Online busy time scheduling

Rick van de Bovenkamp
Utrecht University

Abstract. In this thesis we study the online busy time scheduling problem with infinite processors, where each job has a release time r_j, a processing time p_j and a deadline d_j. The objective of busy time scheduling is to use multiple processors to schedule jobs concurrently in order to minimize the time a machine has to be processing jobs. We present $(1 + \frac{4}{1+\lambda})$-consistent and $(1 + \frac{4}{1-\lambda})$-robust algorithm using machine learned advice that is able to achieve better results than a pure online algorithm. This algorithm will use a trust parameter, $\lambda \in [0, 1]$, which allows us to control the tradeoff between consistency and robustness. Moreover, for purely online busy time problem, we introduce a lower bound of 2 for eager algorithms, disprove the currently claimed upper bound of 4, and present a general framework for analysis.

1 Introduction

Scheduling jobs on parallel machines is a computer science problem that has been studied extensively by the scientific community. One need only look at the books and surveys that have been written to find that many settings of this problem exist, and have been researched. See for example [1], [2], where many of these problem settings are discussed. Most of these settings focus on minimizing the makespan, the maximum lateness, the total completion time or the tardiness. But what if we want to schedule jobs in an energy-, or, cost-efficient manner? In this thesis we will discuss the online busy time problem which, among other things, models energy-efficient computing closely.

1.1 Busy time scheduling

In 2010, Flammini et al. introduced the busy time problem [3]. In this problem, we want to schedule a set of jobs on multiple machines, each having one or more processors. The objective is to schedule the jobs in such a way that the total time that the machines are actively processing jobs is minimized. In other words, the time that a machine is idle does not count towards the busy time.

Formally, we have a set of *jobs* J. Each of these jobs has a *release time* $r_j \geq 0$, a required *processing time* $p_j > 0$ and a *deadline* $d_j \geq r_j + p_j$. Any instance also defines a parameter $g \geq 1$ which corresponds to the maximum number of jobs that can be ran on a machine in parallel, in later sections we will focus on the case where $g = \infty$. Any algorithm must then decide what the *starting times* s_j of each job will be, such that the busy time is minimized [4]. These starting times should be chosen such that $s_j + p_j \leq d_j$. The objective of busy time scheduling is to use multiple processors to schedule jobs concurrently in order to minimize the time a machine has to be processing jobs.

Of course there are many variations. The jobs can either be *rigid* ($d_j = r_j + p_j$) or *flexible*, the amount of processors g can be finite or infinite, machines can have varying properties like processing speed, cost, etc. We will discuss these variations and their current research on them later, in Section 3.

In 2003, P. Winkler et al. have shown an offline algorithm for solving the wavelength assignment minimization problem in fiber-optical networks [5]. Here, the objective is to assign wavelengths to a set of lightpaths such that the switching cost of the network is minimized. Coincidentally the rigid busy time scheduling problem can be converted into

a wavelength assignment problem and vice versa. They show that the offline problem is NP-hard once $g \geq 2$.

Besides varying the instance parameters one could also vary the objectives of the problem. A well known objective is to maximize the throughput instead of minimizing the busy time, where we are given a busy time budget T , and want to maximize the total amount or total weight of the scheduled jobs [6][7]. Within this thesis, however, we will focus on busy time minimization as opposed to throughput maximization.

1.2 Online scheduling

While offline scheduling algorithms are useful when one can know in advance what, or even how many jobs will have to be scheduled, and what their properties are, we often do not have this information readily available in practice. Instead, jobs may have to be scheduled as they are revealed to us, and, in this way it might not always be possible to schedule all requests optimally compared to an offline algorithm. In this section we will discuss various techniques of developing and analysing algorithms for online scheduling problems.

1.2.1 Online-time and online-list

There are two main models for such online problems; the *online-time* model and the *online-list* model.

In the online-time model, requests are revealed in order of time. For example, in the context of the busy time scheduling problem, this means that jobs are revealed in order of their respective release times. This model may be advantageous for developing an algorithm since it limits the range of possibilities.

In contrast to the online-time model, the online-list model offers no such guarantees, and requests may come in at any order.

Which of these models is best suited for developing and analyzing online algorithms depends on the problem and the applications of a given algorithm. For busy time scheduling specifically, all of the previous results we mention in Section 3 use online-time. This makes sense for the problem domain since the release times of jobs can be seen as the time the jobs are revealed to the algorithm, which could very well be the case in practical applications.

One thing to note here is that, when using online-time for job scheduling, there is also a distinction between *clairvoyant* and *non-clairvoyant* problems. In a clairvoyant setting the processing time of a job is revealed to the algorithm once it is released, while in the non-clairvoyant setting the algorithm does not know the processing time of a job. Most of the results mentioned in this thesis will consider the clairvoyant setting, unless otherwise mentioned.

1.2.2 Competitive analysis

The primary goal of an online algorithm is not necessarily to compute a solution as fast as possible. Instead, we are looking to optimize the solution quality produced by an algorithm, compared to the optimal solution. Let $ALG(I)$ denote some *algorithm's solution's cost* given input instance I, and $OPT(I)$ denote the respective *optimal solution*. We can then define the *competitive ratio* to be some c if for any input instance I, and some constant a, $ALG(I) \leq c \cdot OPT(I) + a$ holds [8]. Such an algorithm is called *c-competitive*.

Different analyses can show different ranges of the possible competitive ratio of an algorithm. Therefore, in order to show that the analysis does not overestimate the competitive

ratio of an algorithm, one can provide an input instance I for which $ALG(I) = c \cdot OPT(I) + a$. If such an input instance is provided we call the analysis *tight*.

Aside from developing and analyzing online algorithms, we are also interested in analyzing the problem as a whole, in order to show what's possible in the online setting. By providing an algorithm with a competitive ratio c we prove what is called the *upper bound* of a problem. This upper bound tells us that the best-possible competitive ratio is no larger than c.

If we want to show that any online algorithm cannot have a competitive ratio lower than $c_{\min}$, we can to provide an *adaptive input instance*, or *adaptive adversary*, for which no online algorithm can possibly perform better than $c_{\min}$ compared to the optimal solution. This bound, $c_{\min}$ is called the *lower bound* of a problem. This adaptive adversary does not have to define which requests are done in advance, instead, it can base its decisions based on what the algorithm has done so far.

One of the downsides to the competitive ratio is that it only tells us something about the worst-case performance of an algorithm. In practice, we may be more interested in how well an algorithm performs on average, or how well it performs on a certain set of input instances. In Sections 1.2.4 and 1.2.5 we will discuss alternative methods for creating and evaluating online algorithms.

1.2.3 Resource augmentation

Imagine, for example, that no online algorithm for a problem with a constant, or sufficiently low competitive ratio, exists. In this case, one can also look into *resource augmentation* [9], [10]. With this type of analysis the online algorithm is given more resources than the adversary, such as more speed or more processors in the context of busy time scheduling.

This method was first introduced by Kalyanasundaram et al. [11] where they analyze two well-known scheduling problems. These problems both had no known constant-competitive algorithm in the non-clairvoyant problem settings. When using resource augmentation the competitive ratios did become constant-competitive, making it potentially as powerful as clairvoyancy. Resource augmentation is a tradeoff, though, since needing to augment the resources of an algorithm by a lot also decreases the applicability of such an algorithm.

Interestingly, for the busy time problem, if we augment the speed of an online algorithm, any rigid job would turn into a flexible job. This could mean that it is a useful technique for analyzing the problem setting where there are rigid jobs and limited processors on multiple machines.

1.2.4 Online scheduling with predictions

A different way of developing a good-performing online algorithm is to make use of what is called *advice*, or *predictions*. In the past, researchers have developed a model where the algorithm can obtain information from a given advisor [12]–[15]. This advisor knows the input instance and can relay information about this input instance, or the optimal solution, to the algorithm.

Unfortunately, communicating large amounts of data on the input instance can be very costly, especially in the setting of communication problems where this model is often used. Therefore the challenge becomes to find a good competitive ratio while keeping the number of bits in the advice, called the *advice complexity*, as small as possible.

In recent years, however, a new model which closely resembles that of the advice model, has become quite popular. Instead of having an advisor that knows the input instance, we can

use an advisor that gives us, potentially unreliable, *predictions* about the input instance [16]–[22]. This advisor can for example be trained using machine-learning techniques.

Since the predictions that are made may contain large errors, we consider it to be untrusted advice, and therefore, we want to find an online algorithm that has two distinct properties [16]:

1. *Consistency.* The algorithm's performance improves if the quality of the prediction improves. If the prediction is perfect then the algorithm should perform close to optimal.
2. *Robustness.* If the prediction is very bad, or even adversarial, the algorithm still limits the resulting competitive ratio. This competitive ratio should not be much worse than the best known online algorithm.

We can analyze the algorithm in terms of how well it performs with respect to these properties. Unfortunately there are multiple definitions used for these parameters. Some compare ALG to an algorithm that fully trusts the advice, while some compare ALG to OPT. We define an algorithm to be (r, w)-*competitive* if its competitive ratio with perfect predictions is at most r and its competitive ratio with adversarial predictions is at most w. In Section 5, we will study the MULTIPLIER algorithm with machine learned advice, and we will compare ALG to OPT to determine both the consistency and robustness.

Since we now have two parameters by which to characterize an algorithm's performance we may have two algorithms of which none is strictly better than the other. Therefore, the ultimate objective is to find an algorithm that is pareto-optimal in both aspects, if one exists.

1.2.5 Randomized algorithms

Another type of online algorithm is a randomized online algorithm [23]–[25]. A randomized algorithm is non-deterministic and is not guaranteed to produce the same result when ran multiple times on the same input instance. Since the worst-case performance can be pretty bad, which is why we are actually interested in is the *expected cost* $E[ALG(I)]$.

A randomized algorithm is considered to be c-competitive if for any oblivious adversary $E[ALG(I)] \leq c \cdot OPT(I) + a$ holds [8]. This only proves the upper bound of a problem though, if we want to define the lower bound of a problem in the randomized setting we can make use of *Yao's principle* [26]. Yao's principle states that, for a maximization problem, the lower bound of a problem can be defined by the expected value of an optimal solution, divided by the expected value of the best-known deterministic algorithm. With this we do not need to come up with an adversarial instance.

This type of analysis is a bit more optimistic since we do not just look at the worst-case performance, but at the expected performance. If one is looking for a practical online algorithm without having the ability to train an advisor, as mentioned in the previous section, then randomized algorithms can be a suitable option.

1.3 Table of contents

We will first introduce the necessary preliminaries in Section 2. Then, we will look into the previous research on online busy time scheduling in Section 3. Of all previous research, the DOUBLER algorithm by Koehler et al. [4] is the most foundational to our research. Therefore, we will recap their competitive analysis in Section 3.4 before we present our contributions.

In Section 4, we will present an adaptive adversary for online busy time scheduling with infinite processors. This results in a new lower bound of 2 for eager algorithms, improving upon the previous lower bound of 1.618 for eager algorithms. Then, in Section 5, we will look at online busy time scheduling with advice and present a $(1 + \frac{4}{1+\lambda})$-consistent and

$(1 + \frac{4}{1-\lambda})$-robust algorithm, where λ is a parameter between 0 and 1 that represents the algorithms' trust in the advice.

In Section 6, we will present some additional results. More specifically, in Section 6.1, we will disprove the upper bound of 4 claimed by Fong et al. [27]. And, in Section 6.2, we will introduce a general CREDITING method for analyzing both the DOUBLER and OVERLAPPER algorithms from sections 3.4 and 6.1 respectively. We also use this method to introduce the SAVER algorithm and show that this has not lead to improvements on the upper bound yet. Finally, we will conclude and reflect upon our research in Section 7.

2 Definitions and notations

Problem definition. In this thesis, we will focus on the online busy time problem with unlimited processors. Each instance consists of a set of *jobs* J, where each job j has a corresponding *release time* r_j, a *processing time* p_j, and a *deadline* d_j. A schedule consists of a *starting time* s_j for each job, and it is valid if all jobs are scheduled such that $s_j + p_j \leq d_j$.

We define $\mathcal{T}$, the set of *active time intervals* for some given schedule, as $\bigcup_{j \in J} \{\ [s_j, s_j + p_j)\ \}$, where we always merge touching and overlapping intervals. For example, $\{\ [0,3)\ \} \bigcup \{\ [2,4)\ \}$ and $\{\ [0,3)\ \} \bigcup \{\ [3,4)\ \}$ both become $\{\ [0,4)\ \}$. We let $\mu(\mathcal{T})$, the *busy time* of a schedule, be the sum of the lengths of the active time intervals.

