *weight* of J_i). J_i is completed after it has been running for w_i time, and the time at which this happens is its *completion time* c_i. We will also be interested in the *flow time* f_i of J_i, which is defined as $f_i = c_i - r_i$: the time that job J_i exists in the system (without being completed).

The output of an algorithm is a schedule π that for each job determines when and on which machine it is run. We denote the starting time of job J_i in schedule π by $S_i(\pi)$, and its flow time in π by $f_i(\pi) = c_i(\pi) - r_i$.

We can allow an algorithm to *preempt* a job, halting its execution and continuing it later, possibly on a different machine. We also consider the situation where each job must be completed by a certain *deadline* (otherwise the algorithm fails). If jobs have deadlines, the deadline of job J_i is denoted by d_i. Furthermore, we will also consider *related* machines, where each machine has a speed which determines how long it takes to complete a job: on a machine with speed s, a job of size w completes in w/s time.

We will discuss several criteria by which machine scheduling algorithms can be measured. This is first of all the maximum completion time $\max c_i$, the time at which the last job completes. This is also known as the *makespan*. In the case that jobs arrive in a list in stead of over time, the problem of minimizing the maximum makespan is equivalent to minimizing the maximum load over all the machines: *load balancing*. Here a job size does not represent the time that the job is

running, but rather the amount that this job adds to the load of a machine when it is assigned to it.

We also consider the problem of minimizing the maximum starting time $\max S_i$ of jobs arriving in a list. Here the jobs are assigned to machines while the list is processed, and then the jobs are run on the machines in the order in which they were assigned to them.

For problems where jobs arrive over time, we also consider the total completion time $\sum c_i$ and the total flow time $\sum f_i$. The last measure is applicable to systems where the load is proportional to the total number of bits (data) that exist in the system over time (both of running jobs and of waiting jobs).

Finally, we will consider the total earnings criterion. Here a job J_i has to start immediately when it arrives, at time r_i, but J_i does not have to be scheduled completely and the benefit to the algorithm is the time it schedules a job. The algorithm may also reject a job completely.

Known results Minimizing the makespan for the case that the jobs arrive one by one (load balancing) was considered in a series of papers [41, 42, 9, 49, 3]. Graham [41] introduced the algorithm GREEDY. This algorithm schedules each arriving job on the least loaded machine. The load of a machine is the sum of the loads of the jobs that are assigned to it. Graham showed that GREEDY has a competitive ratio of $2 - 1/m$, which is optimal for $m = 2$ and $m = 3$. Currently, the best upper bound for general m is $1 + \sqrt{(1 + \ln 2)/2} \approx 1.920$ due to Fleischer and Wahl [36] and the best lower bound is 1.853 [40] based on Albers [3].

Chakrabarti, Phillips, Schulz, Shmoys, Stein and Wein [18] gave a 4-competitive algorithm for minimizing the total completion time on parallel machines when jobs arrive over time, while Vestjens [72] showed a lower bound of 1.309. For a single machine, Hoogeveen and Vestjens [43] gave a 2-competitive on-line algorithm and showed that it is optimal.

For minimizing the total flow time on a single machine, the competitive ratio is $\Theta(n)$ in the standard version of the problem (no preemptions). When preemptions are allowed, the algorithm SRPT gives an optimal schedule [56].

For parallel machines, Leonardi and Raz [59] gave a preemptive algorithm with competitive ratio of $O(\log \min(n/m, \Delta))$ which is optimal up to a constant factor, where Δ is the ratio between the largest and the smallest possible job size.

Scheduling under availability constraints In this problem, the machines on which jobs are to be scheduled are not available for use continuously. This has been studied in a number of papers (see the surveys [57, 64]). Most attention has gone to deterministic availability constraints, both for the case in which preemptions are allowed and the case in which they are not. Also the non-resumable case has been studied, in which a job must be restarted (losing all the work done on that job) if its machine goes down. The job cannot wait for the machine to become

available again. In [23], stochastic scheduling is studied: here the job sizes are stochastic and the availability periods of the machines are deterministic.

We consider a model that is opposite to stochastic scheduling. We have a collection of machines. A machine may become unavailable, but the scheduler does not know in advance when this will happen. A job J needs to be scheduled on at most one machine at a time. At the start, the scheduler must pick one machine to run the job on. If the machine becomes temporarily unavailable, the scheduler is allowed to move the job and restart it from scratch on a different machine. The goal is to minimize the expected completion time of the job. This was mentioned as an open problem in [64].

We study the case where the availability of the machines is captured by a Markov chain. This has similarities with the modeling in [51], where the paging problem (see section 1.1) was addressed in a similar way, i. e. by modeling the behaviour of the program that requests the pages by a Markov chain.

Bin packing This is one of the oldest and most well-studied problems in computer science [25, 28]. In this problem, we receive a sequence σ of pieces $p_1, p_2, \ldots, p_n$. Each piece has a fixed size in $(0, 1]$. We have an infinite number of bins each with capacity 1. Each piece must be assigned to a bin. Further, the sum of the sizes of the pieces assigned to any bin may not exceed its capacity. The goal is to minimize the number of bins used.

Known results The classical on-line bin packing problem, where the pieces arrive on-line (one by one), was first investigated by Johnson [44, 45]. He showed that the NEXT FIT algorithm has a competitive ratio of 2. Subsequently, it was shown by Johnson, Demers, Ullman, Garey and Graham that the FIRST FIT algorithm has a competitive ratio of $\frac{17}{10}$ [46]. Yao presented REVISED FIRST FIT and showed that it has a competitive ratio of $\frac{5}{3}$, and further showed that no on-line algorithm has a competitive ratio less than $\frac{3}{2}$ [75].

Ramanan, Brown, Lee and Lee [62] designed the algorithm MODIFIED HARMONIC and showed it has a competitive ratio of approximately 1.616. Currently, the best known algorithm is due to Seiden [66] and has a competitive ratio of at most 1.58889. Brown and Liang independently improved the lower bound to 1.53635 [17, 60]. This was subsequently improved by Van Vliet to 1.54014 [71]. Chandra [19] shows that the preceding lower bounds also apply to randomized algorithms.

1.3 Extensions to competitive analysis

A disadvantage of competitive analysis is that it sometimes gives an unrealistically bad impression of an algorithm, in that algorithms that perform well in practice have a high competitive

ratio. Furthermore, it sometimes fails to differentiate between algorithms whose performance is observed empirically to be very different. An example of both of these shortcomings is given by the paging algorithms LRU and FIFO. These both have a competitive ratio of k, where k is the number of pages in the system, although LRU performs much better than FIFO and much better than k times optimal in practice. A more sophisticated analysis using access graphs was needed to differentiate between these algorithms [24].

We discuss two extensions to the standard competitive analysis.

Randomization Perhaps the most important and best-known extension is allowing the on-line algorithm to use random bits in its decision making. In this case we can only consider expectations of costs or benefits. This can improve the competitive ratio markedly. The competitive ratio for cost minimization problems is defined as

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\mathbb{E}(\mathcal{A}(\sigma))}{\text{OPT}(\sigma)}$$

and an analogous definition can be given for maximization problems.

In the analysis of randomized algorithms, it is possible to distinguish between three kinds of adversaries. For a detailed discussion, see e. g. [13]. We will use only the most common *oblivious* adversary, which does not know the (subsequent and actual) actions of the on-line algorithm and constructs the request sequence in advance, knowing only the description of the on-line algorithm.

Resource augmentation This is a general method to circumvent the shortcomings mentioned above, introduced in 1995 by Kalyanasundaram and Pruhs [47]. They compare an on-line scheduling algorithm that has machines of speed $s > 1$ to an off-line algorithm that has machines of speed 1; this means that the on-line algorithm completes each job in $1/s$ the time that it takes the off-line algorithm to run it. For certain scheduling problems with unbounded competitive ratio, they show that it is possible to attain a good competitive ratio if s is slightly larger than 1.

Resource augmentation has been applied to a number of problems. It was already used in 1985, in the paper where the competitive ratio was introduced [70]: here the performance of some paging algorithms was studied, where the on-line algorithm has more memory than the optimal off-line algorithm has. Since 1997, in several machine scheduling and load balancing problems [12, 32, 47, 48, 55, 61] the effect of adding more or faster machines has been studied.

An important case is the case where the on-line algorithm has some constant number ℓ of machines for every machine that the off-line algorithm has. Such on-line algorithms are also called ℓ-machine algorithms, since they use ℓ times as many machines as the optimal off-line

algorithm. Similarly, an algorithm that uses the same number of machines as the off-line algorithm, but uses machines which are $s > 1$ times faster, is called an s-speed algorithm.

For a job sequence σ and an on-line algorithm $\mathcal{A}$, denote the total cost of σ (using a certain cost criterion) in the schedule of $\mathcal{A}$ on M machines by $\mathcal{A}_M(\sigma)$. Denote the optimal total cost for σ on m machines by $\text{OPT}_m(\sigma)$. Then the competitive ratio using resource augmentation is defined as

$$\mathcal{R}_{m,M}(\mathcal{A}) = \sup_{\sigma} \frac{\mathcal{A}_M(\sigma)}{\text{OPT}_m(\sigma)},$$

and an analogous definition can be given for maximization problems.

For the problem of minimizing the makespan, Brehob et al [16] showed in 2000 that no matter how many machines the on-line algorithm has, it can never perform optimally: $\mathcal{R}_{m,M}(\mathcal{A}) > 1$ for all $M > m \geq 2$. However, one may expect that for reasonable algorithms $\mathcal{R}_{m,M}(\mathcal{A})$ would approach 1 when $\gamma = M/m$ increases. In fact, [16] showed that GREEDY has a competitive ratio which approaches 1 in a rate depending linearly on $1/\gamma$.

Phillips, Stein, Torng and Wein [61] study the problem of minimizing the total flow time using resource augmentation. They give algorithms with augmentation on the number of machines. These are an $O(\log n)$-machine algorithm (which has a competitive ratio $1+o(1)$) and an $O(\log \Delta)$-machine algorithm (which achieves the competitive ratio 1), where Δ is the maximum ratio between job sizes. Both algorithms are valid for every m.

1.4 Outline of the thesis

Having discussed the topics of this thesis, we now give an outline of the thesis and its main results.

In chapter 2, we consider the problem of minimizing the total flow time. On a single machine, the competitive ratio of this problem is $\Theta(n)$. Examining the effect of resource augmentation for this problem, we give an algorithm with competitive ratio $O(n^{1/\gamma})$ for the case that the off-line algorithm has a single machine and show that no algorithm can do better. Here γ is the ratio between the number of on-line and the number of off-line machines.

In Chapter 3, we consider the effect of resource augmentation on the problem of minimizing the makespan in on-line scheduling when jobs arrive one by one. We show that the competitive ratio of an on-line algorithm decays at best exponentially in γ, and we give an algorithm that achieves such a competitive ratio.

In Chapter 4 we consider scheduling with availability constraints, where the availability of the machines is modeled using Markov chains. The objective is to minimize the expected completion time of a job. For several types of Markov chains, we present elegant and optimal policies.

Next, in Chapter 5 we consider the problem of scheduling jobs that do not have to be scheduled completely: the scheduler also gains something for jobs that are not completely scheduled. However, jobs must be executed immediately when they arrive, or not at all. We adjust a known algorithm for another problem to this problem for the case that jobs can have different sizes, and present a new algorithm and a lower bound for the case where all jobs have the same size.

In Chapter 6, we consider the problem of minimizing the maximum starting time of jobs arriving in a list. We show that the greedy algorithm has a competitive ratio which is logarithmic in the number of machines, and give a constant competitive algorithm. We also give tight bounds on the amount of resource augmentation (in the form of extra machines) required to obtain a competitive ratio of 1.

The last part of this thesis, Chapter 7, considers variable-sized bin packing, where the on-line algorithm has several bin sizes that it can choose from and tries to minimize the total size of the used bins. We present the first lower bounds for this problem and give algorithms that are better than VARIABLE HARMONIC [27].

An overview of the most important notations is given in Table 1.1. An overview of where the chapters in this thesis were previously published is given on page 113.

Table 1.1: An overview of the notation

$\mathcal{A}$	an on-line algorithm
σ	input sequence for the algorithm, e. g. a job sequence
$\mathcal{A}(\sigma)$	cost or benefit of algorithm $\mathcal{A}$ on input σ
OPT	optimal off-line algorithm
$\mathcal{R}(\mathcal{A})$	competitive ratio of $\mathcal{A}$
n	number of items in σ
m	number of machines (or number of off-line machines)
γ	ratio between number of on-line and number of off-line machines
Δ	ratio between biggest and smallest possible job size
M	number of on-line machines (if not equal to m)
J_i	the i-th job
r_i	its release time
w_i	its size or weight
d_i	its deadline
f_i	its flow time
c_i	its completion time
S_i	its starting time

Chapter 2

Minimizing the total flow time using resource augmentation

We consider the problem of minimizing the total flow time using resource augmentation. We design an algorithm of competitive ratio $O(\min(\Delta^{1/\gamma}, n^{1/\gamma}))$ if it is compared to an off-line algorithm that has only one machine. Here Δ is the maximum ratio between two job sizes, n is the number of jobs in the sequence and γ is the ratio between the number of on-line and the number of off-line machines. (Thus in this chapter, γ is the number of off-line machines.) Furthermore, we provide a lower bound which shows that the algorithm is optimal up to a constant factor for any constant γ. The algorithm works for a hard version of the problem where the sizes of the smallest and the largest jobs are not known in advance, only Δ is known. Our results give an exact trade-off between the resource augmentation and the competitive ratio.

2.1 Introduction

In this chapter, we consider on-line scheduling, where the goal is to minimize the total flow time. This is a well-known and hard problem, which has been studied widely both in on-line and in off-line environments [7, 52, 59]. Recall that the flow time f_j of a job J_j is defined as the difference between its completion time c_j and its release time r_j, which is the time at which J_j arrives. The *total flow time* is the sum of the flow times of all the jobs in a sequence. Note that the problem of minimizing the total flow time is equivalent to minimizing the avarage flow time, since they only differ by a factor of n, the number of jobs in the sequence.

We give an on-line algorithm LEVELS that uses resource augmentation and has a competitive ratio of $\mathcal{R}_{1,\gamma}(\text{LEVELS}) = O(\min(n^{1/\gamma}, \Delta^{1/\gamma}))$, where n, γ and Δ are defined as in Table 1.1; for a definition of $\mathcal{R}_{1,\gamma}(\mathcal{A})$, where $\mathcal{A}$ is an on-line algorithm, see Section 1.3. We also show

that for all on-line algorithms $\mathcal{A}$ and number m_1 of off-line machines we have $\mathcal{R}_{m_1,\gamma}(\mathcal{A}) = \Omega\left(\frac{\min(n^{1/\gamma},\Delta^{1/\gamma})}{(12\gamma)^\gamma}\right)$. Note that this is independent of m_1.

This shows that LEVELS is optimal up to a constant factor for any constant γ against an adversary on one machine. Furthermore, LEVELS works for a hard version of this problem where the sizes of the smallest and the largest jobs are not known in advance; only Δ is known in advance. The algorithm in [61] for minimizing the total flow time works only if the job size limits are known in advance.

In [39], a related problem on a network of links is considered. It immediately follows from our lower bounds, that any constant competitive algorithm has a polylogarithmic number of machines. More precisely, if $\mathcal{A}$ has a constant competitive ratio and γm machines, while OPT has m machines, we have $\gamma = \Omega\left(\frac{\sqrt{\log(\min(n,\Delta))}}{m\sqrt{\log\log(\min(n,\Delta))}}\right)$. This result can also be deduced from Theorem 10 in [39]. However, using their proof for the general lower bound would give only an exponent of $\frac{1}{2\gamma}$. Improving the exponent to be the tight exponent $\frac{1}{\gamma}$ is non-trivial. Our results imply that by choosing the amount of resource augmentation, the competitive ratio is fixed.

We also adapt the lower bound for the case where the on-line algorithm has faster machines than the off-line algorithm. This results in a lower bound of $\Omega(n^{1/(2m^2)})$ on the speed of on-line machines necessary to obtain a constant competitive ratio, if $\gamma = 1$ and m is the number of (off-line and on-line) machines.

We also consider the following scheduling problem studied in [31, 61]. Each job J_j has a deadline d_j. Instead of minimizing the flow time, we require that each job is finished by its deadline, effectively limiting the flow time of job J_j to $d_j - r_j$. The goal is to complete all jobs on time. For this problem, we give lower bounds on the speed and the number of machines required for a non-preemptive on-line algorithm to succeed on any sequence.

2.2 Algorithm LEVELS

We begin this section with two preliminary lemmas. We have the following results for the case where n is not known and the case where OPT has the same number of machines as the on-line algorithm.

Lemma 2.1 *For all on-line algorithms $\mathcal{A}$ that do not know the number n of jobs in advance, we have $\mathcal{R}_{1,m}(\mathcal{A}) = \Omega(n)$.*

Proof We use a number $N >> 1$.

One job of size 1 arrives at time 0. When it is started, N jobs of size $1/N$ arrive with intervals of $1/N$ during the next 1 time unit. If they are all delayed until time 1, no more jobs arrive and we are done. The optimal flow time on one machine is 3 and the on-line flow time is $\Theta(N)$.

On the other hand, if one of those jobs is started while the first job is running, N jobs of size $1/N^2$ arrive with intervals of $1/N^2$ during the next $1/N$ time. Depending on the decision by the on-line algorithm, we continue in this way or stop as soon as it delays N jobs (or reaches the last machine).

When all machines are in use, the on-line algorithm will have a flow time of $\Theta(N)$. □

Lemma 2.2 *For all on-line algorithms $\mathcal{A}$, $\mathcal{R}_{m,m}(\mathcal{A}) = \Omega(n/m^2)$.*

Proof A single unit job arrives at time 0. Let t be the time at which $\mathcal{A}$ starts this job. Let $x = \frac{m}{2(n-1)}$. For $i = 0, \ldots, \frac{n-1}{m}$, m jobs of length x are released at time $t + ix$. It is easy to see that the optimal total flow time is $\Theta(m)$ whereas the flow time of $\mathcal{A}$ will be $\Omega(n/m)$. Consequently, $\mathcal{R}_{m,m}(\mathcal{A}) = \Omega(n/m^2)$. □

These negative results motivate our study of algorithms that know n and have $\gamma > 1$ as many machines as the off-line algorithm. We will now define such an algorithm, called LEVELS. LEVELS uses a priority queue Q_i and a variable D_i for each machine $1 \leq i \leq \gamma$. We initialize $Q_i = \emptyset$ and $D_i = 0$ ($i = 1, \ldots, \gamma$).

The idea of LEVELS is to put jobs that are about the same size on the same machine, and jobs that are (much) smaller on other machines. The reason for this is that it does not matter very much for the total (or average) flow time if a job has to wait for similar-sized or smaller jobs to complete, but it is important that jobs should not have to wait too long for jobs that are larger than they are. This should be avoided as much as possible.

Since LEVELS has γ machines at its disposal, it tries to put jobs with sizes that are less than a factor of $\lambda = n^{1/\gamma}$ apart together on the same machine. Recall that n is the number of jobs. The variables D_i are maintained to determine on which machine a new job should go, based on its size.

An *event* is either an arrival of a new job or a completion of a job by a machine.

Algorithm LEVELS

- If a few events occur at the same time, the algorithm first deals with all arrivals before it deals with job completions.
- On completion of a job on machine i, if $Q_i \neq \emptyset$, a job of minimum release time among jobs with minimum processing time in Q_i is scheduled immediately on machine i. (The job is dequeued from Q_i.)
- On arrival of job J_j, let i be a minimum index of a machine for which $D_i \leq \lambda w_j$. If there is no such index, take $i = \gamma$. If machine i is idle, J_j is immediately scheduled on machine i, and otherwise, J_j is enqueued into Q_i. If $w_j > D_i$, D_i is modified by $D_i \leftarrow w_j$.

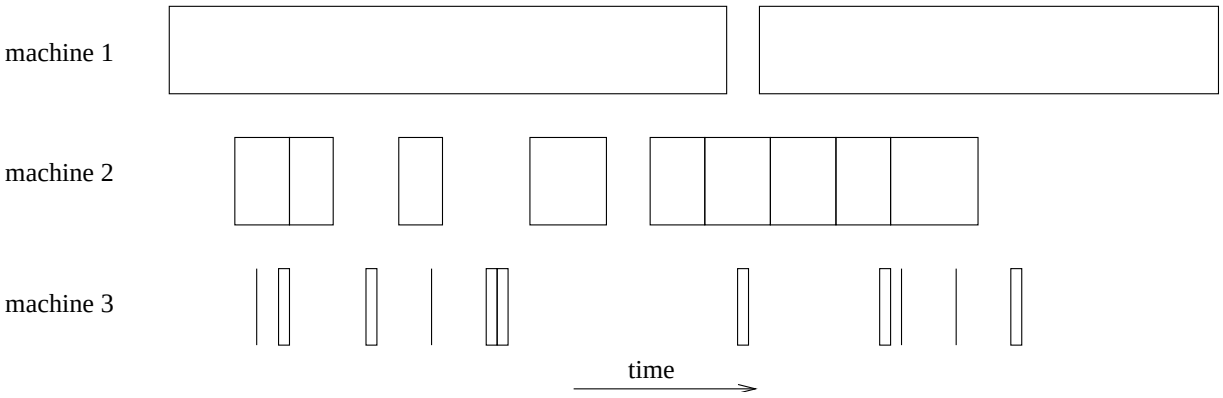

Figure 2.1: Example of a schedule created by LEVELS

Figure 2.1 shows the schedule that LEVELS creates on 3 machines for an example job sequence. We analyze the performance of LEVELS compared to a preemptive off-line algorithm OPT on a single machine. This also gives us an upper bound of LEVELS against a nonpreemptive OPT.

Denote the schedule of LEVELS by π. Partition the schedule of each machine into blocks. A block is a maximal sub-sequence of jobs of non-decreasing sizes, that run on one machine consecutively, without any idle time. We now introduce some notation.

- Let N_i be the number of blocks in the schedule of LEVELS on machine i.
- Let $B_{i,k}$ be the k^{th} block on machine i.
- Let $b(i,k,j)$ be the index of the j^{th} job in block $B_{i,k}$.
- Let $N_{i,k}$ be the number of jobs in $B_{i,k}$.
- Let $W_{i,k}$ be the size of the largest job in blocks $B_{i,1},\ldots,B_{i,k}$ i.e.

$$\begin{aligned} W_{i,k} &= \max_{1\le h\le k}\ \max_{1\le j\le N_{i,h}} w_{b(i,h,j)} \\ W_{i,0} &= 0 \quad \text{for all } 1\le i\le\gamma. \end{aligned}$$

- Let $I=\bigcup_{1\le i\le\gamma,1\le k\le N_i} B_{i,k}$, i.e. I is the set of all jobs.

Similarly to the proof in [43], we define a pseudo-schedule ψ on γ machines, in which job $J_{b(i,k,j)}$ is scheduled on machine i at time $S_{b(i,k,j)}(\pi)-W_{i,k-1}$. Note that ψ is not necessarily a valid schedule, since some jobs might be assigned to the same machine at the same time, and some jobs may start before their arrival times.

The amount that jobs are shifted backwards increases with time. Therefore, if there is no idle time between jobs in π, there is no idle time between them in ψ either. Note that in ψ, the flow time of a job J_j can be smaller than w_j, and even negative.

To analyze the performance of LEVELS, we introduce an *extended flow problem*. Each job J_j has two parameters r_j and r'_j, where $r'_j \leq r_j$. r'_j is the pre-release time of job J_j. Job J_j may be assigned to a machine starting from time r'_j. The flow time is still defined by the completion time minus the release time, i.e. $f_j = c_j - r_j$. Changing an input σ for the original problem into an input σ' of the extended problem, requires a definition of the values of r'_j for all jobs. Clearly, the optimal total flow time for an input σ' of the extended problem is no larger than the optimal total flow time of σ in the original problem.

Let I_i be the set of jobs that run on machine i in π. We define an instance I'_i for the extended problem. I'_i contains the same jobs as I_i. For each $J_j \in I_i$, r_j remains the same. Define $r'_j = \min\{r_j, S_j(\psi)\}$. Clearly $\text{OPT}(I) \geq \sum_{i=1}^{\gamma} \text{OPT}(I_i) \geq \sum_{i=1}^{\gamma} \text{OPT}(I'_i)$, where $\text{OPT}(I_i)$ is the preemptive optimal off-line cost for the jobs that LEVELS scheduled on machine i. We consider a preemptive optimal off-line schedule ϕ_i for I'_i on a single machine. In ϕ_i, jobs of equal size are completed in the order of arrival. Ties are broken as in π, the schedule of LEVELS. The following lemma is similar to [43].

Lemma 2.3 *For each job $J_j \in I'_i$, $f_j(\phi_i) \geq f_j(\psi)$.*

Proof Since the release times of the jobs are the same in ψ and ϕ, we only have to show that in ϕ_i, no job starts earlier than it does in ψ. Assume to the contrary this is not always the case. Let J_j be the first job in ϕ_i for which $S_j(\phi_i) < S_j(\psi)$. Note that in this case $r'_j < S_j(\psi)$ and hence $r_j < S_j(\psi)$. Let t be the end of the last period of idle time before $S_j(\psi)$, and let $B_{i,k}$ be the block that contains J_j.

Suppose $W_{i,k-1} \leq w_j$. Then all jobs that run on machine i from time t until time $S_j(\psi)$ in ψ are either smaller than w_j or have the same size, but are released earlier. Moreover, these jobs do not arrive earlier than time t, hence in ϕ_i they do not run before time t. They do run before $S_j(\phi_i)$ because they have higher priority, hence $S_j(\phi_i) \geq S_j(\psi)$, a contradiction.

Suppose $W_{i,k-1} > w_j$. J_j was available to be run in ψ during the interval $[r_j, S_j(\psi)]$ since $r_j < S_j(\psi)$. In π, all jobs running in the interval $[r_j, S_j(\pi)]$ are smaller than J_j (or arrived before, and have the same size), except for the first one, say $J_{j'}$. Since in ψ, all these jobs are shifted backwards by at least the size of $J_{j'}$, during $[r_j, S_j(\psi)]$ only jobs with higher priority than J_j are run in ψ. J_j is the first job which starts later in ψ than it does in ϕ_i, so these jobs occupy the machine until time $S_j(\psi)$, hence $S_j(\psi) \leq S_j(\phi_i)$. □

We are now ready to prove the main result of this chapter.

Theorem 2.1 $\mathcal{R}_{1,\gamma}(\text{LEVELS}) = O(n^{1/\gamma})$.

Proof Using Lemma 2.3 we can bound the difference in total flow time between ψ and π.

Since $\text{LEVELS}(J_{b(i,k,j)}) = C_{b(i,k,j)}(\psi) + W_{i,k-1} - r_{b(i,k,j)}$, we have

$$\begin{aligned}
\text{LEVELS}(I) &= \sum_{i=1}^{\gamma}\sum_{k=1}^{N_i}\sum_{j=1}^{N_{i,k}} \left(C_{b(i,k,j)}(\psi) + W_{i,k-1} - r_{b(i,k,j)}\right) \\
&\leq \sum_{J_{b(i,k,j)}\in I} \left(C_{b(i,k,j)}(\psi) - r_{b(i,k,j)}\right) + \sum_{J_{b(i,k,j)}\in I} W_{i,k-1} \\
&\leq \text{OPT}(I) + \sum_{J_{b(i,k,j)}\in I} W_{i,k-1}.
\end{aligned}$$

Let W the maximum job size. We show the following properties:

$$W_{i,k-1} \leq \lambda w_{b(i,k,j)} \qquad \text{for each job } J_{b(i,k,j)},\ 1 \leq i \leq \gamma - 1 \tag{2.1}$$

$$W_{\gamma,k-1} \leq \frac{W}{\lambda^{\gamma-1}} \qquad \text{for each job } J_{b(\gamma,k,j)} \tag{2.2}$$

Adding both properties together we get

$$\begin{aligned}
\text{LEVELS}(I) &\leq \text{OPT}(I) + \sum_{\substack{J_{b(i,k,j)}\in I \\ i\neq\gamma}} W_{i,k-1} + \sum_{J_{b(\gamma,k,j)}\in I} W_{\gamma,k-1} \\
&\leq \text{OPT}(I) + \lambda \sum_{J_{b(i,k,j)}\in I} w_{b(i,k,j)} + \sum_{J_{b(\gamma,k,j)}\in I} \frac{W}{\lambda^{\gamma-1}} \\
&\leq \text{OPT}(I) + \lambda\text{OPT}(I) + n \cdot \frac{\text{OPT}(I)}{n^{(\gamma-1)/\gamma}} = (2\lambda+1)\cdot\text{OPT}(I)
\end{aligned}$$

This holds since $\text{OPT}(I) \geq W$ and $\text{OPT}(I)$ is at least the sum of all job sizes, and since $|I| = n$.

To prove (2.1) we recall that $J_{b(i,k,j)}$ was assigned to machine i because it satisfied $D_i \leq \lambda w_{b(i,k,j)}$. If $W_{i,k-1} \leq w_{b(i,k,j)}$ we are done. Otherwise the job of size $W_{i,k-1}$ arrived before $J_{b(i,k,j)}$ and hence when $J_{b(i,k,j)}$ arrived, D_i satisfied $D_i \geq W_{i,k-1}$, hence $W_{i,k-1} \leq D_i \leq \lambda w_{b(i,k,j)}$.

To prove (2.2) we show by induction that every job J_j on machine i in LEVELS satisfies $w_j \leq W/\lambda^{i-1}$. This is trivial for $i = 1$. Assume it is true for some machine $i \geq 1$, then at all times $D_i \leq W/\lambda^{i-1}$ holds. Hence, a job J' that was too small for machine i satisfied $w' \leq D_i/\lambda \leq W/\lambda^i$. This completes the proof. □

We give a variant of LEVELS with a competitive ratio which depends on Δ, the ratio between the size of the largest job and the size of the smallest job.

Algorithm REVISED LEVELS: Run LEVELS with $\lambda = \Delta^{1/\gamma}$.

Theorem 2.2 $\mathcal{R}_{1,\gamma}(\text{REVISED LEVELS}) = O(\Delta^{1/\gamma})$.

Proof The proof is very similar to the proof of Theorem 2.1. The only difference in the proof is that property (2.1) also holds for machine γ (this follows from property (2.2) and the definition of Δ), hence the competitive ratio is now $\lambda + 1$. □

If we take $\lambda = \min(n^{1/\gamma}, \Delta^{1/\gamma})$ in the definition of LEVELS, Theorems 2.1 and 2.2 imply that it has a competitive ratio of $O(\min(n^{1/\gamma}, \Delta^{1/\gamma}))$.

2.3 Lower bounds

We show the following lower bound on the competitive ratio of this problem.

Theorem 2.3 *Let $\mathcal{A}$ be an on-line scheduling algorithm to minimize the total flow time on γ machines. Then for any $1 \leq m_1 \leq \gamma$ and sequences consisting of $\Theta(n)$ jobs, $\mathcal{R}_{m_1,\gamma}(\mathcal{A}) = \Omega\left(\frac{n^{1/\gamma}}{(12\gamma)^{\gamma-1}}\right)$.*

The idea of the lower bound is to let successively smaller jobs arrive, forcing $\mathcal{A}$ to start using a new machine each time, until all its machines are in use. Again, this works because $\mathcal{A}$ should try to avoid having jobs wait for jobs that are smaller than them. In fact we show that $\mathcal{A}$ should behave very similarly to LEVELS on this job sequence.

We first describe a job sequence $\sigma_{\mathcal{A}}$, that depends on $\mathcal{A}$, and then show that it implies the theorem. Let n be an integer. There will be at most $\sum_{i=0}^{\gamma} n^{i/\gamma}$ jobs in $\sigma_{\mathcal{A}}$ (note $\sum_{i=0}^{\gamma} n^{i/\gamma} = \Theta(n)$). We build $\sigma_{\mathcal{A}}$ recursively, defining the jobs according to the behavior of the on-line algorithm $\mathcal{A}$.

Definition A job J of size w is considered *active*, if the previous active job of size w is completed by $\mathcal{A}$ at least w units of time before J is assigned, and J finishes no later than the job that caused its arrival.

Note that this definition depends on $\mathcal{A}$. The first job in $\sigma_{\mathcal{A}}$ has size n and arrives at time 0. We consider it to be an active job. On an assignment of a job J of size w by $\mathcal{A}$, do the following:

- If J is active, and all other machines are running larger jobs (all machines are consequently busy for at least w units of time), n jobs of size 0 arrive immediately. No more jobs will arrive.
- Otherwise, if J is active, then J causes the arrival of $n^{1/\gamma}$ jobs of size $\frac{1}{3} \cdot \frac{w}{n^{1/\gamma}}$. These jobs arrive starting the time that J is assigned, every $\frac{w}{n^{1/\gamma}}$ units of time, until they all have arrived.
- In all other cases (J is not active), no jobs arrive till the next job that $\mathcal{A}$ starts.

We give an example of a job sequence for the case that $\mathcal{A}$ schedules all arriving jobs immediately on 3 machines in Figure 2.2. Only the nonzero jobs are shown.

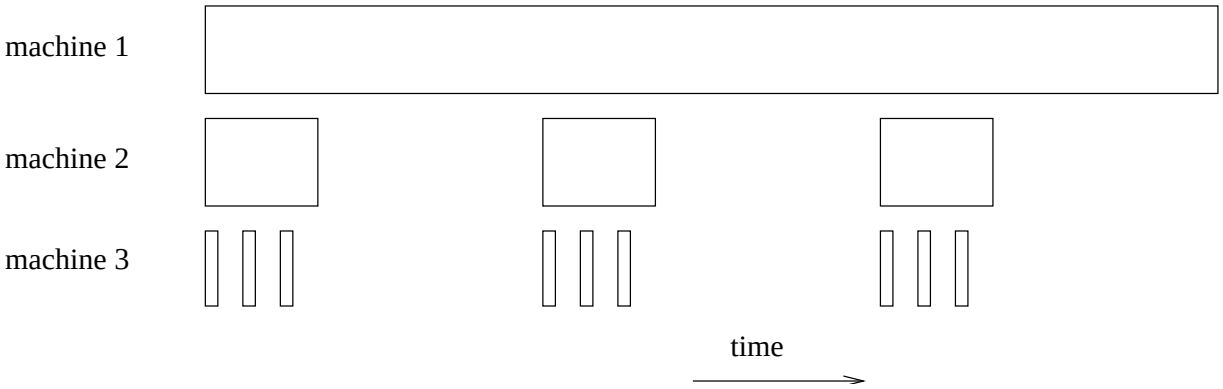

Figure 2.2: An example job sequence for the lower bound

Lemma 2.4 $\text{OPT}(\sigma_{\mathcal{A}}) \leq 6n$, *even if* OPT *only has a single machine.*

Proof We show that all jobs can be assigned on a single machine, during an interval of length $2n$, so that a job of length w has a flow time of at most $3w$. The total flow time then follows.

We show how to assign all jobs of a certain size w so that no active jobs of size w are running at the same time on on-line machines, i. e. the intervals used by $\mathcal{A}$ to run active jobs of size w, and the intervals that are used by OPT to run jobs of size w, are disjoint. Smaller jobs are assigned by OPT during the intervals in which $\mathcal{A}$ assigned active jobs of size w. Hence, the time slots given by the optimal off-line for different jobs are disjoint.

Finally, we show how to define those time slots. A job J of size w, that arrives at time t, is not followed by other jobs of size w until time $t + 3w$. Since an active on-line job starts at least w units of time after the previous active job of this size (w) is completed, there is a time slot of size at least w during the interval $[t, t + 3w]$ where no active job of size w is running on any of the on-line machines. The optimal off-line algorithm can assign J during that time. This is true also for the first job. Finally, the optimal algorithm can also manage the jobs of size 0 easily by running them immediately when they arrive. Hence, the total time that the optimal off-line machine is not idle is at most $2n$. □

Having described the construction of the sequence $\sigma_{\mathcal{A}}$ and its optimal total flow time, we now derive a bound on $\mathcal{A}(\sigma_{\mathcal{A}})$. We partition jobs into three types, according to the on-line assignment. A job that arrived during the processing of a job of size w, and has size $\frac{1}{3}\frac{w}{n^{1/\gamma}}$ is either active or *passive* (if it is not active, but completed before the job of size w is completed). Otherwise, the job is called *late*. Let $P(w), T(w)$ and $L(w)$ denote the number of passive, active and late jobs of size w (respectively). Let $N(w) = P(w) + T(w) + L(w)$.

Claim 2.1 $T(w) \geq \lceil \frac{1}{2\gamma}(P(w) + T(w)) \rceil$.

Proof The number of jobs of size w that the on-line algorithm can complete during $2w$ units of time (until a job can be active again) is at most 2γ. □

Now we are ready to prove Theorem 2.3.

Proof (Of Theorem 2.3.) According to the definition of the sequence, $N(w) = n^{1/\gamma} \cdot T(3wn^{1/\gamma})$. We distinguish two cases.

Case 1. In all phases $L(w) \leq \frac{1}{2}N(w)$. Hence $T(w) \geq \frac{1}{4l}N(w)$ for all w. This is true for $w = (\frac{1}{3})^{\gamma-1}n^{1/\gamma}$ (the smallest non-zero jobs) and hence there are at least $n^{\gamma-1/\gamma}(\frac{1}{4\gamma})^{\gamma-1} > 0$ such jobs. Therefore the zero jobs arrive and are delayed by at least $(\frac{1}{3})^{\gamma-1}n^{1/\gamma}$ units of time. Since their flow time is at least $n \cdot n^{1/\gamma}(\frac{1}{3})^{\gamma-1}$, and the optimal total flow time is at most $6n$, the competitive ratio follows.

Case 2. There is a phase where $L(w) > \frac{1}{2}N(w)$. Consider the phase with largest w in which this happens. Since for larger sizes w' we have $L(w') \leq \frac{1}{2}N(w')$, we can bound the number of jobs of size w (for $w = (\frac{1}{3})^i n^{1-i/\gamma}$) by $N(w) \geq n^{i/\gamma}(\frac{1}{4\gamma})^i$. The late jobs are delayed by at least $\frac{1}{4} \cdot 3wn^{1/\gamma}$ on average. (This is the delay if for each job of size $3wn^{1/\gamma}$ the *last* $\frac{1}{2}n^{1/\gamma}$ jobs of size w that arrive are the ones that are late; in all other cases, the delay is bigger.)

