SAPIENZA
UNIVERSITÀ DI ROMA

Online Scheduling with Predictions

Department of Computer Science
Applied Computer Science and Artificial Intelligence

Gabriele Matiddi
ID number 1985899

Advisor
Prof. Alessandro Panconesi

Academic Year 2023-2024

Online Scheduling with Predictions
Bachelor Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: gabrimat1@protonmail.ch

Abstract

This thesis investigates the integration of machine learning predictions with online scheduling algorithms to improve efficiency in resource allocation tasks. Scheduling is crucial in fields like cloud computing and manufacturing, which directly affect productivity. Three main contributions are made: a review of existing online algorithms and their competitive ratios, the introduction of a new algorithm built upon one of those in the study with theoretical performance bounds, and extensive simulations validating the proposed and existing algorithms utilizing the total completion time metric. The simulations highlight the effectiveness of incorporating predictions in online scheduling; real-world data has been used to evaluate how the algorithm would perform outside the theoretical sandbox.

Contents

Chapter 1

Introduction

1.1 Online algorithms

Due to rapid technological advancements and increasing demand for computational efficiency, improving the logic behind processing systems is of great importance. Online algorithms, which make decisions based on information available up to the current time without knowledge of future events, have gained significant attention, as they are essential in various real-world applications. The primary challenge in online algorithms is evaluating their performance compared to offline algorithms, which instead have complete knowledge of the input sequence in advance. The competitive ratio is a key metric that pivoted the study of online algorithms [22, 3] in a theoretical setting and granted them theoretical guarantees. The competitive ratio is used to evaluate online algorithms, measuring how well an online algorithm performs relative to the optimal offline algorithm [7]. Lately, with the explosion in popularity of machine learning models, these two concepts were found to mix well to tackle real-world problems [19, 16].

1.2 Contributions

This thesis explores the integration of machine learning predictions with online algorithms to enhance their performance in scheduling problems. Scheduling, a fundamental issue in computer science, is the decision regarding resource allocation given a set of processes, in particular, we consider the problem of *CPU Scheduling*, which concerns the processes in an operating system to which we must allocate computing resources [21]. We investigate several online scheduling algorithms that leverage predictions to improve decision-making. Our focus is on the jobs' total completion time, a crucial metric in many practical scenarios. By incorporating predictions, we aim to bridge the gap between online and offline performance, providing robust and efficient solutions even in the presence of prediction errors. The thesis makes three main contributions: first, it reviews existing online algorithms and their competitive ratios, highlighting their strengths and limitations; second, it introduces a new variation of an online algorithm for scheduling and proves its bounds theoretically; and third, it conducts extensive simulations to evaluate the proposed algorithm and those already present in the literature, demonstrating their

effectiveness compared to traditional approaches. The structure of this thesis is as follows.

1.3 Thesis structure

- Chapter 2 provides a background on online algorithms, competitive ratios, and the basics of job scheduling.
- Chapter 3 details the proposed algorithms and their theoretical notions.
- Chapter 4 presents the experimental setup and results, showcasing the various performances of the algorithms achieved through our methods.
- Finally, Chapter 5 concludes the thesis with a summary of findings and potential directions for future research.

The tests and the code utilized can be found at the following github repository.

Chapter 2

Background

2.1 Online Algorithms and Competitive Ratio

In online computation, an algorithm must make decisions without knowing future information, and the algorithms utilized in this framework are referred to as *online algorithms* [7]. These algorithms are essential for problems that cannot be solved using offline algorithms, which operate with the assumption of having access to the entire input at the start of their execution, and it is clear that in many real-world scenarios, this assumption does not hold. Examples of tasks that heavily rely on an online approach include paging in virtual memory systems [2], routing in communication networks [28], caching [19], and various applications in finance [8]. Problems in computational theory can be classified into inherently offline and inherently online problems. For inherently online problems, even though offline algorithms cannot be used to solve actual instances due to their requirement of complete future knowledge, they are still valuable for performance evaluation by acting as a benchmark. In some cases, both the online and offline versions of a problem are relevant and arise in different scenarios. A notable example is job scheduling, which appears in various contexts with different underlying assumptions [3, 18, 20]. Initially, the performance of online algorithms was studied using the framework of average-case complexity. However, in the 1990s, the introduction of competitive analysis revolutionized the field [7]. Competitive analysis provides a robust method to evaluate the performance of online algorithms by comparing their outcomes with those of an optimal offline algorithm. This approach has since become a standard in studying online algorithms, offering a clearer and more rigorous understanding of their efficiency and effectiveness across different problem domains. This new system introduced a new metric for the quality of online algorithms called *competitive ratio*. With this different framework, the rendition of an online algorithm is assessed by comparing its associated cost (or score) on each input sequence against an ideally *optimal offline algorithm*, an algorithm which has full knowledge of the future, or in other words, the entirety of the input sequence [7].

The two frameworks for performance evaluation have different downsides. Competitive analysis, for example, is considerably pessimistic, as it tests solutions on worst-case scenarios over all possible inputs, losing the capability to exploit any known prior structure of the inputs [29]. Conversely, average-case analysis or *distri-*

butional complexity requires a good knowledge of the underlying distribution of the inputs for mathematical tractability, which rarely is the case [7].

2.1.1 Cost Minimization Problems

In the following subsection, some information on competitive analysis will be explained to deal with the main concepts of the thesis. Most of the information is originally found in [7].
An **optimization problem** can be of two types: cost minimization or profit maximization. We will focus on the former, as is the case for the total completion time problem in scheduling. An optimization problem $\mathcal{P}$ of cost minimization consists of two elements: an input set $\mathcal{I}$ and cost function $\mathcal{C}$. For each input $I \in \mathcal{I}$, there is a set of feasible solutions $F(I)$, and for each possible solution $O \in F(I)$, there is a cost of the output O considering the input I denoted as $\mathcal{C}(I, O)$ to minimize.
Given a legal input I, an algorithm ALG for a problem $\mathcal{P}$ computes a feasible solution $\mathrm{ALG}[I] \in F(I)$. The cost of this solution is denoted as $\mathrm{ALG}(I) = \mathcal{C}(I, \mathrm{ALG}[I])$. An **optimal algorithm** denoted OPT, is one that possesses the following property:

$$\mathrm{OPT}(I) = \min_{O \in F(I)} \mathcal{C}(I, O), \forall\ I \in \mathcal{I}$$