Definitions for algorithms. The *latest start time* of a job j is denoted as LST_j, and we define it as $d_j - p_j$. Let U_t denote all *unscheduled jobs*, that is, those jobs which are released but are not yet scheduled at time t. When the value of t is clear in a given context we will refer to U_t as U. Moreover, let T denote the *times in the problem instance*, which starts at 0 and ends at $\max_{j \in J}\{d_j + p_j\}$. We will use these definitions to define the algorithms in later sections.

Definitions for analysis. Given two jobs in a schedule, j_1 and j_2, we note that they overlap iff $s_{j_1} \leq s_{j_2} + p_{j_2} \wedge s_{j_2} \leq s_{j_1} + p_{j_1}$. The *interval graph of the schedule* is defined as a graph where each job has a corresponding vertex and there is an edge between their vertices iff their corresponding jobs overlap.

A *connected component* in a graph is a maximal set of vertices in the graph for which each pair of vertices have a path between them. We define a connected component in a schedule to be a connected component in the interval graph of the given schedule. Since ALG's solution may contain multiple connected components, we will denote the set of connected components as $\mathcal{C}$, where the i-th connected component in $\mathcal{C}$ will be denoted as C_i.

Let $J(C_i)$ denote the set of *jobs that are inside a connected component* C_i. And, let $S(C_i)$ denote the *start time* of a connected component C_i. By definition of connected components, this will be equal to $\min\{s_j : j \in J(C_i)\}$. Similarly, let $E(C_i)$ denote the *end time* of a connected component. By definition of connected components this will be equal to $\max\{s_j + p_j : j \in J(C_i)\}$.

Note that an instance may not have a single optimal solution. Let $\mathcal{O}$ be the set of all *optimal solutions* for a given instance. Each solution $O_n \in \mathcal{O}$ has the same busy time $\mu(O_n)$ and no better solution can be achieved.

▶ **Lemma 1** (Tardy optimal solutions). *There exists a tardy optimal solution O_n in $\mathcal{O}$ for which every connected component C_i in O_n has $S(C_i) = u_j = s_j$ for some job j.*

Proof: Let $f(C_i)$ denote whether $S(C_i) = u_j = s_j$ for some job j holds for C_i. Assume no such solution exists and pick the solution O_n which has the most number of connected

components C_i for which $f(C_i)$ holds. From this solution pick a connected component C_i for which $f(C_i)$ doesn't hold.

Let J' be all jobs j for which $S(C_i) = s_j$ holds. Calculate the minimum slack SL to be $\min\{ u_j - s_j : j \in J' \}$. Delay all jobs $j \in J'$ by this amount and call this new connected component C'_i. Now $S(C'_i) = S(C_i) + SL$. The end time $E(C'_i)$ is delayed by at most SL compared to $E(C_i)$. Thus $E(C'_i) - S(C'_i) \leq E(C_i) - S(C_i)$.

Thus, there should be a solution $O_{n'}$ that has one more connected component for which $f(C_i)$ holds than O_n. Since O_n was picked to have the maximum amount of C_i for which $f(C_i)$ holds this leads to a contradiction. ◀

Let O^* denote some *tardy optimal solution* for which every connected component C_i in O^* has $S(C_i) = u_j = s_j$ for some job j. In the rest of this thesis, whenever we refer to an optimal solution, we refer to a tardy optimal solution. Moreover, for all previously mentioned variables, we will use a '*' to denote that this this variable relates O^* in the rest of this thesis. For example, s^*_j is the starting time of job j in the optimal solution, and C^*_i is a connected component in the optimal solution.

3 Previous results

In this section we will list and describe the relative previous results on the online busy time problem. Table 1 shows the best previous results for the various busy time scheduling settings. Our contributions are marked in bold text.

Problem setting	**Remarks**	**Lower bound**	**Upper bound**
Flexible jobs,	-	1.618 [4], [28]	5 [4] ~~4~~ [27]
unlimited processors	With advice	-	$\mathbf{(1 + \frac{4}{1+\lambda}, 1 + \frac{4}{1-\lambda})}$
	Eager algorithms	**2**	-
Rigid jobs,	-	g [6]	5 log $p_{\max}$ [6]
limited processors	One-sided clique	2 [6]	$1 + \frac{1+\sqrt{5}}{2}$ [6]
	Clique	-	$2(1 + \frac{1+\sqrt{5}}{2})$ [6]
	Heterogeneous machines, non-clairvoyant	-	$O(\log \frac{p_{\max}}{p_{\min}})$ [29]
Flexible jobs, limited processors	-	-	$O(\log \frac{p_{\max}}{p_{\min}})$ [4]

Table 1 An overview of the current results

3.1 Flexible jobs, unlimited processors

In this problem setting, we consider only instances where the number of processors is unlimited. This eliminates the decision factor of deciding which machine a job should be scheduled on. Instead, any algorithm needs to solely focus on *when* a job should be scheduled.

In 2017 three seperate papers were published on this problem setting, by Koehler et al. [4], Fong et al. [27] and by Ren et al. [28]. The authors of these papers have presented upper bounds of 5, 4, and $2\sqrt{2} + 4$ respectively. For this thesis, the most relevant algorithm is the DOUBLER algorithm by Koehler et al., which we study in Section 3.4. According to Koehler et al., the analysis of the upper bound of 4 by Fong et al. is incorrect. However,

they do not prove this claim, so we will study the results by Fong et al. in Section 6.1 and disprove the claimed upper bound of 4 by Fong et al.

Interestingly, all three papers present online algorithms that are very similar. All of them wait until some unscheduled job reaches its latest possible starting time, at which point it is scheduled. Then this job is marked as a *primary*, or *flag* job. The algorithm by Koehler et al. then reserves an open window, relative to the processing time of the scheduled job, under which all jobs that fit in it are also scheduled. The other two algorithms define a minimum overlap ratio with the primary/flag job, for which all jobs that satisfy this condition are scheduled.

The current lower bounds by both Koehler et al. and Ren et al. are 1.618 so there's still some improvement possible, either by improving upon the lower bound, or by developing a better algorithm.

3.2 Rigid jobs, limited processors

In this problem setting, the number of processors we can use is limited, but we do have unlimited machines available. In contrast to the flexible jobs, unlimited processors setting, this setting only considers *where* each job should be scheduled.

In 2014, Shalom et al. first analyzed this setting [6]. They show that the problem has a lower bound of g and, since this is not a constant-competitive ratio, they also study some special problem instances. Moreover, they show a $5 \log p_{\max}$-competitive algorithm for any instance where the length of jobs does not grow exponentially with the number of processors. This is done by classifying jobs based on their processing time, and putting them into buckets. Each machine will then have a bucket assigned to it, and it will only process jobs out of that particular bucket. They also prove an upper bound of $1 + \frac{1+\sqrt{5}}{2}$ and a lower bound of 2 for one-sided clique instances, and an upper bound of $2(1 + \frac{1+\sqrt{5}}{2})$ for general clique instances. And, they show results for the throughput setting and extend their results to the previously mentioned context of optical network optimization. Other than considering special instances, or non-deterministic algorithms, their general results are unlikely to be improved upon, since they provide a tight analysis for their lower, and upper bounds.

In 2020, Ren et al. introduced a setting that closely resembles the problems faced in cloud computing [29]. They use a model in which machines may have different numbers of processors, and the machines may also have varying costs associated to them. Their algorithm is also non-clairvoyant, meaning the processing times are unknown until a job has finished running. The competitive ratio of their algorithm is $O(\log \frac{p_{\max}}{p_{\min}})$ and their analysis is tight.

3.3 Flexible jobs, limited processors

In this problem setting, we have both flexible jobs and a limited number of processors, with unlimited machines. Any algorithm should therefore decide both *when* and *where* jobs should be scheduled. Since any instance containing only rigid jobs is also a valid instance for this problem setting, the lower bound is also at least g, as shown by Shalom et at. [6].

As of now there exists only one paper with results on this problem by Koehler et al. [4]. Their goal was not only to minimize busy time, but also to offer a trade-off between the busy time and the number of machines used, for which they show a result that uses a mixing parameter α. They provide a $O(\log \frac{p_{\max}}{p_{\min}})$-competitive online algorithm by making use of the bucketing algorithm by Shalom et al. They estimate the true competitive ratio to be $9 \log \frac{p_{\max}}{p_{\min}}$ based on their lower bound of busy time scheduling with infinite processors.

Similar to the rigid job setting, one would need to consider special instances or non-deterministic algorithms if one wants to find an online algorithm that is constant-competitive. It is plausible that randomized algorithms or algorithms with machine learned advice could result in constant-competitive results.

3.4 The doubler algorithm

In this section we will recap and explain the DOUBLER algorithm by Koehler et al. [4]. Then, we will recap the analysis, since we will adapt some of the proofs in Section 5.

3.4.1 Algorithm description

The DOUBLER algorithm keeps track of a set P of *primary jobs*, which are jobs that are scheduled at the latest possible starting time, and they will determine how long the machine is busy. Each time a primary job is scheduled, DOUBLER activates the machine for twice the job's processing time p_j during which time it can also be busy processing other jobs. While the machine is busy, DOUBLER will schedule any job that is available and can be scheduled during this time. Since DOUBLER needs to schedule all jobs before their respective deadlines, any unscheduled job is scheduled once its latest possible starting time is reached. If this job does not fit in the current busy time, it will be marked as a primary job.

Algorithm 1 The DOUBLER algorithm

Let $P = \emptyset$
for all times $t \in T$ **do**
  Schedule every unscheduled job j for which $[t, t + p_j) \in \mathcal{T}$
  if $t = LST_j$ for some unscheduled job **then**
    Let j be some job with $t = LST_j$ and $p_j = \max\{p_{j'} | j' \in U \text{ and } t = LST_{j'}\}$
    $\mathcal{T} = \mathcal{T} \bigcup \{[t, t + 2p_j)\}$
    Schedule job j and add it to P, the set of primary jobs
    Schedule every unscheduled job j for which $[t, t + p_j) \in \mathcal{T}$
  end if
end for

In Algorithm 1 we show the precise flow of DOUBLER. Note that, the step where jobs are scheduled if they fit in the busy time is executed twice. This is necessary because some jobs may not fit in the busy time initially, while they do fit after a new primary job is scheduled. If one skips this step, then t is increased for the next iteration and the job may not fit anymore.

3.4.2 Analysis recapped

In this section, we will recap the DOUBLER analysis by Koehler et al. [4] and show that DOUBLER is 5-competitive. Koehler et al. also show that the analysis is tight, but for the purpose of this thesis we will not go into those details. The general idea is that one can split the total cost of the algorithm into the costs of the optimal and non-optimal busy time intervals. Then, one can analyze these parts per connected component in the optimal solution.

Remember that $\mathcal{T}^*$ denotes the busy time intervals in the optimal solution. Then, let P_1 be the set of all primary jobs which, including their open window, fully fit inside the

optimal solution. Formally, $P_1 = \{j \in P \mid [s_j, s_j + 2p_j) \subset \mathcal{T}^*\}$. Let P_2 be the set of all other primary jobs, formally, $P_2 = P \setminus P_1$.

Note that, in the DOUBLER algorithm, any job that is scheduled must be either primary, or it must fit inside the open window $[s'_j, s'_j + 2p'_j)$ of some primary job j'. Therefore $\mu(\mathcal{T}) = \mu\left(\bigcup_{j \in P} [s_j, s_j + 2p_j)\right)$. Let $\mu(P_1)$ denote $\mu\left(\bigcup_{j \in P_1} [s_j, s_j + 2p_j)\right)$ and $\mu(P_2)$ denote $\mu\left(\bigcup_{j \in P_2} [s_j, s_j + 2p_j)\right)$ for the rest of this analysis. Since P_1 fully fits inside the optimal solution by definition of P_1, it must be that $\mu(P_1) \leq \mu(\mathcal{T}^*)$. Combined with $\mu(\mathcal{T}) = \mu\left(\bigcup_{j \in P} [s_j, s_j + 2p_j)\right)$, it follows that $\mu(\mathcal{T}) \leq \mu(\mathcal{T}^*) + \mu(P_2)$. This means that, if one can find an upper bound on the cost of P_2 in terms of $\mu(\mathcal{T}^*)$, one can find an upperbound for the DOUBLER algorithm.

In order to analyze the cost of the algorithm in terms of connected components, one can partition the jobs in P_2 by the connected components that they originally belong to in OPT. Let P_2^i denote the jobs in P_2 that are in connected component C_i^* in the optimal solution. Recall that $\mathcal{C}^*$ is a set of connected components $C_i^* \ldots C_k^*$ in OPT, and that they do not overlap by definition. Therefore $\mu(P_2) \leq \sum_{C_i^* \in \mathcal{C}^*} \sum_{j \in P_2^i} 2p_j$. Now one needs to find a bound on the cost of $\sum_{j \in P_2^i} 2p_j$ in order to get a bound on the competitive ratio.