The total flow time is at least

$$\begin{aligned}
\mathcal{A}_l(\sigma_\mathcal{A}) &\geq L(w) \cdot \frac{1}{4} \cdot 3wn^{1/\gamma} \geq \frac{1}{2}n^{i/\gamma}\left(\frac{1}{4\gamma}\right)^i \frac{1}{4}\left(\frac{1}{3}\right)^{i-1} n^{1-i/\gamma+1/\gamma} \\
&= \frac{1}{(4\gamma)^i} \cdot \frac{1}{8}\left(\frac{1}{3}\right)^{i-1} \cdot n^{1+1/\gamma} = \left(\frac{1}{12\gamma}\right)^i \frac{3}{8}n^{1+1/\gamma} \\
&\geq \frac{1}{(12\gamma)^{\gamma-1}} \cdot \frac{3}{8} \cdot n^{1+1/\gamma} = \Omega\left(\frac{n^{1/\gamma}}{(12\gamma)^{\gamma-1}}\right) \cdot \Theta(n).
\end{aligned}$$

Since the optimal total flow time is $\Theta(n)$, the competitive ratio follows. □

Theorem 2.4 *Let $\mathcal{A}$ be an on-line scheduling algorithm to minimize the total flow time on γ machines. Then $\mathcal{R}_{m_1,\gamma}(\mathcal{A}) = \Omega\left(\frac{\Delta^{1/\gamma}}{(12\gamma)^\gamma}\right)$ for any $1 \leq m_1 \leq \gamma$ if the maximum ratio between job sizes is Δ.*

Proof We adjust $\sigma_\mathcal{A}$ by starting with a job of size Δ and fixing $n = \Delta/3^\gamma$. We assume $\Delta \geq 6^\gamma$ so that $n \geq 2^\gamma$ and $n^{1/\gamma} \geq 2$, which is needed for the construction of the sequence.

Starting from here, we build a sequence $\sigma'_\mathcal{A}$ in exactly the same way as $\sigma_\mathcal{A}$, except that we do not let jobs of size 0 arrive. Clearly, $\text{OPT}_1(\sigma'_\mathcal{A}) \leq 6\Delta$. We can follow the proof of Theorem 2.3. However, in this case we know that all the smallest jobs will be late. If they arrive we are in the second case of the proof; but if they do not, then for an earlier w we must have $L(w) > \frac{1}{2}N(w)$. So only Case 2 remains of that proof.

The total flow is at least $\frac{3}{8}\frac{n^{1/\gamma}}{(12\gamma)^\gamma}\Delta = \frac{1}{8}\frac{\Delta^{1/\gamma}}{(12\gamma)^\gamma}\Delta$ (because now $i \leq \gamma$ instead of $i \leq \gamma - 1$), giving the desired competitive ratio. □

A direct consequence of Theorems 2.3 and 2.4 is the following bound on the number of machines needed to maintain a constant competitive ratio. This corollary can also be proved using a simple adaptation of Theorem 10 in [39].

Corollary 2.1 *Any on-line algorithm for minimizing the total flow time on m machines that has a constant competitive ratio using resource augmentation, is an $\Omega\left(\frac{\sqrt{\log(\min(n,\Delta))}}{m\sqrt{\log\log(\min(n,\Delta))}}\right)$-machine algorithm (on sequences of $\Theta(n)$ jobs).*

Next we consider resource augmentation on the speed as well as on the number of machines. We consider an on-line algorithm which uses machines of speed $s > 1$. The optimal off-line algorithm uses machines of speed 1.

Theorem 2.5 *Let $\mathcal{A}$ be an on-line scheduling algorithm to minimize the total flow time on γ machines. Let $s > 1$ be the speed of the on-line machines. Then $\mathcal{R}_{m_1,\gamma}(\mathcal{A}) = \Omega\left(\frac{n^{1/\gamma}}{s(12\gamma s^2)^{\gamma-1}}\right)$ for any $1 \leq m_1 \leq \gamma$ and sequences consisting of $O(n)$ jobs. Furthermore, $\mathcal{R}_{m_1,\gamma}(\mathcal{A}) = \Omega\left(\frac{\Delta^{1/\gamma}}{s(12\gamma s^2)^\gamma}\right)$ for any $1 \leq m_1 \leq \gamma$.*

Proof Again, we use a job sequence similar to $\sigma_{\mathcal{A}}$. The jobs of phase i now have size $1/(3s^2n^{1/\gamma})^i$. For the Δ-part of the proof, we fix $n = \Delta/(3s^2)^\gamma$. Similar calculations as in the previous proofs result in the stated lower bounds. □

Corollary 2.2 *Any on-line algorithm for minimizing the total flow time on m machines that has a constant competitive ratio using resource augmentation on the speed, is an $\Omega(n^{1/(2m^2)})$-speed algorithm (on sequences of $\Theta(n)$ jobs) and an $\Omega(\Delta^{1/(2m^2)})$-speed algorithm.*

2.4 Hard deadlines

The lower bounding method used in the previous section can also be applied to another problem. In this section, we consider the problem of non-preemptive scheduling of jobs with hard deadlines. Each arriving job J_j has a deadline d_j by which it must be completed. The goal is to produce a schedule, in which all jobs are scheduled such that all of them are completed on time (i.e. by their deadlines). We give a lower bound on the resource augmentation required so that all jobs finish on time. We allow the on-line algorithm resource augmentation in both the number of machines and their speed. We compare an on-line algorithm that schedules on γ machines of speed s to an optimal off-line algorithm that uses a single machine of speed 1.

Let Δ denote the ratio between the largest job in the sequence and the smallest job. The lower bound sequence consists of $\gamma + 1$ jobs $J_0, \ldots, J_\gamma$ where $w_i = 1/(2s+1)^i$. We define release times and deadlines recursively; $r_0 = 0$ and $d_0 = 2 + 1/s$. Let π be the on-line schedule, then $r_{i+1} = S_i(\pi)$ and $d_{i+1} = C_i(\pi)$. Hence J_{i+1} runs in parallel to all jobs $J_0, \ldots, J_i$ in any feasible schedule π.

Lemma 2.5 *An optimal off-line algorithm on a single machine of speed* 1 *can complete all jobs on time.*

Proof For each $i > 0$, $w_i = 1/(2s+1)^i$, hence $d_i - r_i = w_{i-1}/s = \frac{2s+1}{s} w_i$. This holds also for J_0, since $w_0 = 1$ and $d_0 - r_0 = \frac{2s+1}{s}$. All jobs arriving after J_i have release times and deadlines in the interval $[S_i(\pi), C_i(\pi)]$. The optimal off-line algorithm can schedule J_i outside this time interval, and avoid conflict with future jobs. By induction, previous jobs are scheduled before r_i or after d_i, so there is no conflict with them either. If $S_i(\pi) - r_i \geq w_i$, schedule J_i at time r_i. Otherwise $C_i(\pi) = S_i(\pi) + w_i/s < r_i + w_i(1 + 1/s)$, hence J_i is scheduled at time $C_i(\pi)$ and completed at $C_i(\pi) + w_i < r_i + w_i(2 + 1/s) = d_i$. □

It is easy to see that the on-line algorithm cannot finish all jobs on time. If the first γ jobs finish on time, then all γ machines are busy during the time interval $[r_\gamma, d_\gamma]$ and it is impossible to start J_γ before time d_γ. We have the following theorem.

Theorem 2.6 *The on-line algorithm fails, if* $\Delta \geq (2s+1)^\gamma$.

From the lower bound on Δ, one can derive several necessary conditions for an on-line algorithm to succeed on any sequence. Given machines of constant speed s, the number of machines γ must satisfy $\gamma \geq \frac{\log \Delta}{\log(2s+1)}$ i.e. $\gamma = \Omega(\log \Delta)$. On the other hand, for a constant number γ of machines, s has to satisfy $2s + 1 \geq \Delta^{1/\gamma}$, i.e. $s = \Omega(\Delta^{1/\gamma})$.

The lower bound on Δ clearly holds also for the case where the optimal off-line algorithm is allowed to use $m_1 > 1$ machines. Consider a k-machine algorithm that always succeeds in building a feasible schedule ($k = \gamma/m_1$), then k satisfies $k = \Omega(\log \Delta/m)$ for constant s. Finally, s satisfies $s = \Omega(\Delta^{1/mk})$ for constant k.

2.5 Conclusions

We have presented an algorithm for minimizing the flow time on γ identical machines with competitive ratio $O(\min(\Delta^{1/\gamma}, n^{1/\gamma}))$ against an optimal off-line algorithm on a single machine, and we have shown a lower bound of $\Omega\left(\frac{\min(n^{1/\gamma}, \Delta^{1/\gamma})}{(12\gamma)^\gamma}\right)$ on the competitive ratio of any algorithm, even against an adversary on one machine. For any given constant number of on-line machines γ, this gives an exact trade-off between the amount of resource augmentation and the competitive ratio.

An interesting remaining open problem is to find an algorithm which is optimally competitive against an off-line algorithm on a single machine for any γ.

Chapter 3

Resource augmentation in load balancing

We consider load balancing in the following setting. An on-line algorithm is allowed to use M machines, whereas the optimal off-line algorithm is limited to m machines, for some fixed $m < M$. We show that while GREEDY (section 1.2) has a competitive ratio which decays linearly in the inverse of M/m, the best on-line algorithm has a ratio which decays exponentially in M/m. Specifically, we give an on-line algorithm BUCKETS with competitive ratio of $1 + 2^{-\frac{M}{m}(1-o(1))}$, and we give a lower bound of $1 + e^{-\frac{M}{m}(1+o(1))}$ for randomized on-line scheduling to minimize the makespan, which can also be applied to our problem.

We also consider load balancing with temporary tasks. Here jobs arrive and depart at arbitrary times. We prove that for $M = m + 1$, GREEDY is optimal. (It is not optimal for permanent tasks, where jobs only arrive.)

3.1 Introduction

In the load balancing problem, jobs arrive on-line and have to be scheduled on identical parallel machines. Each job has a size (or weight) that represents the load that that job adds to the machine that it is assigned to. The goal is to minimize the maximum load over all the machines. For the *permanent tasks model*, in which jobs that have arrived do not leave, this is equivalent to on-line scheduling of jobs in a list, where job sizes represent running times of jobs and the goal is to minimize the *makespan*: the maximum completion time over all the jobs. However, we will also consider the *temporary tasks model*, where jobs can also leave. This is not equivalent to any scheduling problem of jobs with running times.

In the setting that we consider, the jobs arrive in a list, one by one (and not over time). The on-line algorithm has M identical machines, and it is compared to an optimal off-line algorithm which has $m < M$ identical machines. We write $\gamma = M/m$.

In the permanent tasks model, the load of a machine is the sum of the loads of the jobs that

are assigned to it; the load of an algorithm is the maximum load over all the machines in the schedule of that algorithm. In particular, the *optimal load* for a sequence is the maximum load over all the machines in the schedule of an optimal algorithm OPT for that sequence.

For this problem, GREEDY has a competitive ratio which tends to an inversely linear function in γ for $\gamma \to \infty$. In contrast, we design an algorithm of which the competitive ratio approaches 1 in a rate depending exponentially on γ. More specifically, we give an algorithm BUCKETS for $\gamma > 3$ of competitive ratio $1 + 2^{-\gamma(1-o(1))}$ for $\gamma \to \infty$.

We then show that it is not possible to do substantially better than BUCKETS by giving a lower bound for minimizing the makespan in on-line scheduling of jobs in a list, where the on-line algorithm is even allowed to preempt jobs, but the optimal off-line algorithm is not. Here each job may be assigned by the on-line algorithm to one or more machines and time slots, where the time slots have to be disjoint. The assignment has to be determined completely at the arrival of a job. Using similar techniques as in [20, 21, 67] we prove a lower bound of $1/(1-(\frac{m-1}{m})^M) = 1 + e^{-\gamma(1+o(1))}$ on the competitive ratio of any randomized preemptive algorithm. This lower bound holds a forteriori for deterministic and/or non-preemptive algorithms. Hence it also holds for our load balancing problem, which implies that no on-line load balancing algorithm can have a competitive ratio that decreases faster than exponentially in γ.

In the temporary tasks model, jobs arrive and depart at arbitrary times and the cost of an algorithm is the maximum load over time and machines. Jobs still arrive one by one but the arrival of a job can now also be followed by the departure of one or more jobs (one by one). It was proved in [8] that for $M = m$, GREEDY is optimal for this model. It is $(2 - 1/m)$-competitive. We show that if M is just slightly larger than m, i.e., $M = m + 1$, then GREEDY which is $(2 - 2/(m + 1))$-competitive is still optimal. We note that the results in [3] imply that for permanent tasks GREEDY is not optimal anymore for general $M > m$.

3.2 Permanent tasks

In this section, we consider the permanent tasks model, where jobs only arrive. We consider the behaviour of the competitive ratio of this problem as a function of $\gamma = M/m$. We start with the competitive ratio of GREEDY. The following lemma is shown in [16] using a similar analysis as in [41]:

Lemma 3.1 *The competitive ratio of* GREEDY *is* $1 + \frac{m-1}{M}$.

The above theorem implies a competitive ratio which tends to a linear function in $1/\gamma$ for $\gamma \to \infty$. Surprisingly, we can give an algorithm called BUCKETS which has a competitive ratio $1 + 2^{-\gamma(1-o(1))}$ for $\gamma > 3$.

3.2.1 Algorithm BUCKETS

We define an algorithm BUCKETS for load balancing of permanent tasks using resource augmentation, specifically for large γ ($\gamma > 3$). If $\gamma \leq 3$ we can take BUCKETS = GREEDY (without affecting our asymptotic analysis). The main idea of BUCKETS is to put "large" jobs on machines by themselves, on machines that have not been assigned "large" jobs earlier. Of course, whether a job is large or not depends on the other jobs that arrive before and after it, and ultimately on its size relative to the maximum (optimal) load. For this reason, BUCKETS maintains an estimate of the optimal maximum load and assigns the arriving jobs based on their size relative to this estimate. We denote the current estimate of the optimal maximum load by β.

Let $0 < \xi < 1$ be some parameter to be determined later. We partition the machines into buckets: $b = \lfloor \gamma - 2/\xi \rfloor$ small buckets, each of which contains m machines, and one big bucket that contains all other machines. Since the small buckets contain $\lfloor \gamma - 2/\xi \rfloor m$ machines, the big bucket contains at least $2m/\xi$ machines.

Denote by β_i the value of β after the arrival of i jobs. The algorithm consists of phases, beginning with phase 0. During a phase j, the algorithm can use only the big bucket and the small bucket number j mod b. (The small buckets are numbered $0, \ldots, b-1$.) We assign the first job of the sequence to a machine in small bucket 0 and initialize $\beta_1 = w_1$. We modify β only when a new phase starts.

On arrival of a job J_i (starting from $i = 2$), we do the following.

- If $w_i \leq \beta_{i-1}/2$ (a *small* job), assign J_i greedily to the least loaded machine in the big bucket.
- If $\beta_{i-1}/2 < w_i \leq \beta_{i-1}$ (a *medium-sized* job), and there is a machine in the small bucket which was not used in the current phase, assign J_i to this machine.
- Finally, if a medium-sized job arrives when all m machines in the current small bucket were already used in the current phase, or if $w_i > \beta_{i-1}$ (a *large* job), then phase $j+1$ begins: we define $\beta_i = \max((2-\xi)\beta_{i-1}, w_i)$ and J_i is assigned to a machine in the small bucket of phase $j+1$.

An example run of BUCKETS is given in Figure 3.1: the last job has just been placed on a machine in small bucket 0, starting phase 3 (it was too large to go in bucket 2).

Definition OPT_i is the optimal load after i jobs have arrived, i. e. the maximum load over all the machines in an optimal schedule for this sequence of i jobs.

Lemma 3.2 *During the execution of* BUCKETS*, we have the following two invariants on* β*:*

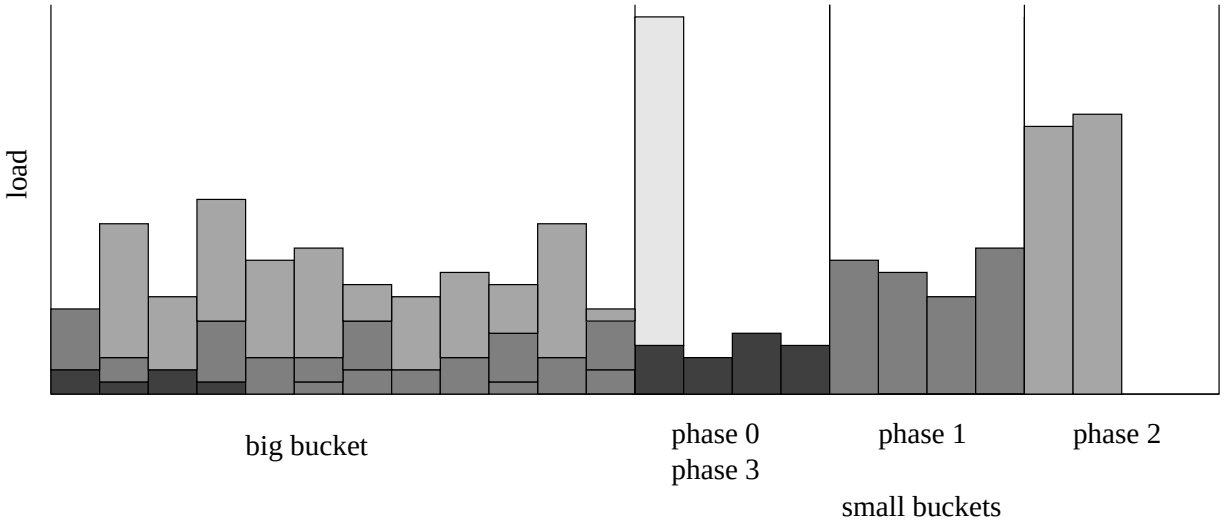

Figure 3.1: A run of the algorithm BUCKETS

- $\max_{j \le i} w_j \le \beta_i$
- $(2-\xi)\text{OPT}_i \ge \beta_i$

Proof We show that both invariants hold after the arrival of a job (and thus hold throughout the execution of BUCKETS). After the assignment of the first job, $\beta_1 = \text{OPT}_1 = w_1$, and both invariants hold since $\xi < 1$.

The first invariant always holds: when a job arrives which is larger than β, β is increased. To show that the second invariant holds, we show that β is increased only when the previous β is smaller than the current OPT, and that β is not increased too much. If β is increased because $\beta_{i-1} < w_i$, then $\text{OPT}_i \ge w_i$ and since $\beta_i = \max((2-\xi)\beta_{i-1}, w_i)$ then $\beta_i \le (2-\xi)w_i \le (2-\xi)\text{OPT}_i$. If β is increased because all the machines in the small bucket were used in the current phase, then there are at least $m+1$ jobs of weight more than $\beta_{i-1}/2$ and hence the optimal off-line schedule has to assign two of them on one machine, yielding $\text{OPT}_i > \beta_{i-1}$. Thus $\beta_i \le (2-\xi)\text{OPT}_i$. □

Theorem 3.1 *The algorithm* BUCKETS *is* $(1+2^{-\gamma(1-o(1))})$*-competitive for an appropriate choice of* ξ *for* $\gamma \to \infty$.

Proof Consider first the big bucket. Here BUCKETS uses GREEDY to assign jobs to the machines. We show that the maximum load in the big bucket never exceeds OPT_i at step i (after arrival of job J_i). It is easy to see that for any $a > 1$, the maximum load of running GREEDY on am machines is at most $\text{OPT}_i/a + \max_{j \le i} w_j$. Since $w_j \le \beta_{i-1}/2$ for any $j < i$

for which J_j is assigned to the big bucket, and $\beta_{i-1}/(2-\xi) \leq \text{OPT}_{i-1}$, the load is bounded by $(\frac{1}{a} + \frac{2-\xi}{2})\text{OPT}_{i-1} \leq (\frac{\xi}{2} + \frac{2-\xi}{2})\text{OPT}_i = \text{OPT}_i$.

Next, we bound the maximum load on the machines in small buckets. When a new phase starts, the value of β is multiplied by at least $2-\xi$. Each machine in a small bucket is used at most once in each phase.

Consider a job which is assigned to a machine in a small bucket machine at the last time that machine is used. Denote this job by $J_{i'}$, and let $\beta' = \beta_{i'}$. Then the previous job assigned to the same machine is of weight at most $\beta'/(2-\xi)^b$, and in general the j-th job before $J_{i'}$ that was assigned to this machine has weight at most at most $\beta'/(2-\xi)^{jb}$. Thus the total weight of all jobs on this machine, except $J_{i'}$, is at most $2\beta'/(2-\xi)^b$. Since $\text{OPT} \geq \beta'/(2-\xi)$ we get that the total weight of jobs on this machine is at most

$$w_{i'} + \frac{2\beta'}{(2-\xi)^b} \leq w_{i'} + \frac{4\text{OPT}}{(2-\xi)^b} \leq (1 + \frac{4}{(2-\xi)^b})\text{OPT} \leq (1 + \frac{4}{(2-\xi)^{\gamma-2/\xi-1}})\text{OPT}.$$

Choosing an appropriate value of ξ gives the required competitive ratio. For example $\xi = \sqrt{3/\gamma}$ is a suitable value. This follows because

$$\begin{aligned} 4/(2-\sqrt{3/\gamma})^{\gamma-2\sqrt{\gamma/3}-1} &= 2^{-\gamma(1-2/\sqrt{3\gamma}-3/\gamma)} \cdot (1-\frac{1}{2}\sqrt{3/\gamma})^{-\gamma-2\sqrt{\gamma/3}-3} \\ &= 2^{-\gamma(1-o(1))} \cdot (1-\frac{1}{2}\sqrt{3/\gamma})^{-\gamma} \cdot (1-\frac{1}{2}\sqrt{3/\gamma})^{-2\sqrt{\gamma/3}-3}. \end{aligned}$$

By taking $x = \sqrt{\gamma}$ we can see that $(1-\frac{1}{2}\sqrt{3}/x)^{-2\sqrt{3}/x-3} = O(1) = 2^{\gamma o(1)}$ and we have $\log_2((1-\frac{1}{2}\sqrt{3}/x)^{-x^2}) = -x^2\log_2(1-\frac{1}{2}\sqrt{3}/x) = x^2(\frac{1}{2}\sqrt{3}/x + O(1/x^2))\log 2 = xO(1)$, hence $(1-\frac{1}{2}\sqrt{3/\gamma})^{-\gamma} = 2^{xO(1)} = 2^{\sqrt{\gamma}O(1)} = 2^{\gamma o(1)}$. This gives the desired result. □

3.2.2 Lower bounds

We begin by giving a simple exponential lower bound.

Theorem 3.2 $\mathcal{R}_{m,\gamma m}(\mathcal{A}) \geq 1 + 2^{-2\gamma+1}$ *for all deterministic on-line load balancing algorithms* $\mathcal{A}$.

Proof Suppose there is an algorithm $\mathcal{A}$ that has a competitive ratio less than $1+2^{-2\gamma+1}$. We give a proof for even m and for integer γ. It is easy to extend the proof to all cases. The sequence that we use consists of at most $M + m/2$ jobs that arrive in at most $2\gamma+1$ phases. Phase 1 consists of $m/2$ unit jobs, and phase i for $i > 1$ consists of $m/2$ jobs of weight 2^{i-2}. The sequence stops (i. e. no more jobs arrive) after a phase in which $\mathcal{A}$ schedules two jobs on one machine. If $\mathcal{A}$ reaches phase $2\gamma+1$, there are more jobs than on-line machines since $M + m/2 > M$, therefore $\mathcal{A}$ has two jobs on one machine after some phase $i \leq 2\gamma+1$.

The optimal off-line load after every phase is the weight of the jobs that arrived in that phase. Figure 3.2 (left) shows the optimal schedule after a phase: the jobs from the last phase are on a machine by themselves, while all the previous jobs are together on the remaining machines.

If $\mathcal{A}$ has two jobs on one machine, its maximum load is at least $1 + x$ where x is the weight of the last job. Hence the competitive ratio in that case is $\frac{1+x}{x}$. The minimum value of $\frac{1+x}{x}$ is $1 + 2^{2-i}$ where $i = 2\gamma + 1$, hence $1 + 2^{-2\gamma+1}$ is a lower bound on the competitive ratio. □

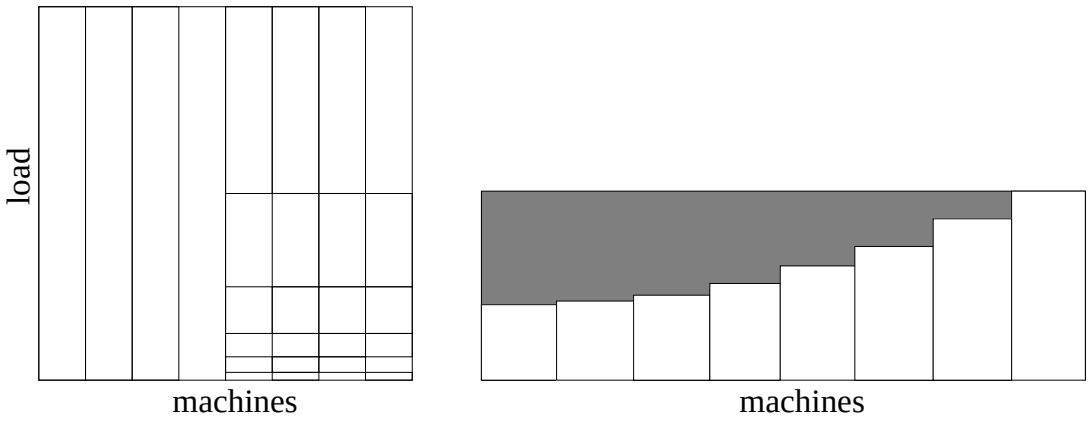

Figure 3.2: The optimal schedules in Theorems 3.2 (left) and 3.3 (right)

We can give a slightly better lower bound, which holds for deterministic and randomized algorithms. To do this, we will make a small excursion to the problem of minimizing the makespan in on-line scheduling of jobs in a list. Hence in this lower bound, the job sizes represent running times instead of loads. We show a lower bound on on-line preemptive algorithms versus a non-preemptive optimal off-line algorithm. Hence our lower bound holds both for the preemptive and non-preemptive models. This holds because for the preemptive model we consider a preemptive optimal off-line algorithm which can perform at least as well as a non-preemptive optimal off-line algorithm, and for the non-preemptive model we consider non-preemptive on-line algorithms which form a subset of the set of preemptive on-line algorithms. Since the non-preemptive model is equivalent to our load balancing model, this gives us also a lower bound for load balancing in the permanent tasks model. The lower bound builds on the lower bounds given by Sgall [67] and independently by Chen, Van Vliet and Woeginger [20, 21].

The main idea here is to use a large amount of infinitesimally small jobs ('sand') and then a sequence of big jobs $J_1, \ldots, J_M$ of increasing size so that the optimal makespan after job J_i is exactly equal to the size of J_i.

Specifically, the sequence begins by very small jobs of total size $m - 1$ followed by the sequence $J_1, \ldots, J_M$. The size of J_i for $1 \leq i \leq M$ is taken as μ^{i-1} where $\mu = \frac{m}{m-1}$. By defining it in this way, it can be seen that each job J_i has size equal to the total size of all

previous jobs divided by $m-1$. We define OPT_i in this case as the optimal makespan after i big jobs have arrived.

Lemma 3.3 *For the above sequence,* $\text{OPT}_i = \mu^{i-1}$ *for* $1 \leq i \leq M$.

Proof We give an off-line algorithm to assign the jobs, and show that the resulting makespan is μ^{i-1} after i big jobs have arrived, which is optimal.

The algorithm assigns jobs to the off-line machines greedily, in non-increasing order (sorted according to size). This is equivalent to using the LPT (longest processing time) rule. We show that no big job is assigned in a way that it completes after time μ^{i-1}. Note that the total size of all small jobs and the first j big jobs is $m-1+\sum_{i=0}^{j-1}\mu^i = m-1+\frac{\mu^j-1}{\mu-1} = m-1+(\mu^j-1)(m-1) = (m-1)\mu^j = \mu^{j-1}m$, since $\mu - 1 = 1/(m-1)$.

Assume that the assignment of job J_j to a certain machine causes it to complete after time μ^{i-1}. This means that all other machines will finish their jobs after time $\mu^{i-1} - \mu^{j-1}$. Hence the total size of assigned jobs until J_j is more than $(\mu^{i-1} - \mu^{j-1})(m-1) + \mu^{i-1}$.

The total size of jobs smaller than J_j is $\mu^{j-1}(m-1)$. These jobs still have to be assigned by the off-line algorithm because of its ordering of the jobs. Hence the total size of all the jobs is more than $\mu^{i-1}m$, which is a contradiction. Therefore the assignment of the small jobs results in a schedule where each machine finishes its last job at time μ^{i-1}, which is optimal.

Note that in the case that the jobs should be assigned to machines in order of their arrival, this can easily be accomplished by taking the above assignment and inverting the order of the jobs on each machine. □

The following lemma, adapted from [67, 33], is the key of lower bounding the competitive ratio.

Lemma 3.4 *For any deterministic or randomized, preemptive or non-preemptive algorithm for minimizing the makespan of jobs in a list, for the sequence above the following holds:*

$$\mathcal{R} \geq \frac{W}{\sum_{i=1}^{M} \text{OPT}_i},$$

where $\mathcal{R}$ *is the competitive ratio and* W *is the total size of all the jobs.*

Proof Denote by $\mathcal{A}(J_i)$ the makespan of the on-line algorithm $\mathcal{A}$ after the assignment of the job J_i. Then

$$\frac{\sum_{i=1}^{M} \mathbb{E}(\mathcal{A}(J_i))}{\sum_{i=1}^{M} \text{OPT}_i} \leq \frac{\sum_{i=1}^{M} \mathcal{R} \cdot \text{OPT}_i}{\sum_{i=1}^{M} \text{OPT}_i} = \mathcal{R}.$$

Hence it is enough to show that $\sum_{i=1}^{M} \mathbb{E}(\mathcal{A}(J_i)) \geq W$.
Assume that $\mathcal{A}$ is deterministic. We sort the machines by non-increasing maximum completion

time. For $1 \leq i \leq M$, let L_i be the completion time of the last job on the ith machine at the end of the sequence. Then $L_1 \geq L_2 \geq \ldots \geq L_M$. Removing any $i-1$ jobs still leaves a machine with a job that does not complete before time L_i. Therefore $\mathcal{A}(J_{M-1}) \geq L_2$, $\mathcal{A}(J_{M-2}) \geq L_3$ and in general $\mathcal{A}(J_i) \geq L_{M-i+1}$ for $1 \leq i \leq M-1$. Since $W = \sum_{i=1}^{M} L_i$ we conclude that

$$\sum_{i=1}^{M} \mathcal{A}(J_i) \geq \sum_{i=1}^{M} L_{M-i+1} = W$$

as needed. If $\mathcal{A}$ is randomized, we average over the deterministic algorithms and conclude that

$$\sum_{i=1}^{M} \mathbb{E}(\mathcal{A}(J_i)) \geq W.$$

This gives the bound. Note that these arguments hold independent of whether $\mathcal{A}$ is preemptive or not. □

Theorem 3.3 *The competitive ratio of any on-line algorithm for minimizing the makespan of jobs in a list, deterministic or randomized, preemptive or non-preemptive, is at least* $1/(1-(\frac{m-1}{m})^M) = 1 + e^{-\frac{M}{m}(1+o(1))}$.

Proof We use the job sequence defined above and apply Lemma 3.4. We have $W = \mu^M(m-1)$,

$$\sum_{i=1}^{M} \mathrm{OPT}_i = \sum_{i=1}^{M} \mu^{i-1} = \frac{\mu^M - 1}{\mu - 1}$$

and

$$\mathcal{R} \geq \frac{\mu^M(m-1)}{(\mu^M - 1)}(\mu - 1) = \frac{\mu^M}{\mu^M - 1} = \frac{1}{1 - \frac{1}{\mu^M}} = \frac{1}{1 - (\frac{m-1}{m})^M}$$

as needed. □

Corollary 3.1 *The competitive ratio of any deterministic or randomized on-line load balancing algorithm is at least* $1/(1-(\frac{m-1}{m})^M) = 1 + e^{-\frac{M}{m}(1+o(1))} = 1 + e^{-\gamma(1+o(1))}$.

We can improve the bound for the special case $\gamma = 2$ for deterministic load balancing (i. e. minimizing the makespan in on-line scheduling without preemptions).

Claim 3.1 $\mathcal{R}_{m,2m}(\mathcal{A}) \geq 5/4$ *for all deterministic on-line load balancing algorithms* $\mathcal{A}$.

Proof Suppose there is an algorithm $\mathcal{A}$ that maintains a competitive ratio less than $5/4$. We use a job sequence consisting of at most four phases:

- m jobs of weight 1
- $\lfloor \frac{m}{2} \rfloor$ jobs of weight $3/2$
- $\lfloor \frac{m}{3} \rfloor + 1$ jobs of weight 3
- $\lfloor \frac{m+1}{6} \rfloor + 1$ jobs of weight 4.

The sequence stops after a phase in which $\mathcal{A}$ schedules two jobs on one machine. Note that the sequence contains more than $2m$ jobs if it reaches the fourth phase, hence there is such a phase. This can be seen in the following table.

m mod 6	0	1	2	3	4	5
Amount of jobs	$2m+2$	$2m+1$	$2m+1$	$2m+1$	$2m+1$	$2m+1$

We show that the optimal load in phase i is i. This is clear for phases 1 and 2. In phase 3, if the machines are packed to a maximum load of 3, at most $5/2$ of space can be lost: 2 is lost if a job of weight 1 has to be assigned to its own machine, and an additional $1/2$ is lost if there is an odd number of jobs of weight $3/2$. The total weight is at most $m + \frac{3m}{4} + (m+3) = \frac{11m}{4} + 3$, which is at most $3m - 5/2$ for $m \geq 22$. This implies that the machines can be packed with a maximum load of 3 for $m \geq 22$. By inspection, the machines can be packed for $8 \leq m \leq 21$ too.

In phase 4, the total weight is at most $\frac{11m}{4} + 3 + \frac{4m}{6} + \frac{14}{3}$. In the optimal packing, if the maximum load is 4, then at most $7/2$ of space is lost, using the same arguments as in phase 3. We have $\frac{41}{12}m + \frac{23}{3} \leq 4m - 7/2$ which holds for $m \geq 20$. Therefore the optimal algorithm can maintain a maximum load of 4 in phase 4, if $m \geq 20$. By inspection, the machines can be packed for $8 \leq m \leq 19$ as well.

As an example, we give the optimal schedules for phases 3 and 4 when $m = 8$ and $m = 9$ (see Figure 3.3).

Depending on the phase in which $\mathcal{A}$ puts two jobs on the same machine, we find competitive ratios of $2, 5/4, 4/3$ and $5/4$, respectively. Hence the competitive ratio is at least $5/4$. □

The method in this proof can be adjusted to give slightly worse lower bounds for $\mathcal{R}_{m,2m}(\mathcal{A})$ for $2 \leq m \leq 7$. The first two phases are the same, and then jobs arrive with sizes as in Table 3.1. The resulting bound is also given.

3.3 Temporary tasks

We now consider load balancing of temporary tasks, where jobs arrive and depart (leave) at arbitrary times. For $M = m$, by [8] GREEDY is $(2 - 1/m)$-competitive for permanent tasks as

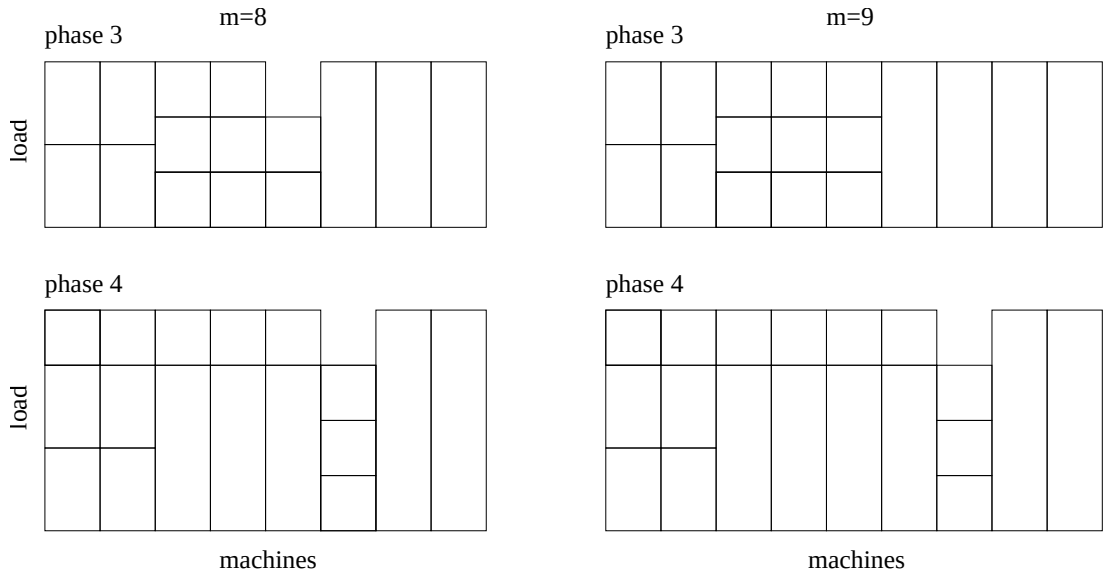

Figure 3.3: The last phases for $m = 8, 9$

m	job sizes	lower bound
2	3, 7	8/7
3	3, 4, 6	7/6
4	3, 4, 5	6/5
5	3, 3, 4, 5	6/5
6	3, 3, 9/2, 9/2	11/9
7	3, 3, 3, 9/2, 9/2	11/9

Table 3.1: Lower bounds for load balancing: $\gamma = 2$, $2 \leq m \leq 7$

well as for temporary tasks. GREEDY is not optimal for permanent tasks, but also by [8] it is optimal for temporary tasks.