It is said that an algorithm ALG is a c-approximation algorithm for a minimization problem $\mathcal{P}$ if there exists $\alpha \geq 0$ such that for all legal inputs we have:

$$\mathrm{ALG}(I) - c \cdot \mathrm{OPT}(I) \leq \alpha$$

2.1.2 The Competitive Ratio

An **online** algorithm ALG is said c-competitive if there exists a constant α such that for all possible input sequences I we have that:

$$\mathrm{ALG}(I) \leq \mathrm{OPT}(I) \cdot c + \alpha$$

More specifically, when $\alpha \leq 0$, we say that the algorithm is *strictly* c-competitive. Then it is clear that if an algorithm ALG is a c-approximation algorithm that computes online, then ALG is also c-competitive. From the previous definitions, we can now say that a (strictly) c-competitive algorithm has a cost within a factor c of the optimal offline algorithm, and the algorithm is called **competitive** if c is a constant with respect to the input of the problem I.
The infimum of all the values in the set $\{c \mid \text{ALG is } c\text{-competitive}\}$ is called the competitive ratio of the algorithm ALG and is denoted by $\mathcal{R}(\mathrm{ALG})$.
The concept of competitive ratio in performance analysis for online algorithms is not affected by the computational complexity of the algorithm at hand. Discovering algorithms with good competitive ratios and polynomial time complexity for real-world scenarios is highly desirable, especially when swift decision-making is crucial. In some instances, due to the nature of online problems, the algorithm may have strict time requirements for delivering solutions.

2.1.3 Adversary models

The competitiveness of online algorithms can be measured against different types of adversaries [7]. This variable is crucial in the case of randomized algorithms. There are mainly three types of adversaries:

- The first is called the **oblivious adversary** or weak adversary. This adversary knows the decisions and structure of the algorithm, but it doesn't know the underlying distribution of the algorithm's randomized results.
- We then have **adaptive online adversary** or medium adversary. This adversary must make its own decision before it is allowed to know the algorithm's decision. Still, it adapts to the algorithm's previous answers and gets stronger in the execution.
- Lastly, we have the **adaptive offline adversary** or strong adversary. This adversary knows everything and serves optimal decisions to hinder the algorithm.

Randomized algorithms do not offer a significant advantage against a strong adversary. Research has shown ([5, 6]) that if a randomized algorithm is α-competitive against any adaptive offline adversary, then there exists also a α-competitive deterministic algorithm.

2.2 Online Algorithms with Machine Learning Oracles

The framework and the information here summarized were presented in the work of Thodoris Lykouris and Sergei Vassilvitskii [15]. It is natural for some problems (as shown before) to require future knowledge to obtain an optimal solution. Not possessing it and aiming for the optimal implies dealing with uncertainty, which is a complicated issue for real-world computational tasks. Online algorithms already try to tackle these challenges by striving to achieve reasonable bounds in *worst-case* scenarios through the measure of the competitive ratio. Another famous approach for dealing with uncertainty is machine learning, where past information is used to build robust models and predict the future. Those models aim to be good *on average* by minimizing some loss function during the *learning* phase. While machine learning predictions perform well on predictable data that follows observed patterns, they tend to falter on outliers or adversarial examples [24].
Those two paradigms blend naturally when trying to obtain a solution that strives to improve as the machine-learned predictions get better and degrade gracefully when they fail instead [15], possibly by not doing worse than the optimal *self-reliant*[1] algorithm. The theoretical framework (summarized in this section) that connects online algorithms and machine learning has been introduced in [15]. From now on, we will refer to the machine learning model equipped to the algorithm as *oracle*. We aim for an approach that has the following properties:

- Make minimal assumptions on the error of the oracle. In particular, we know only the error η without information about the underlying distribution.

[1]The online algorithm that computes without using external predictions.

- The algorithm should yield better results with lower error η. This property is known as *robustness*.
- Are worst-case competitive, or in other words, the algorithm aims to behave at least as well as the best online algorithm regardless of the prediction error on the instance.

More formally, let $\mathcal{A}$ be an online algorithm for a problem Π, $\mathcal{L}$ a loss function for a prediction model $\mathcal{H}$, and $\mathcal{R}(\mathcal{A}, \epsilon)$ the competitive ratio of $\mathcal{A}$ with error (of the predictor) ϵ. Then, the Online with Machine Learned Advice ($OMLA$) framework specified in [15] states:

- $\mathcal{A}$ is β-consistent if $\mathcal{R}(\mathcal{A}, 0) = \beta$.
- $\mathcal{A}$ is α-robust for a function $\alpha(\cdot)$, if $\mathcal{R}(\mathcal{A}, \epsilon) = O(\alpha(\epsilon))$.
- $\mathcal{A}$ is γ-competitive if for all values of ϵ we have $\mathcal{R}(\mathcal{A}, \epsilon) \leq \gamma$.

With these three properties formalized, the objective is to find an online algorithm 1-consistent and α-robust with $\alpha(\cdot)$ being a slowly growing function.
In [15], a new algorithm for caching is developed by using the principles mentioned above, called **PredictiveMarker**, that obtains a competitive ratio of $2 + O(\min(\sqrt{\eta/\mathrm{OPT}}, \log k))$, in contrast to the best known randomized algorithm which instead has a competitive ratio of $\Theta(\log k)$.
In addition to the main algorithm developed in [15], the research presents a so-called "blind algorithm" denoted as $\mathcal{B}$ due to the *Bélády*'s rule. In caching, the optimal offline algorithm is the one that evicts the element that will appear the furthest in time, and $\mathcal{B}$ tries to mimic it by evicting the element which is **predicted** to appear the furthest in time. This algorithm is 1-consistent since it behaves like the clairvoyant algorithm if the predictions are perfect. Unfortunately, the degradation with respect to the increase of the predictor error is far from the best possible, giving an unbounded competitive ratio of $\mathcal{R}(\mathcal{B}, \epsilon) = \Omega(\epsilon)$, and so the algorithm is not robust.
To obtain an algorithm that improves with better predictions but still has reasonable bounds in worst-case scenarios, a possible approach is to mix two algorithms and make them work together towards a specific task. We will see that in [20], this approach is used to get a consistent and robust algorithm by mixing a blind, greedy algorithm and a self-reliant algorithm. We will delve more in-depth into this in the next chapter.