▶ **Definition 2** (P_2^i ordered and ρ_j)**.** *Let $\vec{P_2^i}$ be the ordered list of the jobs in P_2^i for any i in a non-descending order with respect to their starting times s_j. In order to avoid overly-complex notation we let ρ_j denote the j^{th} job in $\vec{P_2^i}$.*

▶ **Lemma 3** (Primary job availability)**.** *All primary jobs in $\vec{P_2^i}$ are released when the first job starts. $r_{\rho_j} \leq s_{\rho_1}$ for all ρ_j.*

Proof: By contradiction, assume that $r_{\rho_j} > s_{\rho_1}$ for some job ρ_j. Note that, the job ρ_j fits inside the optimal connected component $[s^*_{\rho_j}, s^*_{\rho_j} + p_{\rho_j}) \subseteq C_i^*$, by definition of connected components and OPT. Therefore $r_{\rho_j} + p_{\rho_j} \leq E(C_i^*)$. Otherwise, it would be impossible for ρ_j to fit in C_i^*. Since ρ_1 is a primary job that is not fully inside the optimal connected component by definition of P_2^i, it must be that $[s_{\rho_1}, s_{\rho_1} + 2p_{\rho_1}) \nsubseteq C_i^*$. Moreover, since this job is primary, it starts at its latest start time, and therefore it must be that the job finishes running in the algorithm's solution after the optimal connected component finishes running. Formally, $s_{\rho_1} + 2p_{\rho_1} \geq E(C_i^*)$. Otherwise, if this job starts at its latest start time, and the end of the job plus its window is before the end of the connected component, it would not be in P_2. Therefore, it must be that $s_{\rho_1} + 2p_{\rho_1} \geq E(C_i^*) \geq r_{\rho_j} + p_{\rho_j}$. Since by contradiction it is assumed that $r_{\rho_j} > s_{\rho_1}$, and $s_{\rho_1} + 2p_{\rho_1} \geq r_{\rho_j} + p_{\rho_j}$, job ρ_j fully fits inside the open window of job s_{ρ_1}. Therefore, ρ_j must have been scheduled as a non-primary job, but since $\rho_j \in P$ this is a contradiction. ◀

▶ **Lemma 4** (Primary job size growth)**.** *The processing time of primary jobs in $\vec{P_2^i}$ doubles, or more than doubles each job. $p_{\rho_j} > 2p_{\rho_{j-1}}$ for all $j \geq 2$.*

Proof: By contradiction, assume that $p_{\rho_j} \leq 2p_{\rho_{j-1}}$. Recall that job ρ_j was already released at $t = s_{\rho_{j-1}}$ per Lemma 3, and $s_{\rho_{j-1}} \geq s_{\rho_1} \geq r_{\rho_j}$. Therefore, job ρ_j must have been scheduled at time $s_{\rho_{j-1}}$ under the window of job ρ_{j-1}, which means it would not be primary. This is a contradiction since $\rho_j \in P_2$. ◀

▶ **Theorem 5.** *Doubler is at most 5-competitive*

Proof: Let $\rho_{\max}$ denote the processing time of the largest job in $\vec{P_2^i}$, and let $j_{\max}$ denote the index of this job. With Lemma 4 in mind one can now see that, by induction, $\sum_{j \in P_2^i} 2p_j \leq \sum_{j=1}^{j_{\max}} 2^{j-j_{\max}} \cdot \rho_{\max}$. And, by the definition of connected components we know that any job in a connected component can not be larger than the connected component, meaning one can replace $\rho_{\max}$ with $\mu(C_i^*)$. This means that $\sum_{j \in P_2^i} 2p_j \leq \mu(C_i^*) \cdot \sum_{j=0}^{\infty} 2^{-j} = 4\mu(C_i^*)$.

If one plugs this definition into $\mu(P_2) \leq \sum_{C_i^* \in \mathcal{C}^*} \sum_{j \in P_2^i} 2p_j$, one can deduce that $\mu(P_2) \leq 4 \cdot \sum_{C_i^* \in \mathcal{C}^*} \mu(C_i^*) = 4 \cdot \mu(\mathcal{T}^*)$. This means that $\mu(\mathcal{T}) \leq \mu(\mathcal{T}^*) + \mu\left(\bigcup_{j \in P_2} [s_j, s_j + 2p_j)\right) \leq 5 \cdot \mu(\mathcal{T}^*)$. And, since $OPT = \mu(\mathcal{T}^*)$ by definition, DOUBLER is at most 5-competitive. ◀

4 Lower bound of 2 for eager algorithms

In this section, we will present the proof for a lower bound of 2 for eager algorithms in online busy time scheduling with unlimited processors.

4.1 Preliminaries

In order limit the number of variations that we have to account for, we will start by defining the family of eager algorithms.

▶ **Definition 6** (Eager algorithms)**.** *Let the family of* eager algorithms *be the family of algorithms where any algorithm schedules jobs immediately if they don't increase the busy time. Formally, the any eager algorithm schedules all jobs $j \in U$ at time t for which $\mu([t, t + p_j) \bigcup \mathcal{T}) = \mu(\mathcal{T})$.*

▶ **Lemma 7** (Optimality of eager algorithms)**.** *Any non-eager algorithm's schedule can be converted into a schedule produced by an eager algorithm with at most the same busy time.*

Proof: Let S denote some schedule which was produced by a non-eager algorithm on some instance. Let S' denote the revised, eager schedule, which starts out as a copy of S. Then, starting from the earliest job, we change the starting time of all jobs which at some point could have been scheduled without increasing busy time. The new starting time of such a job becomes the first point in time where scheduling the job does not increase busy time. And, because the busy times of our machine might change, we repeat this process until no such jobs can be found.

Since changing a jobs starting time is the same as removing and adding a job to the schedule, we can say that performing this operation will change the cost of S' from $\mu(S)$ to $\mu(S) + remove + add$. Scheduling a job eagerly, by definition, does not increase busy time. This means that $add = 0$. Note that, removing a job from a schedule can never increase busy time, meaning $remove \leq 0$. Since $add = 0$ and $remove \leq 0$, it must be that $\mu(S') \leq \mu(S)$. ◀

4.2 Definition

The adversary in *Algorithm 2* will release n^2 jobs j at the very start, with processing times that increase linearly by $\frac{j}{n}$. The deadlines also increase exponentially, namely n^{j+2}, such

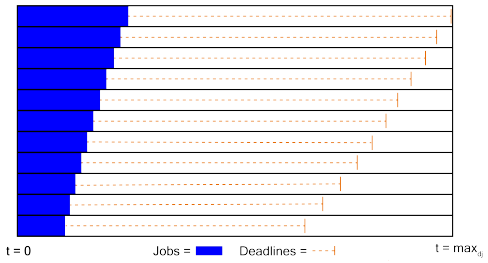

Figure 1 Initial jobs

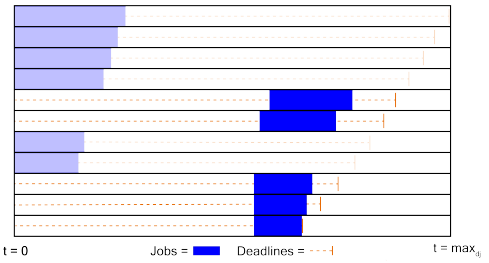

Figure 2 Not possible since ALG must be eager

that spreading all jobs never spans two deadlines.

Once the algorithm, which we require to be eager, decides to schedule some set of jobs, the adversary waits until all of those jobs have finished running. Let α denote some parameter that we will later use to maximize the lower bound. Then, if the ratio between the largest and smallest jobs in the previous connected component is α or greater, the adversary releases copies of all jobs but the smallest. We use $\mathcal{D}_1$ to classify the set of connected components for which the ratio was α or greater. The set $\mathcal{D}_2$ classifies the set of jobs for which this is not the case.

Figure 1 shows the initial setup for some small instance at $t = 0$. Figure 2 shows an impossible set of scheduled jobs for this instance, since we require the algorithm to be eager. Figures 3 and 4 show examples where some algorithm scheduled connected components that will be marked as $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. The deadlines in these figures are not at true scale.

Algorithm 2 Adversarial instance for $g = \infty$

Require: $ALG \in A$.
  Release n^2 jobs with $r_j = 0$, $p_j = 1 + \frac{j}{n}$, $d_j = n^{j+2}$
  $\mathcal{D}_1 \leftarrow \emptyset$, $\mathcal{D}_2 \leftarrow \emptyset$
  while $t < \max_{j \in J} d_j$ **do**
    if $t - \epsilon \in \mathcal{T}$ and $t \notin \mathcal{T}$ **then** ▷ Wait until all jobs have finished running
      $C_i \leftarrow$ the latest connected component of ALG
      $\max_{C_i} \leftarrow \max_{j \in C_i} p_j$ ▷ $\max_{C_i}$ is the longest job processing time in C_i
      $\min_{C_i} \leftarrow \min_{j \in C_i} p_j$ ▷ $\min_{C_i}$ is the shortest job processing time in C_i
      if $\max_{C_i} > \alpha \ \min_{C_i}$ **then**
        for all $j \in C_i \mid p_j > \min_{C_i}$ **do**
          Release a new job j' with $r_{j'} = t$, $p_{j'} = p_j$ and $d_{j'} = d_j$
        end for
        Add C_i to $\mathcal{D}_1$
      else:
        Add C_i to $\mathcal{D}_2$
      end if
    end if
  end while

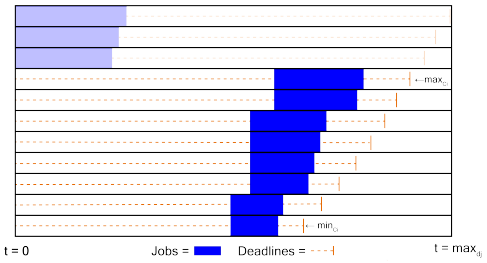

Figure 3 Connected component in $\mathcal{D}_1$

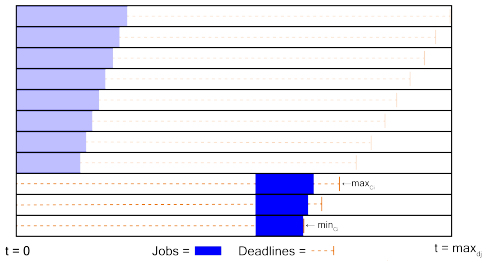

Figure 4 Connected component in $\mathcal{D}_2$

4.3 Analysis

▶ **Lemma 8** (Cost of ALG). *The cost of ALG is bounded by the cost of each connected component in $\mathcal{D}_1$ and $\mathcal{D}_2$. That is, $ALG \geq \sum_{C_i \in \mathcal{D}_1} \max_{C_i} + \sum_{C_i \in \mathcal{D}_2} \max_{C_i}$*

Proof: Each of the connected components $C_i \in \mathcal{C}$ has a cost of $\mu(C_i) \geq \max_{C_i}$, since we can never make the busy time of a connected component smaller than the processing time of its largest job. Each connected component is included in either $\mathcal{D}_1$ or $\mathcal{D}_2$, but never in both. Therefore we can partition the cost of ALG into $\sum_{C_i \in \mathcal{D}_1} \max_{C_i}$ and $\sum_{C_i \in \mathcal{D}_2} \max_{C_i}$. ◀

▶ **Lemma 9** (Cost of OPT). *The cost of OPT is bounded by the maximum job size and the connected components in $\mathcal{D}_1$. That is, $OPT \leq n + 1 \sum_{C_i \in \mathcal{D}_1} \min_{C_i}$*

Proof: Once the jobs in some connected component C_i have finished processing, we know whether C_i will be in $\mathcal{D}_1$ or $\mathcal{D}_2$. Let OFF denote some algorithm that can change its decisions offline. By providing the decisions that OFF could make we can determine an upper bound on the cost of OPT.

If $\mathcal{D}_1 = \emptyset$, which it always is initially, OFF could schedule all jobs at the same time with a busy time of $n + 1$, since no additional jobs are released yet. As new jobs are released, due to some connected component $C_i \in \mathcal{D}_1$, OFF can change its decisions by delaying some previously scheduled jobs.

When ALG schedules a connected component C_i such that $C_i \in \mathcal{D}_1$, the adversary will release additional jobs with processing times $\min_{C_i} + \frac{1}{n}$ up to $\max_{C_i}$ after the jobs in C_i have finished running. Since these newly released jobs have a release time greater than zero, OFF can no longer schedule all jobs at the same time. Instead, OFF can keep the starting times for all jobs with processing times up to $\min_{C_i}$ the same, and set the startings times for all jobs with processing times greater than $\min_{C_i}$ to the release time of the newly released jobs after C_i. Figures 5 and 6 illustrate this scenario. For illustrative purposes, the deadlines are not at true scale.