Also for $M > m$, it is easy to see that GREEDY has the same competitive ratio for temporary tasks as for permanent tasks, which is $1 + (m - 1)/M$. This holds because we can apply the analysis for permanent tasks at any time that a job arrives, and taking into consideration only the jobs that are still present at that time. We can do that because this analysis only uses arguments concerning the total size of all the jobs that are present and the largest size of those jobs, and is still valid even if there have been other jobs present in the past. (When jobs leave, the cost of GREEDY and OPT is unaffected since this cost is the maximum load over machines and over time.)

However, in contrast to the case $M = m$, GREEDY is not optimal for temporary tasks. We

can define an algorithm BUCKETS′ for temporary tasks based on BUCKETS. The only change in the definition is that if a medium-sized job leaves during a phase, we will use that machine again if a new medium-sized job arrives. (Note that in each phase, medium-sized jobs are assigned only to the small bucket of that phase; jobs that were medium-sized in previous phases are now small.) Hence, a new phase only starts when a large job arrives or when all m machines in the current small bucket have a medium-sized job (that has not left yet).

With this modification, the same analysis of the competitive ratio of algorithm BUCKETS for permanent tasks also holds for temporary tasks. The proofs of Lemma 3.2 and Theorem 3.1 only use arguments about the currently present jobs and still hold if some jobs have left before the current time.

Although GREEDY is thus not optimal in general for $M > m$, we show that if the on-line algorithm has one more machine than the optimal off-line algorithm, then GREEDY is still optimal.

Theorem 3.4 *Suppose $M = m + 1$. Then* GREEDY *is optimal for temporary tasks.*

Proof We need to show a lower bound of $\mathcal{R}_1 = \frac{2m}{m+1}$ on the competitive ratio of any on-line algorithm $\mathcal{A}$. The proof consists of two parts: one for odd m and one for even m. In the proof we mention the value of the optimal off-line load only when the value increases, i. e. when jobs arrive that OPT cannot place on the machines without exceeding the previous maximum optimal load on some machine.

Case A. m is odd. We start the sequence by $m^2(m-1)$ unit-weight jobs. The optimal off-line load is $m(m-1)$. We distinguish between two cases:
Case A1. $\mathcal{A}$ places at least $m(m-1)$ unit jobs on one machine, say machine K.
Case A2. All machines have load at least $m-1$.

This covers all the cases, since if all machines have load at most $m(m-1)-1$ and there is also a machine with a load of at most $m-2$, the total load on all the machines is no more than $m(m(m-1)-1)+(m-2) = m^2(m-1)-2 < m^2(m-1)$, a contradiction.

Case A1. $\mathcal{A}$ places at least $m(m-1)$ unit jobs on one machine, say machine K.

We extend the input sequence as follows. All the jobs leave except $m(m-1)$ jobs on K. We have the situation in Figure 3.4, left.

Now, a set S_1 of $m(m-1)$ jobs of weight $m-1$ arrives. The optimal off-line load is still $(m(m-1)+m(m-1)^2)/m = m(m-1)$.

Suppose A assigns at least $m-1$ jobs in S_1 to K. Then the load on K is at least $m(m-1)+(m-1)^2 = (2m-1)(m-1)$, and since $\frac{(2m-1)(m-1)}{m(m-1)} > \frac{2m}{m+1} = \mathcal{R}_1$, we are done.

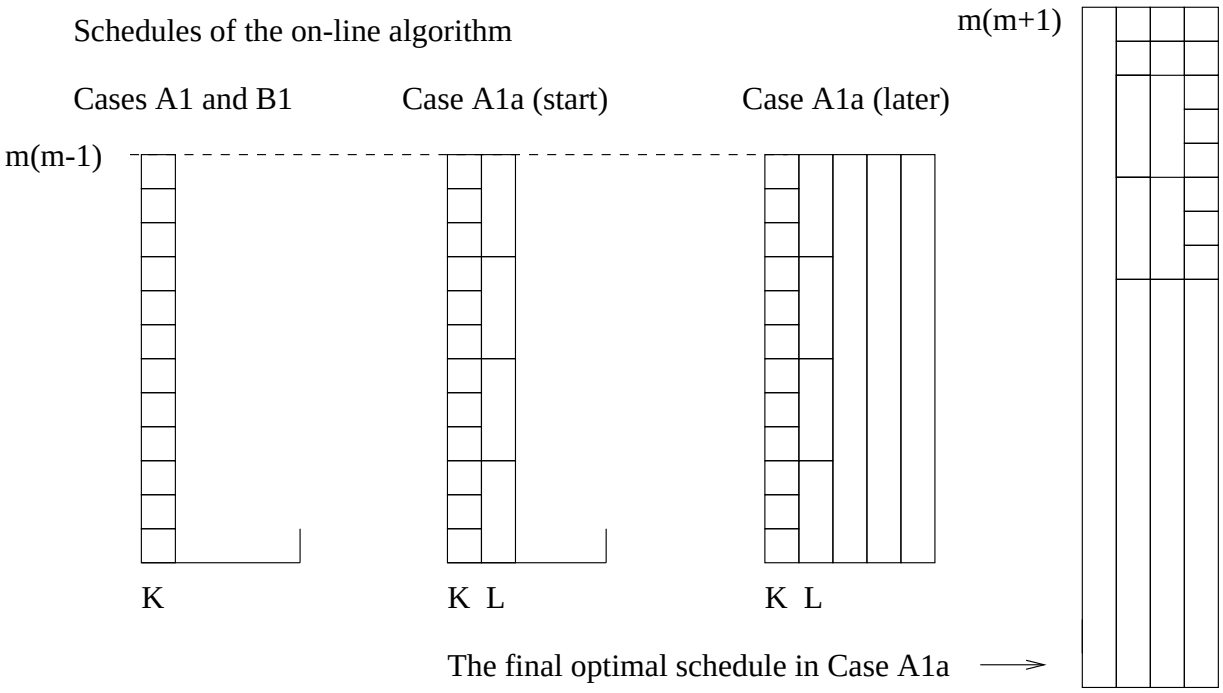

Figure 3.4: Schedules in the lower bound for temporary tasks

Suppose $\mathcal{A}$ assigns at most $m-2$ jobs in S_1 to K. Then at least $m(m-1)-(m-2) = (m-1)^2+1$ jobs in S_1 must be assigned by $\mathcal{A}$ to the m other machines. There are two sub-cases:
Case A1a. There is a machine L ($L \neq K$) which has at least m jobs in S_1.
Case A1b. All machines (except machine K) have at least one job in S_1.

This covers all the cases, since if all machines besides K have at most $m-1$ jobs in S_1 and there is also a machine (not K) without jobs in S_1, then the machines besides K have only at most $(m-1)^2$ jobs in S_1 in total, a contradiction.

Case A1a. There is a machine L ($L \neq K$) which has at least m jobs in S_1.

All jobs in S_1 leave except m jobs on L. (The $m(m-1)$ jobs of weight 1 on K remain.) See Figure 3.4, second schedule.

Now, a set S_2 of $m-1$ jobs of weight $m(m-1)$ arrive. The new optimal off-line load is $(2m(m-1)+m(m-1)^2)/m = m^2-1 = (m+1)(m-1)$.

Suppose $\mathcal{A}$ assigns one of the jobs in S_2 to K or L, or two jobs from S_2 to the same machine. Then the load on that machine is $2m(m-1)$, and $\frac{2m(m-1)}{(m+1)(m-1)} = \mathcal{R}_1$.

Suppose $\mathcal{A}$ assigns all the jobs in S_2 to empty machines. We have the situation shown in Figure 3.4, third schedule. Now, a job of weight $m(m+1)$ arrives. The on-line load becomes $2m^2$. The total weight of all the jobs that have not left is $m^2(m+1)$. OPT can schedule these

as follows: the job of weight $m(m+1)$ has its own machine, the other machines all have one job of weight $m(m-1)$. Among those machines, there are $m/2$ machines with 2 jobs of weight $m-1$ and two jobs of weight 1. The other $m/2-1$ machines have $2m$ jobs of weight 1. The off-line load is precisely $m(m+1)$ on each machine. See Figure 3.4, right. Hence $\mathcal{R}(\mathcal{A}) \geq 2m^2/(m(m+1)) = \mathcal{R}_1$.

Case A1b. All machines (except machine K) have at least one job in S_1.

All jobs in S_1 leave except m jobs, one job from S_1 remains on each machine except machine K (which still has $m(m-1)$ jobs of weight 1). See Figure 3.5, left.

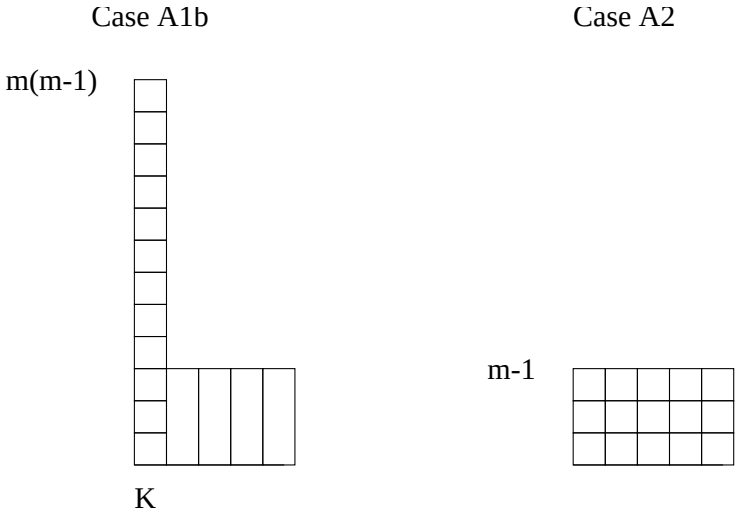

Figure 3.5: Schedules in the lower bound for temporary tasks

Next, a set S_3 of $\frac{m^2-2m-1}{2}$ jobs of weight $2(m-1)$ arrives. The optimal off-line load is still $m(m-1)$ since $((m^2-2m-1)(m-1)+2m(m-1))/m = (m^2-1)(m-1)/m < m(m-1)$.

Suppose $\mathcal{A}$ assigns at least $\frac{m-1}{2}$ jobs in S_3 to machine K. Then the load on K is at least $2(m-1)\frac{m-1}{2} + m(m-1) = (2m-1)(m-1)$ and $\frac{(2m-1)(m-1)}{m(m-1)} > \mathcal{R}_1$.

Suppose $\mathcal{A}$ assigns at most $\frac{m-3}{2}$ jobs in S_3 to machine K. There are $(\frac{m^2-2m-1}{2} - \frac{m-3}{2})/m = \frac{m-3}{2} + \frac{1}{m}$ jobs in S_3 on average on the other machines, so there is at least one machine L ($L \neq K$) with at least $\frac{m-1}{2}$ jobs from S_3 and hence a load of at least $m(m-1)$. All jobs leave except the unit jobs on K and jobs of total weight precisely $m(m-1)$ on machine L. The loads are now the same as in Figure 3.4, second schedule, but the jobs on L have weight $2(m-1)$ now. We continue as in Case A1a: the set S_2 arrives followed by one job of weight $m(m+1)$. The optimal algorithm can again balance its jobs to a load of $m(m+1)$, which completes this case.

Case A2. All machines have load at least $m-1$.

All jobs leave except $m-1$ jobs on each machine. See Figure 3.5, right.

Now, a set S_4 of $m^2 - m - 1$ jobs of weight $m - 1$ arrives. The average number of jobs in S_4 per machine is $(m^2 - m - 1)/(m + 1) = m - 2 + \frac{1}{m+1}$, and hence there is a machine with at least $m - 1$ jobs from S_4 and a load of at least $m(m - 1)$. We call this machine K. All jobs in S_4 leave apart from $m - 1$ jobs on K. The loads are now the same as in Figure 3.5, left, but K now has $m - 1$ unit-weight jobs and $m - 1$ jobs of weight $m - 1$, and all other jobs now have $m - 1$ unit-weight jobs. Hence there are in total more unit jobs and less larger jobs now. We continue as in Case A1b with the set S_3. (Since there are now more unit jobs than in Case A1b, but the total weight of all the jobs is the same, the optimal loads are not higher than in that case.)

Case B. m is even. The structure of this proof is very similar to that of Case A. Only some numbers change because m is even now.

We start the sequence by $m^2(m - 1)$ unit jobs. The optimal off-line load is $m(m - 1)$. We distinguish between two cases:

Case B1. One machine, say K, has at least $m(m - 1)$ unit jobs.

Case B2. There are at least m unit jobs on each machine.

This covers all the cases, since if all machines have load at most $m(m - 1) - 1$ and there is also a machine with a load of at most $m - 1$, the total load on all the machines is no more than $m(m(m - 1) - 1) + (m - 1) = m^2(m - 1) - 1 < m^2(m - 1)$, a contradiction.

Case B1. One machine, say K, has at least $m(m - 1)$ unit jobs.

All jobs leave except $m(m - 1)$ jobs on K. See Figure 3.4, left.

Now a set S_5 of $(m - 1)^2$ jobs of weight m arrive. The optimal off-line load is $(m(m - 1) + m(m - 1)^2)/m = m(m - 1)$.

Suppose $\mathcal{A}$ assigns at least $m - 1$ jobs in S_5 to K. Then the load on K is at least $2m(m - 1)$, and $\frac{2m(m-1)}{m(m-1)} = 2 > \mathcal{R}_1$.

Suppose $\mathcal{A}$ assigns at most $m - 2$ jobs in S_5 to K. Then $\mathcal{A}$ must assign at least $(m - 1)^2 - (m-2) = m^2-3m+3$ jobs in S_5 to the empty machines. We distinguish between two sub-cases:

Case B1a. There is a machine L ($L \neq K$) which has $m - 1$ jobs in S_5.

Case B1b. Each machine except K has one job in S_5.

This covers all the cases, since if all machines besides K have at most $m - 2$ jobs in S_1 and there is also a machine (not K) without jobs in S_1, then the machines besides K have only at most $(m - 1)(m - 2) = m^2 - 3m + 2 < m^2 - 3m + 3$ jobs in S_1 in total, a contradiction.

Case B1a. There is a machine L ($L \neq K$) which has $m - 1$ jobs in S_5.

All jobs of weight m leave except $m - 1$ jobs on L. The loads now are the same as in the second schedule in Figure 3.4, but the jobs on L now have weight m. We continue as in Case

A1a, with the set S_2 followed by a job of weight $m(m+1)$, which completes this case.

Case B1b. Each machine except K has one job in S_5.

All jobs of weight m leave except m jobs, one remains on each machine except on machine K. The schedule is very similar to Figure 3.5, left, but the jobs on the machines other than K now have weight m.

Now, a set S_6 of $\frac{m^2-3m}{2}$ jobs of weight $2m$ arrive.

Suppose $\mathcal{A}$ assigns $m/2$ jobs from S_6 to machine K. Then the load on K is $m^2+m(m-1)$, and $\frac{m^2+m(m-1)}{m(m-1)} > 2 > \mathcal{R}_1$.

Suppose $\mathcal{A}$ assigns at most $\frac{m-2}{2}$ jobs to K. Then the average number of jobs of weight $2m$ on machines besides K is $\frac{m^2-3m-(m-2)}{2m} = \frac{m}{2} - 2 + \frac{1}{m}$. Thus, one machine L must have $\frac{m}{2} - 1$ jobs of weight $2m$ and a load of at least $m(m-1)$. All jobs leave except the unit jobs on K and jobs of total weight $m(m-1)$ on L. The loads are the same as in the second schedule in Figure 3.4, but the jobs on L now have weight m. We continue as in Case A1a with the set S_2 followed by the job of weight $m(m+1)$.

Case B2. There are at least m unit jobs on each machine.

All jobs leave except m unit jobs on each machine. Next, $\frac{m^2(m-2)-m}{2}$ jobs of weight 2 arrive. The total load of all the jobs is $m^2(m-1)$ and the optimal load is again $m(m-1)$.

If there is a machine with load at least $m(m-1)$, all other jobs leave and we continue as in Case B1 with the set S_5.

Otherwise, each machine has load at least $2m$, since if all machines have a load of at most $m(m-1)-2$ and there is also one machine with a load of at most $2m-2$, the total load is at most $m(m(m-1)-2)+(2m-2) = m^2(m-1)-2$, a contradiction.

Now, some jobs of weight 2 leave in such a way that the load on each machine is $2m-2 \geq m$. Next, m^2-2m-2 jobs of weight $m-1$ arrive. On average, each machine will have $m-2-\frac{2}{m}$ of these jobs, so one must have $m-2$ of them. Therefore, one machine K will have a load of at least $m(m-1)$. Jobs of weight $m-1$ on that machine leave such that the load becomes $m(m-1)$. All non-unit jobs on the other machines leave. The schedule is identical to the one in Case B1b just before the set S_6 arrives, and we continue as in that case. □

3.4 Conclusions

We have examined the effects of resource augmentation for several load balancing problems. For the permanent tasks model, which is equivalent to the problem of scheduling jobs on identical

machines to minimize the makespan, we have shown an algorithm with a competitive ratio which decreases exponentially in m/M, while GREEDY has a competitive ratio that is linear in m/M.

An open question is whether it is possible to close the gap between the lower bound and the upper bound on identical machines. Both bounds are decreasing exponentially, and we conjecture that the true value of the competitive ratio is closer to the lower bound.

Chapter 4

Running a job on a collection of partly available machines, with on-line restarts

We consider the problem of running a background job on a selected machine of a collection of machines, e.g. workstations in a network. Each of these machines may become temporarily unavailable (busy) without warning. If the currently selected machine becomes unavailable, the scheduler is allowed to restart the job on a different machine. The behaviour of machines is characterized by a Markov chain, which can be compared to [51]. The objective is to minimize expected completion time of the job. For several types of Markov chains, we present elegant and optimal policies.

4.1 Introduction

In networks of workstations, a considerable amount of capacity is unused, since the primary users are only using the workstations part of the time. Such machines could therefore be used for large(r) jobs that can be executed in the background or with low priority. This means that such a job gets the "free time" of the machine, i.e., the time that no higher-priority job is using it. However, this does not mean that the job does not have any objectives in completion time.

When a workstation is used (for a higher-priority job), there is information available on the type of job that is executed. This is e.g. available from the process manager (process statuses). With this information, it could be decided what to do: e.g., to just wait until the workstation is available again, or to restart the larger job on another machine. In this way, the completion time of this job could be minimized.

The above situation is not only true for workstations, but for e.g. supercomputers or other scarce high-performance computers that are available in smaller quantities as well. We therefore

study the problem of executing a large job as a background job, where the completion time should be minimized and the job can be executed on one machine at a time.

The problem To be precise, we study the problem of scheduling a job J on a machine out of a collection of machines that are not available continuously, without having full knowledge about when they are available. At the start, the scheduler must pick one machine to run the job on. If the machine becomes temporarily unavailable, the scheduler is allowed to move the job and restart it from scratch on a different machine. The goal is to minimize the expected completion time. This was mentioned as an open problem in [64].

In [5], a method is discussed for a different but related situation, viz., where a job J must be run in a specific time interval. The assumption made on the availability of the workstations was that at least one of the workstations would be available for a certain amount of time (significantly larger than the time required to run the job) during the interval in which the job was to be run. Using this assumption, a method was shown which had an $1 - O(1/m)$ probability of choosing a "good" workstation, so that J is completed on time, where m is the number of workstations. However, with this approach it is not possible to minimize the expected running time of J. As it turns out, in order to give bounds for the completion time, it is necessary to use a different approach.

We study the case where the availability of the workstations is captured by a Markov chain. We will consider Markov chains that, if the idle state is deleted, become acyclic. In practical situations, a Markov chain is obtained and approximated from statistical information, and approximating "infinite cycling behaviour" by just one or a couple of states will fall within the statistical and practical accuracies. We also refer to [51] for some general comments on Markov chains.

Note that every behaviour of workstations can indeed be modeled by a Markov chain, depending on the grain of description. E.g, the most simple chain can be obtained by having, besides a state for "available" (idle), one state for "unavailable", with the expected unavailability time as its cost. Making more elaborate Markov chains based on (on-line) system statistics and additional information, enables finer-grained description and improved scheduling strategies, yielding lower completion times.

Our results We present elegant and optimal scheduling strategies. The actual job size does not need to be known (but it does not help to know it either). The computational complexity of our strategies is $O(n \cdot e)$, where n is the number of nodes in the Markov chain and e is the number of edges. This only needs to be computed once for all future large jobs. The strategy only depends locally on the machine the job is running on. This is in contrast with [5], where global decisions are needed.

Chapter outline We begin by looking at a Markov chain where only one user-job size can occur. We then examine more complex Markov chains, where jobs of different sizes can occur. Finally, we look at the case where the interrupting jobs themselves form a Markov chain (i. e. more is known about the sequences in which jobs are often started), thus enabling a fine-grained description of machine behaviour.

4.2 The model

We have a job J which takes d units of time to complete. (Although d does not need to be known in advance, throughout this chapter, we use d as if it were known.) At any time, we can allocate not more than one machine from a collection of machines to run J. If the machine becomes temporarily unavailable, the scheduler is allowed to restart the job from scratch on a different machine. The goal is to minimize the expected completion time of J.

The behaviour of every machine is characterized by a Markov chain. One state of this chain, called the idle state, represents the situation that the machine is available for executing a (new) large job. Any other state represents a local job or a job session, that makes the machine unavailable for the scheduler. Such local jobs or job sessions can have different sizes. Only the expectation of the size of each such job or job session needs to be known, since we minimize the expected completion time of J. However, for reasons of simplicity, we henceforth consider a state to correspond to just one job with a fixed size. The conversion to job sessions and expected size of those is not made explicit any more, but this is trivial since only expected (completion) times are considered.

The machines are identical, in the sense that they are modeled by the same Markov chain. All machines behave independently of each other and of the decisions made by the scheduler. The scheduler may use the information of the Markov chain. We assume that if the scheduler wants to restart J, there is always a machine available. This is realistic, since we will show that in a network of some non-trivial size, the expected time for the first machine to become available, starting in a randomly chosen time step, is very small as long as the Markov chain does not yield extreme occupation in this network. (And if there is extreme occupation, no scheduler can hope to perform well.)

The Markov chain of a machine, together with the possibility of a restart, induces a Markov decision process on the job J, as follows. We define J to be in *time state* t if it has been worked on for t time steps since its latest restart, not counting the time that the current machine was busy. For every time state $0, \ldots, d-1$ of J and every possible interrupting job, we need to decide what to do in case of an interruption. Do we restart J or wait? J is completed when it reaches time state d.

4.3 The basic case

In the basic case, all machines behave according to the Markov chain shown on the left in Figure 4.1. A more compact way of picturing this is shown on the right in the same figure, where the

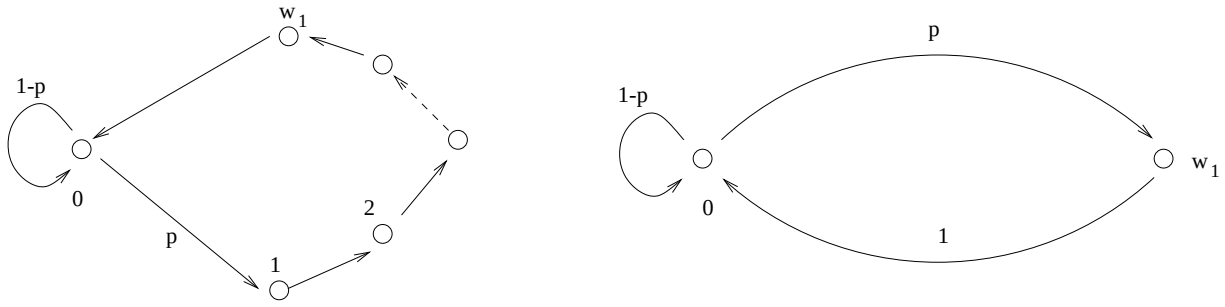

Figure 4.1: Markov chain of one machine in two forms

node marked w_1 costs w_1 units of time. Note that this is just a simplifying picture and cannot be used as the basis of calculations. The induced Markov chain on J is shown in Figure 4.2. Costs are in bold type.

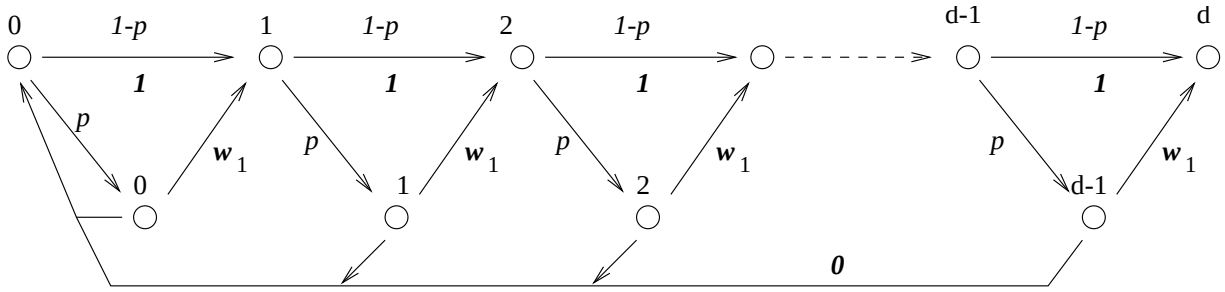

Figure 4.2: Markov chain of our job

In Markov decision theory, this is known as a first-passage problem [30]. Such problems can be solved using a linear program, but this requires introducing $2d$ variables, one for each node in the Markov chain. Solving a linear program with $2d$ variables can be done in $O(8d^3)$ time. This is clearly impractical, as this is far more than the running time of the (large) job itself. Furthermore, such a linear program would have to be solved for every occurring job size d. We show an optimal policy with time complexity $O(1)$, that is independent of, and does not need to know, d.

Clearly, in the top row of this Markov chain (representing the idle node), it is always optimal to continue. Only when the process moves to one of the nodes in the lower row, meaning that the current machine is busy because of a higher priority job, we need to make a choice.

4.3.1 The optimal policy

It is known [30] that for any first-passage problem there is an optimal policy that is stationary: it does not depend on the total time that the job has been running, or the number of times it has been in the current time state. Also, it is deterministic.

A policy will be denoted by a vector $a = (a_0, \ldots, a_{d-1})$, where $a_t = 1$ means the scheduler will restart J if it gets interrupted in time state t, and $a_t = 0$ means he will not restart. Define $\text{OPT}(t)$ to be the expected minimal costs (running time) to complete J, starting in time state t. These costs satisfy $\text{OPT}(d) = 0$ and

$$\begin{aligned} \text{OPT}(t) &= (1-p)(1+\text{OPT}(t+1)) \\ &+ p \cdot \min\{\text{OPT}(0), w_1 + \text{OPT}(t+1)\} \qquad t = 0, \ldots, d-1. \end{aligned} \tag{4.1}$$

This holds because the probability of going directly to the next time state is the probability of remaining in the idle node, $1-p$, and the optimal costs in that case are $1+\text{OPT}(t+1)$. When the machine becomes busy, the minimal costs are the minimum of the two choices there: restarting costs $\text{OPT}(0)$, and waiting costs $w_1 + \text{OPT}(t+1)$.

It follows from (4.1) that a restart in time state t is optimal if and only if

$$\text{OPT}(0) \leq w_1 + \text{OPT}(t+1), \tag{4.2}$$

and in that case restarting is optimal in all the previous time states as well, since $\text{OPT}(t)$ is monotonically decreasing. Thus $a_t = a_{t-1} = \ldots = a_0 = 1$.

It follows that an optimal policy is a *threshold policy*: interruptions cause restarts only up to a certain point. Therefore an optimal policy is of the form $a(k)$:

$$a(k) = (1, \overset{k}{\ldots}, 1, 0, \overset{d-1-k}{\ldots}, 0) \qquad k \in \{0, \ldots, d-1\}.$$

Here k indicates the number of steps for which an interruption causes a restart, e. g. $k = 2$ means $a_0 = 1, a_1 = 1, a_2 = \ldots = a_{d-1} = 0$. We have $k \geq 1$ since in time state 0, restarting is always cheapest.

We define $C(k)$ as the expected cost to reach k for the first time, using strategy $a(k)$.

Theorem 4.1 *The optimal policy for the basic case is given by* $a(k^*) = (1, \overset{k^*}{\ldots}, 1, 0, \overset{d-1-k^*}{\ldots}, 0)$, *where* k^* *is determined by*

$$k^* = \lceil \frac{\log(1 + (w_1 - 1)p)}{-\log(1-p)} \rceil. \tag{4.3}$$

The expected completion time is at most $w_1 + (d - k^*)(1 + (w_1 - 1)p)$.

Proof. It follows from (4.2) that restarting is optimal as long as $\text{OPT}(0) - \text{OPT}(t+1) < w_1$. Since $\text{OPT}(0) - \text{OPT}(t+1)$ is the expected total optimal cost minus the expected optimal cost starting in $t+1$, it is equal to $C(k^*)$, where k^* is the optimal choice of k. Thus, we need to calculate $C(k)$ for general k and we need to find the smallest k for which $C(k) > w_1$.

We first calculate $C(k)$. Define R_k as the event that a restart occurs before reaching time state k, then $\mathbb{P}(R_k) = 1 - (1-p)^k$. We write $C_R(k)$ for the cost until a restart, given that this occurs before k is reached. After a restart the costs to reach k are again $C(k)$. Using that the expectation of a random variable $\mathbb{E}X = \mathbb{E}(X|Y)\mathbb{P}(Y) + \mathbb{E}(X|\neg Y)\mathbb{P}(\neg Y)$ we can see that

$$C(k) = (C_R(k) + C(k))(1 - (1-p)^k) + k(1-p)^k \tag{4.4}$$

or

$$C(k) = C_R(k)\frac{1-(1-p)^k}{(1-p)^k} + k. \tag{4.5}$$

Since $\mathbb{E}(X|Y) = \sum_{x \in Y} x\mathbb{P}(X = x)/\mathbb{P}(Y)$, we have that

$$C_R(k) = \sum_{t<k} t \cdot \mathbb{P}(\text{restart after } t \text{ steps})/(1-(1-p)^k).$$

Using $\mathbb{P}(\text{restart after } t \text{ steps}) = (1-p)^t$, we can derive

$$C(k) = \frac{1-p}{p} \cdot \frac{1-(1-p)^k}{(1-p)^k} = \frac{1-p}{p}((1-p)^{-k} - 1).$$

If $C(k) > w_1$, it is no longer advantageous to restart J in time state k. Since $C(k^*) \leq w_1$ implies $(1-p)^{-k^*} - 1 \leq w_1 \frac{p}{1-p}$, we have

$$k^* = \lfloor \frac{\log(1 + \frac{w_1 p}{1-p})}{-\log(1-p)} \rfloor = \lceil \frac{\log(1 + (w_1 - 1)p)}{-\log(1-p)} \rceil.$$

After time state k^*, the job is not restarted anymore. The expected completion time from then on is at most $(d-k^*)(1+(w_1-1)p)$, since $d-k^*$ more units of work need to be done on J, which are each expected to take $(1-p)\cdot 1 + pw_1$ time. Therefore, the total costs are at most $w_1 + (d-k^*)(1+(w_1-1)p)$. □

If we compare this to [5], where the job was completed with probability $1 - O(1/m)$ if at least one machine was available for $\alpha d \log m$ time, we see that we now have a bound that does not depend on m. Note that on the other hand, the behaviour of the machines is now more precisely modeled.

Note that $\mathbb{P}(R_k)/\mathbb{P}(\neg R_k) = 1/\mathbb{P}(\neg R_k) - 1$ is the expected number of failed runs (runs that ended in a restart), and $\frac{1-p}{p} = \frac{1}{p} - 1$ is the expected number of time steps *before* a job gets interrupted. Therefore $C(k) = \mathbb{E}(\text{length of a failed run}) \cdot \mathbb{E}(\#\text{failed runs})$.

Since k^* is the largest k so that $C(k) \leq w_1$, we have $(1-p)w_1 + p - 1 \leq C(k^*) \leq w_1$ and $C(k^*) \to w_1 - 1$ if $p \to 0$.

4.3.2 Restarting and availability

In the above calculations, it is assumed that whenever the scheduler wants to restart, an idle machine is immediately available. Of course, this does not always have to be the case, but it is not difficult to see that we can always expect some machine to be available quickly. The time until this happens is called the waiting time.

First we need the stationary distribution of the Markov chain on a single machine. This is fairly straightforward, and it turns out that the stationary probabilities are $w_1p/(w_1p+1)$ for node w_1 and $1/(w_1p+1)$ for the idle node.

The probability that all machines are busy when one is needed is

$$\left(\frac{w_1p}{w_1p+1}\right)^{m-1}.$$

On each *busy* machine the time until it is again available is distributed homogeneously on the values $1, 2, \ldots, w_1$. The expectation of the minimum of m homogeneously distributed variables is $w_1/(m+1)$. Therefore, the expected waiting time is

$$\left(\frac{w_1p}{w_1p+1}\right)^{m-1}\frac{w_1}{m+1}.$$

Until now we used (4.1) and compared $\text{OPT}(0)$ to $w_1 + \text{OPT}(i+1)$ to determine the optimal policy in state i. Now we should compare $\text{OPT}(0)$ plus the waiting time to $\text{OPT}(i+1)$ plus w_1, so we need to check if

$$\text{OPT}(0) - \text{OPT}(i+1) < w_1' = w_1 - \left(\frac{w_1p}{w_1p+1}\right)^{m-1}\frac{w_1}{m+1}$$

instead of (4.2). The calculations do not change, so the only result is that in (4.3), w_1 must be replaced by w_1'. But the waiting time is much smaller than w_1 and therefore negligible. The situation in state 0 does not change either: although restarts are now no longer free, they still cost far less than w_1. So we still have that in the starting state, restarting is always optimal. Similar results hold if there are interrupting jobs of different sizes, and for general Markov chains. However, for general Markov chains, calculating the expected waiting time is more involved.

4.4 Two jobs

Suppose there are two jobs that can interrupt J, of sizes w_1 and w_2, where $w_1 < w_2$. The probabilities of interruptions by jobs w_1 and w_2 are p_1 and p_2, respectively. We assume that

these interruptions do not occur simultaneously. Then for the optimal completion costs OPT we have

$$\begin{aligned} \text{OPT}(t) &= (1-p_1-p_2)(1+\text{OPT}(t+1)) \\ &\quad +p_1 \cdot \min\{\text{OPT}(0), w_1+\text{OPT}(t+1)\} \\ &\quad +p_2 \cdot \min\{\text{OPT}(0), w_2+\text{OPT}(t+1)\}. \end{aligned}$$

A policy for this problem has the form

$$a = \begin{pmatrix} a_0^1 & a_1^1 & \dots & a_{d-1}^1 \\ a_0^2 & a_1^2 & \dots & a_{d-1}^2 \end{pmatrix},$$

where for all t and i, $a_t^i \in \{0,1\}$. $a_t^i = 1$ means that J will be restarted if interruption w_i occurs in time state t, and $a_t^i = 0$ means J will continue. We have again

$$a_i^j = 1 \text{ is optimal } \Leftrightarrow \text{OPT}(0) - \text{OPT}(i+1) \leq w_j. \tag{4.6}$$

Because $\text{OPT}(i)$ is strictly decreasing, there is a largest l_1 such that $\text{OPT}(0) - \text{OPT}(l_1+1) < w_1$. For states $i > l_1$, a restart is no longer optimal when w_1 interrupts J. Since $w_1 < w_2$, there can be states $i > l_1$ where it is still optimal to restart in case of w_2. This holds until state, say, $l_1 + l_2$. After that, J must never be restarted.

This implies that we can divide the time states of J in three consecutive *phases* (time state intervals). In the first phase, J gets restarted whenever it is interrupted. In the second phase, it is only restarted when w_2 (the largest of w_1 and w_2) interrupts it, and in the third phase it is not restarted at all.

4.4.1 Calculations

Because in the first phase J is always restarted when interrupted, we can determine the optimal length of this phase using the method of the previous section: it ends at the point where a restart becomes too expensive, that is, more expensive than w_1. This follows directly from (4.6), and these costs do not depend on the second or third phases, so l_1 can be determined independently. When we know the optimal l_1^*, we can derive l_2.

The event that a restart occurs in phase i is denoted by R_i. We denote the total expected costs from state 0 to reach phase $i+1$ for the first time by $C_i(l_i)$. In general, we can only calculate costs of reaching a certain state for the first time, since it is always possible that a restart occurs after that state. $C_i(l_i)$ depends on the l_j where $j \leq i$, but when we are going to calculate it, all l_j with $j < i$ will be known, so l_i is the only unknown.

We will also need to look at the expected cost from the beginning of a phase until a restart within that phase, which we will denote by $C_{R_i}(l_i)$. This cost depends only on the length l_i of phase i. Finally, we denote the expected time to go from one state to the next in phase i, given that there is no restart, by T_i. We have $1 = T_1 < T_2 < T_3$.

First we need to calculate for which states $\text{OPT}(0) - \text{OPT}(i+1) < w_1$, or for which l_1 the expected cost of reaching state l_1 for the first time (while restarting whenever J is interrupted) become larger than w_1. As noted above, this does not depend on l_2.

The probability that J gets interrupted in phase 1 is $p_1 + p_2 =: q_1$. Analogously to section 4.3, we find for the optimal length l_1^* of phase 1

$$l_1^* = \lceil \frac{\log(1 + (w_1 - 1)q_1)}{-\log(1 - q_1)} \rceil, \tag{4.7}$$

The cost until l_1^* is reached is $C_1 := C_1(l_1^*) \leq w_1$. We need C_1 to calculate l_2, since $C_2(l_2)$ is equal to C_1 plus the expected cost in case of a restart, plus the expected cost if there is no restart:

$$C_2(l_2) = C_1 + \{C_{R_2}(l_2) + C_2(l_2)\}\mathbb{P}(R_2) + l_2 T_2 \mathbb{P}(\neg R_2).$$

This equation is similar to (4.4), except here there is a contribution of C_1 for the first phase, and the cost of taking one step is now greater than 1.