2.3 Job scheduling

Job scheduling is a fundamental problem in computer science with an incredible variety of real-world applications. For example, the efficiency of CPU scheduling is directly tied to the device's productivity. Most of the information summarized here is originally found in [21].
A real-world process alternates between "CPU Bursts" and "I/O bursts". CPU Bursts occur when the CPU for the process performs active work, while I/O bursts are periods when the process is idle, waiting for some external action. In real-world

scenarios, the processes are usually swapped out from the processor during those I/O bursts in the case of a *multiprogramming* [2] system. In our analysis, we will assume that the processes consist of a single CPU burst segment.

2.3.1 Process States and Scheduling

In the context of operating systems and process management, a process or job can be in various states during its lifetime. These states can be broadly categorized into the following [21]:

1. **New or Initial State**: This is the state when a process is first created or initialized. The process is being set up, and resources are being allocated for execution.
2. **Ready State**: A process is ready when it is waiting to be assigned to a processor by the scheduler. The process is in the main memory and has all the resources it needs to execute except for the CPU.
3. **Running or Execution State**: This is the state where the process is actively executing its instructions on the CPU.
4. **Blocked or Waiting State**: This is the state where a process cannot execute because it is waiting for some event to occur, such as the completion of an I/O operation, the arrival of a signal, or the availability of a resource. In a multiprogramming system, the process would typically be swapped out of the CPU during this time to allow other processes to execute.
5. **Terminated or Exit State**: This is the final stage of a process. The process has completed its execution or been terminated due to an error or a signal. The process is removed from the main memory, and its resources are deallocated.

In our analysis, processes are already in the ready state from the start, and we assume no constraints in main memory or other resources.

2.3.2 Preemptive and nonpreemptive scheduling

We will now use the term "job" to refer to processes, threads, and other computational tasks to be scheduled. Scheduling decisions normally may happen after four types of events:

1. When a job switches from a running state to a waiting state.
2. When a job switches from a running state to a ready state.
3. When a job switches from a waiting state to a ready state.
4. When a job finishes.

[2] Technique to maximize the output, organizing code and data such that the processor always has something to execute [21].

In our study, the first three situations do not apply to us; instead, there is a new case to consider:

- When, due to a change in the predictions or some other computational reasoning, another job becomes the preferred one to be scheduled.

When scheduling decisions occur only in situations 1 and 4 (when the processor has only the choice to remain idle or pick another job), the scheduling is called nonpreemptive. Otherwise, it is called preemptive [21]. In our study, all the algorithms used assume the capability of preemptive scheduling.
Preemption, if not handled correctly, may give rise to a certain set of problems like race conditions between processes and excessive time expenditure for context switches[3]. However, we will disregard all those problems as they are irrelevant to our theoretical setting.

2.3.3 Scheduling criteria

In scheduling theory, algorithms try to optimize one particular criterion. Some of the most common are [21]:

1. **Throughput**, number of jobs completed per time unit.
2. **Waiting time**, the total amount of time that the jobs spend in the ready queue.
3. **Response time**, time lapsed between submission and first response by the CPU.
4. **Turnaround time**, time lapsed between submission and completion. When all the jobs are present at the start of the execution, it is also called **Completion time**.

An elementary algorithm for scheduling jobs is the First Come First Served (FCFS) policy. Under FCFS, jobs are executed in the order they arrive in the system [21]. While simple to implement, FCFS can perform very poorly for completion times, as short jobs arriving later can get stuck waiting behind long-running jobs that arrived earlier. There are much more complicated algorithms used for the problem of scheduling when we talk about operating systems. Some examples of those are *Priority Scheduling*, *Multi-Level Queue Scheduling*, *Multi-Level Feedback Queue Scheduling* [21]. In the next chapter, we will introduce different scheduling algorithms from those mentioned above in a preemptive setting. Our algorithms focus only on the optimization of the total completion time metric.

[3]When a CPU switches its current process with another, saving the state of the original process and restoring the one of the incoming process (if any) [21].

Chapter 3

Online Scheduling

We will now explain the theoretical setting we will use, which, as mentioned before, doesn't consider many of the problems of real-world scheduling done by CPUs, and its focus is on evaluating the metric of total completion time. We now formally define our problem. Our setting starts similar to the one of the foundational research [18, 12].

Definition 3.1. A scheduling problem of size n is composed by a set of jobs $J = \{J_1, ..., J_n\}$. A job can be considered a triplet $J_i = (x_i, a_i, \boldsymbol{\beta}_i)$ where x_i is the size of the job, a_i is the arrival time (in our setting, $a_i = 0$, $\forall i \in \{1, ..., n\}$. In other words, all the jobs are present at the starting time). $\boldsymbol{\beta}_i$ is a vector representing the characteristics of the job.

In general, computing can be done by a set of processors $P = \{P_1, ..., P_n\}$, but we will consider the case of a single machine, so $|P| = 1$. Much of the research in scheduling before the nineties was concerned with the problem of *clairvoyant scheduling*, where arrival time a_i and size x_i of a job J_i are already known to the scheduler [18]. We will refer to this version of the problem with the term *offline* and to the algorithms that compute in this setting as *offline algorithms*. Likewise, we will refer to the version of the problem where some information is unknown as *online* and the algorithms in that setting as *online algorithms*. There are many different possible settings; in the next chapter, we will see that our setting assumes information knowledge such as to mimic a real-life scenario.
A solution to the scheduling problem is called a *schedule* and is an assignment of one or more intervals to each job on the processor such that the total time assigned to job J_i is equal to x_i, $\forall i \in \{1, ..., n\}$. Those time intervals should not overlap.