Each time OFF changes its decisions offline, due to some $C_i \in \mathcal{D}_1$, the cost of the final connected component in OFF will be $n + 1$, since this is the maximum job size and we can schedule all jobs at the same time. And, since OFF may not be able to schedule the job with $p_j = \min_{C_i}$ at the same time as the final connected component, the cost of scheduling this job is at most $\min_{C_i}$. Therefore the total cost of OFF is at most $n + 1 + \sum_{C_i \in \mathcal{D}_1} \min_{C_i}$. And, since the solution of OPT will have the lowest possible busy time, by definition of an optimal solution, we can say that $OPT \leq OFF \leq n + 1 + \sum_{C_i \in \mathcal{D}_1} \min_{C_i}$.

◀

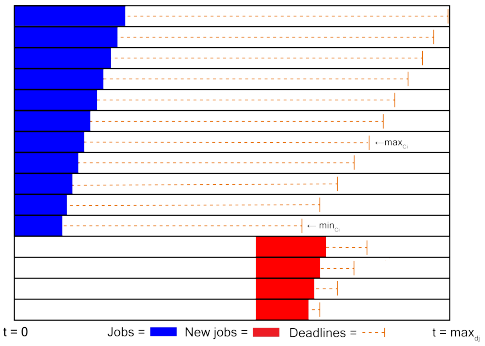

Figure 5 Initial solution OFF with new jobs

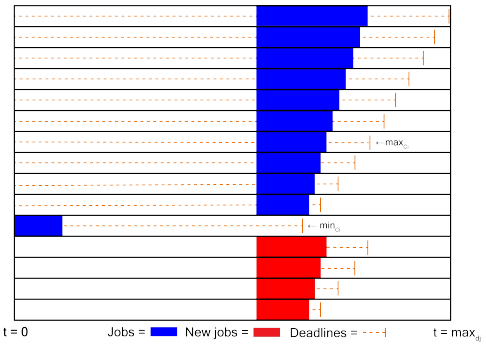

Figure 6 Revised solution OFF after $C_i \in \mathcal{D}_1$

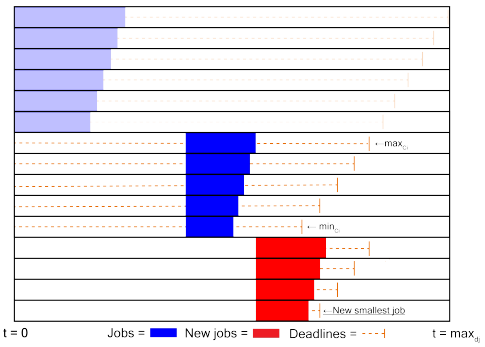

Figure 7 New smallest job after $\mathcal{D}_1$

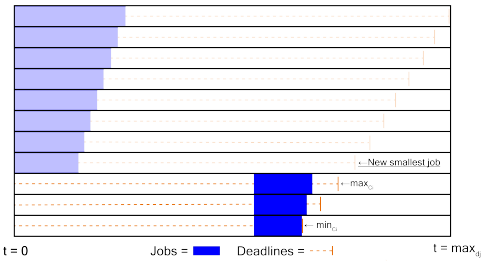

Figure 8 New smallest job after $\mathcal{D}_2$

▶ **Lemma 10** (Charged cost of OPT). *We can charge the n-term in the upperbound of the cost of OPT to $\mathcal{D}_1$ and $\mathcal{D}_2$ such that* $n = \sum_{C_i \in \mathcal{D}_1} \left(\min_{C_i} + \frac{1}{n}\right) + \sum_{C_i \in \mathcal{D}_2} \left(\max_{C_i} - \min_{C_i} + \frac{1}{n}\right)$.

Let f_i denote the smallest processing time of the jobs in U, right after connected component C_i has finished running and, if applicable, the adversary has released any new jobs. We define f_0 to be 1, because before any job has been scheduled, the smallest unscheduled job is the smallest possible job, which has a processing time of 1. When U is empty after k connected components (meaning all jobs have been scheduled), we define f_k to be $n+1$, which is the largest job size.

Let Δf_i be $f_i - f_{i-1}$. Note that f_i is a monotonic function, since we require ALG to be in the eager family of algorithms, and the adversary never releases smaller jobs than $\min_{C_i}$ after C_i has finished running. Therefore, since $f_0 = 1$ and the last term $f_k = n+1$, we know that $\sum_{i=0}^{k} \Delta f_i = n$. Moreover, since $\mathcal{D}_1$ and $\mathcal{D}_2$ are disjoint, we can partition $\sum_{i=0}^{k} \Delta f_i$ into $\sum_{C_i \in \mathcal{D}_1} \Delta f_i$ and $\sum_{C_i \in \mathcal{D}_2} \Delta f_i$.

Next we will determine the value of Δf_i via case distinction.

- If $C_i \in D_1$, then the algorithm schedules all jobs with processing times $\min_{C_i}$ to processing times $\max_{C_i}$. Afterwards, the adversary releases new jobs with processing times $\min_{C_i} + \frac{1}{n}$ up to $\max_{C_i}$. Therefore, $\Delta f_i = \min_{C_i} + \frac{1}{n} - \min_{C_i} = \frac{1}{n}$. Figure 7 illustrates this scenario.
- If $C_i \in D_2$, then the algorithm also schedules all jobs with processing times $\min_{C_i}$ to processing times $\max_{C_i}$. Afterwards, the adversary doesn't release new jobs as part of this decision. Therefore, $\Delta f_i = \max_{C_i} + \frac{1}{n} - \min_{C_i}$. Figure 8 illustrates this scenario.

Since $n = \sum_{C_i \in \mathcal{D}_1} \Delta f_i + \sum_{C_i \in \mathcal{D}_2} \Delta f_i$, we can now say that $n = \sum_{C_i \in \mathcal{D}_1} \left(\min_{C_i} + \frac{1}{n}\right) + \sum_{C_i \in \mathcal{D}_2} \left(\max_{C_i} - \min_{C_i} + \frac{1}{n}\right)$ by filling in the costs of Δf_i. ◀

▶ **Lemma 11** (Last connected component cost). *The last connected component, C_k, is in $\mathcal{D}_2$ and it will have a busy time of $\max_{j \in J} p_j = n + 1$.*

Proof: The largest job $j \in J$ has size $1 + n(\frac{1}{n}) = n + 1$. At some point, *ALG* will have to schedule this job or the produced schedule is infeasible. Once this job is scheduled and finished processing one of two things will happen:

1. The job j is added to $\mathcal{D}_1$, and the algorithm will schedule more jobs.
2. The job j is added to $\mathcal{D}_2$, and no jobs will be released anymore.

If $C_k \in \mathcal{D}_2$, then $U = \emptyset$, since we require *ALG* to be eager. Therefore, all smaller jobs must have been scheduled as well. And, by definition of $\mathcal{D}_2$, no more jobs will be released, meaning this is a possible way to finish scheduling jobs, while preventing the adversary from releasing more jobs.

If, however, $C_k \in \mathcal{D}_1$, then the algorithm will release jobs with processing times between $\min_{C_k} + \frac{1}{n}$ and $\max_{C_k}$. Due to the definition of these newly released jobs, it is only possible for U to be empty afterwards if $\max_{C_k} = \min_{C_k}$. If $\alpha > 1$ this cannot be the case for any $C_i \in \mathcal{D}_1$ since $\max_{C_i} \geq \alpha \ \min_{C_i}$. Therefore, it is impossible to finish scheduling jobs with a connected component $C_k \in \mathcal{D}_1$, while preventing the adversary from releasing more jobs. ◀

▶ **Lemma 12** (Total cost of $\mathcal{D}_2$). $\sum_{C_i \in \mathcal{D}_2} \max_{C_i} \geq n + |\mathcal{D}_2| + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n}$.

Proof: As per *Lemma 11*, one $C_i \in \mathcal{D}_2$ will have $\max_{C_i} = n + 1$. Since the minimum job size is 1 and all other jobs are at least $\frac{1}{n}$ larger, the other $i = [1 \dots |\mathcal{D}_2| - 1]$ connected components in $\mathcal{D}_2$ will each increase the total busy time of *ALG* by at least $1 + \frac{i-1}{n}$.

Therefore, these jobs will be scheduled with a busy time of at least $\sum_{i=1}^{|\mathcal{D}_2|-1} \left(1 + \frac{i-1}{n}\right) = |\mathcal{D}_2| - 1 + \sum_{i=1}^{|\mathcal{D}_2|-1} \left(\frac{i-1}{n}\right) = |\mathcal{D}_2| - 1 + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n}$.

Moreover, when we add the largest job the required busy time becomes at least $|\mathcal{D}_2| - 1 + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n} + n + 1 = n + |\mathcal{D}_2| + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n}$. ◀

▶ **Lemma 13** (Contribution of $\frac{1}{n}$). *As n grows to infinity, the contribution of $\frac{1}{n}$ for each connected component in $\mathcal{D}_2$ in the lower bound of the cost of OPT to the competitive ratio becomes 0. Formally, $\lim_{n \to \infty} \frac{\sum_{C_i \in \mathcal{D}_2} \frac{1}{n}}{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}} = 0$.*

Proof: As per *Lemma 12*, we know that $\frac{\sum_{C_i \in \mathcal{D}_2} \frac{1}{n}}{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}} = \frac{\frac{|\mathcal{D}_2|}{n}}{n + |\mathcal{D}_2| + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n}}$. If we increase n to infinity while $|\mathcal{D}_2|$ is very small then $\frac{|\mathcal{D}_2|}{n}$ approaches 0, and therefore the whole fraction becomes 0. However, $|\mathcal{D}_2|$ could be as large as n^2 if all $C_i \in \mathcal{D}_\in$ contain only a single job, which means $\frac{|\mathcal{D}_2|}{n}$ can be as large as n.

When $|\mathcal{D}_2|$ increases by one, $\sum_{C_i \in \mathcal{D}_2} \frac{1}{n}$ is increased by $\frac{1}{n}$, while $n + |\mathcal{D}_2| + \frac{|\mathcal{D}_2|^2 - 3|\mathcal{D}_2| + 2}{2n}$ is increased by more than one. Therefore, even if $|\mathcal{D}_2|$ increases by a lot, the term $\frac{|\mathcal{D}_2|}{n}$ is only increased by less than $\frac{1}{n}$-th of the same amount. Meaning as n grows to infinity the fraction approaches 0, no matter how large $|D_2|$ is. ◀

▶ **Theorem 14.** *No eager online algorithm can achieve better a competitive ratio better than 2*

Proof: We will prove this by looking at the ratio between ALG and OPT and then take the minimum ratio of the fractions for $\mathcal{D}_1$ and $\mathcal{D}_2$ separately. $\frac{ALG}{OPT} \geq$
$\frac{\sum_{C_i \in \mathcal{D}_1} \max_{C_i} + \sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_1} \left(\min_{C_i} + \frac{1}{n}\right) + \sum_{C_i \in \mathcal{D}_2} \left(\max_{C_i} - \min_{C_i} + \frac{1}{n}\right)} \geq \min \left\{ \frac{\sum_{C_i \in \mathcal{D}_1} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_1} \left(\min_{C_i} + \frac{1}{n}\right)}, \frac{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_2} \left(\max_{C_i} - min_{C_i} + \frac{1}{n}\right)} \right\}$

For the first term, $\max_{C_i} > \alpha \ \min_{C_i}$ for any $C_i \in \mathcal{D}_1$, and our jobs increase in size by exactly $\frac{1}{n}$. Therefore, $\frac{\sum_{C_i \in \mathcal{D}_1} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_1} \left(\min_{C_i} + \frac{1}{n}\right)} \geq \alpha$. Which means that the ratio of the first term is α.

For the second term, $\max_{C_i} \leq \alpha \ \min_{C_i}$ since $C_i \in \mathcal{D}_\in$ and therefore $\max_{C_i} - \min_{C_i} \leq (1 - \frac{1}{\alpha}) \max_{C_i}$. Moreover, in Lemma 13 we have shown that $\frac{1}{n}$ is insignificant if n is large enough. Therefore, we can rewrite the ratio of the second term as $\frac{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_2}^{k} \left(\max_{C_i} - \min_{C_i}\right)}$. Combining $\max_{C_i} - \min_{C_i} \leq (1 - \frac{1}{\alpha}) \max_{C_i}$ with $\frac{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_2}^{k} \left(\max_{C_i} - \min_{C_i}\right)}$ gives us $\frac{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_2}^{k} \left(\max_{C_i} - \min_{C_i}\right)} \geq$
$\frac{\sum_{C_i \in \mathcal{D}_2} \max_{C_i}}{\sum_{C_i \in \mathcal{D}_2} (1 - \frac{1}{\alpha}) \max_{C_i}}$. This results in a ratio of $\frac{\alpha}{\alpha - 1}$ for the second term.