It follows that

$$\mathbb{P}(\neg R_2)C_2(l_2) = C_1 + C_{R_2}(l_2)\mathbb{P}(R_2) + \mathbb{P}(\neg R_2)l_2 T_2.$$

As was shown previously,

$$\begin{aligned} C_{R_2}(l_2)\mathbb{P}(R_2) &= \sum_{i=0}^{l_2-1} i \cdot T_2 q_2 (1-q_2)^i \\ &= (\frac{1-q_2}{q_2}\mathbb{P}(R_2) - l_2\mathbb{P}(\neg R_2))T_2. \end{aligned}$$

Therefore

$$C_2(l_2) = \frac{N_1}{\mathbb{P}(\neg R_2)} + \frac{1-q_2}{q_2} \cdot \frac{\mathbb{P}(R_2)}{\mathbb{P}(\neg R_2)} \cdot T_2. \tag{4.8}$$

Looking at this equation, we see that the expected cost of reaching $l_1^* + l_2$ for the first time is equal to (cost in phase 1)·(#times phase 1 is traversed) + $\mathbb{E}$(length of a run in phase 2)·$\mathbb{E}$(#failed runs in phase 2)·(step size in phase 2).

This cost must be at most M_2. It follows that

$$l_2^* = \left\lfloor \log\left(\frac{C_1 + \frac{1-q_2}{q_2}T_2}{w_2 + \frac{1-q_2}{q_2}T_2} \right) \Big/ \log(1-q_2) \right\rfloor, \tag{4.9}$$

We can now combine everything into the following theorem.

Theorem 4.2 *The optimal policy for two interrupting jobs is*

$$\begin{pmatrix} 1 & \overset{l_1^*}{\ldots} & 1 & 0 & \overset{l_2^*}{\ldots} & 0 & 0 & \overset{d-1-l_1^*-l_2^*}{\ldots} & 0 \\ 1 & \ldots & 1 & 1 & \ldots & 1 & 0 & \ldots & 0 \end{pmatrix},$$

where l_1^* and l_2^* are determined by (4.7) and (4.9).

4.5 n Interrupting jobs

The calculations are completely analogous to those used in the previous section. First we find

$$l_1^* = \lfloor \frac{\ln(1+(w_1-1)q_1)}{-\ln(1-q_1)} + 1 \rfloor. \tag{4.10}$$

Define $C_i(l_i)$ as the expected cost to reach the $i+1$-st phase for the first time, starting in state 0, where l_i is the length of phase i. Define furthermore $C_i = C_i(l_i^*)$. Thus we find

$$C_i(l_i) = C_{i-1} + (C_{R_i}(l_i) + C_i(l_i))\mathbb{P}(R_i) + l_i T_i \mathbb{P}(\neg R_i).$$

For every phase, this gives us a formula of the form (4.8). Thus for $i = 1, \ldots, n$ we find

$$l_i^* = \left\lfloor \log\left(\frac{C_{i-1} + \frac{1-q_i}{q_i} T_i}{w_i + \frac{1-q_i}{q_i} T_i} \right) \Big/ \log(1-q_i) \right\rfloor, \tag{4.11}$$

This leads to the following theorem.

Theorem 4.3 *For n interrupting jobs $w_1, \ldots, w_n$, the optimal policy for each w_i is given by*

$$(1, \overset{l_1^*+\cdots+l_i^*}{\ldots}, 1, 0, \ldots, 0),$$

where the l_i's are given by (4.7) and (4.11).

4.6 General Markov chains

Finally, we consider the situation where n different jobs, connected by a Markov chain $\mathcal{M}$, can interrupt the scheduler's job J. We assume that $\mathcal{M}$, including the 'idle' node, denoted by 0, is irreducible (all states communicate), and that $\mathcal{M} \backslash \{0\}$ is acyclic.

Node i of $\mathcal{M}$ represents a job of size w_i $(i = 1, \ldots, n)$. For each node i, the costs associated with a restart are 0 and the cost of continuing (waiting) is w_i. Define $OUT(i)$ as the set of nodes j where edge (i, j) exists. The probability that a machine moves from node i to node j is denoted by p_{ij}. Recall that this is independent of the scheduler's decisions.

4.6.1 Definitions and notations

In the following, we always use the word *node* to refer to the state of the current machine, and keep using *time state* to refer to the state of J. When describing policies, we continue to use subscripts to indicate time states, and we introduce superscripts do indicate nodes (states of the machine). A policy a for this problem consists of n policies a^i, one for each node i, and we write $a^i = (a^i_0, a^i_1, \ldots, a^i_{d-1})$, where the subscript denotes the time state of J. $a^i_t = 1$ means that J will be restarted if i is visited in time state t, and $a^i_t = 0$ means J will continue. The optimal policy is deterministic and stationary.

A node i of $\mathcal{M}$ is said to be *blocking* in time state t if $a^i_t = 1$, and it is *reachable* in time state t if there exists a path in $\mathcal{M}$ from 0 to i without blocking nodes. We say that an algorithm *unblocks* node i in time state t when $a^i_{t-1} = 1$ and $a^i_t = 0$.

We introduce two important costs:

- $B_t(i)$ is the total cost of reaching time state t, starting in time state 0 in the idle node, if i is blocking before time state t.

- $U_t(i)$ is the total cost of reaching time state $t+1$, starting in time state 0 in the idle node, if i is unblocked in time state t.

Say a and b are time states and x and y are nodes in the chain. Assume $b \neq 0$. We will write

- $D^{a,x}_{b,y}$ is the cost of going **d**irectly (without restart) from $\{a, x\}$ to $\{b, y\}$ (which we call the goal),

- $p^{a,x}_{b,y}$ is the probability of this happening.

- $R^{a,x}_{b,y}$ is the *total* cost of this move, including possible **r**estarts and assuming no restarts take place in $\{a, x\}$.

Note that the optimal decision in $\{a, x\}$ does not depend on the number of times this node and state have been visited.

We write $D^{a,x}_{b,y}(\neg i)$ and $p^{a,x}_{b,y}(\neg i)$ to indicate costs and probabilities when it is assumed that node i is not visited in the meanwhile. If $a = t$, but $b \leq t+1$, then it is assumed that $t+1$ is not reached before the goal. We use the notations $D^{a,x}_{0,0}$ and $p^{a,x}_{0,0}$ to indicate the cost and probability of a restart when starting in $\{a, x\}$. (As an example, $p^{t,0}_{0,0}(\neg i)$ is the probability of a restart, starting in time state t and node 0, without visiting node i or time state $t+1$.)

Finally, the *cost of node* x in time state t is given by $R^{t,x}_{t+1,0}$.

4.6.2 Nodes should be unblocked exactly once

Lemma 4.1 *From one time state to the next, when using the optimal policy, the cost of a node in $\mathcal{M}$ can never increase more than the cost of a restart in that same node.*

Proof. Take a time state t. We use an induction. Note that as t grows, the cost of restarting grows, because more work is lost by restarting. To simplify the wording, we will now color non-blocking nodes green and blocking nodes red.

Consider a green node i and suppose that the cost of all its successors has not increased more than that of a restart since time state $t-1$. In that case, no successor turned red.

We divide $OUT(i)$ in two sets, $OK_t(i)$ and $BAD_t(i)$, where the "bad" nodes are nodes where J is restarted in time state t. We need to show that the cost of i can not increase faster than the cost of a restart, which is $R^{0,0}_{t+1,0}$. Write

$$R^{t,i}_{t+1,0} = w_i + \sum_{k \in OUT(i)} p_{ik} z_k(t+1),$$

where

$$z_k(t+1) = \begin{cases} R^{t,k}_{t+1,0} & k \in OK_t(i) \\ R^{0,0}_{t+1,0} & k \in BAD_t(i) \end{cases}.$$

Write the increase in cost starting from i as $\delta = R^{t,i}_{t+1,0} - R^{t-1,i}_{t,0}$, the increase in the costs of a successor k as $\delta_k = z_k(t+1) - z_k(t)$ for all $k \in OUT(i)$ and finally the increase in the cost of a restart (while i is green, non-blocking) as $\delta_{YES} = R^{0,0}_{t+1,0} - R^{0,0}_{t,0}$.

We need to show $\delta \leq \delta_{YES}$. For $k \in OUT(i)$ we have three cases:

- k remained green, then $\delta_k = R^{t,k}_{t+1,0} - R^{t-1,k}_{t,0} < \delta_{YES}$ because of the induction hypothesis.
- k remained red. Then $\delta_k = R^{0,0}_{t+1,0} - R^{0,0}_{t,0} = \delta_{YES}$.
- k turned green: $\delta_k = R^{t,k}_{t+1,0} - R^{0,0}_{t,0} < R^{0,0}_{t+1,0} - R^{0,0}_{t,0} = \delta_{YES}$, otherwise k could not be green now.

In other words, $\delta_k \leq \delta_{YES}$ for all k, so $\delta = \sum_{k \in OUT(i)} p_{ik}\delta_k \leq \delta_{YES}$. □

Corollary 4.1 *If a node of $\mathcal{M}$ is non-blocking, it must remain non-blocking until the job is completed, unless the job is restarted.*

Proof. The previous lemma implies that if a node of $\mathcal{M}$ is non-blocking in time state t, it will not be blocking in $t+1$, for any t. □

4.6.3 Thresholds

We begin by looking at individual nodes, and show locally optimal strategies. Later we will describe the global policy.

If we write $\text{OPT}(t, i)$ for the optimal completion costs, starting in time state t and node i, we have that

$$\text{OPT}(t, i) = \min\{\text{OPT}(0, 0), w_i + \sum_{k \in OUT(i)} p_{ik}\text{OPT}(t, k)\}, \tag{4.12}$$

similar to the earlier cases. We do not have a simple interpretation for $\text{OPT}(0, 0) - \text{OPT}(t, k)$ anymore. Instead, we use the following lemma.

Lemma 4.2 *The optimal policy in time state t and node i can be determined by minimizing the cost from there until time state $t + 1$.*

Proof. An optimal policy will minimize the cost to reach t for all t: if it costs the policy more to reach t_1, it will cost more to reach any point beyond t_1. □

Let $\text{OPT}_{t+1}(t, i)$ denote the optimal cost of reaching $t + 1$ for the first time, starting in (t, i). Then $\text{OPT}_{t+1}(t, i) = \min\{\text{OPT}_{t+1}(0, 0), w_i + \sum_{k \in OUT(i)} p_{ik}\text{OPT}_{t+1}(t, k)\}$. Since the decision in time state t and node i is the same each time this pair is visited, we can in fact replace this equality by

$$\text{OPT}_{t+1}(t, i) = \min\{B_{t+1}(i), R^{t,i}_{t+1,0}\} \tag{4.13}$$

if we calculate these costs for the optimal policy. See Figure 4.3.

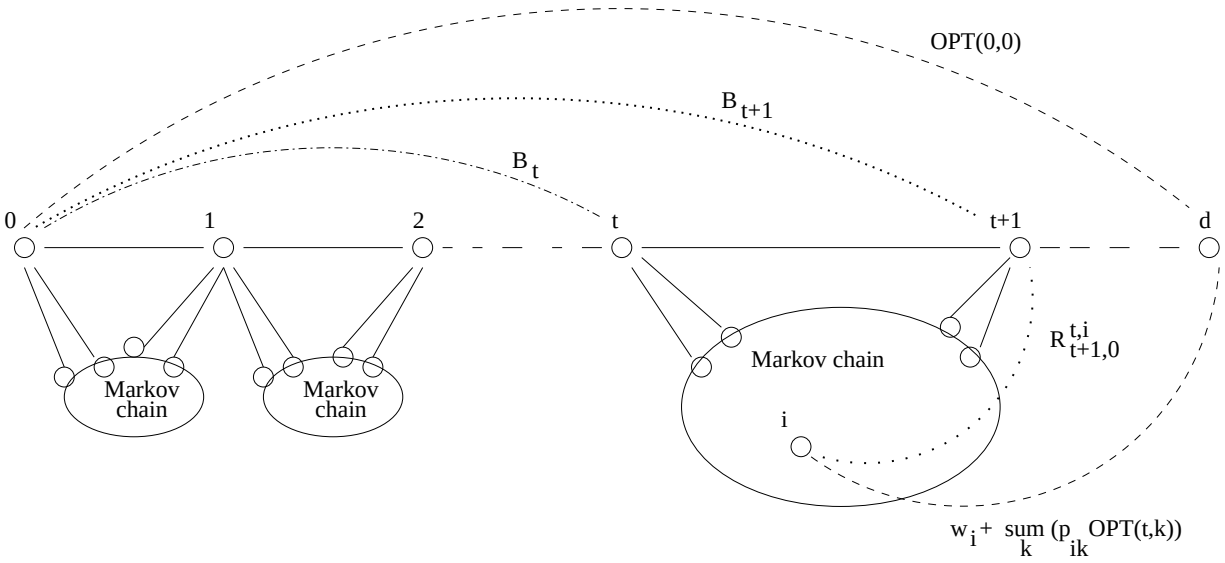

Figure 4.3: Markov chain of J in the general case

We will now formulate equations for three important costs. All costs naturally depend on the chosen strategy, but we will not denote this explicitly in every equation.

- $R^{t,i}_{t+1,0}$ is equal to the cost of node i, which is w_i, plus the expected cost if there is no restart, plus finally the expected cost if there is one. We have

$$R^{t,i}_{t+1,0} = w_i + p^{t,i}_{t+1,0} D^{t,i}_{t+1,0} + \; p^{t,i}_{0,0}(D^{t,i}_{0,0} + U_t(i)), \tag{4.14}$$

  where $p^{t,i}_{t+1,0} + p^{t,i}_{0,0} = 1$.

- Similarly, we can derive the following connection between $B_{t+1}(i)$ and $B_t(i)$:

$$\begin{aligned} B_{t+1}(i) \;\; = \;\; & B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + \; p^{t,0}_{t,i}(D^{t,0}_{t,i} + B_{t+1}(i)) \\ & + p^{t,0}_{0,0}(\neg i)(D^{t,0}_{0,0}(\neg i) + B_{t+1}(i)) \end{aligned}$$

  Thus,

$$\begin{aligned} B_{t+1}(i) \;\; = \;\; & \big(B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i} D^{t,0}_{t,i} \\ & + p^{t,0}_{0,0}(\neg i) D^{t,0}_{0,0}(\neg i)\big) \,/ p^{t,0}_{t+1,0}(\neg i). \end{aligned} \tag{4.15}$$

  Note that $p^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i} + p^{t,0}_{0,0}(\neg i) = 1$.

- For $U_t(i)$ we find similarly

$$\begin{aligned} U_t(i) \;\; = \;\; & \big(B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i}(D^{t,0}_{t,i} + R^{t,i}_{t+1,0}) \\ & + p^{t,0}_{0,0}(\neg i) D^{t,0}_{0,0}(\neg i)\big) \,/ (p^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i}). \end{aligned}$$

  Using (4.15), we can write this as

$$U_t(i) = \xi B_{t+1}(i) + (1-\xi) R^{t,i}_{t+1,0} \quad \text{for some } \xi \in [0,1]. \tag{4.16}$$

According to (4.13), the optimal policy in each node is to unblock it if this is cheaper than keeping it blocking; in other words, if $R^{t,i}_{t+1,0} < B_{t+1}(i)$. Note that if $t = 0$, restarting is always cheaper.

We are therefore especially interested in those values of t, where $R^{t,i}_{t+1,0} = B_{t+1}(i)$, because this is a time state where i should be unblocked. Using (4.16), this implies $R^{t,i}_{t+1,0} = U_t(i) = B_{t+1}(i)$. Using these equalities in (4.14), we find

$$B_{t+1}(i) = \frac{w_i + p^{t,i}_{t+1,0} D^{t,i}_{t+1,0} + p^{t,i}_{0,0} D^{t,i}_{0,0}}{p^{t,i}_{t+1,0}}. \tag{4.17}$$

Finally, we have from (4.15) that

$$B_t(i) = p^{t,0}_{t+1,0}(\neg i)\{B_{t+1}(i) - D^{t,0}_{t+1,0}(\neg i)\} - p^{t,0}_{0,0}(\neg i)D^{t,0}_{0,0}(\neg i) - p^{t,0}_{t,i}D^{t,0}_{t,i}.$$

Combining this with (4.17), we can see that $B_{t+1}(i) > R^{t,i}_{t+1,0}$ is equivalent to $B_t(i) > TH_t(i)$, where

$$\begin{aligned} TH_t(i) &= -p^{t,0}_{0,0}(\neg i)D^{t,0}_{0,0}(\neg i) - p^{t,0}_{t,i}D^{t,0}_{t,i} \\ &+p^{t,0}_{t+1,0}(\neg i)\left(\frac{w_i + p^{t,i}_{t+1,0}D^{t,i}_{t+1,0} + p^{t,i}_{0,0}D^{t,i}_{0,0}}{p^{t,i}_{t+1,0}} - D^{t,0}_{t+1,0}(\neg i)\right) \end{aligned} \tag{4.18}$$

$TH_t(i)$ is called the *threshold* of node i in time state t. Thresholds determine when nodes should be unblocked.

Lemma 4.3 *From time state t, the optimal policy is that the subset of blocking nodes remains constant until the first $t_0 \geq t$ for which there is a blocked node i for which $B_{t_0}(i) \geq TH_{t_0}(i)$ and $B_{t_0-1}(i) < TH_{t_0-1}(i)$. In time state t_0, i is unblocked.*

Proof. The threshold $TH_t(i)$ depends only on which nodes are blocking in time state t and which are not. This is because it consists of costs and probabilities of moves in $\mathcal{M}$ in time state t, without including costs after restarts. Therefore, as long as the subset of blocking nodes does not change, $TH_t(i)$ is a constant. Furthermore, $B_t(i)$ is strictly increasing in t, since it is not cheaper to reach t than it is to reach $t-1$.

This implies that, as long as the subset of non-blocking nodes does not change over time, for every node i there is one specific time state t_i, where for $t < t_i$ we have $R^{t,i}_{t+1,0} > B_{t+1}(i)$ and for $t \geq t_i$ we have $R^{t,i}_{t+1,0} \leq B_{t+1}(i)$. This time state is determined by the threshold for each node i. A node for which the threshold is first reached should be unblocked at that time. Before then, the optimal subset of blocking nodes does not change, because blocking nodes must never become non-blocking by Corollary 4.1. □

4.6.4 Calculating thresholds

We have seen that thresholds play an important part in a policy for this problem. We will now show how to calculate thresholds. According to (4.18), a threshold depends on no less than five different costs and five probabilities. But we can see immediately that $p^{t,i}_{t+1,0} + p^{t,i}_{0,0} = 1$ and $p^{t,0}_{0,0}(\neg i) + p^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i} = 1$. Consider a node i which has a set of outgoing edges $OUT(i)$. For each state t we divide $OUT(i)$ in two sets, $OK_t(i)$ and $BAD_t(i)$, where the "bad" edges lead to nodes where J is restarted in state t. The associated end nodes are also called bad.

We call 0 a good node; for the following calculations (equations (4.19)–(4.21)) we put $p^{t,0}_{t+1,0} = 1, D^{t,0}_{t+1,0} = 0$ and $w_0 = 0$ whenever they appear in a right member. The success

probability $p^{t,i}_{t+1,0}$ is the total probability of going to a good node, weighted by the success probabilities of those nodes:

$$p^{t,i}_{t+1,0} = \sum_{(i,k)\in OK_t(i)} p_{ik} \cdot p^{t,k}_{t+1,0}. \tag{4.19}$$

To calculate $D^{t,i}_{t+1,0}$, we must assume that J is not restarted, and therefore that one of the "good" outgoing edges is chosen when leaving i. If we normalize the transition probabilities on these "good" edges by putting $p'_{ik} = p_{ik} / \sum_{(i,i')\in OK_t(i)} p_{ii'}$, we get

$$D^{t,i}_{t+1,0} = \sum_{(i,k)\in OK_t(i)} p'_{ik}(w_k + D^{t,k}_{t+1,0}). \tag{4.20}$$

For $D^{t,i}_{0,0}$, a similar equation holds. If $p^{t,i}_{t+1,0} = 1$, we can put $D^{t,i}_{0,0} = 0$. Otherwise, since we assume that a restart actually occurs, we must disallow the edge $(i, 0)$ if it exists and in general any edge (i, k) for which $p^{t,k}_{t+1,0} = 1$. We then normalize the transition probabilities on the set of remaining edges, called $REM_t(i)$, by putting $p'_{ik} = p_{ik} / \sum_{(i,k)\in REM_t(i)} p_{ik}$. Note that $REM_t(i)$ may contain bad nodes. If the system moves to a bad node, no more costs are incurred, because the job is then immediately restarted. In a good node k however, an extra cost of expected size $w_k + D^{t,k}_{0,0}$ is incurred. Therefore

$$D^{t,i}_{0,0} = \sum_{(i,k)\in OK_t(i)\cap REM_t(i)} p'_{ik}(w_k + D^{t,k}_{0,0}). \tag{4.21}$$

Similar relations hold for $p^{t,0}_{t+1,0}(\neg i)$, $D^{t,0}_{t+1,0}(\neg i)$ and $D^{t,0}_{0,0}(\neg i)$, except that here it needs to be taken into account that i must not be visited. Finally, $D^{t,0}_{t,i}$ and $p^{t,0}_{t,i}$ can also be calculated using standard techniques. These last five values only change when a node gets allowed that is reachable from 0.

4.6.5 The algorithm

We are now finally ready to describe a method to determine the global strategy (for the whole Markov chain). We will eventually tag each node with the time state in which it is unblocked. We will begin at the "end" of $\mathcal{M}$ (which is well defined, since the chain is acyclic), and work our way back to nodes that can be reached from 0, each time calculating which node should be unblocked next. This way, some nodes will be unblocked before they can even be reached from 0, but this has no adverse effect on the costs of the algorithm.

The method is divided into steps. In each step x we calculate the next time step t_x on which a node i_x gets unblocked. We do this until all nodes are non-blocking (or the job finishes). Within each interval $[t_{x-1}, t_x - 1]$ the thresholds are constant. We put $i_0 = t_0 = 0$.

Each step x consists of a number of calculations. There are always a number of relevant nodes, for which some calculations are necessary, in the correct order. We will define two sets of nodes: A_x and B_x. Each step consists of the following sub-steps.

- Define A_x as the set of *non-blocking* nodes from which i_{x-1} can be reached. For every node i in this set, recalculate $D^{t,i}_{t+1,0}$, $D^{t,i}_{0,0}$ and $p^{t,i}_{t+1,0}$. Do this in reverse order (walking backwards through the graph), beginning with the nodes that have no successor in A_x.
- Define B_x as the set of *blocking* nodes from which 0 or a non-blocking node can be reached in a single step. Calculate the thresholds of this set in any order, using the new data from the set A_x when necessary.
- Determine the node $i_x \in B_x$ which first reaches its threshold. Calculate t_x and unblock i_x in time state t_x.

A few notes on this algorithm:

1. For blocking nodes $i \notin B_x$ we always have that the threshold is not reached yet, since it is always cheaper to restart immediately than to wait until the next (blocking) node and then to restart.
2. For nodes $i \in B_x \cap B_{x-1}$ from which i_{x-1} can not be reached, the threshold now is the same as in step $x - 1$.
3. It is possible for two or more nodes to have the minimal threshold value. In this case, unblock the last one in the acyclic ordering.

Theorem 4.4 *This algorithm is optimal.*

Proof. This follows from Lemma 4.3, the structure of the algorithm and the notes above. □

Theorem 4.5 *This algorithm runs in $O(n \cdot e)$ time, where e is the number of edges in $\mathcal{M}$.*

Proof. Consider one step in the algorithm. The calculations for A_x take $O(\text{number of outgoing edges from } A_x)$, which is certainly $O(e)$.

For B_x, some costs and probabilities starting in t need to be recalculated if i_{x-1} is reachable. This requires walking backwards through the graph starting in i_{x-1}, and doing $O(\text{number of outgoing edges})$ calculations in each node. Since each edge is used only once, and $\mathcal{M}$ is acyclic, the total costs of this are $O(e)$ as well.

The entire process therefore is $O(n \cdot e)$. □

A first-passage problem like this can also be solved using a linear program; however, in this case for each node in the Markov decision process a variable needs to be introduced. Solving a

linear programming problem with nd variables can be done in $O(d^3n^3)$ time. This is clearly far worse, especially if $\mathcal{M}$ is fairly small relative to the size of the job, so that $d >> n$. Moreover, this linear program needs to be solved for every occurring job size d. Since the time complexity is the third power of the job size itself, this is impractical.

4.6.6 On the use of this strategy

All the calculations for this strategy can be done before the job starts, and in fact this is necessary. When for every node it is known when it should be unblocked, this can be represented internally by tagging each node with the time state in which it is unblocked. Every time the machine switches to a different node, the scheduler can check its tag and compare it to the current time state to see whether the job needs to be restarted. Since the strategy does not depend on the length of the job, this tagging only needs to happen once and then many jobs could be run.

Finally, more than one scheduler can use this strategy: if all schedulers have different priorities, they can disregard jobs (and schedulers) with lower priorities. They can then construct their own Markov chain, their own model of the availability of the workstations, and run jobs (one scheduler must run only one job at a time). As long as there are not too many background jobs being run, the important assumption that it does not take too long to restart will still hold.

4.7 Conclusions

We have shown optimal policies for scheduling on Markovian machines, for several types of Markov chains, and an efficient way of calculating them. These policies can be readily extended to run many jobs simultaneously on one network, or to run jobs with checkpoints [26] (by considering each part of the job as a separate job). Also, note that our solutions do not depend on d, neither in their computational complexity, nor as input parameter. This means that it only needs to be computed once, for all possible occurring jobs. Then the nodes can be tagged with the time state in which they are unblocked, and any job can be run on the network.

An open question is whether it is possible to do this for a Markov chain that has cycles, since in our method and proofs we use heavily that it is possible to walk backwards through the graph. Perhaps in this case one would have to resort to using a linear programming formulation, using dn variables. This can be solved in $O(d^3n^3)$ time, which is much more than d itself. As noted, this is substantially worse than in the acyclic case, and requires new computations for every possible job size d. Hence, this would be an impractical approach.

Recall however, that we can approximate a cyclic Markov chain by our acyclic ones (after deleting $\{0\}$), see Section 4.1.

Chapter 5

Partial servicing of on-line jobs

We consider the problem of scheduling jobs on-line, where jobs may be served partially in order to optimize the overall use of the machines. Service requests arrive on-line to be executed immediately. The scheduler must decide how long and if it will run a job (that is, it must fix the Quality of Service level of the job) at the time of arrival of the job. We assume that preemption is not allowed.

We give lower bounds on the competitive ratio and present algorithms for jobs with varying sizes and for jobs with uniform size, and for jobs that can be run for an arbitrary time or only for some fixed fraction of their full execution time.

Partial execution or computation of jobs has been an important topic of research in several papers [10, 22, 29, 34, 54, 68, 69, 76, 77]. Some problems that have been considered are imprecise computation, anytime algorithms and two-level jobs (see below).

5.1 Introduction

Problem settings As before, we use competitive analysis to measure the quality of the scheduling algorithms. We consider three different settings.

- In section 5.2, we consider jobs with different job sizes. We show that the amount by which the sizes can differ determines how well an algorithm can do: if all job sizes are between 1 and M, the competitive ratio is $\Omega(\log M)$. We adapt the algorithm HARMONIC from [1] and show a competitive ratio of $O(\log M)$.

- Subsequently, and most important, we focus on scheduling uniform sized jobs, where each job can be run for any length of time between 0 and 1. In sections 5.3 and 5.4, we prove a randomized lower bound of 1.5, and we present a deterministic scheduling algorithm with a competitive ratio slightly above $2\sqrt{2} - 1 \approx 1.828$.

- Finally, in section 5.5 we consider the case of uniform sized jobs, where each job can only be run for either some fixed $\alpha < 1$ time or for 1 time. We derive a lower bound of $1 + \alpha - \alpha^2$.

The performance measure used is the total usage of all the machines (the total amount of time that machines are busy). For each job, a scheduling algorithm earns the time that it serves that job. The goal is to use the machines most efficiently, in other words, to serve as many requests as possible for as long as possible.

Throughout this chapter, $\varepsilon > 0$ is an arbitrarily small constant.

Applications As an example, one could consider the transmission of pictures or other multimedia data, where the quality of the transmission has to be set in advance (like quality parameters in JPEG), cannot be changed halfway and transmissions should not be interrupted.

Another example considers the scheduling of excess services. For instance, a (mobile) network guarantees a basic service per request. Excess quality in continuous data streams can be scheduled instantaneously if and when relevant, and if sufficient resources are available (e.g. available buffer storage at a network node).

Finally, when searching in multimedia databases, the quality of the search is adjustable. The decision to possibly use a better resolution quality on parts of the search instances can only be made on-line and should be serviced instantly if excess capacity is available [15].

Related Work We give a short overview of some related work.

In overloaded real-time systems, a well-known method to ensure graceful degradation is *imprecise computation* [34, 54, 68]. On-line scheduling of imprecise computation jobs is studied in [10, 69], but mainly on task sets that already satisfy the *(weak) feasible mandatory constraint*: at no time may a job arrive which makes it infeasible to complete all mandatory subtasks (for the off-line algorithm). This is quite a strong constraint.

Anytime algorithms are introduced in [29] and studied further in [77]. This is a type of algorithm that may be interrupted at any point, returning a result with a quality that depends on the execution time.

In [22], a model similar to the one in this chapter is studied, but on a single machine and using stochastic processes and analysis instead of competitive analysis. Jobs arrive in a Poisson process and can be executed in two ways, full level or reduced level. If they cannot start immediately, they are put in a queue. The execution of jobs can either be switched from one level to the other, or it cannot (as is the case in our model). For both cases, a threshold method is proposed: the approach consists of executing jobs on a particular level depending on whether the length of the queue is more or less than a parameter N. The performance of this algorithm,

which depends on the choice of N, is studied in terms of mean task waiting time, the mean task served computation time, and the fraction of tasks that receive full level computation. The user can adapt N to optimize his desired objective function. There are thus no time constraints (or deadlines) in this model, and the analysis is stochastic. In [76], this model is studied on more machines, again using probabilistic analysis.

5.2 Different job sizes

We begin with two cases where the ratio between the smallest and the biggest possible job is very large and show that in these cases it is not possible to get a good competitive ratio.

Then we show that even for limited ratios between job sizes, the competitive ratio of an on-line algorithm is not improved much by having the option of scheduling jobs partially. The most important factor is the size of the accepted and rejected jobs, and not how long they run.

The proofs in this section hold both when α is fixed, and when jobs may be run for an arbitrary length of time between 0 and 1.

Lemma 5.1 *If job sizes can vary without bound, no algorithm to schedule jobs on $m \geq 1$ machines can attain a finite competitive ratio.*

Proof. Suppose there is a $\mathcal{R}$-competitive on-line algorithm $\mathcal{A}$, and the smallest occurring job size is 1. We consider the job sequence $\{J_1, \ldots, J_{m+1}\}$ with sizes $w_1 = 1, w_2 = \mathcal{R}, w_i = \mathcal{R}^{i-1} (i = 3, \ldots, m), w_{m+1} = 2\mathcal{R}(1 + \ldots + \mathcal{R}^{m-1})$. Job J_i arrives at time $i\varepsilon$ where $\varepsilon < 1/(m+1)$. As soon as $\mathcal{A}$ refuses a job, the sequence stops and no more jobs arrive.

Suppose $\mathcal{A}$ refuses job J_i, where $i \leq m$. Then $\mathcal{A}$ earns at most $1 + \mathcal{R} + \ldots + \mathcal{R}^{i-2}$, while the adversary earns $1 + \mathcal{R} + \ldots + \mathcal{R}^{i-1}$. We have

$$\frac{1 + \mathcal{R} + \ldots + \mathcal{R}^{i-1}}{1 + \mathcal{R} + \ldots + \mathcal{R}^{i-2}} > 1 + \frac{\mathcal{R}^{i-1} - 1}{1 + \mathcal{R} + \ldots + \mathcal{R}^{i-2}} = 1 + \mathcal{R} - 1 = \mathcal{R}.$$

This implies $\mathcal{A}$ must accept the first m jobs. However, it then earns at most $1 + \ldots + \mathcal{R}^{m-1}$. The adversary serves only the last job and earns $2\mathcal{R}$ times as much. □

Note that this lemma holds even when all jobs can only run completely.

5.2.1 Two machines

Lemma 5.2 *If for all jobs J_i we have $1 \leq w_i \leq M$, then for all algorithms $\mathcal{A}$ on two machines we have*

$$\mathcal{R}(\mathcal{A}) \geq \sqrt{4M + 1} - 1.$$

Proof. Consider any on-line algorithm $\mathcal{A}$. The job sequence used is 1, $\frac{\sqrt{4M+1}-1}{2}$, $\frac{\sqrt{4M+1}-1}{2}$, M, M, where the jobs arrive ε time apart with $\varepsilon < 1/5$. The first job must be accepted by $\mathcal{A}$ to attain a finite competitive ratio, the second or the third to attain $\sqrt{4M+1}-1$. (If only the first job is accepted, the adversary can earn $\sqrt{4M+1}-1$.) The adversary then serves the last two jobs and earns $2M$, while $\mathcal{A}$ earns at most $\frac{\sqrt{4M+1}+1}{2}$ (less if some job is not run completely). □

Algorithm $\mathcal{A}_M$ for two machines is defined as follows: if both machines are available, it accepts any job. Otherwise, it accepts only jobs with size at least $\sqrt{M+1}-1$. All accepted jobs are run completely.

Since $\mathcal{A}_M$ runs all jobs completely, the strongest adversary is one that may serve jobs for any length of time. This adversary is used in the following lemma.

Lemma 5.3 $\mathcal{R}(\mathcal{A}_M) = 2\sqrt{M+1}$.

Proof. If $\mathcal{A}_M$ can accept every job in σ, there is nothing to prove, because it runs all jobs completely. Suppose it refuses some jobs, then we need to show that $\mathcal{A}_M$ earns enough from the jobs it accepted. We can do this by allocating the missed earnings to the accepted jobs, and never allocating more than $(2\sqrt{M+1}-1)w$ of missed earnings to a job of size w. We distinguish two cases.

Case 1. For intervals in which $\mathcal{A}_M$ serves only one job at a time (it keeps one machine available during the complete execution of the jobs), we allocate all the jobs that arrive during such a job to that job. Say $\mathcal{A}_M$ runs a job of size w, then all of the jobs arriving while it runs have sizes less than $\sqrt{M+1}-1$.

The worst case is when the adversary runs two jobs of size w for $w-\varepsilon$ time, and then two jobs of size $\sqrt{M+1}-1-\varepsilon$, which arrive at time $t = w-\varepsilon$, completely. We have $\frac{\mathrm{OPT}(\sigma)}{\mathcal{A}_M(\sigma)} = \frac{2(w+\sqrt{M+1}-1-2\varepsilon)}{w} \to 2\sqrt{M+1}$ for $w = 1$ and $\varepsilon \to 0$.

Case 2. If $\mathcal{A}_M$ is running a job of size w_1, and another job arrives that $\mathcal{A}_M$ serves as well, we can allocate the missed earnings to this pair of jobs. Consider $w_1 = 1, w_2 = \sqrt{M+1}-1$. The adversary can serve two jobs of size 1 for $1-\varepsilon$ time, and then two jobs of size M. If $w_1 = 1+\varepsilon_2$, the adversary can earn $2\varepsilon_2$ more while $\mathcal{A}_M$ earns ε_2 more than when $w_1 = 1$, so that its relative performance improves. If $w_2 > \sqrt{M+1}-1$, the adversary earns nothing more. This shows that our choices of w_1 and w_2 represent the worst case. We have $\frac{\mathrm{OPT}(\sigma)}{\mathcal{A}_M(\sigma)} = \frac{2M+2-2\varepsilon}{\sqrt{M+1}} = 2\sqrt{M+1}$.

Suppose a job arrives after either the job of size w_1 or the job of size w_2 is finished, but before both are finished. Only if $\mathcal{A}_M$ serves it do the possible missed earnings of $\mathcal{A}_M$ increase, because this creates a new interval in which $\mathcal{A}_M$ cannot serve any jobs. However, in this case

the new job has size at least $w_3 = \sqrt{M+1}$ and the missed earnings increase by at most w_3, because the total earnings of the adversary increase by at most $2w_3$. □

Note that an on-line algorithm can do at best only slightly better than $\mathcal{A}_M$ if it runs some jobs partially, because Lemma 5.2 implies $\mathcal{A}_M$ is already almost optimal.

5.2.2 $m > 2$ machines

Lemma 5.4 *For $M > 2^m$, we have $\mathcal{R}(\mathcal{A}) > m$ for all algorithms $\mathcal{A}$ on m machines.*

Proof. Let $a = \sqrt[m]{M} - 1$ and suppose $\mathcal{A}$ maintains a competitive ratio of m. If $M > 2^m$, then $a > 1$. Consider the following job sizes: $w_0 = 1,\ w_i = a(1+a)^{i-1} (i = 1, \ldots, m-1), w_m = M$. We build a job sequence in steps. In each step $i \geq 0$, at most m jobs of size w_i arrive ε time apart with $\varepsilon < 1/m^3$. We move to the next step as soon as $\mathcal{A}$ accepts a job. The sequence stops when $\mathcal{A}$ refuses m jobs of a certain size. This happens in step m at the latest.

$\mathcal{A}$ has to accept the first job (with size 1) and one of every size w_i ($1 < i < m$). For instance, if it refuses m jobs of size w_2, it earns at most $1 + w_1 = 1 + a$ while the adversary can earn mw_2, and then $\mathcal{R}(\mathcal{A}) \geq \frac{mw_2}{1+a} = ma > m$, a contradiction. Therefore, $\mathcal{A}$ earns at most $1 + w_1 + \ldots + w_{m-1} = (1+a)^{m-1} = (\sqrt[m]{M})^{m-1}$, and we have $\mathcal{R}(\mathcal{A}) \geq \frac{mM}{(\sqrt[m]{M})^{m-1}} = m\sqrt[m]{M} > ma > m$. □

Corollary For $M > 2^m$, we have $\mathcal{R}(\mathcal{A}) > m(\sqrt[m]{M} - 1)$ for all algorithms $\mathcal{A}$.

We can also show a bound of $\log M$ by using a method similar to [1]. This bound holds for all M and m (also if m is large). However, if $M > 2^m$, the bound above is still the strongest.