Definition 3.2. The associated **completion time** c_i for a job J_i is the total time required for a job to complete its execution, or in other words, the time when the last interval assigned to the job ends, having the job reached a total processing time of x_i. The **total completion time** is defined as $C = \sum_{i=1}^{n} c_i$. [18]

Suppose we are given a set of jobs J, and let $C_{\mathrm{ALG}}(J)$ be the total completion time corresponding to a schedule created by algorithm ALG. We define the competitive

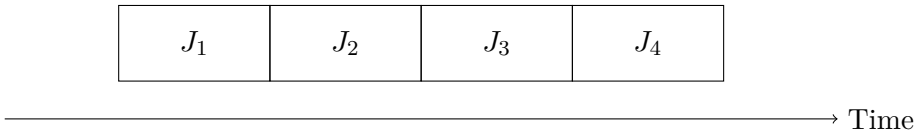
ratio of ALG as:

$$\mathcal{R}(\text{ALG}) = \frac{C_{\text{ALG}}(J)}{C_{\text{OPT}}(J)}$$

It is interesting to notice that in the case of *static*[1] scheduling, total completion time, average completion time, total waiting time, and average waiting time are all equivalent measures for comparing scheduling algorithms and result in equal competitive ratios on instances of the problem [18].

3.1 Online Optimal and Offline Optimal

Let's consider the algorithms that perform best without any prediction. The optimal offline algorithm is called *Shortest Job First* (SJF) [12]. The algorithm has clairvoyant properties and orders the execution of the jobs by their real duration in non-decreasing order. It is natural to see that this is the best possible algorithm, as it "stays ahead" of other algorithms (similarly to how other greedy algorithms work). The completion time of each job is determined by two elements: the job duration and the delay the job has been subject to due to other jobs already executed by the processor. To minimize the total delay the jobs suffer, we must execute them in non-decreasing order. First, we delay $n-1$ jobs with the smallest duration, then $n-2$ jobs with the second smallest duration (summed to the already present delay of the first executed job), and so on. The choice that minimizes the delay is always picking the smallest job to execute. So suppose we have four jobs J_1, J_2, J_3, J_4 and $x_1 \leq x_2 \leq x_3 \leq x_4$ then the schedule given by SJF will look like the following:

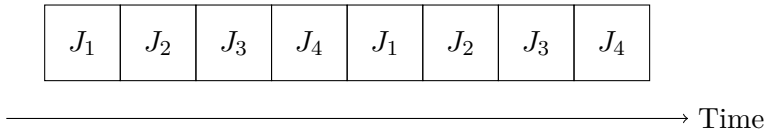


To compute the total completion time for this schedule, we can use the formula:

$$\sum_{i=1}^{n} x_i \cdot (n - i + 1)$$

Which sums up the job sizes and all the delays.
The optimal online algorithm is called *round-robin* (RR). As the name suggests, this algorithm executes all the jobs cyclically for ϵ. This algorithm has been shown to have a competitive ratio of $2 - \frac{2}{n+1}$, which is the best possible for deterministic online algorithms [18].
The interesting property of RR is that it has reliable performance regardless of input by blindly processing all the jobs. In this situation, all the jobs delay each other equally, stabilizing the algorithm's performance on all inputs. A schedule for RR, drawn in the same fashion as before, looks like this:

[1] A static scheduling problem is one where all the jobs are present at the starting time (like in our study) [18].



The next approaches presented are summarized from recent research on the topic. To continue the exposition, we define a prediction function for the processing time $f : \mathbb{R}^n \to \mathbb{R}_+$ that maps the characteristics to the predicted time $f : \boldsymbol{\beta}_i \mapsto y_i$. We assume that this function, whose implementation is not yet discussed, is the general **oracle** for all the algorithms with predictions presented. Depending on the algorithm, this function has an error, which we will define later for each case.

3.2 Shortest Predicted Job First: Greedy Approach with Predictions

One algorithm considered in [20] for the non-clairvoyant job scheduling problem is the *Shortest Predicted Job First* (SPJF) algorithm. This algorithm is the scheduling counterpart of the blind predictor used for caching in [15]. Although it is used to construct a consistent and robust algorithm, this greedy algorithm is used for testing as it signals how good the predictions are since it is 1-consistent. This algorithm just sorts the jobs in non-decreasing order of their predicted processing times y_i and executes them to completion in that order. The SPJF algorithm is a natural extension of the well-known SJF algorithm for the clairvoyant setting where job processing times x_i are known in advance. Sorting jobs by the y_i values aims to approximate the ordering SJF would use if it had perfect information about the x_i values. This allows SPJF to perform very well and produce optimal schedules when the predictions are accurate (i.e., when every y_i is good enough to avoid any displacement of the jobs, basically obtaining the same schedule as SJF).
Lemma 3.2 in [20] proves that SPJF has a competitive ratio of at most $(1 + \frac{2\eta}{n})$, where η is the total L1 error of the predictions $\eta = \sum_{i=1}^{n} \eta_i$ with $\eta_i = |x_i - y_i|$. An example is provided to prove that this bound is asymptotically tight. We can also note from the error definition that the performance of this algorithm does not degrade gracefully, and severe errors (for example, executing the largest job first, a possible situation) will greatly impact the performance.

3.3 Preferential round-robin: Mixed Algorithm with Predictions

To address the drawbacks of the SPJF algorithm when job length predictions are inaccurate, [20] introduces the *Preferential Round Robin* (PRR) scheduling algorithm. PRR combines SPJF with the classical RR algorithm in a preferential execution scheme. The idea is to run the algorithms concurrently and set their rates using a hyperparameter $\lambda \in (0, 1)$ (RR runs at rate λ while SPJF runs at rate $1 - \lambda$). By combining SPJF and RR, PRR aims to achieve the best of both worlds. When predictions are accurate, the SPJF component can mimic the optimal *Shortest job first schedule*, allowing PRR to perform well. However, the RR component provides a

constant competitive ratio guarantee to prevent degrading too much with inaccurate predictions. The analysis shows that PRR achieves a competitive ratio bound that depends on λ, the total prediction error η, and the number of jobs n. Specifically, its competitive ratio is at most $\min\{\frac{1+2\eta}{n\lambda}, \frac{2}{1-\lambda}\}$. PRR can beat RR's competitive ratio of 2 when predictions are sufficiently good while ensuring a constant worst-case guarantee. If we set $\lambda = 0.5$, we are assured that whatever the error of the predictor is, we get a competitive ratio of at most 4.