To find the competitive ratio, we need to find the best α to maximize $\frac{ALG}{OPT} \geq \min \left\{ \alpha, \frac{\alpha}{1-\alpha} \right\}$, which is maximized at $\alpha = 2$. Therefore, $\frac{ALG}{OPT} \geq 2$ if $\alpha = 2$, showing this is a lower bound for busy time scheduling with infinite processors. ◀

5 Busy time scheduling with machine learned advice

In this section we will introduce a $(1 + \frac{4}{1+\lambda})$-consistent and $(1 + \frac{4}{1-\lambda})$-robust online algorithm for online busy time scheduling with machine learned advice.

5.1 Preliminaries

In this section, we will not use the widely-used definition of robustness and consistency, that uses an error-measure. Instead, we will apply a more general definition, first introduced by Angeloupolos et al. [16]. The ultimate objective function, if using machine learning, is then to optimize the competitive ratio for an instance, instead of optimizing for some pre-defined error measure.

▶ **Definition 15** (Consistency). *We define the* consistency *of algorithms with machine learned advice to be the worst-case competitive ratio given that the advice is as good as possible. Formally, Consistency*$(ALG) := \sup_I \inf_{\hat{S}} \frac{ALG(I, \hat{S})}{OPT(I)}$

▶ **Definition 16** (Robustness). *We define the* robustness *of algorithms with machine learned advice to be the worst-case competitive ratio, irrespective of the advice. Formally, Robustness*$(ALG) :=$
$\sup_I \sup_{\hat{S}} \frac{ALG(I, \hat{S})}{OPT(I)}$

Commonly, the advice can be given in the form of a prediction, where the prediction should be as close to the problem instance as possible. Instead of such a prediction we require the advice to be a series of time intervals. The intuition is that these time intervals advice the algorithm on when the machine should be busy processing jobs. Note that, if one has a prediction on what jobs will arrive, then this prediction can be converted into some advice in the format of a series of time intervals.

5.2 The multiplier algorithm

The MULTIPLIER algorithm is very similar to the DOUBLER algorithm by Koehler et al. [4], described in Section 3.4, but instead of always doubling the algorithm will reserve a variable window with a relative size w_j for each primary job. In order to limit the size of this window we require that w_j is between $\alpha = \frac{1-\lambda}{2}$ and $\beta = \frac{1+\lambda}{2}$. Here, λ is the trust parameter which is between 0 and 1. Semantically, a λ of 1 means the algorithm fully trusts the advice, whereas a λ of 0 means the algorithm completely ignores the advice.

Algorithm 3 The multiplier algorithm with advice

Let $P = \emptyset$
for all times $t \in T$ **do**
  Schedule every unscheduled job j for which $[t, t + p_j) \in \mathcal{T}$
  if $t = LST_j$ for some unscheduled job **then**
    Let j be some job with $t = LST_j$ and $p_j = \max\{p_{j'} | j' \in U \text{ and } t = LST_{j'}\}$
    Let $\hat{w_j} = \max_{t' \in T} \{t' - t\}$ s.t. $[t, t') \subseteq \hat{S}$
    Let $w_j = \max\{ \frac{1}{\beta}, \min\{ \frac{1}{\alpha}, \frac{\hat{w_j}}{p_j} \} \}$
    $\mathcal{T} = \mathcal{T} \bigcup \{[t, t + w_j \cdot p_j)\}$
    Schedule job j and add it to P, the set of primary jobs
    Schedule every unscheduled job j for which $[t, t + p_j) \in \mathcal{T}$
  end if
end for

We require the advice, $\hat{S}$, to be a set of time intervals. Then, once a primary job is scheduled, the algorithm computes the advised busy time which is defined as $\hat{w_j} = \max_{t' \in T} \{t' - t\}$ s.t. $[t, t') \subseteq \hat{S}$. From this, the algorithm can determine the relative open window by clamping it between β and α. More formally, $w_j = \max\{ \frac{1}{\beta}, \min\{ \frac{1}{\alpha}, \frac{\hat{w_j}}{p_j} \} \}$. The algorithm then activates the machine for $w_j \cdot p_j$ time during which other jobs can be scheduled. In Figures 9-12 we illustrate how w_j is determined and in Algorithm 3 we show the precise flow of the MULTIPLIER algorithm.

5.3 Analysis

We will start out largely following the analysis of Koehler et al. [4], which we recap in Section 3.4. The difference between the start of our analysis and the analysis of Khuller et al is that, instead of a constant relative window, each job may have a different relative window w_j in our algorithm.

First, we redefine $\mathcal{T}$, P_1 and P_2. Let $\mathcal{T} = \bigcup_{j \in P} [s_j, s_j + w_j \cdot p_j)$, since jobs are either scheduled as primary jobs, or their processing time fully fits inside the window of some primary job. Let $\mathcal{T}^*$ denote the busy time intervals in the optimal solution. Then, let P_1 be the set of all primary jobs which, including their open window, fully fit inside the optimal

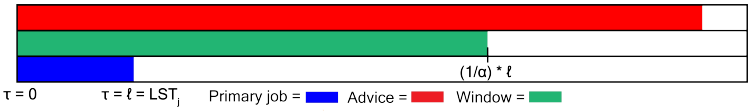

Figure 9 Example 1

Figure 10 Example 2

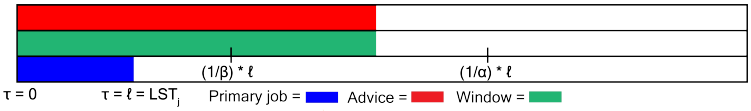

Figure 11 Example 3

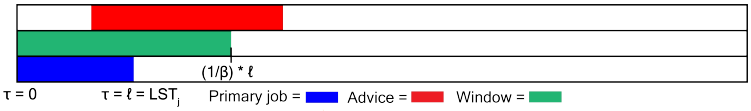

Figure 12 Example 4

Examples for determining the size of the open window

solution. Formally, $P_1 = \{j \in P \mid [s_j, s_j + w_j \cdot p_j) \subset \mathcal{T}^*\}$. Let P_2 be the set of all other primary jobs. Formally, $P_2 = P \setminus P_1$.

Then, by definition of $\mathcal{T}$, $\mu(\mathcal{T}) \leq \sum_{j \in P_1} w_j \cdot p_j + \sum_{j \in P_2} w_j \cdot p_j \leq \mu(\mathcal{T}^*) + \sum_{j \in P_2} w_j \cdot p_j$. Therefore if we can find a bound on $\sum_{j \in P_2} w_j \cdot p_j$, we can find a bound on the competitive ratio of our algorithm.

Note that, since any connected component is a maximal set in the interval graph, connected components never overlap. Therefore $\mathcal{T}^* = \bigcup_{C_i^* \in \mathcal{C}^*} C_i^*$. This means that we can express the cost of OPT as the sum of the lengths of the connected components.

In order to get a bound on the cost of P_2, we want to do something similar to analyze P_2 per connected component. We denote P_2^i as the set of jobs $J(C_i^*) \bigcap P_2$. Consequently, $\mu(P_2) \leq \sum_{C_i^* \in \mathcal{C}^*} \sum_{j \in P_2^i} w_j \cdot p_j$.

Let $\vec{P_2^i}$ be the jobs in P_2^i for any i in a non-descending order with respect to their starting times s_j. In order to avoid overly-complex notation we let ρ_j denote the jth job in $\vec{P_2^i}$.

▶ **Lemma 17** (Primary job availability). *All primary jobs are released when the first job starts. Formally, $r_{\rho_j} \leq s_{\rho_1}$ for all ρ_j.*

Proof: For this lemma we can look at the proofs in Lemma 3. When replacing 2 by w_j, for every job j, the proof still holds, since we use a very similar definition of P_1 and P_2.

By contradiction, assume that $r_{\rho_j} > s_{\rho_1}$ for some job ρ_j. Note that, by definition of connected components and OPT, the job ρ_j fits inside the optimal connected component $[s^*_{\rho_j}, s^*_{\rho_j} + p_{\rho_j}) \subseteq C_i^*$. Therefore, $r_{\rho_j} + p_{\rho_j} \leq E(C_i^*)$. Otherwise, it would be impossible for ρ_j to fit in C_i^*. Since ρ_1 is a primary job that is not fully inside the optimal connected component by definition of P_2^i, it must be that $[s_{\rho_1}, s_{\rho_1} + w_{\rho_1} \cdot p_{\rho_1}) \nsubseteq C_i^*$. Moreover, since this job is primary it starts at its latest start time, and therefore it must be that the job finishes running in the algorithm's solution after the optimal connected component finishes running. Formally, $s_{\rho_1} + w_{\rho_1} \cdot p_{\rho_1} \geq E(C_i^*)$. Otherwise, if this job starts at its latest start time, and, the end of the job plus its window is before the end of the connected component, it would not be in P_2. Therefore, it must be that $s_{\rho_1} + w_{\rho_1} \cdot p_{\rho_1} \geq E(C_i^*) \geq r_{\rho_j} + p_{\rho_j}$. Since, by contradiction, it is assumed that $r_{\rho_j} > s_{\rho_1}$, and $s_{\rho_1} + w_{\rho_1} \cdot p_{\rho_1} \geq r_{\rho_j} + p_{\rho_j}$, job ρ_j fully fits inside the open window of job ρ_1. And therefore ρ_j must have been scheduled as a non-primary job, but since $\rho_j \in P$ this is a contradiction. ◀

▶ **Lemma 18** (Primary job size growth). *The processing time of each primary job increases by at least a factor of their respective windows. Formally, $p_{\rho_j} > w_{\rho_{j-1}} \cdot p_{\rho_{j-1}}$ for all $j \geq 2$.*

Proof: For this lemma we can look at the proofs in Lemma 4. Though, instead of assuming $p_{\rho_j} \leq 2p_{\rho_{j-1}}$, we assume that, by contradiction, $p_{\rho_j} \leq w_{\rho_{j-1}} \cdot p_{\rho_{j-1}}$.

Recall that job ρ_j was already released at $t = s_{\rho_{j-1}}$ per Lemma 17, and $s_{\rho_{j-1}} \geq s_{\rho_1} \geq r_{\rho_j}$. Therefore job ρ_j must have been scheduled at time $s_{\rho_{j-1}}$ under the window of job ρ_{j-1}, which means it would not be primary. This is a contradiction since $\rho_j \in P_2$. ◀

▶ **Observation 19** (Clamped window size). *w_j is always between $\frac{1}{\beta}$ and $\frac{1}{\alpha}$ since it is defined as* $\max\{\ \frac{1}{\beta}, \min\{\ \frac{1}{\alpha}, \frac{\hat{w_j}}{p_j}\ \}\ \}$.

By the definition of connected components, Observation 19 and Lemma 18, we note that $\sum_{j \in P_2^i} w_j \cdot p_j = \sum_{j \in \rho} w_j \cdot p_j$ has the following properties:

1. $p_{\rho_j} \leq \mu(C_i^*)$ for all j

2. $\frac{1}{\beta} \leq w_{\rho_j} \leq \frac{1}{\alpha}$ for all j
3. $p_{\rho_j} > w_{\rho_{j-1}} \cdot p_{\rho_{j-1}}$ for all $j \geq 2$

▶ **Lemma 20** (A bound on P_2^i). $\sum_{j \in P_2^i} w_j \cdot p_j \leq w_{\rho_{\max}} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{j=1}^{j_{\max}-1} \frac{1}{w_{\rho_{j+1}} \cdot w_{\rho_{j+2}} \cdots w_{\rho_{j_{\max}-1}}}$

Let the last job in P_2^i be denoted as $\rho_{\max}$ and the index of this job be denoted as $j_{\max}$. If we look at this last job, we can state that $p_{\rho_{\max}} \leq \mu(C_i^*)$ because of property 1, and therefore $\sum_{j \in \rho} w_j \cdot p_j \leq w_{\rho_{\max}} \cdot \mu(C_i^*) + \sum_{j=1}^{j_{\max}-1} w_j \cdot p_j$.