Theorem 5.1 *For the competitive ratio $\mathcal{R}$ of this scheduling problem with different job sizes we have $\mathcal{R} > \log M$.*

We will first introduce a job sequence, and then show that it implies the theorem. The adversary generates the job sequence in steps; in each step i, m jobs arrive of size i. This process ends when (and if) the on-line algorithm has assigned each machine to a job.

Suppose there is a $\mathcal{R}$-competitive on-line algorithm $\mathcal{A}$. To maintain a competitive ratio of $\mathcal{R}$, $\mathcal{A}$ has to serve at least $m/\mathcal{R}$ jobs of the first m jobs. In the second step, it must serve x jobs so that $2m/(m/\mathcal{R} + 2x) \leq \mathcal{R}$. It follows that $x \geq m/(2\mathcal{R})$. Note that if $\mathcal{A}$ serves more jobs in the first step, it has to use more machines in total in the first two steps. In step i, $\mathcal{A}$ has to serve $m/i\mathcal{R}$ jobs, earning $m/\mathcal{R}$ in each step. This means that $\mathcal{A}$ will exhaust its supply of machines in step k, where k is the smallest solution of $\frac{m}{\mathcal{R}}(1 + \frac{1}{2} + \ldots + \frac{1}{k}) \geq m$. It follows that

$$\mathcal{R} \leq H_k = \sum_{i=1}^{k} \frac{1}{i}, \tag{5.1}$$

and that $\mathcal{A}$ has then earned $km/\mathcal{R}$. Note that serving some jobs partially only lowers $\mathcal{A}$'s profit. We must have $k \leq M$ for this method to work, otherwise not all machines are busy after the jobs of size M.

On the other hand, if this method works the adversary can serve a job of size M on every machine after $\mathcal{A}$ has used all its machines. Then we have $\mathcal{R} \geq \frac{mM}{km/\mathcal{R}}$, implying that $k \geq M$. We conclude that $k = M$ in this case. We are now ready to prove the theorem.

Proof. Suppose there exists an on-line algorithm $\mathcal{A}$ with competitive ratio $\mathcal{R} \leq H_{M-1}$. Then the above job sequence will cause $\mathcal{A}$ to use all its machines in the final step, since (5.1) is satisfied with $k = M$. Consider step $k-1$. After this step, m jobs of size M arrive, which are all served by the adversary. We find that

$$\mathcal{R} \geq \frac{Mm}{\frac{m(k-1)}{\mathcal{R}} + M(m - \frac{mH_{k-1}}{\mathcal{R}})} \Rightarrow \mathcal{R} \geq H_{k-1} + \frac{1}{M} > H_{M-1}.$$

This is a contradiction. We conclude $\mathcal{R} > H_{M-1} \geq \log M$. □

Compare this result to [1], where a central server had to decide which movies to show on a limited number of channels. Each movie has a certain value determined by the amount of people that have requested that movie, and the goal is to use the channels most profitably. The result there is $\mathcal{R} \geq \log \eta$, where η is the total number of customers divided by the number of channels.

Algorithm ADAPTED HARMONIC [1] divides the machines into $M-1$ sets $S_1, \ldots, S_{M-1}$. Each set S_i contains $\lfloor \frac{m}{iH_{M-1}} \rfloor$ machines. Machines in set S_i serve only jobs of size at least i. They are served completely.

Again we use the strongest possible adversary, that can run jobs for any length of time, in the following proof. We will show that ADAPTED HARMONIC is only a constant factor away from the optimal competitive ratio. This implies that an algorithm that serves some jobs partially could do at most a constant factor better. This is independent of which running times are allowed for the jobs.

Theorem 5.2 $\mathcal{R}(\text{ADAPTED HARMONIC}) = O(\log M)$.

Proof. For every amount of sets that are busy in ADAPTED HARMONIC's schedule π, we will show that the adversary cannot earn more than $3H_{M-1}$ times what ADAPTED HARMONIC earns on the busy sets.

Denote the end time of π by t and consider the longest job J_j that finishes in the interval $[t-1, t]$. We distinguish two cases, depending on whether or not all machines are busy at some point during the execution of this job.

Case 1 Suppose all machines are in use by ADAPTED HARMONIC. Then it earns at least $m(M-1)/H_{M-1}$, while an adversary can earn at most $2mM$ in the worst case, by using all its machines continuously while these jobs are running and serving an additional m jobs of size M that arrive just before one of the machines becomes available again. ADAPTED HARMONIC serves none of these last jobs and we have

$$\frac{\text{OPT}(\sigma)}{\text{ADAPTED HARMONIC}(\sigma)} \leq \frac{2mMH_{M-1}}{m(M-1)} = \frac{2M}{M-1}H_{M-1} < 2H_{M-1}+1.$$

Case 2 If not all machines are in use, ADAPTED HARMONIC will only refuse jobs that are currently too small for any non-filled set. Suppose all sets $S_1, \ldots, S_k$ are in use. Then ADAPTED HARMONIC earns at least mk/H_{M-1}. The adversary can earn at most $m(2k+1-\varepsilon)$, by using all its machines continuously while these jobs are running and serving m additional jobs of size $k+1-\varepsilon$, which arrive just before ADAPTED HARMONIC's machines become available again. We have

$$\frac{\text{OPT}(\sigma)}{\text{ADAPTED HARMONIC}(\sigma)} \leq \frac{m(2k+1)}{mk/H_{M-1}} \leq 3H_{M-1}.$$

We can now remove all jobs that finish after J_j starts from π and repeat these arguments. □

5.3 Uniform job sizes

We will now study the case of identical job sizes, which we take to be 1.

5.3.1 Lower bounds

The simplest algorithm is GREEDY, which serves all jobs completely if possible. Clearly, GREEDY maintains a competitive ratio of 2, because it can miss at most 1 in earnings for every job that it serves. The following lemma shows that, like in section 5.2, the case of two machines forms an exception where partial scheduling is not advantageous.

Lemma 5.5 *For two machines and jobs of size 1,* GREEDY *is optimal among algorithms that are free to choose the execution times of jobs between 0 and 1, and it has a competitive ratio of 2.*

Proof. Assume some algorithm $\mathcal{A}$ has a competitive ratio less than 2, say $2-\delta$ where $\delta > 0$. The adversary lets two jobs arrive at time $t = 0$. Say $\mathcal{A}$ serves them for $0 < \alpha_1 \leq 1$ and $\alpha_1 \leq \alpha_2 \leq 1$ time respectively. (If $\alpha_1 = 0$, $\mathcal{A}$ earns at most 1 and has a competitive ratio of at least 2.) At time $t = \alpha_1 - \varepsilon$, two jobs arrive. The adversary can now earn $2(1+\alpha_1-\varepsilon)$, while $\mathcal{A}$ earns $\alpha_1+\alpha_2$. We have $\frac{2(1+\alpha_1-\varepsilon)}{\alpha_1+\alpha_2} \geq 2(1-\frac{\varepsilon}{1+\alpha_1}) > 2-\delta$ for ε small enough: a contradiction. □

We give a lower bound for the general case, which even holds for randomized algorithms.

Lemma 5.6 *For jobs of size 1 on $m > 2$ machines, no (randomized) algorithm that is free to choose the execution times of jobs between 0 and 1 can have a competitive ratio lower than $3/2$.*

Proof. We use Yao's Minimax Principle [74].

We examine the following class of random instances. At time 0, m jobs arrive. At time $0 < t \leq 1$, m more jobs arrive, where t is uniformly distributed over the interval $(0, 1]$. The expected optimal earnings are $3m/2$: the first m jobs are served for such a time that they finish as the next m jobs arrive, which is expected to happen at time $1/2$; those m jobs are served completely.

Consider a deterministic algorithm $\mathcal{A}$ and say $\mathcal{A}$ earns a on running the first m jobs (partially). If $\mathcal{A}$ has $v(t)$ machines available at time t, when the next m jobs arrive, it earns at most an additional $v(t)$. Its expected earnings are at most $a + \int_{t=0}^{1} v(t)dt = m$, since $\int_{t=0}^{1} v(t)dt$ is exactly the earnings that $\mathcal{A}$ missed by not serving the first m jobs completely: $a = m - \int_{t=0}^{1} v(t)dt$. Therefore $\mathcal{R}(\mathcal{A}) \geq 3/2$. □

5.3.2 Algorithm SL

We now present an algorithm SL which makes use of the possibility of choosing the execution time. Although SL could run jobs for any time between 0 and 1, it runs all jobs either completely (*long* jobs) or for $\frac{1}{2}\sqrt{2}$ of the time (*short* jobs). We denote the number of running jobs of these types at time t by $l(t)$ and $s(t)$. The arrival time of job J_j is denoted by t_j.

The idea is to make sure that each short job is related to a unique long job which starts earlier and finishes later. To determine which long jobs to use, marks are used. Short jobs are never marked. Long jobs get marked to enable the start of a short job, or when they have run for at least $1 - \frac{1}{2}\sqrt{2}$ time. The latter is because a new short job would always run until past the end of this long job. In the algorithm, at most $s_0 = \lceil (3 - \sqrt{2})m/7 \rceil \approx 0.22654 \cdot m$ jobs are run short simultaneously at any time. We will ignore the rounding and take $s_0 = (3 - \sqrt{2})m/7$ in the calculations. The algorithm is defined as follows.

Algorithm SL If a job arrives at time t, refuse it if all machines are busy.

If a machine is available, first mark all long jobs J_j for which $t - t_j \geq 1 - \frac{1}{2}\sqrt{2}$. Then if $s(t) < s_0$ and there exists an unmarked long job $J_{j'}$, run the new job for $\frac{1}{2}\sqrt{2}$ time and mark $J_{j'}$. Otherwise, run it completely.

Theorem 5.3 SL *maintains a competitive ratio of*

$$\mathcal{R} = 2\sqrt{2} - 1 + \frac{8\sqrt{2} - 11}{m} \approx 1.8284 + \frac{0.31371}{m},$$

where m is the number of machines.

Proof. We will give the proof in the next section.

5.4 Analysis of Algorithm SL

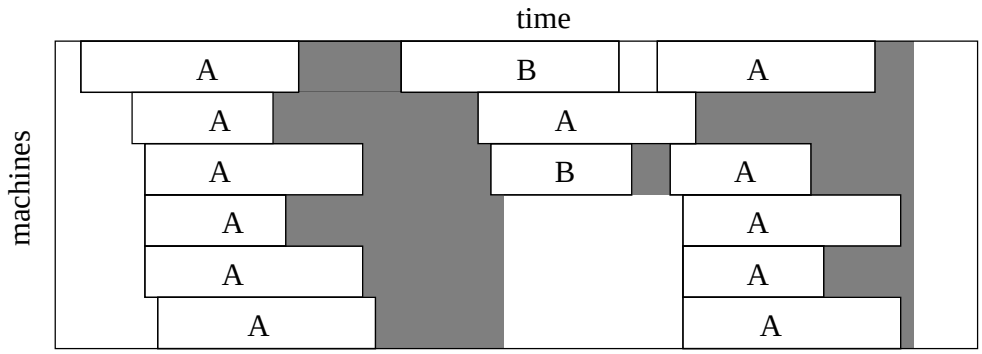

Figure 5.1: A run of SL

Below, we analyze the performance of algorithm SL, which was given in Section 5.3, and prove Theorem 5.3.

Consider a run of SL as in Figure 5.1. We introduce the following concepts.

- A job is of type A if at some moment during the execution of the job, all machines are used; otherwise it is of type B. (The jobs are marked accordingly in Figure 5.1.)
- *Lost earnings* are earnings of the adversary that SL misses. (In Figure 5.1, the lost earnings are marked grey.) Lost earnings are caused because jobs are not run or because they are run too short.
- A job or a set of jobs *compensates* for an amount a of lost earnings, if SL earns b on that job or set of jobs and $(a + b)/b \leq \mathcal{R}$ (or $a/b \leq \mathcal{R} - 1$). I.e., it does not violate the anticipated competitive ratio $\mathcal{R}$.
- A *critical interval* is an interval of time in which SL is using all its machines, and no jobs start or finish. (Such an interval is called critical because only in those intervals does SL refuse jobs, causing lost earnings.)

A job of type B can only cause lost earnings when it is run short, because no job is refused during the time a job of type B is running. However, this causes at most $1-\frac{1}{2}\sqrt{2}$ of lost earnings, so there is always enough compensation for these lost earnings from this job itself.

When jobs of type A are running, the adversary can earn more by running any short jobs among them longer. But it is also possible that jobs arrive while these jobs are running, so that they have to be refused, causing even more lost earnings. The worst case is if n jobs arrive simultaneously just before the earliest job finishes. This explains why the lost earnings in Figure 5.1 after the sets of jobs of type A have a shared endpoint on all the machines. We will show that SL compensates for these lost earnings as well.

We begin by deriving some general properties of SL. Note first of all that if m jobs arrive simultaneously when all of SL's machines are empty, it serves s_0 of them short and earns $\frac{1}{2}s_0\sqrt{2} + (m - s_0) = (6 + 5\sqrt{2})m/14 \approx 0.93365m$. We denote this amount by x_0.

Properties of SL

1. Whenever a short job starts, a (long) job is marked that started earlier and that will finish later. This implies $l(t) \geq s(t)$ for all t.

2. When all machines are busy at some time t, SL earns at least x_0 from the jobs running at time t. (Since $s(t) \leq s_0$ at all times.)

3. Suppose that two consecutive jobs, J_a and J_b, satisfy that $t_b - t_a < 1 - \frac{1}{2}\sqrt{2}$ and that both jobs are long. Then $s(t_b) = s_0$ (and therefore $s(t_a) = s_0$), because J_b was run long although J_a was not marked yet.

Lemma 5.7 *If at some time t all machines are busy, at most $m - s_0$ jobs running at time t will still run for $\frac{1}{2}\sqrt{2}$ or more time after t.*

Proof. Suppose all machines are busy at time t. Consider the set L of (long) jobs that will be running for more than $\frac{1}{2}\sqrt{2}$ time, and suppose it contains $k \geq m - s_0 + 1$ jobs. We derive a contradiction.

Denote the jobs in L by $J_1, \ldots, J_k$, where the jobs are ordered by arrival time. At time t_k, the other jobs in L must have been running for less than $1 - \frac{1}{2}\sqrt{2}$ time, otherwise they would finish before time $t + \frac{1}{2}\sqrt{2}$. This implies that jobs in L can only be marked because short jobs started.

Also, if at time t_k we consider J_k not to be running yet, we know not all machines are busy at time t_k, or J_k would not have started. We have

$$m > s(t_k) + l(t_k) \geq s(t_k) + m - s_0,$$

so $s(t_k) < s_0$. Therefore, between times t_1 and t_k, at most $s(t_k) \leq s_0 - 1$ short jobs can have been started and as a consequence, less than s_0 jobs in L are marked at time t_k. But then there is an unmarked job in L at time t_k, so J_k is run short. This contradicts $J_k \in L$. □

This lemma implies that the length of a critical interval is at most $\frac{1}{2}\sqrt{2}$.

We denote the jobs that SL runs during a critical interval I by $J_1^I, \ldots, J_m^I$, where the jobs are ordered by arrival time. We denote the arrival times of these jobs by $t_1^I, \ldots, t_m^I$; I starts at time t_m^I. We will omit the superscript I if this is clear from the context. We denote the lost earnings that are caused by the jobs in I by X_I.

We say that a job sequence *ends with a critical interval*, if no more jobs arrive after the end of the last critical interval that occurs in SL's schedule. In subsection 5.4.1, we will show that SL can compensate for the lost earnings X_I if it ends with a critical interval I. In 5.4.2, we will generalize this result to job sequences that end with a group of critical intervals. Finally we prove Theorem 5.3 in 5.4.3.

5.4.1 One critical interval

Lemma 5.8 *If a job sequence ends with a critical interval I, then* SL *can compensate for the lost earnings X_I.*

We begin by showing a special case of Lemma 5.8 and then prove the general case.

Lemma 5.9 *If a job sequence ends with a critical interval I, and no other jobs besides $J_1^I, \ldots, J_m^I$ arrive in the interval $[t_1^I, \ldots, t_m^I]$, then* SL *can compensate for the lost earnings X_I.*

Proof. Note that J_1 is long, because a short job implies the existence of an earlier, long job in I by Property 1. SL earns at least x_0 from $J_1, \ldots, J_m$ by Property 2. There are three cases to consider, depending on the size and timing of J_2.

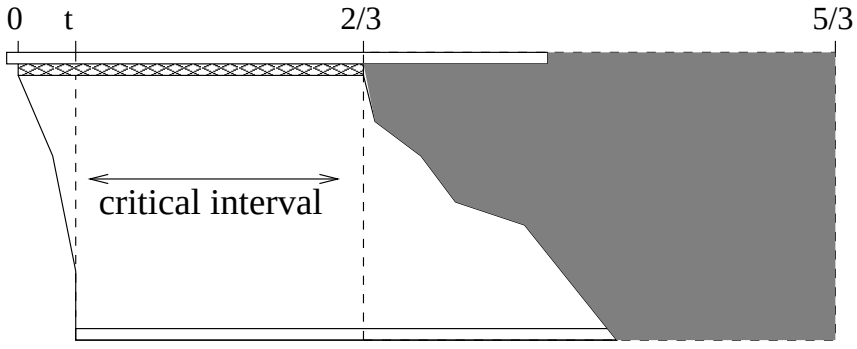

Figure 5.2: J_2 is short

Case 1. *J_2 is short.* See Figure 5.2, where we have taken $t_2 = 0$. Note that J_1 must be the job that is marked when J_2 arrives, because any other existing jobs finish before I starts and hence before J_2 finishes. Therefore, $t_2 - t_1 < 1 - \frac{1}{2}\sqrt{2}$, so before time t_2 the adversary and SL earn less than $1 - \frac{1}{2}\sqrt{2}$ from job 1. After time t_2, the adversary earns at most $(1 + \frac{1}{2}\sqrt{2})m$ from $J_1, \ldots, J_m$ and the jobs that SL refuses during I. We have

$$(1 + \frac{1}{2}\sqrt{2})m + (1 - \frac{1}{2}\sqrt{2}) = \mathcal{R} \cdot x_0,$$

so SL compensates for X_I.

Case 2. *J_2 is long and $t_2 - t_1 < 1 - \frac{1}{2}\sqrt{2}$.*

Since no job arrives between J_1 and J_2, we have by Properties 3 and 1 that $s(t_1) = s_0$ and $l(t_1) \geq s_0$. Denote the set of short jobs running at time t_1 by S_1 and the set of long jobs that were marked because of these short jobs by L_1. Then L_1 contains s_0 jobs. All jobs in $S_1 \cup L_1$ finish before I. (During I, SL does not start or finish any jobs.)

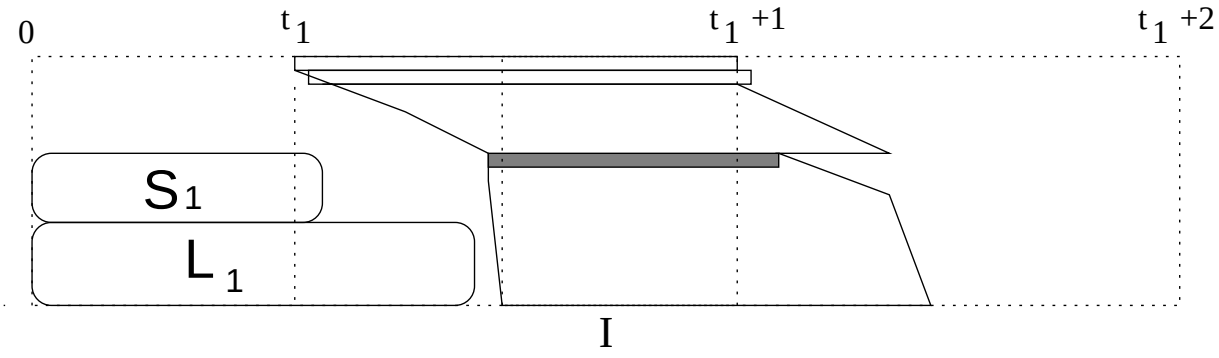

Figure 5.3: J_2 is long

Case 2a. *There is no critical interval while the jobs in S_1 and L_1 are running.*

Hence, the jobs in S_1 and L_1 are of type B. We consider the jobs that are running at time t_1 and the later jobs. After time t_1 the adversary earns at most $2m$, because I ends at most at time $t_1 + 1$. SL earns $\frac{1}{2}s_0\sqrt{2} + s_0$ from S_1 and L_1 and at least x_0 on the rest. For the adversary, we must consider only the earnings on S_1 and L_1 before time t_1; this is clearly less than $\frac{1}{2}s_0\sqrt{2} + s_0$.

We have

$$\frac{2m + \frac{1}{2}s_0\sqrt{2} + s_0}{x_0 + \frac{1}{2}s_0\sqrt{2} + s_0} < \mathcal{R}.$$

This shows SL compensates for X_I (as well as for the lost earnings caused by S_1 and L_1).

Case 2b. *There exists a critical interval before I which includes a job from S_1 or L_1.*

Call the earliest such interval I_2. If I_2 starts after t_1, we can calculate as in Case 2a. Otherwise, we consider the earnings on each machine after the jobs in I_2 started. Say the first job in S_1 starts at time t'. See Figure 5.4.

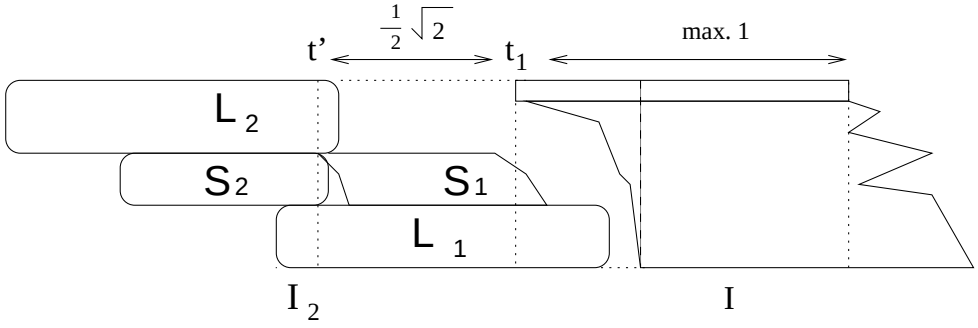

Figure 5.4: J_2 is long and there is another critical interval

Denote the end of I_2 by t''. Note that until time t'', SL and OPT earn exactly the same on the jobs in I_2. We assume that the adversary can keep all its machines occupied from the start of I_2 until 1 time past the end of I_1. Then it follows that any job that SL runs between I_2 and I_1 and is not in $S_1 \cup L_1$ improves the ratio for SL; we may assume there are no such jobs.

There are two cases: $t'' \leq t'$ and $t'' > t'$. If $t'' \leq t'$, the number of short jobs in I_2 should be maximized, because that maximizes the amount of lost earnings between the two critical intervals. Furthermore, the worst case is if all jobs in L_1 are in I_2, since for every long job that is in I_2 but not in L_1, SL and OPT earn 1: the number of these jobs should be minimized. Then $|L_2| = m - 2s_0$. We have $t_1 - t'' < 1$ and I_1 ends at most 1 after t_1. Hence the adversary earns at most

$$3m \text{ (after } t'') \ + |L_2| + \frac{1}{2}\sqrt{2}|S_2| = (3 + \frac{1}{2}\sqrt{2})m$$

whereas SL earns $2x_0 + |S_1| = (12 + 17\sqrt{2})/14$, and we have $\frac{\text{OPT}}{\text{SL}} = 2\sqrt{2} - 1 < \mathcal{R}$.

If $t'' > t'$, we have similarly that the worst case is if all jobs in L_1 and S_1 are in I_2, and $|S_1| = s_0$. We have $t_1 - t' < \frac{1}{2}\sqrt{2}$ so in this case the adversary earns at most

$$(2 + \frac{1}{2}\sqrt{2})m \text{ (after } t') \ + (1 - \frac{1}{2}\sqrt{2})|L_1| + |L_2| = \frac{38 + 6\sqrt{2}}{14}m$$

whereas SL earns $2x_0$. The ratio is less than $\mathcal{R}$.

Case 3. *J_2 is long and $t_2 - t_1 \geq 1 - \frac{1}{2}\sqrt{2}$.* We consider job J_3.

- If J_3 is short, then after time $t_1 + (1 - \frac{1}{2}\sqrt{2})$ the adversary earns at most $(1 + \frac{1}{2}\sqrt{2})m - (m-2)((t_3 - t_1) - (1 - \frac{1}{2}\sqrt{2})) - ((t_2 - t_1) - (1 - \frac{1}{2}\sqrt{2}))$. Before that time, it earns of

course $(1 - \frac{1}{2}\sqrt{2})$ (only counting the jobs in I). So in total, it earns less than it did in Case 1.

- If J_3 is long, we have two cases. If $t_3 - t_2 < 1 - \frac{1}{2}\sqrt{2}$, then again the sets S_1 and L_1 are implied and we are in Case 2. Finally, if $t_3 - t_2 \geq 1 - \frac{1}{2}\sqrt{2}$ we know that $t_4 - t_3 < 1 - \frac{1}{2}\sqrt{2}$, so this reduces to Case 1 or 2 as well.

In all cases, we can conclude that SL compensates for X_I. □

Proof of Lemma 5.8. We can follow the proof of Lemma 5.9. However, it is now possible that a short job J_1' starts after J_1, but finishes before I.

Suppose the first short job in I arrives at time $t' = t_1 + \tau$. If the job sets S_1 and L_1 exist, we can reason as in Case 2 of Lemma 5.9. Otherwise, all long jobs in I that arrive before time t_1' except for maybe one are followed by short jobs not in I. (If there are two such long jobs, they arrived more than $1 - \frac{1}{2}\sqrt{2}$ apart, and the adversary earns less than in Case 1 of Lemma 5.9 (cf. Case 3 of that lemma).)

For each pair of jobs (J_a, J_b), where J_a is long and $J_b \notin I$ is short, we have that J_b will run for at least $\frac{1}{2}\sqrt{2} - \tau$ more time after t', while J_a has run for at most τ time. One such pair is shown in Figure 5.5.

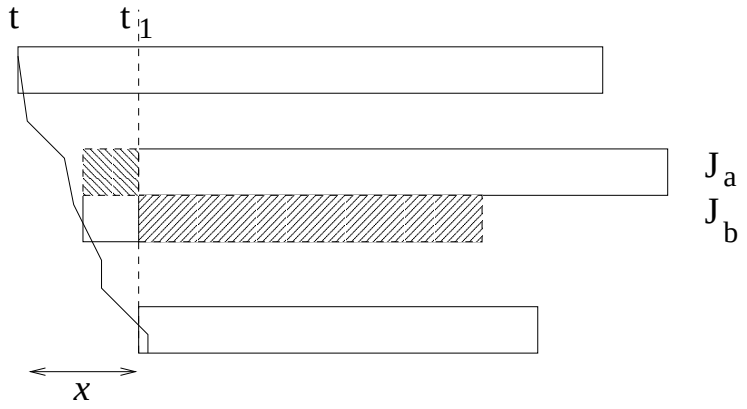

Figure 5.5: Pairs of long and short jobs

We compare the adversary's earnings now to its earnings in Case 1 of Lemma 5.9. Since $J_b \notin I$, it earns less on the machine running J_b and more on the machine running J_a (because there it earns something before time t', which was not taken into account earlier). If $\tau \leq \frac{1}{4}\sqrt{2}$, the adversary loses more on the machines running these pairs than it gains. On the other hand, if $\tau > 1 - \frac{1}{2}\sqrt{2}$, then I is shorter than $\frac{1}{2}\sqrt{2}$: the adversary earns $\tau - (1 - \frac{1}{2}\sqrt{2})$ less on every machine. □

5.4.2 Two or more critical intervals

It is possible that two or more critical intervals follow one another. In that case, we cannot simply apply Lemma 5.8 repeatedly, because some jobs may be running during two or more successive critical intervals. Thus, they would be used twice to compensate for different lost earnings. We now show that SL compensates for all lost earnings in this case as well. We begin with the special case of two critical intervals and then prove the general case.

Definition A *group* of critical intervals is a set $\{I_i\}_{i=1}^k$ of critical intervals, where I_{i+1} starts at most 1 time after I_i finishes $(i = 1, \ldots, k-1)$.

A job sequence *ends with a group of critical intervals* $\{I_i\}_{i=1}^k$, if SL only runs jobs from $\{I_i\}_{i=1}^k$ after I_1 starts.

Note that if there are small jobs that SL starts after one critical interval from a group and finishes before the next, this can only help SL, because it decreases the amount of lost earnings between those two intervals.

Lemma 5.10 *If a job sequence ends with a group of two critical intervals,* SL *compensates for the lost earnings X_{I_1} and X_{I_2}.*

Proof. If all jobs from I_1 have finished before I_2 starts, we know X_{I_1} is compensated for by Lemma 5.8. On I_2 we can also use Lemma 5.8 to see that X_{I_2} is compensated for as well. Note that no jobs are used twice to compensate for lost earnings.

In case some jobs from I_1 are still running in I_2, I_2 ends within 1 time of the start of I_1. We denote the set of long jobs that are running in I_1 but not in I_2 by L_1, the set of long jobs in I_2 but not in I_1 by L_2 and the set of long jobs in both by L_3. Define S_1, S_2 and S_3 analogously. The sizes of these sets are denoted by l_1, l_2, l_3, s_1, s_2 and s_3, respectively. Of the s_1 jobs in S_1, s_{11} jobs had jobs in L_1 marked when they started, and s_{13} jobs had jobs in L_3 marked.

Case 1. If I_2 starts within $\frac{1}{4}\sqrt{2}$ time after I_1 has finished, the jobs that are in I_1 but not in I_2 can compensate for X_{I_1}, since the size of such a job is at least twice the size of the lost earnings following it on the same machine.

Case 2. Say I_2 starts after $\frac{1}{4}\sqrt{2}$, but within $\frac{1}{2}\sqrt{2}$ time after I_1 has finished. Then $X_{I_1} \leq \frac{1}{2}\sqrt{2}(s_{11} + l_1) + (1 - \frac{1}{2}\sqrt{2})s_{13}$. SL earns $l_1 + \sqrt{2}(s_{11} + s_{13})/2$ from the jobs in S_1 and L_1. We have

$$\frac{\frac{1}{2}\sqrt{2}(s_{11} + l_1) + (1 - \frac{1}{2}\sqrt{2})s_{13}}{l_1 + \frac{1}{2}\sqrt{2}(s_{11} + s_{13})} \leq \mathcal{R} - 1 \text{ for all } s_{13}, l_1 \text{ and } s_{11}.$$

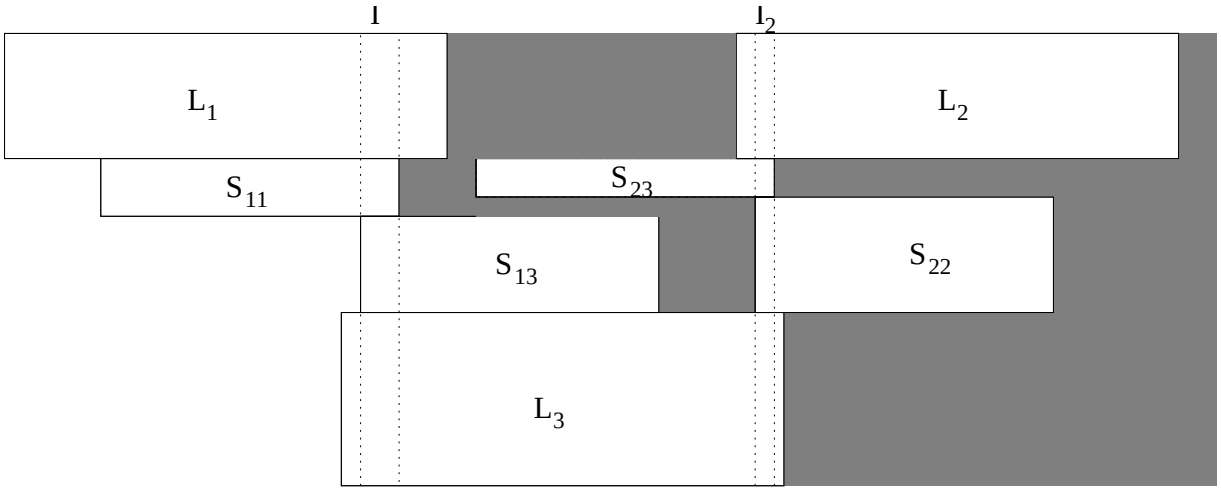

Figure 5.6: The jobs in two critical intervals divided into sets

(The worst case is $s_{13} = 0$ and $l_1 = s_{11}$, since $l_1 \geq s_{11}$). In other words, SL earns enough on S_1 and L_1 to compensate for X_{I_1}. Lemma 5.8 shows that X_{I_2} is compensated for too, again without using jobs twice to compensate for different things: S_1 and L_1 are not in I_2.

Case 3. Suppose I_2 starts more than $\frac{1}{2}\sqrt{2}$ time after I_1 has finished. We have $s_3 = 0$, $s_2 \leq s_0$ and $l_2 = m - l_3 - s_2$.

Case 3a. If $l_3 \leq s_0$, then $l_2 \geq m - 2s_0 \geq m/\mathcal{R}$. In a figure like Figure 5.6, we can move jobs around to aid in the calculations, without affecting profits. Here we move all the jobs in L_2 and S_2 forward, so that they start 1 time after the end of I_1. We then see that after that time, SL earns at least $m/\mathcal{R}$ while the adversary earns m, and before that time, we can apply Lemma 5.8.

Case 3b. The last case is $l_3 > s_0$ and I_2 starts more than $\frac{1}{2}\sqrt{2}$ after I_1. Since all jobs in L_3 start within $1 - \frac{1}{2}\sqrt{2}$ time (because I_1 and I_2 are at least $\frac{1}{2}\sqrt{2}$ apart), and $l_3 > s_0$, it must be that $s_1 = s_0$, because each job in L_3 is available to be marked for a short job before I_1 starts. But then SL already earns $x_0 + \frac{1}{2}s_0\sqrt{2}$ from S_1, L_2, S_2 and L_3 alone, because it earns at least x_0 from the jobs in I_2. Starting at the beginning of I_1, the adversary earns at most $2m$. Since

$$\frac{2m}{x_0 + \frac{1}{2}s_0\sqrt{2}} = \mathcal{R},$$

we have that SL compensates for both X_{I_1} and X_{I_2}. □

We now generalize this result in the following lemma.

Lemma 5.11 *If a job sequence ends with a group of critical intervals,* SL *compensates for all the lost earnings after the first critical interval.*

Proof. We use an induction on the number of critical intervals k. We have already proven the cases $k = 1$ and $k = 2$. Say $k \geq 3$, and consider the last three critical intervals, I_1, I_2 and I_3. See Figure 5.7.

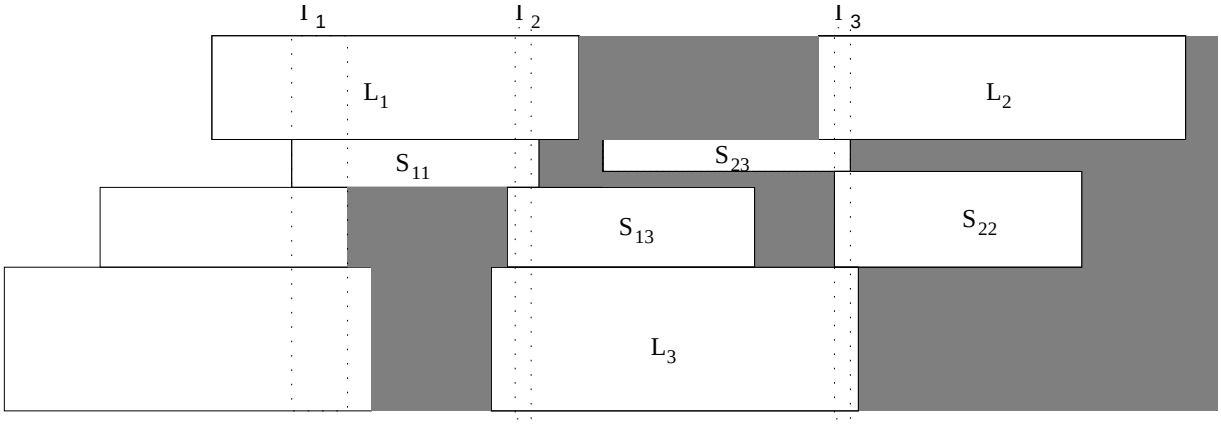

Figure 5.7: The last three critical intervals in a group

There are a number of cases to consider. We will abuse notation and use I_i to refer both to the i-th critical interval and the jobs that SL runs during I_i.

Case 1. There are no short jobs in $I_1 \cap I_2$. Denote the number of long jobs in $I_2 \cap I_3$ by l_3.

Case 1a. There are no short jobs in $I_2 \cap I_3$.

Case 1a1. $l_3 \leq s_0$. Now the jobs in $I_3 \backslash I_2$ are worth at least $m/\mathcal{R}$ and (using induction) we are done as in Lemma 5.10.

Case 1a2. $l_3 > s_0$. As in Lemma 5.10 we have that $s_2 = s_0$, where s_2 is the number of the short jobs in I_2. We find that these jobs together with the jobs in I_3 compensate for the lost earnings after I_2, and can use an induction for the rest, because we never need these jobs to compensate for earlier lost earnings.

Case 1b. There are short jobs in $I_2 \cap I_3$. The adversary earns at most $(1 + \frac{1}{2}\sqrt{2})m$ after I_2, while SL earns at least x_0 from all jobs after I_1. We are again done using induction.

Case 2. There are short jobs in $I_1 \cap I_2$.

Case 2a. $I_1 \cap I_3 = \emptyset$. By induction we know that the jobs up to and including those from I_1 can compensate for lost earnings until 1 time after the end of I_1. Since I_2 ends at most $\frac{1}{2}\sqrt{2}$ after the start of I_1, the jobs after I_1 have to compensate only for the lost earnings starting from $1 - \frac{1}{2}\sqrt{2}$ after I_2. From then on, the adversary earns at most $(1 + \frac{1}{2}\sqrt{2})m$ while SL earns at least x_0 from the jobs in I_3.