3.4 Preferential round-robin with Dynamic λ Hyper-Parameter

We present an original variation of the PRR algorithm tailored to real-life scenarios. Our approach introduces a new hyperparameter $\theta \in (0, 1)$ that acts as a lower bound threshold for the original λ parameter. Rather than fixing λ to a constant value, we allow it to dynamically vary within the range $(\theta, 1)$. This maintains a guaranteed level of robustness while providing more flexibility to leverage accurate predictions when available. The main idea is to periodically simulate the SPJF and RR algorithms on a sample set of completed jobs $\mathcal{S}$ to assess the quality of predictions. By comparing the total completion times C_{SPJF} and C_{RR} achieved by each algorithm on $\mathcal{S}$, we can adjust λ accordingly:

- If $C_{\text{RR}} < C_{\text{SPJF}}$, this signals that predictions may be inaccurate. We then increase λ by a step $\gamma \in (0, 1 - \lambda)$ to rely more on the robust RR component.
- If $C_{\text{SPJF}} < C_{\text{RR}}$, predictions appear to be performing well. We decrease λ by a step $\gamma \in (0, \lambda - \theta)$ to give more weight to the prediction-aware SPJF component.

This dynamic adjustment of λ based on periodic sampling allows the algorithm to adapt to changing prediction quality over time while still providing theoretical guarantees through the θ threshold.
We now show that it is possible to compute C_{RR} and C_{SPJF} in $O(n \log n)$ time complexity where $n = |\mathcal{S}|$. We may assume that we have information about the real durations $x_1, ..., x_n$ of the jobs since we have already completed them.

Theorem 3.1. *It is possible to compute the metrics C_{RR}, C_{SPJF} on a given instance of jobs of size n in $O(n \log n)$ time if x_i for all jobs J_i of the instance are known.*

Proof. We show that both subtasks require $O(n \log n)$ time. Starting with C_{SPJF}, we can sort the jobs (resulting in $O(n \log n)$ cost) and implement the formula

$$\sum_{i=1}^{n} x_i \cdot (n - i + 1)$$

where $\hat{x}_1 \leq \hat{x}_2 \leq ... \leq \hat{x}_n$ (indexes are ordered in non-increasing order of prediction size). This task requires $O(n)$ time, resulting in $O(n \log n)$ time for computing C_{SPJF}. This is a direct simulation of the algorithm behavior and is straightforward.
For computing C_{RR}, we can observe that jobs will always finish computing in non-increasing size order since every job gets the same amount of processing time. So

when a job terminates, all the remaining jobs have received the same processing time as the terminating job. So we can sort by non-increasing order (requiring $O(n \log n)$ time) $x_1 \leq x_2 \leq ... \leq x_n$ and iterate through the sizes. When job J_i is completed, the total elapsed time from the job J_{i-1} is $x_i - x_{i-1}$ times the number of remaining jobs after the termination of J_{i-1}. So, after sorting, only a linear scan is required to compute C_{RR}, resulting in a total complexity of $O(n \log n)$. □

The time complexity that has just been proven shows that this evaluation requires only polynomial time. The underlying logic for adjusting the hyperparameter will ensure that dLambda does not fall behind PRR in running times, especially in real-world settings where this computation can be run during the processing times of the jobs (which in our setting are just simulated and not processed). We now show that by using this version of the algorithm, we obtain the same bounds in worst-case scenarios.

Theorem 3.2. *If we set $\theta = \lambda$ where λ is the fixed value of the hyperparameter in PRR, then dLambda performs at least as good as PRR in a worst-case scenario.*

Proof. By setting θ as a threshold for our dynamic hyperparameter, we redefine the rates for our two algorithms. In a worst-case scenario, where the predictions make the RR component perform better than SPJF, the competitive ratio of PRR is bounded by $\frac{2}{1-\lambda}$. When we set $\theta = \lambda$ to the same fixed value, our dLambda algorithm may not do worse than $\frac{2}{1-\theta}$ as it is a hard bound on the performance as we have to dedicate at least θ time to running round robin. So, in the worst case, we always do at least as good as PRR. □

By having the values of C_{RR} and C_{SPJF}, we compute the γ step. Let $\mathbb{1}_{\{\text{condition}\}}$ be an indicator function that is 1 if the condition is true and 0 otherwise. Specifically, we define:

$$\mathbb{1}_{\{C_{\text{RR}} < C_{\text{SPJF}}\}} = \begin{cases} 1 & \text{if } C_{\text{RR}} < C_{\text{SPJF}} \\ 0 & \text{otherwise} \end{cases}$$

Then, the formula can be written as:

$$\lambda \mathrel{+}= (1 - \lambda) \cdot \mathbb{1}_{\{C_{\text{RR}} < C_{\text{SPJF}}\}} \cdot \frac{C_{\text{SPJF}} - C_{\text{RR}}}{C_{\text{SPJF}}} - \lambda \cdot \mathbb{1}_{\{C_{\text{RR}} < C_{\text{SPJF}}\}} \cdot \frac{C_{\text{RR}} - C_{\text{SPJF}}}{C_{\text{RR}}}$$

This adjustment technique favors the algorithm that has performed better based on past knowledge. The adjustment is proportional to the algorithms' performance and the extent to which we already use that algorithm. This approach enables consistent improvement if the predictor works reasonably well while ensuring smooth changes without swinging too much between algorithms. Although the justification behind this technique is intuitive and not rigorously proven, we show how the performance of this algorithm is experimentally strong in Chapter 4 in real-world scenarios. In the tests, we also set θ to 0, allowing for an ample range of variations of our dynamic parameter, and we observe that the self-regulatory capabilities work consistently.