Because of property 3 we can say that $w_{\rho_j} \cdot p_{\rho_j} < \mu(C_i^*)$ for all $j \leq j_{\max} - 1$, no matter what the value of w_{ρ_j} actually is. Then, by applying property 2, we can rewrite this as $w_{\rho_j} \cdot w_{\rho_{j-1}} \cdot p_{\rho_{j-1}} < w_{\rho_j} \cdot p_{\rho_j} < \mu(C_i^*)$ if $j \geq 2$. Which means that $w_{\rho_{j-1}} \cdot p_{\rho_{j-1}} \leq \frac{\mu(C_i^*)}{w_{\rho_j}}$. Repeating the same logic for $j-2$ we get $w_{\rho_{j-1}} \cdot w_{\rho_{j-2}} \cdot p_{\rho_{j-2}} < w_{\rho_{j-1}} \cdot p_{\rho_{j-1}} < \frac{\mu(C_i^*)}{w_{\rho_j}}$. Therefore, $w_{\rho_{j-2}} \cdot p_{\rho_{j-2}} \leq \frac{\mu(C_i^*)}{w_{\rho_j} \cdot w_{\rho_{j-1}}}$, which we can repeat for all jobs in P_2^i to generalize to $w_{\rho_j} \cdot p_{\rho_j} \leq \mu(C_i^*) \cdot \frac{1}{w_{\rho_{j+1}} \cdot w_{\rho_{j+2}} \cdots w_{\rho_{j_{\max}-1}}}$ for all $j < j_{\max}$.

And therefore, $\sum_{j=1}^{j_{\max}} w_j \cdot p_j \leq w_{\rho_{\max}} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{j=1}^{j_{\max}-1} \frac{1}{w_{\rho_{j+1}} \cdot w_{\rho_{j+2}} \cdots w_{\rho_{j_{\max}-1}}}$ ◀

With this bound on the competitive ratio we can further analyze MULTIPLIER's consistency and robustness.

5.4 Robustness

Note the definition of robustness, we want to show that, no matter the instance and advice, we will always have a competitive ratio compared to OPT of at most the robustness.

Looking at the bound on P_2^i in Lemma 20 we see that there are two terms, one on the cost of the last job in ρ and one term on the cost of the other jobs. $w_{\rho_{\max}} \cdot \mu(C_i^*)$ is maximized when $w_{\rho_{\max}}$ is maximal, which is $\frac{1}{\alpha}$. Therefore, $w_{\rho_{\max}} \cdot \mu(C_i^*) \leq \frac{1}{\alpha} \cdot \mu(C_i^*)$.

If we look at the second term on the bound on P_2^i in Lemma 20, we see that the overall term is maximized when each w_{ρ_j} is minimized. Therefore, since $w_{\rho_j} \geq \frac{1}{\beta}$, we know that $\mu(C_i^*) \cdot \sum_{j=1}^{j_{\max}-1} \frac{1}{w_{\rho_{j+1}} \cdot w_{\rho_{j+2}} \cdots w_{\rho_{j_{\max}-1}}} \leq \mu(C_i^*) \cdot \sum_{i=0}^{j_{\max}-1} \beta^i$. And, since there can be arbitrarily many jobs, we can rewrite this to $\mu(C_i^*) \cdot \sum_{n=0}^{-\infty} \beta^n$.

With this, we have an upper bound on the cost of P_2, $\mu(P_2) \leq \sum_{C_i^* \in \mathcal{C}^*} \sum_{j \in P_2^i} w_j \cdot p_j \leq \sum_{C_i^* \in \mathcal{C}^*} \frac{1}{\alpha} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{n=0}^{-\infty} \frac{1}{\beta}^n$. Recall that the total cost of ALG_λ is bounded by $\mu(P_1) + \mu(P_2)$, therefore the total cost of ALG_λ is at most $\sum_{C_i^* \in \mathcal{C}^*} \mu(C_i^*) + \frac{1}{\alpha} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{n=0}^{-\infty} \frac{1}{\beta}^n$. When we divide the total cost by $OPT = \sum_{C_i^* \in \mathcal{C}^*} \mu(C_i^*)$ we get a competitive ratio of $1 + \frac{1}{\alpha} + \sum_{n=0}^{-\infty} \frac{1}{\beta}^n$. Which, by definition of α and β is equal to $1 + \frac{4}{1-\lambda}$ for all $\lambda < 1$.

▶ **Corollary 21.** *ALG_λ is $1 + \frac{4}{1-\lambda}$-robust for all $\lambda < 1$.*

5.5 Consistency ($\lambda < 1$)

In order to prove the consistency of our algorithm we will need some way to define the best possible advice. Instead of reasoning what the best possible advice would be, we will start out by providing an advisor $ADV(I) \to \hat{S}$. By showing a worst-case competitive ratio using this advisor we can provide a bound on the consistency of our algorithm.

Let $ALG_\lambda(I, \hat{S}, t)$ denote the schedule of running ALG_λ on instance I, with advice $\hat{S}$ until time t. From the schedule resulting from ALG_λ we can deduce P and U at time t. Similarly let $ADV_\lambda(I, t)$ denote the advice when T contains all times before t.

We will denote $C^*(j)$ as the time interval of the connected component in OPT that contains job j. With these definitions, let $ADV_\lambda(I)$ be $\bigcup_{t \in T} ADV'(I, t, ALG_\lambda(I, ADV_\lambda(I, t - \Delta t), t - \Delta t))$. This definition allows us to define ADV' such that we can iteratively update the advice based on the decisions ALG_λ has made.

We define ADV' such that, once a primary job j has to be scheduled at time t, it advices MULTIPLIER to activate the machine for the amount of time that OPT needs to run the connected component of job j. The idea behind this is that MULTIPLIER, for some optimal connected component C_i^*, can then increase the size of primary jobs as fast as possible while not increasing this size much more than $\mu(C_i^*)$. Formally, let $ADV'(I, t, ALG)$ be $\hat{S} = \{[t, t + \mu(C^*(j)))\}$ if $t = LST_j$ for some job $j \in (U \in ALG)$ and $[t, t + p_j) \notin (P \in ALG)$, and $\emptyset$ otherwise. Now we can use this definition to analyze what the worst-case competitive ratio is when this advice is followed.

▶ **Lemma 22** (Cost of $\rho_{\max}$). *$w_{\rho_{\max}} \cdot p_{\rho_{\max}} \leq \frac{1}{\beta} \cdot \mu(C_i^*)$*

Proof: We will do a case distinction based on the size of $p_{\rho_{\max}}$.

- If $\beta \cdot \mu(C_i^*) \leq p_{\rho_{\max}} \leq \mu(C_i^*)$ then $\max\{ \frac{1}{\beta}, \min\ \frac{1}{\alpha}, \frac{\hat{w}_{\rho_{\max}}}{p_{\rho_{\max}}} \ \} \} \leq \max\{\frac{1}{\beta}, \min\{\frac{1}{\alpha}, \frac{\mu(C_i^*)}{\beta \cdot \mu(C_i^*)}\}\}$ $= \frac{1}{\beta}$ which means the total cost is at most $\frac{1}{\beta} \cdot \mu(C_i^*)$.
- If $\alpha \cdot \mu(C_i^*) < p_{\rho_{\max}} < \beta \cdot \mu(C_i^*)$ then $\frac{1}{\beta} < \frac{\mu(C_i^*)}{p_{\rho_{\max}}} < \frac{1}{\alpha}$ and therefore $w_{\rho_{\max}} = \frac{\mu(C_i^*)}{p_{\rho_{\max}}}$ which means that $w_{\rho_{\max}} \cdot p_{\rho_{\max}} = \mu(C_i^*)$.
- If $p_{\rho_{\max}} \leq \alpha \cdot \mu(C_i^*)$ then $\frac{\mu(C_i^*)}{p_{\rho_{\max}}} \geq \frac{1}{\alpha}$ and therefore $w_{\rho_{\max}} = \frac{1}{\alpha}$ which means that $w_{\rho_{\max}} \cdot p_{\rho_{\max}} \leq \mu(C_i^*)$.

Recall that $p_j \leq \mu(C_i^*)$ for all $j \in J(C_i^*)$. Therefore $w_{\rho_{\max}} \cdot p_{\rho_{\max}} \leq \max\{ \frac{1}{\beta} \cdot \mu(C_i^*), \mu(C_i^*) \}$ $= \frac{1}{\beta} \cdot \mu(C_i^*)$. ◀

▶ **Corollary 23.** $\sum_{j=1}^{j_{\max}} w_j \cdot p_j \leq \frac{1}{\beta} \cdot \mu(C_i^*) + \sum_{j=1}^{j_{\max}-1} w_j \cdot p_j$.

Since we now have a bound on the last job in P_2^i, we need to find a bound on the window for the other jobs.

▶ **Lemma 24** (All other jobs have a window of $\frac{1}{\alpha}$). *$w_{\rho_j} = \frac{1}{\alpha}$ for all $j < j_{\max}$.*

Proof by contradiction: We know $w_{\rho_j} \leq \frac{1}{\alpha}$ by definition, therefore we assume $w_{\rho_j} = \frac{1}{\alpha} - x$ for some x between ϵ and $\frac{1}{\alpha} - \frac{1}{\beta}$. Then $\frac{w_{\rho_j}^*}{p_{\rho_j}} = \frac{\mu(C_i^*)}{p_{\rho_j}} = \frac{1}{\alpha} - x$ which means that $\mu(C_i^*) = p_{\rho_j} \cdot (\frac{1}{\alpha} - x)$. We also know that $w_{\rho_{j_{\max}-1}} \cdot p_{\rho_{j_{\max}-1}} < p_{j_{\max}}$, so $w_{\rho_{j_{\max}-1}} \cdot p_{\rho_{j_{\max}-1}} < \mu(C_i^*)$. However $\mu(C_i^*) = p_{\rho_{j_{\max}-1}} \cdot (\frac{1}{\alpha} - x) = p_{\rho_{j_{\max}-1}} \cdot w_{\rho_{j_{\max}-1}}$ which leads to a contradiction for $j = j_{\max} - 1$. And for all $j < j_{\max} - 1$ we know that $p_{\rho_j} < p_{\rho_{j_{\max}-1}}$ which means that $\frac{\mu(C_i^*)}{p_{\rho_j}} \geq \frac{1}{\alpha}$, since the same holds for $p_{\rho_{j_{\max}-1}}$. Therefore $w_{\rho_j} = \frac{1}{\alpha}$ for all $j < j_{\max}$. ◀

Now we can rewrite the bound on the cost of P_2^i to $\sum_{j \in P_2^i} w_j \cdot p_j \leq \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{j=1}^{j_{\max}-1} \frac{1}{w_{\rho_{j+1}} \cdot w_{\rho_{j+2}} \cdots w_{\rho_{j_{\max}-1}}} \leq \frac{1}{\beta} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{n=0}^{-\infty} \alpha^n$ This means that the algorithm's cost is now bounded by $\mu(\mathcal{T}) \leq \sum_{C_i^* \in \mathcal{C}^*} \mu(C_i^*) + \frac{1}{\beta} \cdot \mu(C_i^*) + \mu(C_i^*) \cdot \sum_{n=0}^{-\infty} \alpha^n$. When we divide this bound by $OPT = \sum_{C_i^* \in \mathcal{C}^*} \mu(C_i^*)$, we get a competitive ratio of $1 + \frac{1}{\beta} + \sum_{n=0}^{-\infty} \frac{1}{\alpha}^n$. Which, by definition of α and β we can rewrite to $1 + \frac{4}{1+\lambda}$ for all $\lambda < 1$.

▶ **Corollary 25.** *ALG_λ is at most $1 + \frac{4}{1+\lambda}$-consistent when $\lambda < 1$.*

5.6 Consistency ($\lambda = 1$)

If $\lambda = 1$ we define $\alpha = \infty$ and $\beta = 1$. For this proof we will re-use the definition of $ADV_\lambda(I)$ as our advisor. Once $t = LST_j$ for some job j, $\hat{w} = \mu(C^*(j))$ at the beginning of the optimal connected component to which j belongs. Then, all jobs that fit inside this connected component are scheduled because $w_j \cdot p_j = \mu(C^*(j))$, since $\alpha = \infty$. Therefore the cost of the connected components is the same as OPT.

5.7 Locally updateable advice

Consistency and robustness are quantifyable measures of the quality of an online algorithm with machine learned advice. However, for scheduling problems and other problems with potentially infinitely large instances, we can see that depending on the required format of the advice, the advice may be infinitely large as well. Such advice may be infeasible to compute.

Moreover, if we use the online-time model for an online algorithm, we may see that it is easier to predict things about the near future, rather than predicting things about the far future. Anectodally, we can look at weather predictions to see that this may be true, and literature confirms this. For reference, see the paper about uncertainty in weather and climate prediction by Slingo et al. for example [30].

Therefore, we would like to introduce the notion of a *locally updateable* online algorithm. An online algorithm with machine learned advice is locally updateable, if the advice about the future can be revised as time goes on. Allowing the advice to be updated, in an online algorithm, allows the problem instance to be potentially infinitely large, and it may help the advisor to provide more accurate advice.