Case 2b. $I_1 \cap I_3 \neq \emptyset$. We have that I_3 finishes at most 1 after I_1 starts and we can treat this as a special case of two critical intervals, thus reducing this case to $k-1$ critical intervals. Note that the analysis in Lemma 5.10 does not assume that there are no critical intervals between I_1 and I_2. □

5.4.3 The competitive ratio of SL

Having shown Lemma 5.11, we are now ready to prove Theorem 5.3.

Theorem 5.3 SL *maintains a competitive ratio of* $\mathcal{R} = 2\sqrt{2} - 1 + \frac{8\sqrt{2}-11}{m}$.

Proof. If no jobs arrive within 1 time after a critical interval, the machines of both SL and the adversary are empty. New jobs arriving after that can be treated as a separate job sequence. Thus we can divide the job sequence into parts. The previous lemmas also hold for such a part of a job sequence.

Consider (a part of) a job sequence. All the jobs arriving after the last critical interval can be disregarded, since they are of type B: they compensate for themselves. Moreover, they can only decrease the amount of lost earnings caused by the last critical interval (if they start less than 1 after a critical interval).

If there is no critical interval, we are done. Otherwise, we can apply Lemma 5.11 and remove the last group of critical intervals from consideration. We can then remove the jobs of type B at the end and continue in this way to show that SL compensates for all lost earnings. □

5.5 Extensions of this model

5.5.1 Fixed levels

In this section, we study the case where jobs can only be run at two levels [22, 76]. This reduces the power of the adversary and should lower the competitive ratio. If the jobs can have different sizes, the proofs from Section 5.2 still hold. For the case of uniform jobs, we have the following bound.

Theorem 5.4 *If jobs can be run at two levels,* $\alpha < 1$ *and 1, then no algorithm can have a better competitive ratio than* $1 + \alpha - \alpha^2$.

Proof. Note that each job is run either for 0, α or 1 time. Let m jobs arrive at time $t = 0$. Say $\mathcal{A}$ serves φm jobs partially and the rest completely. It earns $(1 - \varphi + \alpha\varphi)m$. If this is less than $m/(1 + \alpha - \alpha^2)$ we are done. Otherwise, we have $\varphi \leq \frac{\alpha}{1+\alpha-\alpha^2}$. Another m jobs arrive at time

$t = \alpha$. $\mathcal{A}$ earns at most $(1 + \alpha\varphi)m$ in total, while the off-line algorithm can earn $m + m\alpha$. Since $\varphi \leq \frac{\alpha}{1+\alpha-\alpha^2}$, we have $r(\mathcal{A}) \geq \frac{1+\alpha}{1+\alpha\varphi} \geq 1 + \alpha - \alpha^2$. □

Note that for $\alpha = \frac{1}{2}\sqrt{2}$, SL yields a competitive ratio for this problem of at most 1.828 (but possibly much better). Extending these results to more values of α is an open problem.

5.5.2 Non-linear rewards

In many applications, it is reasonable to assume that serving a job partially, say for time α, an algorithm earns more than α (but strictly less than 1). For instance, if a video is transmitted using only half of all the pixel data, the perceived quality is much more than $1/2$ and therefore the algorithm should earn more, say $3/4$.

Consider rewards of the form $f(\alpha)$, where $f(0) = 0, f(1) = 1$, $f(\alpha) > \alpha$ and $f''(\alpha) < 0$ (concave) for all $0 < \alpha < 1$. One example is

$$f(\alpha) = \sqrt{\alpha}.$$

If α is fixed, the only change that this causes in the lower bound is that all earnings of α are replaced by $f(\alpha)$. The timing of the jobs by the adversary does not change, and the structure of the proof remains unchanged.

In other applications, $f(\alpha) < \alpha$ is possible. Then the above still holds as long as $f(\alpha)$ is monotonic non-decreasing. Other methods would be required for a non-monotonic $f(\alpha)$.

5.6 Conclusions

We have studied the problem of scheduling jobs that do not have a fixed execution time on-line. We have first considered the general case with different job sizes.

Subsequently, we have given a randomized lower bound of 1.5 and a deterministic algorithm with competitive ratio ≈ 1.828 for the scheduling of uniform jobs. An open question is by how much either the lower bound or the algorithm could be improved. Especially using randomization it could be possible to find a better algorithm.

An extension of this model is to introduce either deadlines or startup times, limiting either the time at which a job should finish or the time at which it should start. Finally, algorithms for fixed level servicing can be investigated.

Chapter 6

Minimizing the maximum starting time

In this chapter, we study the scheduling problem of minimizing the maximum starting time on-line. The goal is to minimize the last time that a job starts. We show that while the greedy algorithm has a competitive ratio of $\Theta(\log m)$, we can give a constant competitive algorithm for this problem. We also show that the greedy algorithm is optimal for resource augmentation in the sense that it requires $2m-1$ machines to have a competitive ratio of 1, whereas no algorithm can achieve this with $2m-2$ machines.

6.1 Introduction

The system that we study consists of three parts. The first part is the set of servers which run the jobs. There are m independent and identical such servers, without any communications channels between them. An additional server, called the input server, is used for communication. This server is in charge of giving input to the identical servers, and is the heart of the system. The third part is a scheduler. This is a computer which runs a scheduling algorithm to decide where each new job is going to run. The input server gets this information from the scheduler, and supplies the servers with all data needed to process the required jobs. The scheduler works in an on-line paradigm where jobs arrive one by one (each job is assigned to a machine without knowledge of future jobs). This needs to be done so that on arrival of a request, it is possible to give an immediate acknowledgement to it and to specify which server is going to run this job. However, the input server moves the data of each job to the server which is going to run it, just before the job starts running. The order in which jobs, that were assigned to a certain server, are given to it is the order of their arrival. See Figure 6.1. After a job has been processed, its output needs to be collected. An output collector is not a part of the system, and is not synchronized with it.

Since the input supplier is a communication channel between scheduler and servers, we

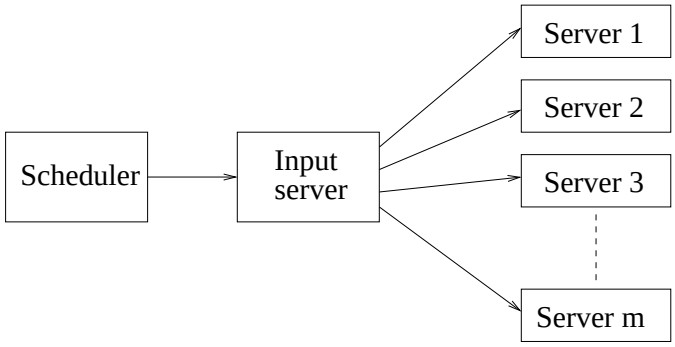

Figure 6.1: The system considered in this chapter

would like to free this server as soon as possible. Both in order to make it available for a different use, and to increase the chances of successful completion of the delivery of all jobs. To do that, the goal should be to minimize the maximum starting time of any job.

An example of this situation is the following. There is a loading station where trucks are loaded with goods. These goods need to be delivered to different places, after which the trucks return to the loading station to pick up a new load. At the end of a work day, the station can close as soon as the truck carrying the last load has left, and does not need to wait for the trucks to return. The loading station is the input server in our model, the servers are the trucks, and the goods are the jobs. The time it takes to deliver the goods in one truck is the size of the job. (Here we consider a truck load to be "one job", e. g. each truck contains only one item, or items for only one destination (client).)

We define the problem in more standard terms of machine scheduling of jobs in a list. We consider the problem of minimizing the maximum starting time. Jobs arrive on-line to be scheduled on m parallel machines. These machines can be either identical or *related*, in which case each machine has a speed that determines how long it takes to run one unit of work. We study the on-line paradigm where jobs arrive one by one. A job J_j is defined by its size and by its order in the input sequence. Denote the starting time of job J_j by S_j. We denote the cost of an algorithm $\mathcal{A}$ on a job sequence $\sigma = \{J_1, \ldots, J_n\}$ by $\mathcal{A}(\sigma) = \max_j S_j$. An algorithm is required to run the jobs on each machine in the order of arrival.

In this chapter, we consider both the competitive ratio of algorithms in the standard setting, and the amount of resource augmentation (in the form of extra machines) required to have a competitive ratio of 1. To the best of our knowledge, no previous work on the above goal function exists.

Note that if a sequence σ contains at most m jobs, then $\textsc{opt}(\sigma) = 0$. Hence, any algorithm with finite competitive ratio needs to have zero cost and run all jobs on different machines in that case.

We show the following results for the competitive ratio on identical machines:

- The greedy algorithm, which assigns each job to the least loaded machine, has competitive ratio $\Theta(\log m)$.

- The greedy algorithm has optimal competitive ratios for 2 and 3 machines, which are 2 and $5/2$ respectively.

- There exists a constant competitive algorithm BALANCE which has competitive ratio 12 for any m (hence the greedy algorithm is far from having optimal competitive ratio for general m).

- For any $\varepsilon > 0$, there exists a value m_1 so that for any $m_1 > m$, the competitive ratio of any on-line algorithm on m machines is at least $4 - \varepsilon$.

The last item implies that any algorithm that works on an arbitrary number of machines has a competitive ratio of at least 4.

For two related machines, we give a matching upper and lower bound of $q + 1$ for the competitive ratio, where q is the speed of the fastest machine relative to the slowest.

We show the following results for resource augmentation:

- The greedy algorithm has competitive ratio 1 if it uses $2m - 1$ machines (and is compared to an optimal off-line algorithm with m machines).

- Any on-line algorithm which uses $2m - 2$ machines has competitive ratio larger than 1, and any on-line algorithm which uses $2m - 1$ machines has competitive ratio of at least 1. Hence the greedy algorithm is optimal in this measure.

Note that the off-line version of minimizing the maximum starting time is strongly NP-hard. The off-line problem of minimizing the maximum completion time (minimizing the makespan) is a special case of our problem. A simple reduction from the makespan problem to our problem can be given by adding m very large jobs (larger than the sum of all other jobs) in the end of the sequence. Each machine is forced to have one such job, and the maximum starting time of the large jobs, is the makespan of the original sequence.

We present results on the greedy algorithm in Section 6.2, the constant competitive algorithm BALANCE in Section 6.3, lower bounds in Section 6.4, results for related machines in Section 6.5 and results for resource augmentation in Section 6.6.

6.2 The greedy algorithm

GREEDY always assigns an arriving job on the machine where it can start the earliest (see [41]). In some upper bound proofs we use the following definition: a *final* job is a job that starts as the last job on some machine in OPT's schedule.

Theorem 6.1 GREEDY *has a competitive ratio of* $\Theta(\log m)$ *on identical machines.*

Proof. Let $\lambda = \text{OPT}(\sigma)$. Note that all on-line machines are occupied until time $\text{GREEDY}(\sigma)$. We cut the schedule of GREEDY into pieces of time length 2λ starting from the bottom.

If there are less than m final jobs, there are less than m jobs, hence GREEDY is optimal. Suppose there are m final jobs.

Claim: At time $2i\lambda$, at most $m/2^i$ final jobs did not start yet.

Proof: By induction. The claim holds for $i = 0$. Assume it holds for some $i \geq 0$.

A final job is called *missing* if it did not start before time $2\lambda i$. Let k be the number of missing jobs. We have $k \leq m/2^i$ starting at time $2\lambda i$ or later. The total size of non-final jobs running at any time after $2\lambda i$ is at most $k\lambda$. This follows because GREEDY schedules the jobs with monotonically increasing start times, hence if there are k missing final jobs, then all the unstarted jobs must have arrived after the $m - k$-th final job. That job is started before time $2\lambda_i$ and hence the unstarted jobs must be scheduled by OPT on the machines where it runs the last k final jobs. Since OPT completes all these (non-final) jobs no later than at time λ, the total size of these jobs is at most $k\lambda$.

At most $k/2$ machines can be busy with these jobs during the entire time interval $[2\lambda i, 2\lambda(i+1)]$. Hence $k/2$ or more final jobs start in this interval (one for every machine that is not busy with non-final jobs during the entire interval and that was also not running a final job already). At most $k/2$ final jobs will be missing at time $2\lambda(i+1)$, and $k/2 \leq m/2^{i+1}$. □

At time $2\lambda \log_2 m$, only one final job is missing, therefore $\text{GREEDY}(\sigma) \leq 2\lambda \log_2 m + \lambda$, hence $\mathcal{R}(\text{GREEDY}) = O(\log m)$.

To show that $\mathcal{R}(\text{GREEDY}) = \Omega(\log m)$, we use a job sequence that consists of a job of size 1 followed by a job of size M (a large constant, e.g. $M = m$), repeated m times. The optimal algorithm can assign the jobs so that no job starts later than at time 1, whereas GREEDY starts the last job at time $1 + \lfloor \log_2 m \rfloor$. □

We now consider the competitive ratio of GREEDY for $m = 2, 3$. In Section 6.4, we will show matching lower bounds. Hence, GREEDY is optimal for $m = 2, 3$.

Lemma 6.1 *On identical machines,* $\mathcal{R}(\text{GREEDY}) \leq 2$ *for* $m = 2$*, and* $\mathcal{R}(\text{GREEDY}) \leq 5/2$ *for* $m = 3$*.*

Proof. We start with the case $m = 2$. We need to show that the competitive ratio of GREEDY is at most 2. Assume by contradiction that GREEDY has competitive ratio of $\rho > 2$. Define $\varepsilon = \frac{1}{2}(\rho - 2)$ and consider a sequence σ for which GREEDY has a ratio of at least $\rho - \varepsilon$. Without loss of generality we assume that $\text{OPT}(\sigma) = 1$. We denote the last job in σ by J_ℓ. This is a final job.

Since J_ℓ was assigned by GREEDY to the least loaded machine, both of the on-line machines are busy until time $\rho - \varepsilon$. Hence the total size of all jobs but J_ℓ is at least $2(\rho - \varepsilon) > 4$. The volume of jobs that OPT runs before time $\text{OPT}(\sigma) = 1$ is at most 2. OPT can run only two additional (final) jobs after time 1, one on each machine. One of those jobs is J_ℓ. Hence the other job, J_0, must have a size greater than $2(\rho - \varepsilon) - 2 > 2$.

Hence there exists a job J_0 of size greater than 2. The volume of the remaining jobs (apart from J_ℓ) is at most 2. Hence GREEDY will not schedule J_ℓ on the same machine as J_0, because the other machine must be less loaded. Scheduled on that machine, J_ℓ starts no later than at time 2, since at most a volume of 2 of jobs is scheduled before it.

For $m = 3$, suppose GREEDY has competitive ratio $\rho > 5/2$ and define σ and J_ℓ as above (taking $\varepsilon = \frac{1}{2}(\rho - 5/2)$). Assume $\text{OPT}(\sigma) = 1$. Denote the total size of all jobs but J_ℓ by V. Note that the size of J_ℓ is irrelevant for the competitive ratio; we may assume it has size 0. Denote the total size of all jobs of size at most 1 by V'. Since $\text{OPT}(\sigma) = 1$, OPT starts all its jobs no later than at time 1; the jobs that it completes before time 1 have total size at most 3.

We have $V \geq 3(\rho - \varepsilon) > 15/2$, since all three of GREEDY's machines are busy until past time $\rho - \varepsilon > 5/2$ when J_ℓ arrives.

- If σ contains no jobs larger than 1, consider the optimal off-line schedule. Two final jobs are of size at most 1, and the third (J_ℓ) is of size 0. The rest of the jobs are completed by time 1, and their total size is at most 3. Hence $V = V' \leq 5$, a contradiction.

- If σ contains one job larger than 1, then $V' \leq 4$: one final job has size 0, and one must have size at most 1 (since only one can be larger than 1). The rest of the jobs are of size at most 1, and have total size at most 3. Consider the least loaded machine among the two machines that do not run the job larger than 1, at the time J_ℓ arrives. Since $V' \leq 4$, it cannot have a load more than 2. But then GREEDY starts J_ℓ no later than at time 2.

- If σ contains two jobs larger than 1, then analogously to the previous cases, $V' \leq 3$. Denote the time that GREEDY starts the second large job by t_2. Similarly to in the previous case, we have $t_2 \leq 3/2 < 5/2$. At most a volume of 1 of jobs starts after t_2, since OPT has to run all these jobs and J_ℓ on one machine if $\text{OPT}(\sigma) = 1$: two of OPT's machines are already running large jobs and cannot be used anymore.

– If $t_2 \geq 1/2$, then in the worst case GREEDY assigns all the jobs that arrive after t_2 to one machine and starts J_ℓ no later than at time $5/2$.

– If $t_2 < 1/2$, then at the time the second large job arrives GREEDY starts no job later than at time $1/2$. Hence the on-line machine that has no large job has load at most $3/2$ at this time, since all jobs on that machine have size at most 1 and GREEDY always uses the least loaded machine. Since after t_2, at most a volume 1 of jobs still arrives, J_ℓ starts no later than at time $5/2$. □

We now turn to the performance of GREEDY on related machines. We set the speed of the slowest machine to 1 and denote the speed of the fastest machine by $q > 1$. I.e. on the fastest machine, it takes w/q time to complete a job of size w.

Lemma 6.2 *For two related machines,* GREEDY *has a competitive ratio of at most* $q + 1$.

Proof. Suppose the competitive ratio of GREEDY is $\rho > 1 + q$. We define J_ℓ and σ analogously to in Lemma 6.1, taking $\varepsilon = \frac{1}{2}(\rho - q - 1)$. In the present case, we find that the total size of all jobs but J_ℓ must be greater than $(q + 1)^2$, OPT can run at most $1 + q$ before time 1 and there must be a job J_0 of size greater than $q(1 + q)$. Again GREEDY will not schedule J_ℓ on the same machine as J_0 (even if J_0 is run on the fast machine), and hence not start it later than at time $q + 1$ (assuming that J_ℓ is run on the slow machine, otherwise it starts not after time $(q + 1)/q$). □

6.3 Algorithm BALANCE

We give an algorithm for identical machines of competitive ratio 12. This algorithm works in phases and uses an estimate on $\text{OPT}(\sigma)$ which is denoted by λ. A job is called *large* if its size is more than λ, and *small* otherwise; if $\lambda \geq \text{OPT}(\sigma)$, OPT can only run one such job on each machine. Also, once OPT has done this, it cannot use that machine anymore for any job.

A phase of BALANCE ends if it is clear from the small jobs that arrived in the phase, and from the large jobs that exist, that if another job arrives then $\lambda \leq \text{OPT}(\sigma)$. In this case we double λ and start a new phase.

In every phase, BALANCE only uses machines that do not already have large jobs. Each such machine will receive jobs according to one of the two following possibilities.

1. Only small jobs, of total weight in that phase less than 3λ.

2. Small jobs of weight less than 2λ, and one large job on top of them.

A machine that received a large job is called *large-heavy*, a machine that received weight of at least 2λ of small jobs in the current phase is called *small-heavy*. Both small-heavy and large-heavy machines are considered *heavy*. A machine that received more than a weight of λ of small jobs in the current phase but at most 2λ (and no large job) is considered *half-heavy*. Other machines are *non-heavy*. A machine that is not heavy (but possibly half-heavy) is called *active*. The algorithm BALANCE also maintains a set Q that contains the active machines.

Define λ_i as the value of λ in phase i. The algorithm BALANCE starts with phase 0 which is different from the other phases. In phase 0, m jobs arrive that are assigned to different machines. We then set λ_0 equal to the size of the smallest job that has arrived. Then the first of the regular phases starts.

Phases: A new phase starts when $Q = \emptyset$, i. e. there are no active machines anymore. (Phase 1 starts when phase 0 ends.) At the start of phase $i > 0$, we set $\lambda_i = 2\lambda_{i-1}$. Then Q contains all machines that do not have a large job. This holds because no machine has yet received any job in the current phase, so no machine can be small-heavy. Note that such a large job has arrived in some previous phase, but that the definition of large jobs has changed compared to the previous phase. I. e. not all the large jobs from previous phases are still large.

At all times, the algorithm only uses active machines. When the phase starts, all active machines are non-heavy. Each phase consists of two parts. The first part continues as long as there is at least one non-heavy machine among the active machines. As soon as no machine is non-heavy, the second part starts.

Part 1 In the first part of the phase, the algorithm works as follows. For small jobs, it uses the machines in Q in a Next Fit-fashion, moving to the next machine as soon as a machine has received a load of more than λ_i in the current phase. An arriving large job is assigned to a machine that already has weight of more than λ_i. If no such machine exists, it is assigned to the active machine that BALANCE is currently using or going to use for small jobs (there is a unique such machine, and all other non-heavy machines did not receive any jobs in the current phase). A machine that receives a large job becomes large-heavy, and is removed from Q.

Part 2 When all machines are either half-heavy or large-heavy we move on to the second part of the phase. We are ready to use the half-heavy machines once again. We again start using the machines in Q in a Next Fit-fashion, moving to the next machine as soon as the machine has received a total load at least $2\lambda_i$ in the current phase. A machine that receives weight of at least $2\lambda_i$ of small jobs in total in this phase becomes small-heavy and hence stops being active (is removed from Q). A machine that receives a large job becomes large-heavy and also stops being active (it is removed from Q).

As long as $|Q| > 0$, there are active machines. When $Q = \emptyset$, a new phase starts. An example

of a run of BALANCE can be seen in Figure 6.2.

We show that as soon as a first job in the new phase arrives, then $\lambda_{i-1} \leq \text{OPT}(\sigma)$. (Note that it is possible that no jobs arrive in a phase; this happens if $Q = \emptyset$ at the beginning of a phase.)

Lemma 6.3 *In each phase $i > 0$ in which jobs arrive, we have $\text{OPT}(\sigma) \geq \lambda_i/2$, where σ is the sequence of jobs that arrived until phase i, including the first job of phase i.*

Proof. The lemma holds for phase 1, since there is at least one machine of the optimal off-line algorithm that has two scheduled jobs after the first job in phase 1 arrives.

Consider a phase $i > 1$. If phase i starts when phase $i-1$ is still in its first part, then no machines are small-heavy. Hence in total m jobs have arrived that were considered large in phase $i-1$ (where some may have arrived before phase $i-1$). After the first job arrives in phase i, we have $\text{OPT}(\sigma) > \lambda_{i-1} = \lambda_i/2$.

If phase i starts while phase $i-1$ is in its second part, let K be the set of large jobs that were assigned to non-heavy machines in phase $i-1$. (If no such jobs exist, $K = \emptyset$). The jobs in K arrived in part 1 of phase $i-1$, since in part 2 only half-heavy machines are used. In part 1 of a phase, the active machines that have already been used are half-heavy or large-heavy.

Assume by contradiction that $\text{OPT}(\sigma) < \lambda_{i-1}$. Suppose $K \neq \emptyset$. Denote the last job in K by J_K and denote the set of machines that are still active after J_K has arrived by Q'. Write $q = |Q'|$. There was no half-heavy machines available for J_K, so all the machines that already received jobs in phase $i-1$, including the one that received J_K, are large-heavy at this point (they cannot be small-heavy in part 1). If $K = \emptyset$, define Q' as the set of active machines at the start of phase $i-1$. Clearly, all machines not in Q' are large-heavy at that point.

From this, we have that there exist $m - q$ large jobs after J_K has arrived (or at the start of phase $i-1$): all machines not in Q' either were large-heavy when phase $i-1$ started, or became large-heavy during it. Hence there are $m - q$ machines of OPT with a large job, since OPT cannot put two large jobs on one machine; OPT cannot put any more jobs on those machines if $\text{OPT}(\sigma) < \lambda_{i-1}$. Consider the set Q'_{OPT} of machines of OPT that do not run any of the $m - q$ large jobs that arrived already. We have $|Q'_{\text{OPT}}| = |Q'| = q$.

We calculate how much weight can be assigned by BALANCE to the machines in Q' (or equivalently, by OPT to the machines in Q'_{OPT}) in the remainder of phase $i-1$. In the schedule of OPT, the machines in Q'_{OPT} have some q jobs running last on them. Apart from that they have at most an amount of $\text{OPT}(\sigma) < \lambda_{i-1}$ small jobs.

Let $q_1 \leq q$ be the number of large jobs assigned by BALANCE to machines in Q' in the remainder of phase $i-1$. At the end of phase $i-1$, each machine in Q' is either small-heavy, or has an amount of at least λ_{i-1} small jobs and a large job. The total weight of small jobs assigned in phase $i-1$ to the machines of Q' by BALANCE is at least $(2q - q_1)\lambda_i$.

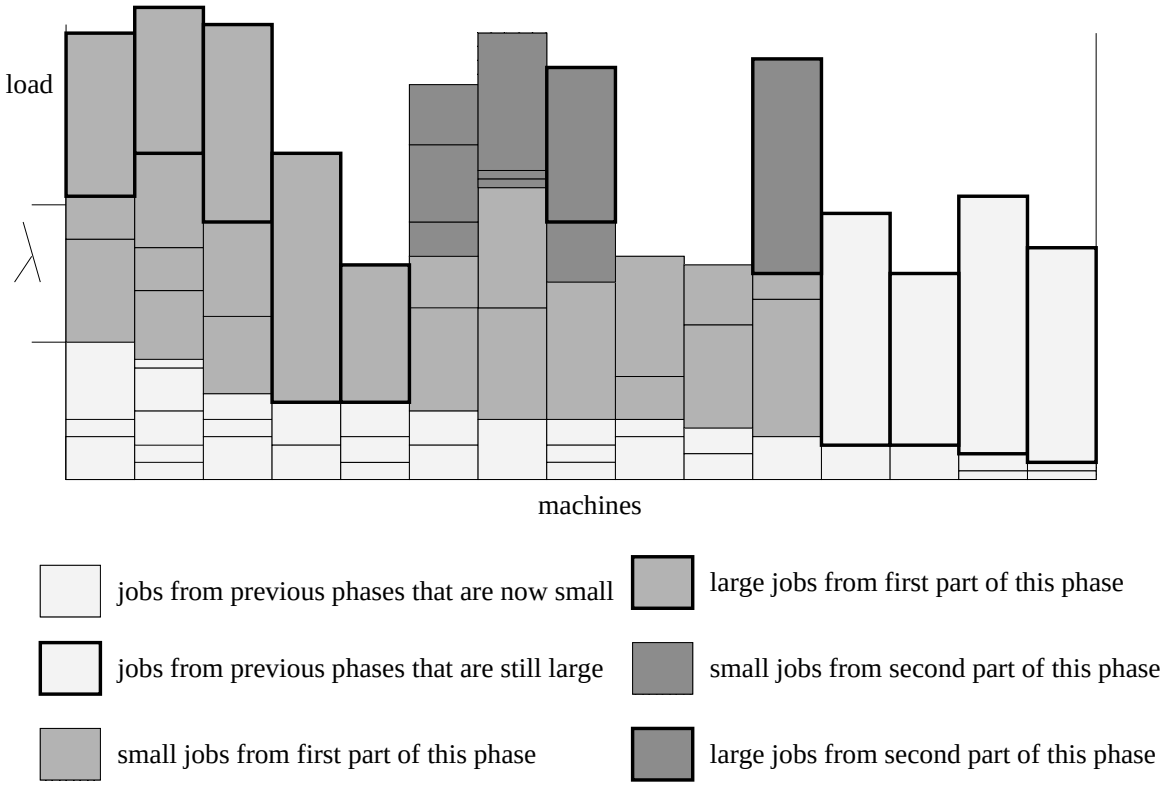

Figure 6.2: A run of BALANCE

Suppose we remove the q largest jobs assigned in phase $i-1$ to the machines of the set Q' in the assignment of BALANCE. This means that we remove q_1 large jobs and $q-q_1$ small jobs. By definition, each small removed job has size of at most λ_{i-1}, so we removed at most an amount of $(q-q_1)\lambda_{i-1}$ small jobs. Therefore we are left with total weight of at least $q\lambda_{i-1}$ on the machines in Q', counting only weight from jobs that arrived in this phase.

This implies that even if OPT runs the largest q jobs last on the machines in Q'_{OPT}, it starts at least one of them at time λ_{i-1} or later, by the total weight of the other jobs. This gives a contradiction, already without the first job in phase i. This proves the lemma. □

Theorem 6.2 *Algorithm* BALANCE *has a competitive ratio of 12.*

Proof. Consider the last phase $\ell > 0$ in which jobs arrived. (If $\ell = 0$, BALANCE is optimal.) Let $\Lambda = \lambda_\ell$. We have $\text{OPT} \geq \Lambda/2$ by Lemma 6.3. Consider the machines that received jobs in phase ℓ, and for each such machine, consider the total size of jobs below the last job that is run on that machine. (For the machines that did not receive jobs in this phase, we have stronger bounds.) This size consists of three parts:

- The small jobs of phase ℓ
- The small jobs of previous phases
- The large jobs of previous phases

For the computation, for phases $0 < i < \ell$ in which a machine got only small jobs, we replace an amount of $2\lambda_i$ of small jobs from that phase by one (large) job. (Possibly a small job is broken in two parts to get a total of exactly $2\lambda_i$.) Because we only consider machines that received jobs in phase ℓ, the maximum starting time is unaffected by this substitution. As a result, each machine receives at most a weight of $2\lambda_i$ of small jobs in phase i.

In phase ℓ, each machine receives at most 2Λ of small jobs before it receives its last job. The value of λ is doubled between phases, hence the total amount of small jobs from previous phases on a machine is at most $\sum_{i<\ell} 2\lambda_i \leq 2\Lambda$.

We still need to consider the large jobs from previous phases. We count the large jobs not by the phases they arrive; instead, each large job is counted in the first phase where it is not large anymore, and the machine is active again. The large jobs that replace 2Λ worth of small jobs as described above, are always already small in the subsequent phase. For each phase $i \leq \ell$, a machine has at most one job that has just become small. This job is of size at most λ_i. Hence in total the size of all these jobs is at most $\sum_{i\leq\ell} \lambda_i \leq 2\Lambda$. Therefore the total load below the last jobs on any machine is at most

$$2\Lambda + 2\Lambda + 2\Lambda \leq 6\Lambda.$$

Since $\Lambda \leq 2\text{OPT}$, we are done. □

6.4 Lower bounds

In the following proofs, we take M to be a large constant. If we construct a job sequence σ that contains a job of size M, then we assume that M is larger than $\mathcal{R}$ times the sum of smaller jobs in σ, where $\mathcal{R}$ is the competitive ratio that we want to show. This choice of M ensures that if a machine is assigned a job of size M, it cannot receive any other job after this without violating the competitive ratio.

Lemma 6.4 *Suppose we have a job sequence σ that shows that $\mathcal{R}(\mathcal{A}) \geq \mathcal{R}$ for all on-line algorithms on m_1 machines. Then for any $m > m_1$, $\mathcal{R}(\mathcal{A}) \geq \mathcal{R}$ for all on-line algorithms on m machines, as well.*

Proof. Construct the sequence σ' by adding $m - m_1$ jobs of size M before the first job of σ. The optimal cost for this sequence is the same as for σ on m_1 machines. On the machines that do not run the first $m - m_1$ jobs, we have that $\mathcal{A}$ must have a cost at least $\mathcal{R}$ times the optimal cost for σ on m_1 machines, and we are done. □

Theorem 6.3 *Take $\alpha = (\sqrt{5}+1)/2 \approx 1.618$ and M a large constant. For all on-line algorithms $\mathcal{A}$, we have the following lower bounds for the competitive ratio on identical machines.*

Number of machines	*Job sequence*	$\mathcal{R}$
2	$1, M, 1, M$	2
3	$1/2, 1/2, M, 1, M, 1, M$	$5/2$
4	$\alpha - 1, \alpha - 1, M, M, 1, M, 1, M$	$\alpha + 1 \approx 2.618$

Moreover, as the number of machines tends to infinity, the competitive ratio tends to at least 4.

Proof. For $m \leq 4$, we use the job sequences described in the table above. For these sequences, any on-line algorithm that has a better competitive ratio than in the last column of the table must assign these jobs in the same way as the greedy algorithm, or violate the competitive ratio. In all cases, after the last job arrives we have $\text{OPT}(\sigma) = 1$ and $\mathcal{A}(\sigma) = \mathcal{R}$.

As an example, for $m = 4$, the first four jobs must be assigned to four different machines, the next two jobs to the machines with the jobs of size $\alpha - 1$, and the last two to the machine that does not have a job of size M yet. The sequence stops as soon as $\mathcal{A}$ assigns a job differently than described here, or after the fourth large job.

For larger m, we use the following job sequence. Assume $m = 2^r$ for some $r \geq 3$, and consider a sequence of real numbers $\{k_i\}_{i=0}^{\infty}$ with properties to be defined later. We will first define the job sequence and then specify for which value of r it works. The job sequence consists of $r + 1$ steps. For $1 \leq i \leq r$, in step i first $m/2^i$ jobs of size k_i arrive, and then $m/2^i$ jobs of size M. In step $r + 1$, one last job of size k_r arrives, followed by a job of size M.

We denote the optimal maximum starting time after step i by OPT_i. If $k_i \leq k_{i+1}$ for all $1 \leq i \leq r-1$, then for $1 \leq i \leq r$, we have $\text{OPT}_i = k_{i-1}$ (we put $k_0 = 0$), which is seen as follows. We describe the optimal schedule after step i. (We note that the optimal schedules after different steps can be very different.) There are $m/2^i$ machines with one job of size k_i, and $m/2^i$ machines with one job of size M. These machines do not have any other jobs. The remaining machines have one job of size k_s for some $s < i$, and after it one job of size M. After the last step we have $\text{OPT}_{r+1} = k_r$. In this case, all machines have one job of size k_s for some $s \leq r$, and after it one job of size M.

We will now define the sequence $\{k_i\}_{i=0}^{\infty}$ in such a way that the on-line algorithm cannot place two jobs on the same machine in one step. By induction we can see that at the start of step i ($1 \leq i \leq r$), $m(1 - 1/2^{i-1})$ jobs of size M have already arrived. Thus if the on-line algorithm places the $m/2^{i-1}$ jobs from step i on different machines (that moreover do not have a job of size M yet), then also by induction, after every step i ($1 \leq i \leq r$), every machine of the on-line algorithm either has a job of size M, or it has one job of *each* size k_j, for $1 \leq j \leq i$.

Define $s_i = \sum_{j=1}^{i} k_j$. If the on-line algorithm does put two jobs on the same machine in some step $i \leq r$, then by the above the last job on that machine starts at time $\sum_{j=1}^{i} k_j$ and the implied ratio is

$$\mathcal{R}_i = \frac{\sum_{j=1}^{i} k_j}{k_{i-1}} = \frac{s_i}{k_{i-1}} \,. \tag{6.1}$$

If the on-line algorithm never does this, then in the final step $r+1$ it has only $m/2^r = 1$ machine left without a job of size M, and this machine has one job of each size k_j for $1 \leq j \leq i$. The on-line algorithm has minimal cost if it places the two jobs from step $i+1$ on this machine, and the implied competitive ratio is thus $\mathcal{R}_{r+1} = (\sum_{j=1}^{r} k_j + k_r)/k_r = (s_r + k_r)/k_r$. Using (6.1), we will define the sequence $\{k_i\}_{i=0}^{\infty}$ so that $\mathcal{R}_i = \mathcal{R}$ is a constant for $1 \leq i \leq r+1$. This implies

$$k_0 = 0,\ k_1 = 1,\ k_i = \mathcal{R}k_{i-1} - \sum_{j=1}^{i-1} k_j = \mathcal{R}k_{i-1} - s_{i-1} \text{ for } i > 1.$$

This proves a competitive ratio of $\mathcal{R}$ if $k_i \leq k_{i+1}$ for $1 \leq i \leq r-1$ and $(s_r + k_r)/k_r \geq \mathcal{R}$ (where this last condition follows from step $r+1$). We have

$$\begin{aligned}
(s_r + k_r)/k_r \geq \mathcal{R} \quad &\Longleftrightarrow \quad k_r + s_r \geq Rk_r = s_{r+1} \qquad \text{using (6.1)} \\
&\Longleftrightarrow \quad k_r \geq s_{r+1} - s_r = k_{r+1} \\
&\Longleftrightarrow \quad s_{r+1} \geq s_{r+2} \iff k_{r+2} \leq 0 \iff s_{r+3} \leq 0.
\end{aligned}$$

Hence it is sufficient to show that the sequence $\{s_i\}_{i=0}^{\infty}$ has its first nonpositive term s_{r+3} for some $r \geq 1$. This value of r determines for which m this job sequence shows a lower bound of $\mathcal{R}$, since $m = 2^r$. Note that if s_{r+3} is nonpositive, we have to stop the job sequence after

step $r+1$ at the latest, because by the above $k_{r+2} \leq 0 < k_1 \leq k_r$: the sequence is no longer non-decreasing. As stated above, we will in fact give one final job of size k_r in step $r+1$, and a job of size M, and thus not use any value k_i for $i > r$. The sequence $\{s_i\}_{i=0}^{\infty}$ satisfies the recurrence $s_{i+2} - \mathcal{R}s_{i+1} + \mathcal{R}s_i = 0$. For $\mathcal{R} < 4$, the solution of this recurrence is given by

$$s_i = \frac{2\sin(\theta i)\sqrt{\mathcal{R}}^{\,i}}{\sqrt{4\mathcal{R} - \mathcal{R}^2}}$$

where

$$\cos\theta = -\frac{1}{2}\sqrt{\mathcal{R}} \qquad \text{and} \qquad \sin\theta = \sqrt{1 - \frac{\mathcal{R}}{4}}.$$

Since $\sin\theta \neq 0$, then $\theta \neq 0$, which implies $s_i < 0$ for some value of i. Furthermore, this value of i tends to ∞ as $\mathcal{R}$ tends to 4 from below. Direct calculations show that for $i = 1, 2, 3, 4$, $s_i > 0$, hence given such minimal integer i, we can define $r = i - 3$. From the calculations it also follows that $\{k_i\}_{i=0}^{r}$ is non-decreasing.

In conclusion, for any value of $\mathcal{R} < 4$ it is possible to find a value r so that any on-line algorithm has at least a competitive ratio of $\mathcal{R}$ on 2^r machines. By Lemma 6.4, this implies that for every $\varepsilon > 0$, there exists a value m_1 such that for any on-line algorithm $\mathcal{A}$ on $m > m_1$ machines, $\mathcal{R}(\mathcal{A}) \geq 4 - \varepsilon$. □

Note that this proof does not hold for $\mathcal{R} \geq 4$, because the solution of the recurrence in that case is not guaranteed to be below 0 for any i.

Corollary 6.1 *On identical machines,* GREEDY *is optimal for* $m = 2, 3$.