3.5 Multi-Stage Algorithm

In [11], a new algorithm is proposed to improve PRR. This algorithm stems from analyzing a new error measure that is monotonic like the L1 norm but also possesses a *Lipschitzness* property. This property ensures that an optimal solution of the predicted instance gets closer to the optimal solution of the real instance. The error measure is given by:

$$\nu(J; \{p_j\}, \{\hat{p}_j\}) = \mathrm{OPT}(\{\hat{p}_j\}_{j \in J_o} \cup \{p_j\}_{j \in J_u}) - \mathrm{OPT}(\{p_j\}_{j \in J_o} \cup \{\hat{p}_j\}_{j \in J_u}),$$

where J_o is the set of jobs with overestimated sizes and J_u is the set of jobs with underestimated sizes.
The proposed algorithm is randomized and uses median and error estimation sampling techniques. During each round, some jobs are partially executed to obtain estimates, which is necessary for the algorithm to work. These estimates are then used to decide whether to use the predictions (greedily processing jobs) or perform RR for the round. This approach introduces an algorithm that is $O(1)$-robust and $(1 + \epsilon)$-consistent. We will use the term *Multi-Stage* Algorithm, which is taken from [14], to refer to this algorithm. We will shorten it with the term NCS, abbreviating the name of the research published that introduced it [11].

Chapter 4

Tests and Results

To evaluate the efficacy of the algorithms in real-world scenarios, we use real-world datasets instead of generating the instances of jobs synthetically. In past research ([20, 14]), a common approach was generating information using a Pareto distribution with parameter $\alpha = 1.1$ as it is considered to model well the job sizes ([4, 10]). Also, we perform simulations that compute using a much wider range of jobs than the previous research. This can also allow us to see how the performance works over longer processing spans.
All the considered datasets have different features that could be used to train the jobs. Still, we consider the executable ID (EID) the main feature for ease of implementation and significance. More rigorously, two jobs are equal in predictions if they have the same EID. Having the same EID implies spawning from the same process and executing the same binary code with different parameters. It is clear why this influences the job size; thus, it is a natural approach to classification.

4.1 Datasets

In this section, we present the datasets utilized to evaluate the performance of the proposed online scheduling algorithms with machine learning predictions. The datasets were selected to cover various job scheduling scenarios, providing a robust foundation for testing and validation. The job counts indicated are the total number of valid jobs extracted for our specific problem analysis. Four of them are from the *Grid workloads archive* website [1].

1. Google Cluster Trace: We extracted approximately 17 million valid jobs from this dataset, representing a large-scale cloud computing environment with diverse job types and resource demands. It serves as a comprehensive source for understanding the behavior of scheduling algorithms under real-world conditions [9].

2. DAS2: With around 1.1 million valid jobs, the DAS2 dataset captures the job scheduling dynamics in a distributed system setting. It is provided by the Advanced School for Computing and Imaging (ASCI) [26].

3. Grid5000: This dataset contains about 770,000 valid jobs and tests scheduling algorithms in a grid computing context. Grid5000 is an experimental grid platform consisting of 9 sites geographically distributed in France, encompassing a total of 15 clusters. The dataset includes job records from the beginning of the Grid5000 project up to November 10, 2006. Grid5000 uses the OAR batch scheduler, which supports reservation capabilities and allows resource co-allocation through OARGrid [27].

4. SHARCNet: Comprising approximately 180,000 valid jobs, the SHARCNet dataset offers a glimpse into workload management in supercomputing environments. SHARCNet is structured as a "cluster of clusters" across southwestern, central, and northern Ontario, designed to meet the computational needs of researchers in diverse research areas. This trace contains up to a year's worth of accounting records from the SHARCNet clusters installed at several academic institutions in Ontario, Canada [17].

5. AuverGrid: This dataset includes around 330,000 valid jobs and represents a production grid platform in the Auvergne region, France. AuverGrid consists of 5 clusters located geographically in the area and employs the LCG middleware as part of the EGEE project (Enabling Grids for E-science in Europe). The clusters, composed of dual 3GHz Pentium-IV Xeons nodes running Scientific Linux, are primarily used for biomedical and high-energy physics research [25].

4.2 Oracles

Different types of oracles can be used for the tests. For some oracles, defining and using a training set is necessary. A training set is a set of jobs $\mathcal{T}$ of which we know precisely the sizes x_i. To obtain predictions y_i for jobs that are instead part of the test set (and so used for the evaluation), we use known features before execution. As mentioned before, we use only the EID and refer to the EID of job J_i as b_i.

4.2.1 Gaussian Perturbation

This oracle was used in [20, 14]. The prediction for a job J_i is set as $y_i = x_i + \epsilon$ where ϵ is sampled from a normal distribution with mean 0. The experiments in past papers want to observe how fast the performance degrades with worsening predictions (by increasing the standard deviation).
This oracle is not used in our tests as the research aims to evaluate the algorithms on real-world data, and this type of prediction is artificial.

4.2.2 Mean of Job Size

In real-world scenarios, submitted jobs have an identifier that we can use to estimate the size. In fact, this identifier represents the program from which the job spawned and may be significant for its size. The mean of the known sizes is one possibility. If we want to obtain a prediction for a job J_i and we have a subset of jobs $J_1, ..., J_k \in \mathcal{T}$

such that $\forall j \in \{1, ..., k\}, b_j = b_i$ we can estimate the size x_i and predict:

$$y_i = \frac{\sum_{g=1}^{k} x_g}{k} = \hat{x}_{b_i}$$

If there is no job in the training set that has the same identifier as the job J_i, we use as a prediction the mean of all the job sizes:

$$y_i = \frac{\sum_{j=1}^{n} x_j}{n} = \hat{x}$$

This prediction method returns overall good results in the tests. The mean has already been used as a predictor in the past when job similarity is present, demonstrating good results ([23]).

4.2.3 Median

It is the same approach as the mean method but with the median to mitigate the weight of outliers on the predictions. However, it experimentally did worse than the mean of the sizes, and the results are not included in this work.

4.2.4 Sampling/Lottery

With this oracle, a different approach is taken. We consider a schedule that concerns the *classes*[1] more than the jobs (basically all the time intervals assigned to the jobs of the same class are contiguous). The oracle was implemented only in a greedy fashion but a mixed version with RR can be considered.
In our version of the problem, all the jobs J_i have $a_i = 0$ and so are present at the starting time. We then perform a competition made of k rounds between the classes where we extract one job per class and give one point to the class with the smallest job. Classes without corresponding jobs in the training set use the total mean of the jobs for each extraction. At the end of the competition, the class with the most points is scheduled next. The algorithm that uses this oracle consistently did worse than SPJF, so it is not included in the tests.