Remember that, in the MULTIPLIER algorithm, we require the advice to be in a series of time intervals. And, we have formulated the algorithm in a way such that the advice has to be given at the beginning of the problem instance. However, we can reformulate the algorithm to query the advisor each time $t = LST_j$ for some unscheduled job, and require the advice to be some number that denotes the advised open window, $\hat{w}_j$. This allows the advisor to give advice that contains a finite amount of bits, and it allows the advice to make use of all the local information about the problem instance. Moreover, this does not affect the consistency and robustness of our algorithm, since the union of all adviced open windows, $\bigcup_{j \in P} [LST_j, LST_j + \hat{w}_j)$, is in the same format as $\hat{S}$. Therefore, the MULTIPLIER algorithm can be used as a locally updateable online algorithm.

6 Other results

6.1 The overlapper algorithm

When looking at the doubler algorithm, one might wonder why the algorithm reserves an open window. Instead, we could look at the potential overlap between a primary job and some unscheduled job to determine whether this unscheduled job should be scheduled. In this section, we will analyze the upper bound of such an algorithm.

In 2017, Fong et al. [27], presented an algorithm using this method, and claim that the algorithm is 4-competitive. We will refer to this algorithm as the *OVERLAPPER algorithm*. However, Koehler et al. [4] mentioned that this result is not correct, but they did not provide a counter example to prove this claim. In this section we disprove the claim that the OVERLAPPER algorithm is 4-competitive.

6.1.1 Algorithm description

Since OVERLAPPER by Fong et al. is not formulated in an online-time manner we will reformulate it such that it works in the online-time model. The OVERLAPPER algorithm starts with waiting until some job reaches its latest starting time. Then, the algorithm will schedule this job, and mark it as a *primary job*. Let the *overlap* between two jobs in a schedule, $o(j_1, j_2)$, be $\min\{s_{j_1} + p_{j_1}, s_{j_2} + p_{j_2}\} - \max\{s_{j_1}, s_{j_2}\}$. And, let the *relative overlap* $ro(j_1, j_2)$ be $\frac{o(j_1,j_2)}{p_{j_2}}$. Once a primary job has been scheduled at time t, the algorithm computes the relative overlap between the primary job and each unscheduled job, as if each unscheduled job is to be scheduled at time t. Each job that has a relative overlap with the primary job of more than 0.5 is then scheduled. In Algorithm 4 we show the precise flow of the OVERLAPPER algorithm.

Algorithm 4 The OVERLAPPER algorithm

```
Let P = ∅
for all times t ∈ T do
    Schedule every unscheduled job j for which ro(ρ, j) ≥ 0.5 for some primary job ρ
    if t = LST_j for some unscheduled job then
        Let j be the longest unscheduled job
        Schedule job j and add it to P, the set of primary jobs
        Schedule every unscheduled job j for which ro(ρ, j) ≥ 0.5 for some primary job ρ
    end if
end for
```

6.1.2 Analysis

As previously shown in the analysis of the DOUBLER algorithm in Section 3.4, the cost of DOUBLER can be bounded by the cost of the primary jobs. In their analysis, Fong et al. claim something similar for the OVERLAPPER algorithm. Note that all jobs are either primary jobs or non-primary jobs, and the non-primary jobs are only scheduled if their relative overlap is larger than, or equal to 0.5. Therefore, any non-primary job must be scheduled in the interval $[s_\rho, s_\rho + 2p_\rho)$ for some primary job ρ. And thus, the cost of the OVERLAPPER algorithm is at most $2 \cdot \sum_{j \in P} p_j$.

In their analysis, Fong et al. partition the schedule resulting from the OVERLAPPER algorithm into *blocks*. A block consists of a primary job ρ, and all jobs that were scheduled because they had a relative overlap with ρ of 0.5 or more. Let B_i denote the block which the i^{th} primary job belongs to. Then, if some job is in B_k, where $k > 1$, one of the following must hold:

- The k^{th} primary job could not fit in B_{k-1} since the release time of the k^{th} primary job is after the finishing time of B_{k-1}
- The k^{th} primary job had a relative overlap with the $(k-1)^{th}$ primary job of less than 0.5

Fong et al. then state that "For any two consecutive jobs j_i and j_k, if $r_i < d_k$, then at most $p_k/2$ will overlap with j_i". We will interpret the statement as "For any two consecutive *primary* jobs j_i and j_k, if $r_i < d_k$, then at most $p_k/2$ will overlap with j_i *in the optimal schedule*" , where consecutive means that there's no other primary job with a starting time between s_{j_i} and s_{j_k}, and $s_{j_k} > s_{j_i}$. If we do not interpret this statement in this way, it is false, since for a primary job j_i and some job j_k within the primary job's block it may be that:

- j_k is scheduled after j_i is scheduled, if $r_k > s_i$
- $r_i < d_k$ is always true in this case, since they are in the same block
- $ro(j_k, j_i) \geq 0.5$, since j_k is in the same block as j_i

Then, from the fact that the overlap in the optimal schedule between two consecutive primary jobs j_i and j_k is at most $p_k/2$, Fong et al. state that at least $p_k/2$ time is required to execute job j_k in the optimal schedule. If this statement were always true, then the optimal schedule has a lower bound of $0.5 \cdot \sum_{j \in P} p_i$. Combined with the upper bound on the cost of the algorithm, this would result in a competitive ratio of at most 4. However, we believe this statement is not always true, since it assumes that j_i and j_k will overlap in the optimal schedule, whereas these jobs may also overlap with other jobs.

6.1.3 A counter example

In order to provide an adversary for the OVERLAPPER algorithm, the adversary will schedule a sequence of jobs that, in the optimal solution, can be scheduled as one connected component. This sequence of jobs will consist of a set of small and rigid jobs, R, and some larger primary, and non-primary jobs, J_p and J_{np} respectively. We start out by defining the sequence of jobs first, but for the analysis to work we will repeat the sequence later in this section. And, on top of this repeating sequence, the adversary will also release some larger jobs, which in the end will also form one connected component in the optimal solution at the very end of the schedule.

Let ℓ denote some length, which will correspond to the size of the repeating connected components in the optimal solution, and let ϵ be some insignificantly small number. Also, since we want to create a repeating sequence, let τ be some time at which we want to generate this sequence of jobs. At times where $t \in \{\tau, \tau + \epsilon, \tau + 2\epsilon, \ldots, \tau + \ell - \epsilon\}$, or formally, $t \in \{\tau + i \cdot \epsilon \mid i \in \mathbb{N} \wedge i \cdot \epsilon < \ell\}$, release a rigid job $j \in R$ with $p_j = \epsilon$ and $d_j = r_j + \epsilon$. This will force both the algorithm, and the optimal solution to create a connected component starting at t, with a busy time of at least ℓ.

Then, let π denote a set of processing times of jobs, for which we want to force the algorithm to schedule them as primary jobs. Since the algorithm schedules jobs as non-primary only if the overlap is 0.5 or greater, we want to release jobs that are more-than doubling for J_p. Let π be $\{\ell, \frac{\ell}{2} - \epsilon, \frac{\ell}{4} - 3\epsilon, \frac{\ell}{8} - 7\epsilon, \ldots\}$, or formally, $\pi \in \left\{\ell \cdot \frac{1}{2^{i-1}} - (2^{i-1} - 1)\epsilon \mid i \in \mathbb{N}\right\}$,

and all processing times are greater than $2 \cdot \epsilon$. Release a job $j \in J_p$ at time $\tau + \ell - p_j$ with processing time p_j and deadline $\tau + \ell + 2i \cdot \ell$, for each processing time $p_j \in \pi$, in ascending order. Here i denotes the index of p_j in π in ascending order. And, release a job $j \in J_{np}$ at time $\tau + \ell - 2p_j$ with processing time $2p_j$ and deadline $\tau + \ell + (2i+1) \cdot \ell$, for each processing time p_j in π, except for the largest processing time, in ascending order.

▶ **Lemma 26** (Single component cost OPT). *The cost of OPT, when the adversary releases one set of jobs with $\tau = 0$, is ℓ.*

Proof: If the adversary releases one set of jobs such that $\tau = 0$, then at times 0 to ℓ, OPT is forced to process the rigid jobs with $p_j = \epsilon$. The rest of the jobs have a processing time of at most ℓ, and their release times are $\ell - p_j$. This means they can all be scheduled such that they start at, or after $t = 0$, and they finish at, or before $t = \ell$. Therefore, all jobs will form a connected component that starts at $t = 0$ and finishes at $t = \ell$, meaning the optimal busy time is ℓ. ◀

In Figure 14 we illustrate a possible optimal connected component.

▶ **Lemma 27** (Single component cost ALG). *The cost of* OVERLAPPER*, when the adversary releases one set of jobs with $\tau = 0$, is $4 \cdot \ell$.*

Proof: If the adversary releases one set of jobs such that $\tau = 0$, then at times 0 to ℓ, OVERLAPPER is forced to process the rigid jobs with $p_j = \epsilon$. Moreover, since the smallest processing time in π is greater than $2 \cdot \epsilon$ by definition of π, no job will have a relative overlap greater than 0.5 with the rigid jobs with $p_j = \epsilon$. Therefore, these jobs will be scheduled after $t = \ell$.

After $t = \ell$, the smallest jobs with $p_j \in \pi$ will become the first job that must be scheduled as a primary job, since it has the earliest lastest starting time. Moreover, since a job with twice the processing time has also been released, with a later deadline, this job will have a relative overlap of 0.5, meaning it will be scheduled at the same time as the first primary job with $p_j \in \pi$. All other jobs with $p_j \in \pi$ will also become primary jobs, via the same logic, since all processing times in π more than double. And, for each of these primary jobs, except for the largest job with $p_j = \ell$, a corresponding job with twice the processing time was released.

Therefore, the total busy time of the jobs with $p_j \in \pi$ and $\frac{p_j}{2} \in \pi$, scheduled by OVERLAPPER, will be $\ell + (\ell - 2\epsilon) + (\frac{\ell}{2} - 6\epsilon)$.... Meaning, as ϵ approaches zero, the cost of these jobs approaches $3 \cdot \ell$. If we add the cost of the rigid jobs with $p_j = \epsilon$, the total cost of the schedule produced by OVERLAPPER will be $4 \cdot \ell$. ◀

In Figure 15 we illustrate OVERLAPPER's solution on a single optimal connected component.

So far, the competitive ratio of OVERLAPPER would be 4, however, the last primary job in the optimal connected component, is currently scheduled such that no other non-primary job is scheduled in the same block. In order to force OVERLAPPER to schedule such a non-primary job we will repeat the jobs released in the previously mentioned connected component at times $\tau \in \{0, X, 2X, 3X, ..., N \cdot X\}$. Where X is some large number such that none of the jobs between optimal connected components can overlap, and N is some large number approaching infinity. As per Lemma 27, OVERLAPPER will schedule these jobs with a total cost of $4 \cdot N \cdot \ell$.

Then, at every time where $t = s_j$ in OVERLAPPER's schedule, where $p_j = \ell$, release a job j' with $p_{j'} = 2\ell$ and $d_{j'} = 2N \cdot X$. This job will be scheduled immediately by the OVERLAPPER algorithm, since the overlap with primary job with $p_j = \ell$ is 0.5. Moreover, since we generate

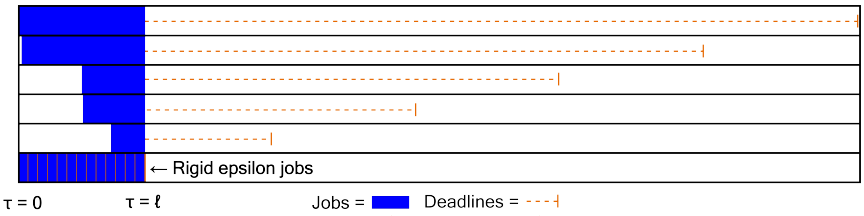

Figure 14 A single optimal connected component

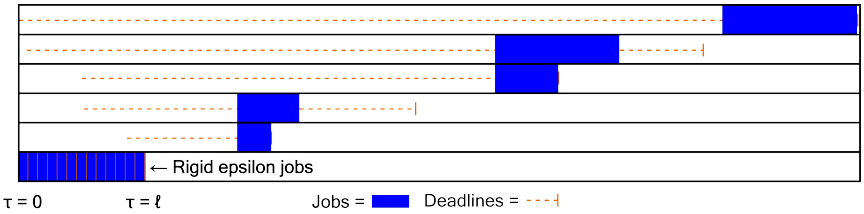

Figure 15 OVERLAPPER for a single connected component in OPT

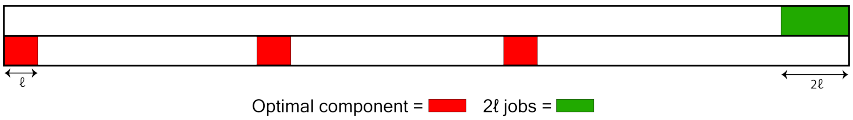

Figure 16 An optimal schedule

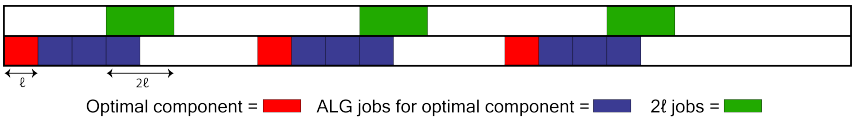

Figure 17 OVERLAPPER's schedule

N optimal connected components, this will add $N \cdot \ell$ to the cost of OVERLAPPER. And, in the optimal schedule, these jobs with $p_j = 2\ell$ can all be scheduled at the same time, meaning the combined cost of these jobs for OPT is at most 2ℓ. We illustrate the resulting schedules of OPT and OVERLAPPER in figures 16 and 17 respectively.