Proof. This follows from Lemma 6.1 and Theorem 6.3. □

6.5 Related machines

We only study the special case $m = 2$ and give a matching lower bound to the upper bound from Lemma 6.2, showing that Greedy is optimal for this case.

Theorem 6.4 *For the problem of minimizing the maximum starting time on two related machines, the competitive ratio is at least* $q + 1$.

Proof. Consider an algorithm $\mathcal{A}$ for this problem and suppose it has a competitive ratio of less than $q+1$. A job of size 1 arrives. If $\mathcal{A}$ places it on the slow machine (the machine with speed 1), then a job of size M arrives (which has to go on the other machine; M is defined as in Section 6.4), followed by a job of size q and another job of size M. The maximum starting time of $\mathcal{A}$ is at least $q + 1$, whereas the optimal maximum starting time is 1, by putting the job of size 1 on

the slow machine, the job of size q on the fast machine, and starting both the large jobs at time 1.

If $\mathcal{A}$ places the first job on the fast machine, then take N a large constant. The second job has size Nq and must be placed on the slow machine. The third job has size NM, where $M = (q+1)N$, and must be placed on the fast machine, otherwise a competitive ratio of $Nq/(1/q) = Nq^2$ is implied. Then a job of size $N-1$ arrives which must go on the slow machine; finally another job of size NM arrives. $\mathcal{A}$ starts its last job at time $Nq + (N-1)$ whereas in the optimal schedule, no job starts after time N. By letting N grow without bound (maintaining $M = (q+1)N$), this proves the ratio. □

Corollary 6.2 GREEDY *is optimal for two related machines.*

Proof. This follows from Lemma 6.2 and Theorem 6.4. □

6.6 Resource augmentation

We now consider on-line algorithms that have more resources than the off-line algorithm. It turns out that in these changed circumstances, GREEDY is optimal in the sense that it requires the minimum possible number of machines to have a competitive ratio of 1.

We only consider identical machines in this section.

Lemma 6.5 GREEDY *has a competitive ratio of 1 if it has at least* $2m-1$ *machines.*

Proof. Let $h = \text{GREEDY}(\sigma)$ and $h^* = \text{OPT}(\sigma)$. Note that the last job J_ℓ that is assigned at time h by GREEDY is a final job for OPT as well, since this is the very last job in the sequence. Let S be the set of on-line machines of GREEDY that only contain non-final jobs or J_ℓ. Since there are at most m final jobs, $|S| \geq 2m - 1 - (m-1) = m$. All of GREEDY's machines are occupied from 0 to h. The machines in S are occupied during this time by non-final jobs. Let W be the total size of non-final jobs. We have $W \geq mh$. But $W \leq h^* m$. Hence $h \leq h^*$. □

Note that a similar proof shows that the competitive ratio of GREEDY tends to zero as the number of on-line machines tends to ∞.

Lemma 6.6 *Any algorithm that has at most* $2m-2$ *machines has a competitive ratio greater than 1.*

Proof. Suppose $\mathcal{A}$ has a competitive ratio of at most 1. We use a construction in phases, where in each phase the size of the arriving jobs is equal to the total size of all the jobs from the previous phases. Let n_i denote the number of jobs in phase i, and M_i denote the size of the jobs in phase

i. We determine the number of phases later. We take $n_0 = m$ and $n_i = 2m - 1$ for $i > 0$. Furthermore, we take $M_0 = 1$, $M_1 = n_0 M_0 = m$ and

$$M_i = \sum_{j=0}^{i-1} n_j M_j = \sum_{j=0}^{i-2} n_j M_j + n_{i-1} M_{i-1} = M_{i-1} + (2m-1)M_{i-1} = 2mM_{i-1} \text{ for } i > 1.$$

Claim: After i phases, at least $\min(m + (m-1)(1 - \frac{1}{2^i}), 2m - 2)$ machines are non-empty.

Proof: We use an induction. All jobs from phase 0 have to be assigned to different machines to have a finite competitive ratio, so m machines are non-empty after phase 0.

Consider phase i for $i > 0$. During each phase $i > 0$, the optimal costs are at most M_i: all the jobs from the previous phases go together on one machine, followed by one job of size M_i. All other machines have two jobs of size M_i. In order to have a competitive ratio of 1, $\mathcal{A}$ can assign at most one job of size M_i on each non-empty machine, and at most 2 such jobs on each empty machine. Let x be the number of non-empty machines at the start of phase i. If $x = 2m - 2$ we are done immediately. Else, we have $x \geq m + (m-1)(1 - \frac{1}{2^{i-1}})$ by induction. The number of machines that become non-empty in phase i is at least $(2m - 1 - x)/2$, so after phase i, at least $m - \frac{1}{2} - \frac{1}{2}x + x$ machines are non-empty. By induction, we have $m - \frac{1}{2} + \frac{1}{2}x \geq m - \frac{1}{2} + (m + (m-1)(1 - \frac{1}{2^{i-1}}))/2 = m + (m-1)(1 - \frac{1}{2^i})$. □

Taking $k = \lceil \log_2 m \rceil$, we have that after k phases, $m + (m-1)(1 - \frac{1}{2^k}) \geq m + (m-1)(1 - \frac{1}{m}) > 2m - 2$, hence $\mathcal{A}$ needs more than $2m - 2$ machines to maintain a competitive ratio of 1. □

Note that no algorithm $\mathcal{A}$ which uses $2m - 1$ machines can have competitive ratio less than 1, due to the sequence $1, \ldots, 1$ ($2m$ jobs). At least two jobs run on the same on-line machine, hence $\mathcal{A}(\sigma) = \text{OPT}(\sigma) = 1$.

6.7 Conclusions

We showed that the greedy algorithm is far from being optimal in one measure (competitive ratio), but optimal in a different measure (amount of resource augmentation). This phenomenon raises many questions. Which of the two measures is more appropriate for this problem? Furthermore, which measure is appropriate for other problems? Is it possible to introduce a different measure that would solve the question: is GREEDY a good algorithm to use?

Chapter 7

Variable-sized on-line bin-packing

In the variable-sized on-line bin packing problem, one has to assign items to bins one by one. The bins are drawn from some fixed set of sizes, and the goal is to minimize the sum of the sizes of the bins used. We present the first algorithms for this problem that use unbounded space, i. e. the number of open bins (bins that can still receive items later) is not bounded by any constant.

We also show the first lower bounds on the asymptotic performance ratio. The case where an algorithm can choose between bins of two sizes, 1 and $\alpha \in (0, 1)$, is studied in detail. An overview of the best upper bounds, using the algorithms in this paper and earlier algorithms, and our lower bounds is given in Figure 7.6.

7.1 Introduction

The bin packing problem is one of the oldest and most well-studied problems in computer science [25, 28]. The influence and importance of this problem are witnessed by the fact that it has spawned off whole areas of research, including the fields of on-line algorithms and approximation algorithms. The on-line variable-sized bin packing problem is a natural generalization of the classical on-line bin packing problem. We show improved upper bounds and the first lower bounds for this problem.

A warning for the reader: in this chapter we do not discuss scheduling problems and we use a different notation from the other chapters, giving some symbols other meanings than they have in other chapters. The reason for this is that we need a lot of symbols in the present chapter and would otherwise run out of "reasonable" symbols to use.

Problem Definition: In the *classical bin packing* problem, we receive a sequence σ of *pieces* $p_1, p_2, \ldots, p_N$. Each piece has a fixed *size* in $(0, 1]$. In a slight abuse of notation, we use p_i to

indicate both the ith piece and its size. The usage should be obvious from the context. We have an infinite number of *bins* each with *capacity* 1. Each piece must be assigned to a bin. Further, the sum of the sizes of the pieces assigned to any bin may not exceed its capacity. A bin is *empty* if no piece is assigned to it, otherwise it is *used*. The goal is to minimize the number of bins used.

The *variable-sized bin packing* problem differs from the classical one in that bins do not all have the same capacity. Instead, we have a set of capacities $0 < \alpha_1 < \alpha_2 < \cdots < \alpha_m = 1$. There are an infinite number of bins of each capacity. The goal now is to minimize the sum of the capacities of the bins used.

In the *on-line* versions of these problems, each piece must be assigned in turn, without knowledge of the next pieces. Since it is impossible in general to produce the best possible solution when computation occurs on-line, we consider approximation algorithms. Basically, we want to find an algorithm which incurs cost which is within a small factor of the minimum possible cost, no matter what the input is. This factor is known as the asymptotic performance ratio.

We define the asymptotic performance ratio more precisely. For a given input sequence σ, let $\mathcal{A}(\sigma)$ be the sum of the capacities of the bins used by algorithm $\mathcal{A}$ on σ. Let $\textsc{opt}(\sigma)$ be the minimum possible cost to pack pieces in σ. The *asymptotic performance ratio* for an algorithm $\mathcal{A}$ is defined to be

$$R_{\mathcal{A}}^{\infty} = \limsup_{n\to\infty} \max_{\sigma} \left\{ \frac{\mathcal{A}(\sigma)}{\textsc{opt}(\sigma)} \middle| \textsc{opt}(\sigma) = n \right\}.$$

The *optimal asymptotic performance ratio* is defined to be

$$R_{\textsc{opt}}^{\infty} = \inf_{\mathcal{A}} R_{\mathcal{A}}^{\infty}.$$

Our goal is to find an algorithm with asymptotic performance ratio close to $R_{\textsc{opt}}^{\infty}$.

Previous Results: The *on-line bin packing problem* was first investigated by Johnson [45, 44]. He showed that the NEXT FIT algorithm has performance ratio 2. Subsequently, it was shown by Johnson, Demers, Ullman, Garey and Graham that the FIRST FIT algorithm has performance ratio $\frac{17}{10}$[46]. Yao showed that REVISED FIRST FIT has performance ratio $\frac{5}{3}$, and further showed that no on-line algorithm has performance ratio less than $\frac{3}{2}$ [75]. Brown and Liang independently improved this lower bound to 1.53635 [17, 60]. This was subsequently improved by van Vliet to 1.54014 [71]. Chandra [19] shows that the preceding lower bounds also apply to randomized algorithms.

Define

$$u_{i+1} = u_i(u_i - 1) + 1, \qquad u_1 = 2,$$

and

$$h_\infty = \sum_{i=1}^{\infty} \frac{1}{u_i - 1} \approx 1.69103.$$

Lee and Lee showed that the HARMONIC algorithm, which uses bounded space, achieves a performance ratio arbitrarily close to h_∞ [58]. They further showed that no bounded space on-line algorithm achieves a performance ratio less than h_∞ [58]. A sequence of further results has brought the upper bound down to 1.58889 [58, 62, 63, 66].

The *variable-sized bin packing problem* was first investigated by Frieson and Langston [37, 38]. Kinnerly and Langston gave an on-line algorithm with performance ratio $\frac{7}{4}$ [53]. Csirik proposed the VARIABLE HARMONIC algorithm, and showed that it has performance ratio at most h_∞ [27]. This algorithm is based on the HARMONIC algorithm of Lee and Lee [58]. Like HARMONIC, it uses bounded space. Csirik also showed that if the algorithm has two bin sizes 1 and $\alpha < 1$, and that if it is allowed to pick α, then a performance ratio of $\frac{7}{5}$ is possible [27]. Seiden has recently shown that VARIABLE HARMONIC is an optimal bounded-space algorithm [65]. The related problem of variable-sized bin covering has been solved by Woeginger and Zhang [73].

Our Results: In this chapter, we present two new algorithms for the variable-sized on-line bin packing problem for the case that the only possible bin sizes are 1 and $\alpha < 1$. These algorithms have the best known performance for many choices of α. Hence we can construct a new algorithm that, given the value of α, chooses whether to use VARIABLE HARMONIC or one of these two algorithms. This construction gives us the best known algorithm for general α. We also show the first lower bounds for the problem with two bin sizes. Prior to this work, no lower bounds were known for the variable-sized bin packing problem.

7.2 Two algorithms

To begin, we present two different unbounded space on-line algorithms for variable-sized bin packing.

We focus in on the case where there are two bin sizes, $\alpha_1 < 1$ and $\alpha_2 = 1$, and examine how the performance ratios of our algorithms change as a function of α_1. Since it is understood that $m = 2$, we abbreviate α_1 using α. We present two algorithms, both of which are combinations of the HARMONIC and REFINED HARMONIC algorithms for the classical problem. Both our algorithms, which we call VRH1(μ) and VRH2(μ), have a real parameter $\mu \in (\frac{1}{3}, \frac{1}{2})$. The algorithm VRH1($\mu$) is defined for all $\alpha \in (0, 1)$, but VRH2(μ) is only defined for

$$\alpha > \max\left\{\frac{1}{2(1-\mu)}, \frac{1}{3\mu}\right\}. \tag{7.1}$$

Before we describe the algorithms, we begin by dividing the interval $(0, 1]$ into smaller intervals. We will use these intervals to determine whether a new item would fit better into a bin of size 1 or a bin of size α. Items of the form $\alpha/k - \varepsilon$, where ε is small compared to α/k and k is some positive integer, fit well into a bin of size α, since we can put k of them in a bin of size α and have only a limited amount of unused space in that bin. On the other hand, items of the form $1/k - \varepsilon$ would fit better in a bin of size 1.

We make the following definitions. Define $n_1 = 50$, $n_2 = \lfloor n_1\alpha \rfloor$, $\epsilon = 1/n_1$ and

$$T = \left\{ \frac{1}{i} \;\middle|\; 1 \le i \le n_1 \right\} \cup \left\{ \frac{\alpha}{i} \;\middle|\; 1 \le i \le n_2 \right\} \cup \{\mu, 1 - \mu\}.$$

Define $n = |T|$. Note that it may be that $n < n_1 + n_2 + 2$, since T is not a multi-set. Rename the members of T as $t_1 = 1 > t_2 > t_3 > \cdots > t_n = \epsilon$. For convenience, define $t_{n+1} = 0$. The interval I_j is defined to be $(t_{j+1}, t_j]$ for $j = 1, \ldots, n + 1$. Note that these intervals are disjoint and that they cover $(0, 1]$. A piece of size s has *type* j if $s \in I_j$. Define the *class* of an interval I_j to be α if $t_j = \alpha/k$ for some positive integer k, otherwise the class is 1.

We begin by describing VRH1. The basic idea of VRH1 is as follows: When each piece arrives, we determine the interval I_j to which it belongs. If this is a class 1 interval, we pack the item in a size 1 bin using a variant of REFINED HARMONIC. If it is a class α interval, we pack the item in a size α bin using a variant of HARMONIC.

We differentiate between bins which are *open* and bins which are *closed*. An open bin is a non-empty bin into which the algorithm may potentially place one or more other pieces. The algorithm does not ever put a piece into a bin which is closed.

VRH1 packs bins in *groups*. All the bins in a group are packed in a similar fashion. The groups are determined by the set T. We define

$$g = \begin{cases} 3 & \text{if } \alpha > 1 - \mu, \\ 2 & \text{otherwise.} \end{cases} \qquad h = \begin{cases} 6 & \text{if } \alpha/2 > \mu, \\ 5 & \text{if } \alpha > \mu \text{ and } \alpha/2 \le \mu, \\ 4 & \text{otherwise.} \end{cases}$$

Note that these functions are defined so that $t_g = 1 - \mu$ and $t_h = \mu$. The groups are named $(g, h), 1, \ldots, g - 1, g + 1, g + 2, \ldots, n$.

Bins in group $j \in \{1, 2, , \ldots, n\} \setminus \{g\}$ contain only type j pieces.

Bins in group (g, h) all have capacity 1. Closed bins contain one type g piece and one type h piece.

Bins in group n all have capacity 1 and are packed using the NEXT FIT algorithm. I.e. there is one open bin in group n. When a type n piece arrives, if the piece fits in the open bin, it is placed there. If not, the open bin is closed, the piece is placed in a newly allocated open group n bin.

VRH1

Initialize $x \longleftarrow 0$ and $y \longleftarrow 0$.

For each item p:

$j \longleftarrow$ type of p.

If $j = n$ then pack p using NEXT FIT in a group n bin.

Else, if $j = g$ then PUT$(p, (g, h))$.

Else, if $j = h$:

$x \longleftarrow x + 1$.

If $y < \lfloor \tau x \rfloor$:

$y \longleftarrow y + 1$.

PUT$(p, (g, h))$.

Else PUT(p, h).

Else PUT(p, j).

PUT(p, G)

If there is no open bin in G then allocate a new bin b.

Else, let b be an arbitrary open bin in G.

Pack p in b.

Figure 7.1: The VRH1(μ) algorithm and the PUT sub-routine.

For group $j \in \{1, 2, , \ldots, n-1\} \setminus \{g\}$, the capacity of bins in the group depends on the class of I_j. If I_j has class 1, then each bin has capacity one, and each closed bin contains $\lfloor 1/t_j \rfloor$ items of type j. Note that t_j is the reciprocal of an integer for $j \neq h$ and therefore $\lfloor 1/t_j \rfloor = 1/t_j$. If I_j has class α, then each bin has capacity α, and each closed bin contains $\lfloor \alpha/t_j \rfloor$ items of type j. Similar to before, t_j/α is the reciprocal of an integer and therefore $\lfloor \alpha/t_j \rfloor = \alpha/t_j$. For each of these groups, there is at most one open bin.

The algorithm has a real parameter $\tau \in [0, 1]$, which for now we fix to be $\frac{1}{7}$. Essentially, a proportion τ of the type h items are reserved for placement with type g items.

A precise definition of VRH1 appears in Figure 7.1. The algorithm uses the sub-routine PUT(p, G), where p is an item and G is a group.

We analyze VRH1 using the technique of *weighting systems* introduced in [66]. A weighting system is a tuple $(\mathbb{R}^\ell, \mathbf{w}, \xi)$, where $\mathbb{R}^\ell$ is a real vector space, $\mathbf{w}$ is a *weighting function*, and ξ is a *consolidation function*. We shall simply describe the weighting system for VRH1, and assure the reader that our definitions meet the requirements put forth in [66].

For VRH1, we use $\ell = 3$, and define $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ to be orthogonal unit basis vectors. The weighting function is:

$$\mathbf{w}(x) = \begin{cases} \mathbf{b} & \text{if } x \in I_g; \\ (1-\tau)\dfrac{\mathbf{a}}{2} + \tau\,\mathbf{c} & \text{if } x \in I_h; \\ \dfrac{\mathbf{a}\,x}{1-\epsilon} & \text{if } x \in I_n; \\ \mathbf{a}\,t_i & \text{otherwise.} \end{cases}$$

The consolidation function is $\xi(x\,\mathbf{a} + y\,\mathbf{b} + z\,\mathbf{c}) = x + \max\{y, z\}$. The following lemma allows us to upper bound the performance of VRH1 using the preceding weighting system:

Lemma 7.1 *For all input sequences σ,*

$$\mathrm{vrh1}(\sigma) \le \xi\left(\sum_{i=1}^{n} \mathbf{w}(p_i)\right) + O(1).$$

Proof. We count the cost for bins in each group.

Consider first bins in group n. Each of these is packed using NEXT FIT, and contains only pieces of size at most ϵ. By the definition of NEXT FIT, each closed bin contains items of total size at least $1-\epsilon$, and there is at most one open bin. Therefore the number of bins used is at most

$$\frac{1}{1-\epsilon}\sum_{p_i \in I_n} p_i + 1 = \mathbf{a} \cdot \sum_{p_i \in I_n} \mathbf{w}(p_i) + O(1).$$

Now consider group j with $j \notin \{h, (g,h), n\}$. There is at most one open bin in this group. The capacity x of each bin is equal to the class of I_j. The number of items in each closed bin is $\lfloor x/t_j \rfloor$. Since $j \notin \{h, (g,h), n\}$, we have $\lfloor x/t_j \rfloor = x/t_j$. Putting these facts together, the cost at most

$$\sum_{p_i \in I_j} \frac{x}{\lfloor x/t_j \rfloor} + 1 = \sum_{p_i \in I_j} t_j + 1 = \mathbf{a} \cdot \sum_{p_i \in I_j} \mathbf{w}(p_i) + O(1).$$

Next consider group h. Let k be number of type h items in σ. The algorithm clearly maintains the invariant that $\lfloor \tau k \rfloor$ of these items go to group (g,h). The remainder are packed two to a bin in capacity 1 bins. At most one bin in group h is open. The total is at most

$$\frac{k - \lfloor \tau k \rfloor}{2} + 1 = \sum_{p_i \in I_h} \frac{1-\tau}{2} + O(1) = \mathbf{a} \cdot \sum_{p_i \in I_h} \mathbf{w}(p_i) + O(1).$$

Finally, consider group (g, h). Let f be the number of type g items in σ. The number of bins is

$$\max\{f, \lfloor \tau k \rfloor\} = \max\{f, \tau k\} + O(1) = \max\left\{ \mathbf{b} \cdot \sum_{p_i \in I_g} \mathbf{w}(p_i),\ \mathbf{c} \cdot \sum_{p_i \in I_h} \mathbf{w}(p_i) \right\} + O(1).$$

Putting all these results together, the total cost is at most

$$\mathbf{a} \cdot \sum_{i=1}^{n} \mathbf{w}(p_i) + \max\left\{ \mathbf{b} \cdot \sum_{i=1}^{n} \mathbf{w}(p_i),\ \mathbf{c} \cdot \sum_{i=1}^{n} \mathbf{w}(p_i) \right\} + O(1) = \xi\left(\sum_{i=1}^{n} \mathbf{w}(p_i) \right) + O(1).$$

□

From [66], we also have

Lemma 7.2 *For any input σ on which* VRH1 *achieves a performance ratio of c, there exists an input σ' where* VRH1 *achieves a performance ratio of at least c and*

1. *every bin in an optimal solution is full, and*
2. *every bin in some optimal solution is packed identically.*

Given these two lemmas, the problem of upper bounding the performance ratio of VRH1 is reduced to that of finding the single packing of an optimal bin with maximal weight/size ratio. We consider the following integer program: Maximize $\xi(\mathbf{x})/\beta$ subject to

$$\mathbf{x} = \mathbf{w}(y) + \sum_{j=1}^{n-1} q_j \mathbf{w}(t_j); \tag{7.2}$$

$$y = \beta - \sum_{j=1}^{n-1} q_j\, t_{j+1} \tag{7.3}$$

$$y > 0, \tag{7.4}$$

$$q_j \in \mathbb{N}, \qquad \text{for } 1 \le j \le n-1, \tag{7.5}$$

$$\beta \in \{1, \alpha\}; \tag{7.6}$$

over variables $\mathbf{x}, y, \beta, q_1, \ldots, q_{n-1}$. Intuitively, q_j is the number of type j pieces in an optimal bin. y is an upper bound on space available for type n pieces. Note that strict inequality is required in (7.4) because a type j piece is strictly larger than t_{j+1}. Call this integer linear program $\mathcal{P}$. The value of $\mathcal{P}$ upper bounds the asymptotic performance ratio of VRH1.

The value of $\mathcal{P}$ is easily determined using a branch and bound procedure very similar to those in [66, 65]. Define

$$\psi_i = \max\left\{ (\mathbf{a} + \mathbf{b} + \mathbf{c}) \cdot \mathbf{w}(t_i),\ \frac{1}{1-\epsilon} \right\}, \qquad \text{for } 1 \le i \le n-1; \qquad \psi_n = \frac{1}{1-\epsilon}.$$

Intuitively, ψ_i is the maximum contribution to the objective function for a type i item relative to its size. We define π so that

$$\psi_{\pi(1)} \geq \psi_{\pi(2)} \geq \cdots \geq \psi_{\pi(n)}.$$

The procedure is displayed in Figure 7.2. The heart of the procedure is the sub-routine TRYALL, which basically finds the maximum weight which can be packed into a bin of size β. Using π, we try first to include items which contribute the most to the objective relative to their size. This is a heuristic. The variables $\mathbf{v}$ and y keep track of the weight and total size of items included so far. The variable j indicates that the current item type is $\pi(j)$. In the for loop at the end of TRYALL, we try each possible number of type $\pi(j)$ items, starting with the largest possible number. First packing as many items as possible is a heuristic which seems to speed up computation. The current maximum is stored in x. When we enter TRYALL, we first compute an upper bound given the packing so far, which is stored in z. When $j = n$, this upper bound is exactly the objective value. If $z \leq x$, we do not have to consider any packing reachable from the current one, and we drop straight through. In the main routine we simply initialize x, call TRYALL for the two bins sizes, and return x. We display the upper bound achieved by VRH1(μ) for several

$x \leftarrow 1$.

TRYALL$(1, 0, 1, 1)$.

TRYALL$(1, 0, \alpha, \alpha)$.

Return x.

TRYALL$(j, \mathbf{v}, y, \beta)$

$z \leftarrow \left(\xi(\mathbf{v}) + y\,\psi_{\pi(j)}\right)/\beta$.

If $z > x$ then:

If $j = n$ then:

$x \leftarrow z$.

Else:

For $i \leftarrow \lceil y/t_{\pi(j)+1} \rceil - 1, \ldots, 0$:

TRYALL$(j+1,\ \mathbf{v} + i\mathbf{w}(t_{\pi(j)}),\ y - it_{\pi(j)+1},\ \beta)$.

Figure 7.2: The algorithm for computing $\mathcal{P}$, along with sub-routine TRYALL.

values of μ in Figure 7.3. By optimizing τ for each choice of α and μ, it is possible to improve the algorithm's performance. However, for simplicity's sake, we keep $\tau = \frac{1}{7}$ in this chapter.

Now we describe VRH2(μ). Redefine

$$T = \left\{ \frac{1}{i} \,\middle|\, 1 \le i \le n_1 \right\} \cup \left\{ \frac{\alpha}{i} \,\middle|\, 1 \le i \le n_2 \right\} \cup \{\alpha\mu,\, \alpha(1-\mu)\}.$$

Define n_1, n_2, ϵ and n as for VRH1. Again, rename the members of T as $t_1 = 1 > t_2 > t_3 > \cdots > t_n = \epsilon$. (7.1) guarantees that $1/2 < \alpha(1-\mu) < \alpha < 1$ and $1/3 < \alpha\mu < \alpha/2 < 1/2$, so we have $g = 3$ and $h = 6$. The only difference from VRH1 is that (g, h) bins have capacity α. Otherwise, the two algorithms are identical. We therefore omit a detailed description and analysis of VRH2. We display the performance ratio of VRH2(μ) for several values of μ in Figure 7.3.

7.3 Lower bounds

We now consider the question of lower bounds for the variable-sized bin packing problem. Prior to this work, no general lower bounds were known.

Our method follows along the lines laid down by Liang, Brown and van Vliet [17, 60, 71]. We give some unknown on-line bin packing algorithm $\mathcal{A}$ one of k possible different inputs. These inputs are defined as follows: Let $\varrho = s_1, s_2, \ldots, s_k$ be a sequence of *item sizes* such that $0 < s_1 < s_2 < \cdots < s_k \le 1$. Let ϵ be a small positive constant. We define σ_0 to be the empty input. Input σ_i consists of σ_{i-1} followed by n items of size $s_i + \epsilon$. Algorithm $\mathcal{A}$ is given σ_i for some $i \in \{1, \ldots, k\}$.

A *pattern* with respect to ϱ is a tuple $p = \langle \text{size}(p), p_1, \ldots, p_k \rangle$ where size(p) is a positive real number and $p_i, 1 \le i \le k$ are non-negative integers such that

$$\sum_{i=1}^{k} p_i\, s_i < \text{size}(p).$$

Intuitively, a pattern describes the contents of some bin of capacity size(p). Define $\mathcal{P}(\varrho, \beta)$ to be the set of all patterns p with respect to ϱ with size(p) $= \beta$. Further define

$$\mathcal{P}(\varrho) = \bigcup_{i=1}^{m} \mathcal{P}(\varrho, \alpha_i).$$

Note that $\mathcal{P}(\varrho)$ is necessarily finite. Given an input sequence of items, an algorithm is defined by the numbers and types of items it places in each of the bins it uses. Specifically, any algorithm is defined by a function $\Phi : \mathcal{P}(\varrho) \mapsto \mathbb{R}_{\ge 0}$. The algorithm uses $\Phi(p)$ bins containing items as described by the pattern p. We define $\phi(p) = \Phi(p)/n$.

Consider the function Φ that determines the packing used by on-line algorithm $\mathcal{A}$ uses for σ_k. Since $\mathcal{A}$ is on-line, the packings it uses for $\sigma_1, \ldots, \sigma_{k-1}$ are completely determined by Φ. We assign to each pattern a *class*, which is defined

$$\text{class}(p) = \min\{i \mid p_i \neq 0\}.$$

Intuitively, the class tells us the first sequence σ_i which results in some item being placed into a bin packed according to this pattern. I.e. if the algorithm packs some bins according to a pattern which has class i, then these bins will contain one or more items after σ_i. Define

$$\mathcal{P}_i(\varrho) = \{p \in \mathcal{P}(\varrho) \mid \text{class}(p) \leq i\}.$$

Then if $\mathcal{A}$ is determined by Φ, its cost for σ_i is simply

$$n \sum_{p \in \mathcal{P}_i(\varrho)} \text{size}(p)\phi(p).$$

Since the algorithm must pack every item, we have the following constraints

$$n \sum_{p \in \mathcal{P}(\varrho)} \phi(p)\, p_i \geq n, \qquad \text{for } 1 \leq i \leq k.$$

For a fixed n, define $\chi_i(n)$ to be the optimal off-line cost for packing the items in σ_i. The following lemma gives us a method of computing the optimal off-line cost for each sequence:

Lemma 7.3 *For $1 \leq i \leq k$, $\chi^* = \lim_{n\to\infty} \chi_i(n)/n$ exists and is the value of the linear program: Minimize*

$$\sum_{p \in \mathcal{P}_i(\varrho)} \text{size}(p)\phi(p) \tag{7.7}$$

subject to

$$1 \quad \leq \quad \sum_{p \in \mathcal{P}(\varrho)} \phi(p)\, p_j, \qquad \textit{for } 1 \leq j \leq i; \tag{7.8}$$

over variables χ_i and $\phi(p), p \in \mathcal{P}(\varrho)$.

Proof. Clearly, the LP always has a finite value between $\sum_{j=1}^{i} s_j$ and i. For any fixed n, the optimal off-line solution is determined by some ϕ. It must satisfy the constraints of the LP, and the objective value is exactly the cost incurred. Therefore the LP lower bounds the optimal off-line cost. The LP is a relaxation in that it allows a fractional number of bins of any pattern, whereas a legitimate solution must have an integral number. Rounding the relaxed solution up to get a legitimate one, the change in the objective value is at most $|\mathcal{P}(\varrho)|/n$. □

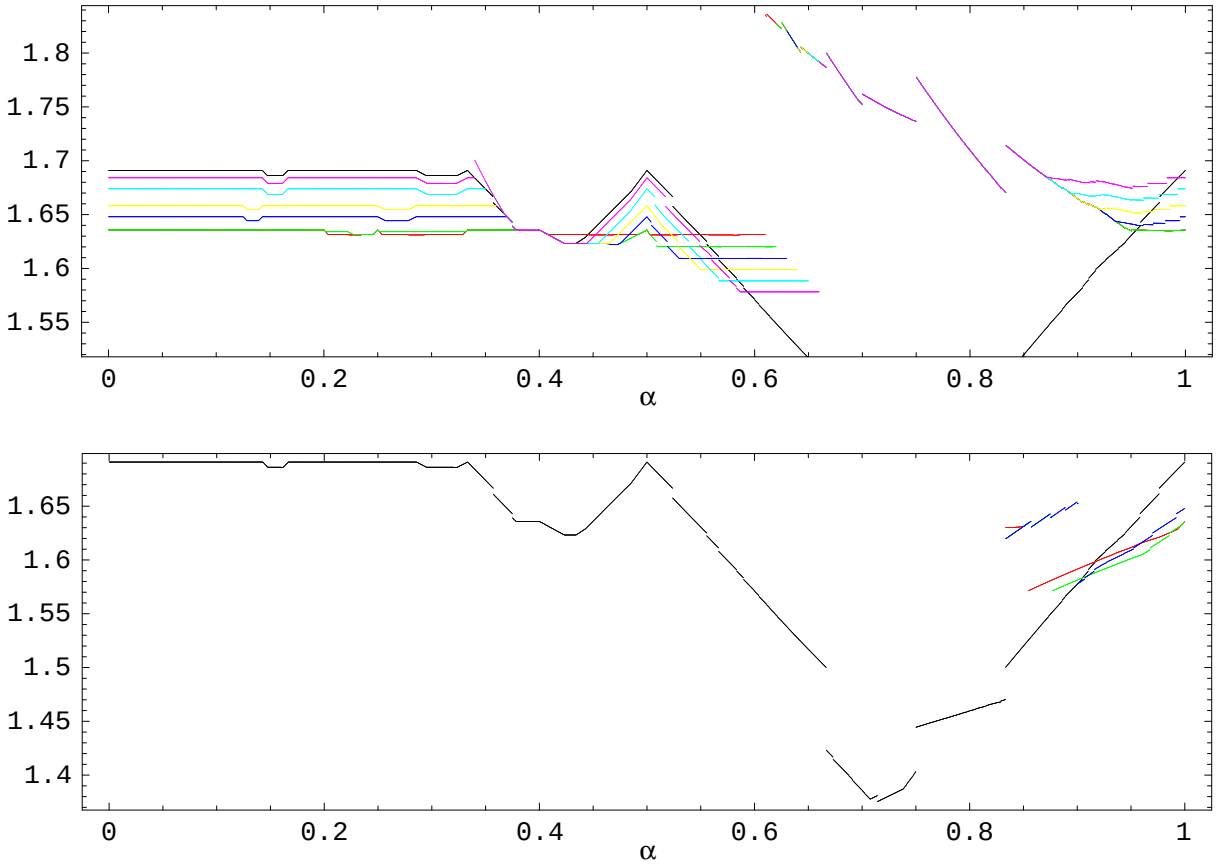

Figure 7.3: Upper bounds for variable sized bin packing. In both figures, VARIABLE HARMONIC is shown in black. In the top figure, we display VRH1$(.61)$ in red, VRH1$(.62)$ in blue, VRH1$(.63)$ in green, VRH1$(.64)$ in yellow, VRH1$(.65)$ in light blue and VRH1$(.66)$ in purple. In the bottom figure, we display VRH2$(.60)$ in red, VRH2$(.61)$ in blue and VRH2$(.62)$ in green.

Given the construction of a sequence, we need to evaluate

$$c = \min_{\mathcal{A}} \max_{i=1,\ldots,k} \limsup_{n\to\infty} \frac{\mathcal{A}(\sigma_i)}{\chi_i(n)} .$$

As $n \to \infty$, we can replace $\chi_i(n)/n$ by χ_i^*. Once we have the values $\chi_1^*, \ldots, \chi_k^*$, we can readily compute a lower bound for our on-line algorithm:

Lemma 7.4 *The optimal value of the linear program: Minimize c subject to*

$$\begin{aligned} c &\geq \frac{1}{\chi_i^*} \sum_{p\in\mathcal{P}_i(\varrho)} \text{size}(p)\phi(p), && \text{for } 1 \leq i \leq k; \\ 1 &\leq \sum_{p\in\mathcal{P}(\varrho)} \phi(p)\, p_i, && \text{for } 1 \leq i \leq k; \end{aligned} \tag{7.9}$$

over variables c and $\phi(p), p \in \mathcal{P}(\varrho)$, is a lower bound on the asymptotic performance ratio of any on-line bin packing algorithm.

Proof. For any fixed n, any algorithm $\mathcal{A}$ has some Φ which must satisfy the second constraint. Further, Φ should assign an integral number of bins to each pattern. However, this integrality constraint is relaxed, and $\sum_{p\in\mathcal{P}_i(\varrho)} \text{size}(p)\phi(p)$ is $1/n$ times the cost to $\mathcal{A}$ for σ_i as $n \to \infty$. The value of c is just the maximum of the performance ratios achieved on $\sigma_1, \ldots, \sigma_k$. □

Although this is essentially the result we seek, a number of issues are left to be resolved.

The first is that these linear programs have a variable for each possible pattern. The number of such patterns is potentially quite large, and we would like to reduce it if possible. We show that this goal is indeed achievable. We say that a pattern p of class i is *dominant* if

$$s_i + \sum_{j=1}^{k} p_j\, s_j > \text{size}(p).$$

Let p be a non-dominant pattern with class i. There exists a unique dominant pattern q of class i such that $p_j = q_j$ for all $i \neq j$. We call q the *dominator* of p with respect to class i.

Lemma 7.5 *In computing the values of the linear programs in Lemmas 7.3 and 7.4, it suffices to consider only dominant patterns.*

Proof. We transform an LP solution by applying the following operation to each non-dominant pattern p of class i: Let $x = \phi(p)$ in the original solution. We set $\phi(p) = 0$ and increment $\phi(q)$ by x, where q is the dominator of p with respect to i. The new solution remains feasible, and its objective value has not changed. Further, the value of $\phi(p)$ is zero for every non-dominant p, therefore these variables can be safely deleted. □

Given a sequence of item sizes ϱ, we can compute a lower bound $L_m(\varrho, \alpha_1, \ldots, \alpha_{m-1})$ using the following algorithm:

1. Enumerate the dominant patterns.
2. For $1 \leq i \leq k$, compute χ_i via the LP given in Lemma 7.3.
3. Compute and return the value of the LP given in Lemma 7.4.

Step one is most easily accomplished via a simple recursive function. Our concern in the remainder of this chapter shall be to study the behavior of $L_m(\varrho, \alpha_1, \ldots, \alpha_{m-1})$ as a function of ϱ and $\alpha_1, \ldots, \alpha_{m-1}$.

7.4 The lower bound sequences

Up to this point, we have assumed that we were given some fixed item sequence ϱ. We consider now the question of choosing a sequence ϱ on which on-line algorithms must perform poorly. We again focus in on the case where there are two bin sizes, and examine properties of $L_2(\varrho, \alpha_1)$. We abbreviate α_1 using α and L_2 using L.