4.3 Offline and Online Training Sets

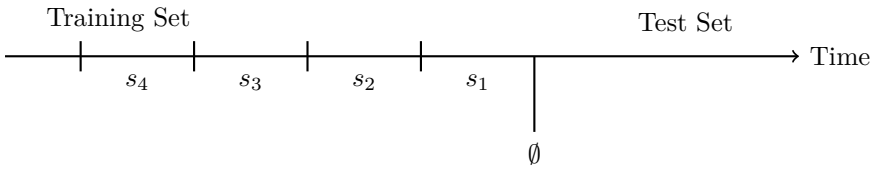
In the context of our testing framework, we defined different types of oracles, some of which require the definition and usage of a training set. There are, however, two ways in which we can make our prediction functions work.
The first method involves utilizing an **offline training set**, or in other words, compiling the training set at the start of the execution and having the prediction function completely determined at the starting time. We basically have a set of jobs executed in the past, and the set remains identical throughout the algorithm.
Another approach is updating the training set every time a job finishes. So we have at each completion of a job J_i, a new training set $\mathcal{T}_{new} = \mathcal{T}_{old} \cup \{J_i\}$. This is particularly significant for predictions when it is the first time a job with an EID b_i

[1] We are referring to a group of jobs with the same executable id as a "class".

joins the training set, and we will see how performance increases when using this approach, which we will refer to as **online training set**. This concept of *online learning* is similar to the one expressed in [13].
The enhanced performance with updated predictions achieved using simpler oracles is a convincing justification for their use. For instance, a traditional neural network would necessitate a backpropagation step to refine its predictions, thereby significantly increasing the oracle overhead, especially in our simulated environment due to the heavy computational load [30]. However, implementing such a sophisticated feature may be feasible in a real-world context where jobs are actually executed.

4.4 Test Methodologies

We perform learning experiments by taking a contiguous slice from the datasets (jobs are sorted by ascending submission time) and splitting it into training and test sets. The test set is always unknown at the starting time. In the experiments, we vary the amount of time we "look back" in the dataset. Basically, we divide the training set in m slices $s_1, ..., s_m$, with s_1 being the one contiguous to the test set. We will check how the algorithms improve their competitive ratio by increasing the training set each time, first $\mathcal{T} = s_1$, then $\mathcal{T} = s_1 \cup s_2$, at last $\mathcal{T} = \bigcup_{i=1}^{m} s_i$. Every test has a maximum training set size of two times the test set if offline or the total number of jobs both in the training and test set if online (since the test set gets eventually "absorbed").



The algorithms are simulated rather than executed in real time, as our primary focus is on the metric of total completion time. This methodology enables us to model a significant volume of jobs over prolonged durations, and the duration of our simulation is contingent solely upon the number of jobs and the intricacy of the implementation. The NCS algorithm, with its sampling activity at each round and the noticeable computational complexity of the subtasks, constrains the pace at which we can simulate it, as it is not easy to predict or streamline the execution, thus posing a bottleneck in our tests. For this reason, our tests on this algorithm are limited in size (up to a test set of 20k jobs). Also, we select for it a hyper-parameter of $\epsilon = 20$, as lower values do not allow the algorithm to use predictions, making it behave like RR. The number of slices utilized for the training set is 10. We will see that the algorithms usually converge relatively quickly to a particular competitive ratio as the chances of finding jobs spawned from the program decrease as we go back in time. All randomized algorithms will have a cloud-like representation representing the standard deviation of the ten runs on which their competitive ratio is averaged. For the greedy blind algorithm, we use only the version with the online training set, as it consistently performs better than its offline version.

4.5 Experiments

We observe that online training sets are much better at predicting future job scheduling. In fact, jobs of the same type usually come in batches, and the possibility of executing one of them allows us to correctly allocate time intervals to the rest of the class (particularly when it's the first time the class is seen in the instance). However, in real-world scenarios where we have multiple machines and real jobs to process, further investigation is required regarding whether it is advantageous to use such a technique.
We will see how our proposed variation of PRR with dynamic lambda (dLambda) performs well whether the predictions are good or bad, converging both at round robin and the greedy algorithms depending on how reliable the predictions are. The values of the competitive ratio of RR are always computed experimentally. In the legends of the plots, algorithms that adopt an online training set methodology have prepended a *d* before their abbreviation.

4.5.1 Google Cluster Trace, great predictions and consistency

In the Google Cluster Trace predictions are of good quality, and the algorithms greatly benefit from them. We will see how greedy algorithms get the upper hand in this environment.
The low standard deviation inside a job class implies that the mean of the class is actually a good representative of the overall sizes of the jobs. The plot in Figure 4.1 captures the mean and standard deviation of the classes of 10 million jobs.

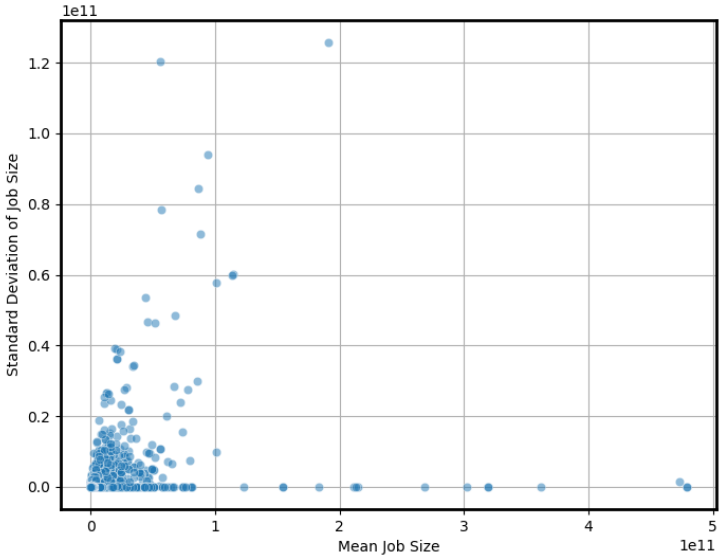


Figure 4.1. Mean versus standard deviation inside the classes of the jobs.

Classes with exact zero standard deviation are those with only one job. We can clearly see that there are a small number of cases with a high standard deviation, and

this makes using real predictions good for testing consistency rather than robustness; the latter will be addressed with other datasets.
We used dashed lines for offline training sets and dot-dash lines for the algorithms using online training sets. On the x-axis, we have the slices of the training set, while on the y-axis, the value of the competitive ratio. In the first test, we compare NCS against the other more straightforward algorithms using a schedule with 20k jobs from the dataset.