▶ **Theorem 28.** *OVERLAPPER is at least 5-competitive*

Combining the additional cost of the jobs with $p_j = 2\ell$ with Lemma 27, the cost of OVERLAPPER becomes $5 \cdot N \cdot \ell$. And, the cost of OPT becomes $N \cdot \ell + 2\ell$. Meaning, as N approaches infinity, the competitive ratio of OVERLAPPER approaches 5, and it is not 4-competitive.

6.2 The crediting method

If we look at both the DOUBLER and the OVERLAPPER algorithms, in sections 3.4 and 6.1, we see both algorithms are very similar. In this section we will introduce generalization between the two, and show that some possible directions to improve the upper bound. To do this, we

introduce the generic CREDITING method, which can be used to define both DOUBLER and OVERLAPPER by defining two functions, *incr* and *decr*.

The CREDITING method, just like the DOUBLER and OVERLAPPER algorithms, keeps track of a list of *primary jobs*, which are jobs that are scheduled at the latest possible starting time. Each time a primary job j is scheduled, the credit, denoted as κ, is increased to $incr(\kappa, j)$. And, when a job has a processing time that's less than, or equal to the remaining credit, the job is scheduled. The credit is not immediately consumed, instead it is consumed over time. The rate at which the credit is decreased depends on the definition of $decr(t, \Delta t)$, where Δt denotes the time that has passed. In order for κ to represent credit that can be used to schedule other jobs, we require $decr(t, \Delta t) \geq \Delta t$ if the machine is processing jobs at time t. In Algorithm 5 we show the precise flow of the CREDITING method.

Algorithm 5 The crediting method

Let $P = \emptyset$
for all times $t \in T$ **do**
  Schedule every unscheduled job j for which $p_j \leq \kappa$
  if $t = LST_j$ for some unscheduled job **then**
    Let j be some job with $t = LST_j$ and $p_j = \max\{p_{j'} | j' \in U \text{ and } t = LST_{j'}\}$
    Schedule job j and add it to P, the set of primary jobs
    Set κ to $incr(\kappa, j)$
    Schedule every unscheduled job j for which $p_j \leq \kappa$
  end if
  Decrease κ by $decr(t, \Delta t)$
end for

If we look at the DOUBLER algorithm, we see that each time a primary job is scheduled, a window of size $2p_j$ is reserved, in which other jobs can be scheduled. We can use the CREDITING method to redefine, by defining $incr(\kappa, j)$ as $2p_j$ and defining $decr(t, \Delta t)$ as Δt if $t \in \mathcal{T}$ and 0 otherwise. This will set the remaining credit to $2p_j$ each time a primary job j is scheduled, and the remaining credit will linearly decrease to 0 over the span of $2p_j$, if no other primary job is scheduled. Therefore, the CREDITING method will behave exactly the same as DOUBLER with these definitions.

Instead of doubling, we can also use the CREDITING method to redefine the OVERLAPPER algorithm. Note that the OVERLAPPER algorithm also schedules jobs in a window of size $2p_j$ for some primary job j, however, it is more restrictive than DOUBLER, since no jobs are scheduled after the primary job has finished running. Therefore, $incr(\kappa, j)$ is still defined as $2p_j$, to allow other jobs to be scheduled within that period of time. Moreover, $decr(t, \Delta t)$ is defined as $2\Delta t$ if $t \in \mathcal{T}$ and 0 otherwise, meaning κ decreases twice as fast as DOUBLER. Since this linearly decreases, the effect is exactly the same as the OVERLAPPER algorithm.

Comparing DOUBLER with OVERLAPPER using the crediting method we see that, for both algorithms, $incr(\kappa, j)$ is defined as $2p_j$. And, if we look at their analyses we see that both of their costs are bound by twice the sum of the processing times of all primary jobs, or, in other words, the sum of $incr(\kappa, j)$ of all primary jobs. But we can actually define a thighter bound on the cost of an algorithm using the crediting method. Remember that we require $decr(t, \Delta t) \geq \Delta t$ if the machine is processing jobs at time t. Then, since κ increases by some factor, $\Delta incr(\kappa, j)$, each time a primary job is scheduled, the total cost of the algorithm can actually be bound by the sum of $\Delta incr(\kappa, j)$ of all primary jobs.

Let SAVER be an algorithm using the crediting method where $incr(\kappa, j)$ is defined as

$\kappa + 2p_j$, and $decr(t, \Delta t)$ is defined as Δt if $t \in P$ and 0 otherwise. Note that the bound on the cost of SAVER is the sum of $2p_j$ of all primary jobs. And, all bounds in the analysis of DOUBLER still hold for the SAVER algorithm, but we leave this up to the reader to verify. Also note that SAVER saves the credit for as long, or longer than DOUBLER and OVERLAPPER. Unfortunately, saving the credit for longer has, so far, not lead to any improvements to the upper bound on busy time scheduling with infinite processors. Whether other definitions of $incr(\kappa, j)$ and $decr(t, \Delta t)$ may lead to improvements to the upper bound could be investigated in future research.

7 Conclusion

In this thesis, we have looked at the online busy time scheduling problem with unlimited processors. We have analyzed existing research on the problem, and presented an adversarial instance to increase the previous lower bound of 1.618 to 2 for eager algorithms. Additionally, we disprove the previous upper bound of 4 by Fong et al. [27], and therefore we establish that the best known upper bound is 5.

We have tried to further improve upon the upper and lower bounds, but despite our efforts, have not succeeded as of yet. Given the large gap between the upper and the lower bounds, future research could focus on narrowing this gap even further. This can be done either by lowering the upper bound or by increasing the lower bound, whether both of these are possible remains to be seen.

In order to provide future researchers with a possible direction for decreasing the upper bound, we have formalized the CREDITING method. We hope that this can eventually be used to improve the upper bound, or in other ways help readers understand the online busy time problem better. Moreover, we found there is not a lot of existing research on the online busy time problem with limited processors and flexible jobs. We think researching this specific problem can be quite difficult, and therefore further analyzing the problem with unlimited processors may help to gain valuable insights into busy time scheduling as a whole.

Additionally, we have also looked into online busy time scheduling with machine learned advice, and have presented the MULTIPLIER algorithm. The MULTIPLIER algorithm is at most $(1 + \frac{4}{1+\lambda})$-consistent and at least $(1 + \frac{4}{1-\lambda})$-robust. We have not proven that this analysis is tight, so it may be that the consistency and robustness are lower than our analysis shows. Moreover, we have introduced the concept of local updateability, and have shown that the MULTIPLIER algorithm is locally updateable. We hope that other researchers also analyze the updateability of their algorithm for problems where the instance may, potentially, be infinitely large.

In summary, we have improved upon existing research on online busy time scheduling by looking at the lower and upper bounds. Furthermore, we have developed an algorithm using machine learned advice, which is currently a hot topic in researching online algorithms. Even though we have not proven the optimality of the algorithms presented in this thesis, we feel that this thesis can form a foundation for future research within this domain. We hope that we have inspired our readers to improve upon our results, and that our thesis will lead to further advancements in online busy time scheduling.

References

[1] J. Y. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC press, 2004.

[2] P. Brucker, *Scheduling algorithms*. Springer, 2007.

[3] M. Flammini, G. Monaco, L. Moscardelli *et al.*, 'Minimizing total busy time in parallel scheduling with application to optical networks,' *Theoretical Computer Science*, vol. 411, no. 40, pp. 3553–3562, 2010.

[4] F. Koehler and S. Khuller, 'Busy time scheduling on a bounded number of machines,' in *Algorithms and Data Structures: 15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31–August 2, 2017, Proceedings*, Springer, 2017, pp. 521–532.

[5] P. Winkler and L. Zhang, 'Wavelength assignment and generalized interval graph coloring,' in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '03, Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 830–831, ISBN: 0898715385.

[6] M. Shalom, A. Voloshin, P. W. Wong, F. C. Yung and S. Zaks, 'Online optimization of busy time on parallel machines,' *Theoretical Computer Science*, vol. 560, pp. 190–206, 2014.

[7] J. Xue, 'Online weighted throughput maximization scheduling with a busy-time budget,' M.S. thesis, 2022.

[8] S. Albers, 'Online algorithms: A survey,' *Mathematical Programming*, vol. 97, pp. 3–26, 2003.

[9] C. Chekuri, A. Goel, S. Khanna and A. Kumar, 'Multi-processor scheduling to minimize flow time with ε resource augmentation,' in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 363–372.

[10] K. Iwama and G. Zhang, 'Online knapsack with resource augmentation,' *Information Processing Letters*, vol. 110, no. 22, pp. 1016–1020, 2010.

[11] B. Kalyanasundaram and K. Pruhs, 'Speed is as powerful as clairvoyance,' *Journal of the ACM (JACM)*, vol. 47, no. 4, pp. 617–643, 2000.

[12] J. Boyar, L. M. Favrholdt, C. Kudahl, K. S. Larsen and J. W. Mikkelsen, 'Online algorithms with advice: A survey,' *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–34, 2017.

[13] Y. Emek, P. Fraigniaud, A. Korman and A. Rosén, 'Online computation with advice,' *Theoretical Computer Science*, vol. 412, no. 24, pp. 2642–2656, 2011.

[14] P. Fraigniaud, D. Ilcinkas and A. Pelc, 'Communication algorithms with advice,' *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 222–232, 2010.

[15] M. P. Renault, A. Rosén and R. van Stee, 'Online algorithms with advice for bin packing and scheduling problems,' *Theoretical Computer Science*, vol. 600, pp. 155–170, 2015.

[16] S. Angelopoulos, C. Dürr, S. Jin, S. Kamali and M. Renault, 'Online computation with untrusted advice,' *arXiv preprint arXiv:1905.05655*, 2019.

[17] S. Angelopoulos and S. Kamali, 'Contract scheduling with predictions,' *Journal of Artificial Intelligence Research*, vol. 77, pp. 395–426, 2023.

[18] J. Boyar, L. M. Favrholdt, S. Kamali and K. S. Larsen, 'Online interval scheduling with predictions,' *arXiv preprint arXiv:2302.13701*, 2023.

[19] A. Antoniadis, C. Coester, M. Eliáš, A. Polak and B. Simon, 'Online metric algorithms with untrusted predictions,' *ACM Transactions on Algorithms*, vol. 19, no. 2, pp. 1–34, 2023.

[20] A. Wei and F. Zhang, 'Optimal robustness-consistency trade-offs for learning-augmented online algorithms,' *Advances in Neural Information Processing Systems*, vol. 33, pp. 8042–8053, 2020.

[21] M. Berg and S. Kamali, *Online bin covering with frequency predictions*, 2024. arXiv: 2401.14881 [cs.DS].

[22] F. Eberle, A. Lindermayr, N. Megow, L. Nölke and J. Schlöter, *Robustification of online graph exploration methods*, 2021. arXiv: 2112.05422 [cs.LG].

[23] K. Pruhs, J. Sgall and E. Torng, *Online scheduling.* 2004.

[24] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge university press, 1995.

[25] Z. Lotker, B. Patt-Shamir and D. Rawitz, 'Rent, lease or buy: Randomized algorithms for multislope ski rental,' *arXiv preprint arXiv:0802.2832*, 2008.

[26] L. Epstein and A. Levin, 'Improved randomized results for the interval selection problem,' *Theoretical Computer Science*, vol. 411, no. 34-36, pp. 3129–3135, 2010.

[27] K. C. Fong, M. Li, Y. Li, S.-H. Poon, W. Wu and Y. Zhao, 'Scheduling tasks to minimize active time on a processor with unlimited capacity,' in *Theory and Applications of Models of Computation: 14th Annual Conference, TAMC 2017, Bern, Switzerland, April 20-22, 2017, Proceedings*, Springer, 2017, pp. 247–259.

[28] R. Ren and X. Tang, 'Online flexible job scheduling for minimum span,' in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 55–66.

[29] R. Ren and X. Tang, 'Busy-time scheduling on heterogeneous machines,' in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2020, pp. 306–315.

[30] J. Slingo and T. Palmer, 'Uncertainty in weather and climate prediction,' *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 369, no. 1956, pp. 4751–4767, 2011.