To begin we define the idea of a *greedy* sequence. Let ϵ denote the empty sequence, and $\wedge$ the sequence concatenation operator. The greedy sequence $\Gamma_\tau(\beta)$ for capacity β with cutoff τ is defined by

$$\gamma(\beta) = \frac{1}{\left\lfloor \frac{1}{\beta} \right\rfloor + 1}; \qquad \Gamma_\tau(\beta) = \begin{cases} \epsilon & \text{if } \beta < \tau, \\ \gamma(\beta) \wedge \Gamma_\tau(\beta - \gamma(\beta)) & \text{otherwise.} \end{cases}$$

The sequence defines the item sizes which would be used if we packed a bin of capacity β using the following procedure: At each step, we determine the remaining capacity in our bin. We choose as the next item the largest reciprocal of an integer which fits without using the remaining capacity completely. We stop when the remaining capacity is smaller than τ. Note that for $\tau = 0$, we get the infinite sequence. We shall use Γ as a shorthand for Γ_0.

The recurrence u_i described in Section 1, which is found in connection with bounded-space bin packing [58], gives rise to the sequence

$$\frac{1}{u_i} = \frac{1}{2}, \frac{1}{3}, \frac{1}{7}, \frac{1}{43}, \frac{1}{1807}, \ldots.$$

This turns out to be the infinite greedy sequence $\Gamma(1)$. Somewhat surprisingly, it is also the sequence used by Brown, Liang and van Vliet in the construction of their lower bounds [17, 60, 71]. In essence, they analytically determine the value of $L_1(\Gamma_\tau(1))$. Liang and Brown lower bound the value, while van Vliet determines it exactly.

This well-known sequence is our first candidate. Actually, we use the first k items sizes in it, and we re-sort them so that the algorithm is confronted with items from smallest to largest. In

general, this re-sorting seems to be a good heuristic to make a given sequence hard (or harder) for an on-line algorithm, since the algorithm has the most decisions to make about how the smallest items are packed, but on the other hand has the least information about which further items will be received. The results are shown in Figure 7.4.

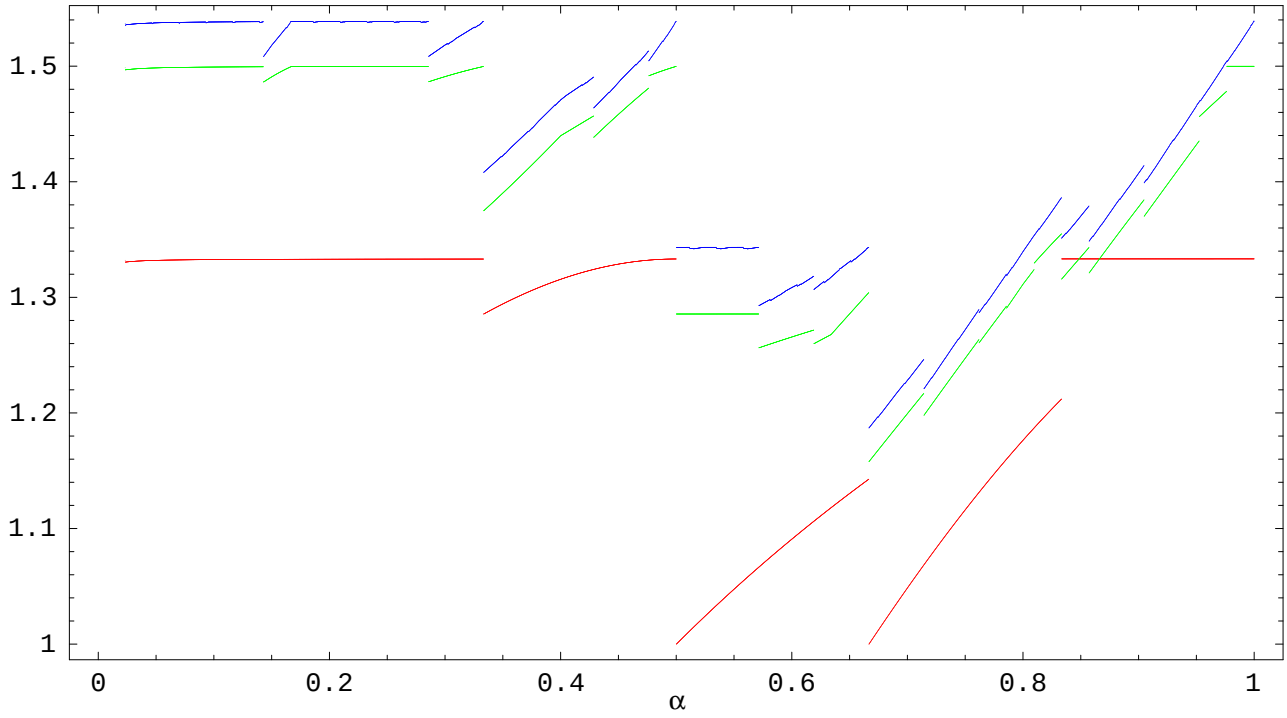

Figure 7.4: The evolution of the curves given by the greedy item sequence. In red is $\frac{1}{2}, \frac{1}{3}$; green is $\frac{1}{2}, \frac{1}{3}, \frac{1}{7}$; blue is $\frac{1}{2}, \frac{1}{3}, \frac{1}{7}, \frac{1}{43}$.

Examining Figure 7.4, one immediately notices that that $L(\Gamma_\tau(1), \alpha)$ exhibits some very strange behavior. The curve is highly discontinuous. Suppose we have a finite sequence ϱ, where each item size is a continuous function of $\alpha \in (0, 1)$. Tuple p is a *potential pattern* if there exists an $\alpha \in (0, 1)$ such that p is a pattern. The set of breakpoints of p with respect to ϱ is defined to be

$$B(p, \varrho) = \left\{ \alpha \in (0, 1) \middle| \sum_{i=1}^{k} p_i \, s_i = \text{size}(p) \right\}.$$

Let $\mathcal{P}^*$ be the set of all potential patterns. The set of all breakpoints is

$$B(\varrho) = \bigcup_{p \in \mathcal{P}^*} B(p, \varrho).$$

Intuitively, at each breakpoint some combinatorial change occurs, and the curve may jump. In the intervals between breakpoints, the curve behaves nicely as summarized by the following lemmas:

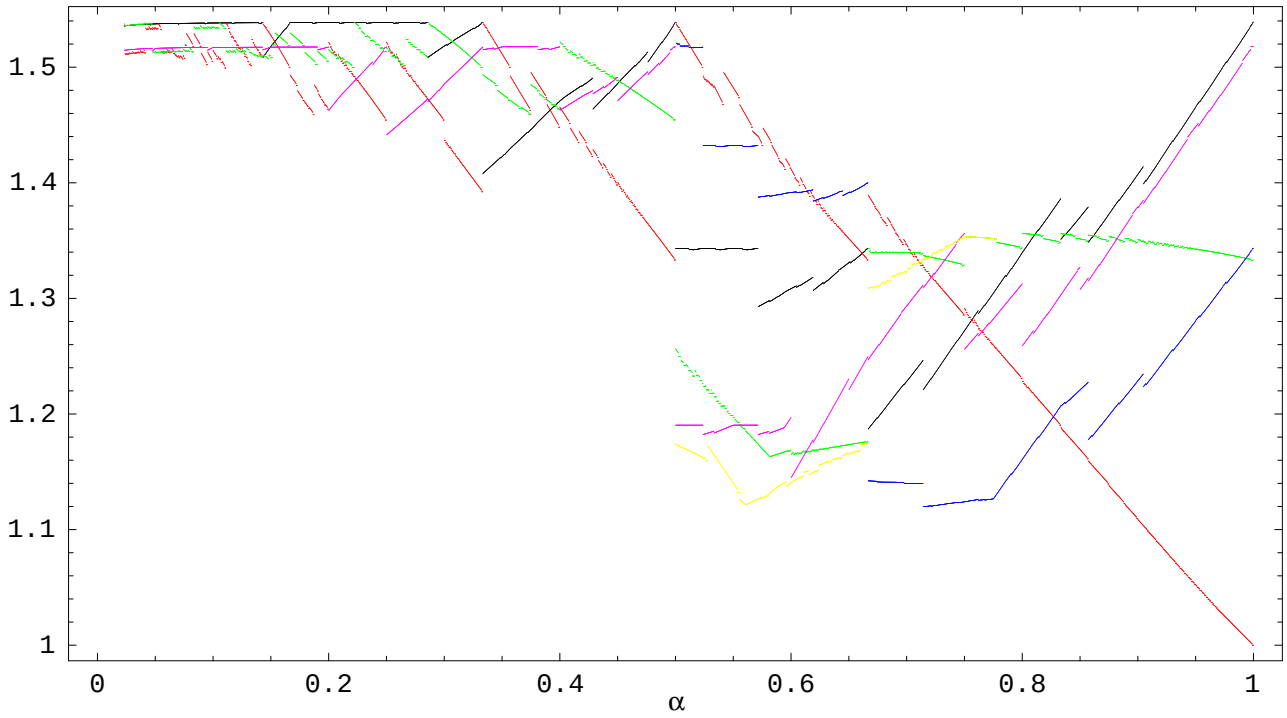

Figure 7.5: A variety of sequences for $\tau = 1/1000$: black is $\frac{1}{2}, \frac{1}{3}, \frac{1}{7}, \frac{1}{43}$; red is $\alpha, \Gamma_\tau(1-\alpha)$; green is $\frac{\alpha}{2}, \Gamma_\tau(1-\frac{\alpha}{2})$; blue is $\alpha, \frac{1}{3}, \frac{1}{7}, \frac{1}{43}$; purple is $\frac{1}{2}, \frac{1}{4}, \frac{1}{5}, \frac{1}{21}$; yellow is $\frac{1}{2}, \frac{\alpha}{2}, \frac{1}{9}, \Gamma_\tau(\frac{7}{18}-\frac{\alpha}{2})$.

Lemma 7.6 *Let ϱ be a finite item sequence, with each item size a continuous function of $\alpha \in (0,1)$. In any interval $I = (\ell, h)$ which does not contain a breakpoint, $L(\varrho, \alpha)$ is continuous. Furthermore, for all $\alpha \in I$,*

$$L(\varrho, \alpha) \geq \min\left\{\frac{\ell+h}{2h}, \frac{2\ell}{\ell+h}\right\} L\left(\varrho, \tfrac{1}{2}(\ell+h)\right).$$

This lemma follows as a corollary from:

Lemma 7.7 *Let ϱ be a finite item sequence, with each item size a continuous function of $\alpha \in (0,1)$. Let I be any interval which does not contain a breakpoint, and let α be any point in I. The following two results hold:*

1. *If $\delta > 0$ is such that $\alpha + \delta \in I$ then*

$$L(\varrho, \alpha+\delta) \geq \left(1 - \frac{\delta}{\alpha+\delta}\right) L(\varrho, \alpha).$$

2. *If $\delta > 0$ is such that $\alpha - \delta \in I$ then*

$$L(\varrho, \alpha-\delta) \geq \left(1 - \frac{\delta}{\alpha}\right) L(\varrho, \alpha).$$

Proof. We first prove statement 1. Denote by $\chi_i^*(x)$ the value of χ_i^* at $\alpha = x$. For $1 \leq i \leq k$ we have

$$\chi_i^*(\alpha + \delta) \leq \frac{\alpha + \delta}{\alpha} \chi_i^*(\alpha).$$

To see this, note that any feasible Φ at α is also feasible at $\alpha + \delta$, since both points are within I and (7.8) does not change within this interval. Each term in (7.7) increases by at most $(\alpha+\delta)/\alpha$. Now consider the linear program of Lemma 7.4. Consider some arbitrary feasible solution ϕ at α. At $\alpha + \delta$ this solution is still feasible (except that possibly c must increase). In the sum $1/\chi_i^* \sum_{p \in \mathcal{P}_i(\varrho)} \text{size}(p)\phi(p)$, the factor $1/\chi_i^*$ decreases by at most $\alpha/(\alpha + \delta)$ and size(p) cannot decrease.

Now consider statement 2. The arguments are quite similar. For $1 \leq i \leq k$ we have

$$\chi_i^*(\alpha - \delta) \leq \chi_i^*(\alpha).$$

Again, a feasible solution remains feasible, Further, its objective value (7.7) cannot increase. Considering the linear program of Lemma 7.4, we find that for each feasible solution, each sum $1/\chi_i^* \sum_{p \in \mathcal{P}_i(\varrho)} \text{size}(p)\phi(p)$ decreases by a factor at most $(\alpha - \delta)/\alpha$. □

Considering Figure 7.4 again, there are sharp drops in the lower bound near the points $\frac{1}{3}$ $\frac{1}{2}$ and $\frac{2}{3}$. It is not hard to see why the bound drops so sharply at those points. For instance, if α is just larger than $\frac{1}{2} + \epsilon$, then the largest items in $\Gamma(1)$ can each be put in their own bin of size α. If $\alpha \geq \frac{2}{3} + 2\epsilon$, two items of size $\frac{1}{3} + \epsilon$ can be put pairwise in bins of size α. In short, in such cases the on-line algorithm can pack some of the largest elements in the list with very little wasted space, hence the low resulting bound.

This observation leads us to try other sequences, in which the last items cannot be packed well. A first candidate is the sequence $\alpha, \Gamma(1 - \alpha)$. As expected, this sequence performs much better than $\Gamma(1)$ in the areas described above.

It is possible to find further improvements for certain values of α. For instance, the sequence $\alpha/2, \Gamma(1 - \alpha/2)$ also works well in some places, and we used other sequences as well. These are shown in Figure 7.5.

As a general guideline for finding sequences, items should not fit too well in either bin size. If an item has size x, then $\min\left\{1 - \lfloor \frac{1}{x} \rfloor x, \alpha - \lfloor \frac{\alpha}{x} \rfloor x\right\}$ should be as large as possible. In areas where a certain item in a sequence fits very well, that item should be adjusted (e.g. use an item $1/(j+1)$ instead of using the item $1/j$) or a completely different sequence should be used. (This helps to explain why the algorithms have a low competitive ratio for α close to 0.7: in that area, this minimum is never very large.)

Furthermore, as in the classical bin packing problem, sequences that are bad for the on-line algorithm should have very different optimal solutions for each prefix sequence. Finally, the item sizes should not increase too fast or slow: If items are very small, the smallest items do

not affect the on-line performance much, while if items are close in size, the sequence is easy because the optimal solutions for the prefixes are alike.

Using Lemma 7.7 we obtain the main theorem of this section:

Theorem 7.1 *Any on-line algorithm for the variable sized bin packing problem with* $m = 2$ *has asymptotic performance ratio at least* $495176908800/370749511199 > 1.33561$.

Proof. First note that for $\alpha <\in (0, 1/43]$, the sequence $\frac{1}{2}, \frac{1}{3}, \frac{1}{7}, \frac{1}{43}$ yields a lower bound of $217/141 > 1.53900$ as in the classic problem: The bin of size α is of no use.

We use the sequences described in the caption of Figure 7.5. For each sequence ϱ, we compute a lower bound on $(1/43, 1)$ using the following procedure:

Define $\varepsilon = 1/10000$. We break the interval (0,1) into subintervals using the lattice points $\varepsilon, 2\varepsilon, \ldots, 1 - \varepsilon$. To simplify the determination of breakpoints, we use a constant sequence for each sub-interval. This constant sequence is fixed at the upper limit of the interval. I.e. throughout the interval $[\ell\varepsilon, \ell\varepsilon + \varepsilon)$ we use the sequence $\varrho|_{\alpha=\ell\varepsilon+\varepsilon}$. Since the sequence is constant, a lower bound on the performance ratio of any on-line bin packing algorithm with $\alpha \in [\ell\varepsilon, \ell\varepsilon+\varepsilon)$ can be determined by the following algorithm:

1. $\varrho' \leftarrow \varrho|_{\alpha=\ell\varepsilon+\varepsilon}$.
2. Initialize $B \leftarrow \{\ell\varepsilon, \ell\varepsilon + \varepsilon\}$.
3. Enumerate all the patterns for ϱ' at $\alpha = \ell\varepsilon + \varepsilon$.
4. For each pattern:
   (a) $z \leftarrow \sum_{i=1}^{k} p_i s_i$.
   (b) If $z \in (\ell\varepsilon, \ell\varepsilon + \varepsilon)$ then $B \leftarrow B \cup \{z\}$.
5. Sort B to get $b_1, b_2, , \ldots, b_j$.
6. Calculate and return the value:

$$\min_{1\le i<j} \min \left\{ \frac{b_i + b_{i+1}}{2b_{i+1}}, \frac{2b_i}{b_i + b_{i+1}} \right\} L\left(\varrho', \tfrac{1}{2}(b_i + b_{i+1})\right).$$

We implemented this algorithm in `Mathematica`, and used it to find lower bounds for each of the aforementioned sequences. The results are shown in Figures 7.5 and 7.6. The lowest lower bound is $495176908800/370749511199$, in the interval [0.7196,0.7197). □

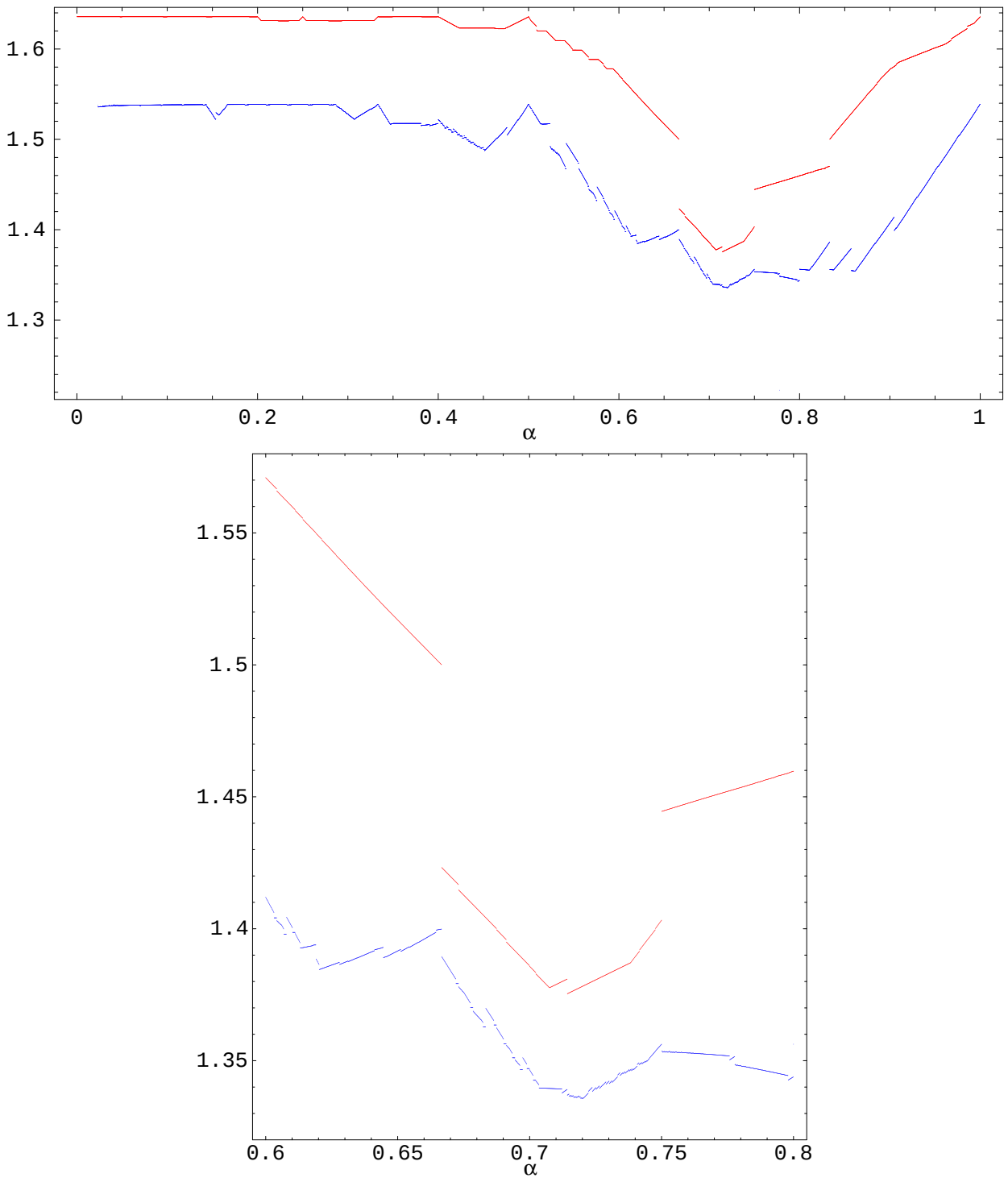

Figure 7.6: The best upper and lower bounds for variable sized on-line bin packing. The bottom figure is a closeup of $[.6, .8]$. The upper bound is best of the VRH1, VRH2 and VARIABLE HARMONIC algorithms.

7.5 Conclusions

We have shown new algorithms and lower bounds for variable-sized on-line bin packing with two bin sizes. The largest gap between the performance of the algorithm and the lower bound is 0.18193, achieved for a second bin of size $\alpha = 0.9071$. The smallest gap is 0.03371 achieved for $\alpha = 0.6667$. Note that for $\alpha \leq \frac{1}{2}$, there is not much difference with the classical problem: having the extra bin size does not help the on-line algorithm much. To be more precise, it helps about as much as it helps the off-line algorithm.

Our work raises the following questions: is there a value of α where it is possible to design a better algorithm and show a matching lower bound? Or, can a lower bound be shown anywhere that matches an existing algorithm? Note that at the moment there is also a small gap between the competitive ratio of the best algorithm and the lower bound in the classical bin packing problem.

Another interesting open problem is analyzing variable-sized bin packing with an arbitrary number of bin sizes.

Publications

In this section we list the papers on which the chapters are based.

Chapter 2 is based on L. Epstein and R. van Stee, “Optimal On-Line Flow Time With Resource Augmentation”. In R. Freivalds, editor, *Fundamentals of Computation Theory, 13th International Symposium, FCT 2001, Proceedings*, volume 2138 of *Lecture Notes in Computer Science*, pp. 472–482. Springer, 2001. This paper was presented at the First Workshop on Efficient Algorithms (WEA).

Chapter 3 is based on Y. Azar, L. Epstein and R. van Stee, “Resource Augmentation in Load Balancing”. *Journal of Scheduling*, 3(5):249–258, 2000. This paper also appeared in M. M. Halldorsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pp. 189–199. Springer, 2000.

Chapter 4 is based on R. van Stee and H. La Poutré, “Running a Job on a Collection of Partly Available Machines, with On-Line Restarts”. *Acta Informatica*, 37(10):727–742, 2001.

Chapter 5 is based on R. van Stee and H. La Poutré, “Partial Servicing of On-line Jobs”. *Journal of Scheduling*, 4(6):379–396, 2001. This paper also appeared in K. Jansen, S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization, Third Internationational Workshop, APPROX 2000*, volume 1913 in *Lecture Notes in Computer Science*, pp. 250–261. Springer, 2000.

Chapter 6 is based on L. Epstein and R. van Stee, “Minimizing the Maximum Starting Time On-line”. Technical Report SEN-R0133. CWI, Amsterdam, 2001.

Chapter 7 is based on L. Epstein, S. Seiden and R. van Stee, “New Bounds for Variable-Sized and Resource Augmented Online Bin Packing”. To appear in *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Proceedings*. Springer, 2002.

Bibliography

[1] S. Aggarwal, J.A. Garay, and A. Herzberg. Adaptive video on demand. In P. Spirakis, editor, *Algorithms - ESA '95, Proceedings Third Annual European Symposium*, volume 979 of *Lecture Notes in Computer Science*, pages 538–553. Springer, 1995.

[2] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18(3):283–305, Jul 1997.

[3] S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, 1999.

[4] B. Awerbuch and Y. Azar. Competitive multicast routing. *Wireless Networks*, 1:107–114, 1995.

[5] B. Awerbuch, Y. Azar, A. Fiat, and F. T. Leighton. Making commitments in the face of uncertainty: How to pick a winner almost every time. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 519–530, 1996.

[6] B. Awerbuch, Y. Azar, A. Fiat, S. Leonardi, and A. Rosen. On-line competitive algorithms for call admission in optical networks. *Algorithmica*, 31(1):29–43, 2001. Also in J. Diaz, M. Serna, editors, *Algorithms - ESA '96, Proceedings Fourth Annual European Symposium*, volume 1136 of Lecture Notes in Computer Science, pages 431–444. Springer, 1996.

[7] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 198–205. ACM, 1999.

[8] Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. In *5th Israeli Symposium on Theory of Computing and Systems*, pages 119–125, 1997.

[9] Y. Bartal, A. Fiat, H. J. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. *Journal of Computer and System Sciences*, 51:359–366, 1995.

[10] S.K. Baruah and M.E. Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47:1027–1032, 1998.

[11] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[12] P. Berman and C. Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6:181–193, 1999. Also in S. Arnborg, L. Ivansson, editors, *Algorithm Theory - SWAT'98, 6th Scandinavian Workshop on Algorithm Theory, Proceedings*, volume 1432 of *Lecture Notes in Computer Science*, pages 189–199. Springer, 1999.

[13] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[14] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50:244–258, 1995.

[15] H. G. P. Bosch, N. Nes, and M. L. Kersten. Navigating through a forest of quad trees to spot images in a database. Technical Report INS-R0007, CWI, Amsterdam, February 2000.

[16] M. Brehob, E. Torng, and P. Uthaisombut. Applying extra-resource analysis to load balancing. *Journal of Scheduling*, 3(5):273–288, 2000.

[17] D. J. Brown. A lower bound for on-line one-dimensional bin packing algorithms. Technical Report R-864, Coordinated Sci. Lab., Urbana, Illinois, 1979.

[18] S. Chakrabarti, C.A. Phillips, A.S. Schulz, D.B. Shmoys, C. Stein, and J.Wein. Improved scheduling algorithms for minsum criteria. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium, ICALP '96, Proceedings*, volume 1099 of *Lecture Notes in Computer Science*, pages 646–657. Springer, 1996.

[19] B. Chandra. Does randomization help in on-line bin packing? *Information Processing Letters*, 43(1):15–19, Aug 1992.

[20] B. Chen, A. van Vliet, and G. J. Woeginger. Lower bounds for randomized online scheduling. *Information Processing Letters*, 51:219–222, 1994.

[21] B. Chen, A. van Vliet, and G. J. Woeginger. An optimal algorithm for preemptive on-line scheduling. *Operations Research Letters*, 18:127–131, 1995.

[22] E.K.P. Chong and W. Zhao. Performance evaluation of scheduling algorithms for imprecise computer systems. *Journal of Systems and Software*, 15:261–277, 1991.

[23] P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[24] M. Chrobak and J. Noga. LRU is better than FIFO. *Algorithmica*, 23:180–185, 1999.

[25] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, chapter 2. PWS Publishing Company, 1997.

[26] E. G. Coffman, Jr., L. Flatto, and P. E. Wright. A stochastic checkpoint optimization problem. *SIAM Journal on Computing*, 22(3):650–659, June 1993.

[27] J. Csirik. An on-line algorithm for variable-sized bin packing. *Acta Informatica*, 26(8):697–709, October 1989.

[28] J. Csirik and G. Woeginger. On-line packing and covering problems. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms—The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, chapter 7, pages 147–177. Springer-Verlag, 1998.

[29] T. Dean and M. Boddy. An analysis of time-dependent planning. In *AAAI 88, The Seventh National Conference on Artificial Intelligence*, pages 49–54. AAAI, 1988.

[30] C. Derman. *Finite State Markovian decision processes*. Academic Press, New York, 1970.

[31] M. L. Dertouzos and A. K.-L. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15:1497–1506, 1989.

[32] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235:109–141, 2000.

[33] L. Epstein and J. Sgall. A lower bound for on-line scheduling on uniformly related machines. *Operations Research Letters*, 26(1):17–22, 2000.

[34] Wu-Chen Feng. Applications and extensions of the imprecise-computation model. Technical report, University of Illinois at Urbana-Champaign, December 1996.

[35] A. Fiat and G. Woeginger, editors. *On-Line Algorithms—The State of the Art*. Lecture Notes in Computer Science. Springer-Verlag, 1998.

[36] R. Fleischer and M. Wahl. Online scheduling revisited. *Journal of Scheduling*, 3(5):343–353, 2000. Also in M. Paterson, editor, *Algorithms - ESA 2000, Proceedings Eighth Annual European Symposium*, volume 1879 in *Lecture Notes in Computer Science*, pages 202–210. Springer, 2000.

[37] D. K. Friesen and M. A. Langston. A storage-size selection problem. *Information Processing Letters*, 18:295–296, 1984.

[38] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM Journal on Computing*, 15:222–230, 1986.

[39] A. Goel, M. R. Henzinger, S. Plotkin, and E. Tardos. Scheduling data transfers in a network and the set scheduling problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 189–197. ACM, 1999.

[40] T. Gormley, N. Reingold, E. Torng, and J. Westbrook. Generating adversaries for request-answer games. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–565. ACM-SIAM, 2000.

[41] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[42] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[43] J.A. Hoogeveen and A.P.A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In W. H. Cunningham, S. T. McCormick, and M. Queyranne, editors, *Integer Programming and Combinatorial Optimization, 5th International IPCO Conference, Proceedings*, volume 1084 of *Lecture Notes in Computer Science*, pages 404–414. Springer, 1996.

[44] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[45] David S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1973.

[46] David S. Johnson, A. Demers, J. D. Ullman, Michael R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:256–278, 1974.

[47] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:617–643, 2000.

[48] Bala Kalyanasundaram and Kirk Pruhs. Maximizing job completions online. In G. Bilardi, G.F. Italiano, A. Pietracaprina, and G. Pucci, editors, *Algorithms - ESA '98, Proceedings Sixth Annual European Symposium*, volume 1461 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1998. To appear in Journal of Algorithms.

[49] D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20:400–430, 1996.

[50] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[51] A. Karlin, S. Phillips, and P. Raghavan. Markov paging. *SIAM Journal on Computing*, 30:906–922, 2000.

[52] H. Kellerer, T. Tautenhahn, and G.J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing*, 28:1155–1166, 1999.

[53] N.G. Kinnersley and M.A. Langston. Online variable-sized bin packing. *Discrete Applied Mathematics*, 22(2):143–148, Feb 1989.

[54] K.J.Lin, S. Natarajan, and J.W.S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the Eighth IEEE Real-Time Systems Symposium*, pages 210–215, 1987.

[55] T. W. Lam and K. K. To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632. ACM-SIAM, 1999.

[56] E.L. Lawler, J.K Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling: algorithms and complexity. In *Handbooks in operations research and management science*, volume 4, pages 445–522. North Holland, 1993.

[57] C.-Y. Lee, L. Lei, and M. Pinedo. Current trends in deterministic scheduling. *Annals of Operations Research*, 70:1–42, 1997.

[58] C.C. Lee and D.T. Lee. A simple on-line bin-packing algorithm. *Journal of the ACM*, 32(3):562–572, Jul 1985.

[59] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29th ACM Symposium on the Theory of Computing*, pages 110–119. ACM, 1997. To appear in Journal of Computer and System Sciences.

[60] F. M. Liang. A lower bound for online bin packing. *Information Processing Letters*, 10:76–79, 1980.

[61] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[62] P. Ramanan, D.J. Brown, C.C. Lee, and D.T. Lee. On-line bin packing in linear time. *Journal of Algorithms*, 10(3):305–326, Sep 1989.

[63] M. B. Richey. Improved bounds for harmonic-based bin packing algorithms. *Discrete Applied Mathematics*, 34:203–227, 1991.

[64] Eric Sanlaville and Günter Schmidt. Machine scheduling with availability constraints. *Acta Informatica*, 35(9):795–811, 1998.

[65] S. S. Seiden. An optimal online algorithm for bounded space variable-sized bin packing. *SIAM Journal on Discrete Mathematics*, 14(4):458–470, 2001.

[66] S.S. Seiden. On the online bin packing problem. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2001.

[67] J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Information Processing Letters*, 63(1):51–55, 1997.

[68] W.-K. Shih. Scheduling in real-time systems to ensure graceful degradation: the imprecise-computation and the deferred-deadline approaches. Technical report, University of Illinois at Urbana-Champaign, December 1992.

[69] W.-K. Shih and J.W.S. Liu. On-line scheduling of imprecise computations to minimize error. *SIAM Journal on Computing*, 25:1105–1121, 1996.

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

[71] A. van Vliet. An improved lower bound for online bin packing algorithms. *Information Processing Letters*, 43(5):277–284, Oct 1992.

[72] A.P.A. Vestjens. *On-line Machine Scheduling*. PhD thesis, Technical University Eindhoven, The Netherlands, 1997.

[73] G. Woeginger and G. Zhang. Optimal on-line algorithms for variable-sized bin covering. Technical Report Woe-22, TU Graz, Institut für Mathematik B, Feb 1998.

[74] A. C. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 222–227. IEEE, 1977.

[75] A. C. C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.

[76] W. Zhao, S. Vrbsky, and J.W.S. Liu. Performance of scheduling algorithms for multi-server imprecise systems. In *Proceedings of the Fifth International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 1992.

[77] S. Zilberstein. Constructing utility-driven real-time systems using anytime algorithms. In *Proceedings of the First IEEE Workshop on Imprecise and Approximate Computation*, 1992.

Samenvatting

On-line algoritmen zijn algoritmen die al een deel van de oplossing van een probleem kunnen leveren voordat alle gegevens bekend zijn. Zulke algoritmen zijn nodig in omgevingen waar de gegevens in de loop van de tijd binnenkomen. Een voorbeeld hiervan is een fabriekshal met een aantal machines die verschillende producten kunnen maken. In de loop van de dag komen orders binnen, maar het is niet de bedoeling dat de machines stil blijven staan totdat alle orders binnen zijn. Ze moeten zo snel mogelijk gaan draaien, en hierbij moet rekening gehouden worden met de nog onbekende orders die later binnen kunnen gaan komen.

Dit proefschrift behandelt met name *on-line scheduling*, waarbij productieschema's voor een of meer machines moeten worden gemaakt aan de hand van bepaalde criteria. Hierbij worden de resultaten van een on-line algoritme vergeleken met de resultaten van een off-line algoritme dat wel alle gegevens van te voren weet, en dus een optimale oplossing kan leveren – er worden geen eisen gesteld aan de complexiteit van de algoritmen, on-line zowel als off-line. Als gevolg hiervan gaat het hier om een slechtste-gevalanalyse en wordt een algoritme beoordeeld op zijn prestaties in de slechtst mogelijke omstandigheden. Het doel is om het in verhouding met het off-line algoritme zo goed mogelijk te doen, er wordt dus gekeken naar de maximale verhouding tussen de on-line en de off-line kosten. Deze verhouding heet *competitive ratio*.

We bekijken verschillende varianten en uitbreidingen op deze analyse. De belangrijkste hiervan is het toestaan van randomisatie, waarbij een on-line algoritme rekening kan houden met verschillende mogelijke toekomstige opdrachten door random bits te gebruiken in zijn beslissingen. Verder bekijken we de situatie waarin het on-line algoritme meer machines tot zijn beschikking heeft dan het off-line algoritme. Dit kan in sommige gevallen een helderder beeld geven van hoe nadelig het is om de toekomst niet te kennen.

De criteria voor scheduling die we in die proefschrift beschouwen zijn de gemiddelde wachttijd over alle opdrachten, de laatste eindtijd (de tijd dat de laatste opdracht klaar is), de laatste starttijd en tenslotte het gemiddelde machinegebruik. In dit laatste geval gaat het om opdrachten die niet volledig uitgevoerd hoeven te worden en is het de bedoeling om de machines zo efficiënt mogelijk te gebruiken. Een voorbeeld van dit soort opdrachten zijn benaderingen van bepaalde waardes, die beter worden naarmate er meer tijd aan besteed wordt.

Hoofdstuk 2 behandelt het gemiddelde-wachttijdcriterium, waarbij het on-line algoritme meer machines heeft dan off-line, die er één heeft. We geven een algoritme en laten zien dat het optimaal is. In hoofdstuk 3 behandelen we het laatste-eindtijdcriterium op dezelfde manier, met dit verschil dat hier het off-line algoritme meer dan één machine heeft: we geven optimale algoritmen voor verschillende situaties.

In hoofdstuk 4 behandelen we scheduling op machines die niet continu beschikbaar zijn, waarbij het probleem is of er gewacht moet worden tot een machine weer beschikbaar komt of dat het beter is om op een nieuwe machine helemaal opnieuw te beginnen. In hoofdstuk 5 behandelen we het schematiseren van opdrachten die niet volledig uitgevoerd hoeven te worden.

Hoofdstuk 6 gaat over het probleem van het minimaliseren van de maximale starttijd van taken die één voor één binnenkomen. Hierbij moeten alle taken toegewezen worden aan machines voordat de machines gaan draaien. Vervolgens moeten ze uitgevoerd worden waarbij ze op elke machine in volgorde van toewijzing gedraaid moeten worden.

Hoofdstuk 7 tenslotte gaat over bin packing. Dit is het probleem van het inpakken van een reeks objecten in dozen (bins) waarbij het de bedoeling is zo weinig mogelijk dozen te gebruiken. We bekijken een variant van het probleem waarbij er dozen met meerdere groottes beschikbaar zijn, en het doel is om het totaal gebruikte volume te minimaliseren.

Curriculum vitae

Rob van Stee werd geboren op 19 september 1973 in Vlissingen. Hij behaalde in juni 1991 zijn gymnasiumdiploma aan het Gertrudislyceum te Roosendaal. In juni 1992 behaalde hij een propedeuse Wiskunde en een propedeuse Informatica aan de universiteit van Leiden. Hij behaalde zijn doctoraal in de Wiskunde op 28 juni 1996 met als afstudeeronderwerp het aantal nulpunten van gegeneraliseerde exponentiële polynomen in algebraï sche getallenlichamen. Zijn begeleider hierbij was dr. J.-H. Evertse.

Vanaf augustus 1996 werkte Rob als onderzoeker in opleiding op een project van NWO, de Nederlandse Organisatie voor Wetenschappelijk Onderzoek. Hij werd begeleid door prof.dr.ir. J.A. La Poutré en prof.dr. J.N. Kok. Het onderzoek vond plaats aan de universiteit van Leiden in het eerste jaar en op het Centrum voor Wiskunde en Informatica van september 1997 tot en met augustus 2001. Op dit moment werkt hij als postdoc aan de Albert-Ludwigs-Universiteit te Freiburg.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bovsnavcki. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09