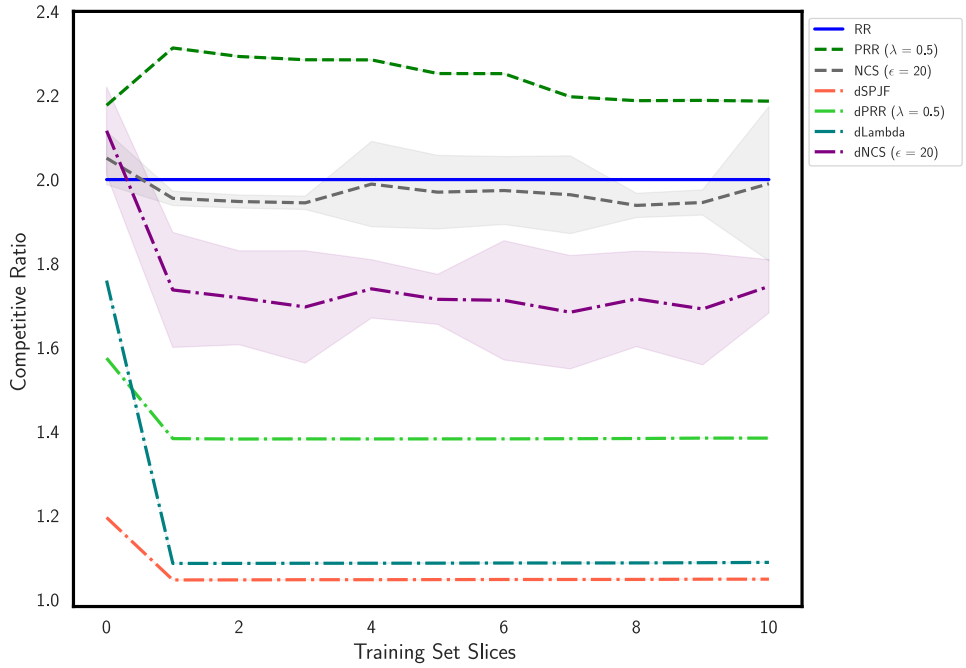


Figure 4.2. Experiment on Google Cluster Trace with a test set of 20k jobs, training set of maximum 40k jobs.

We can observe how online predictions help the algorithms perform better in this situation, as both the online training set versions of NCS and PRR are better than their offline training set counterparts. The greedy algorithm wins all over the ten slices of the training set with a competitive ratio of about 1.1, while the dLambda algorithm follows closely. It is clear (and we will see in other cases later) that the Multi-Stage algorithm doesn't capitalize on good predictions like its competitors but is more focused on robustness. We now show some other tests in Figure 4.3 with a much larger amount of jobs, removing NCS from the computation.

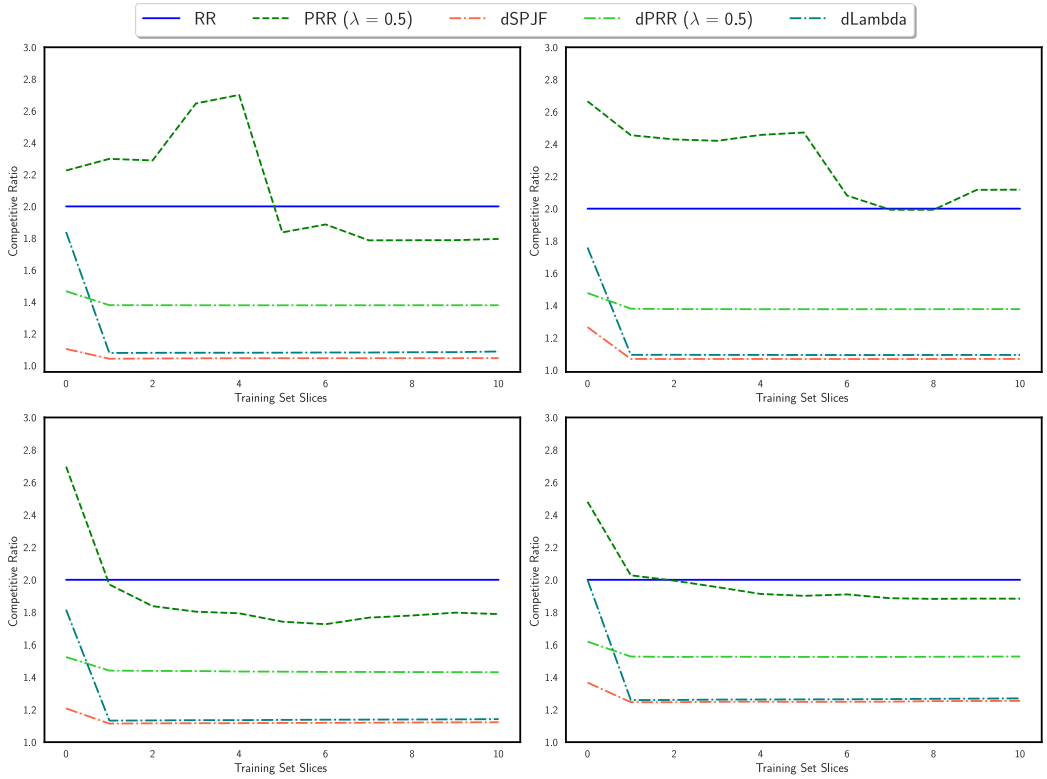


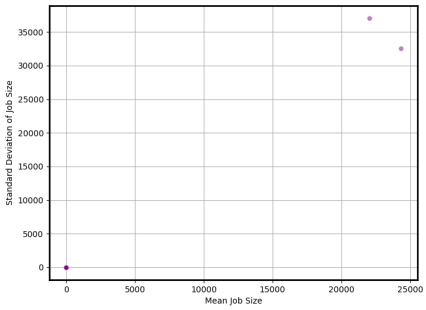
Figure 4.3. Different experiments on the Google Cluster Trace without utilizing NCS, using test sizes of 50k, 100k, 1.6M, and 3.2M, respectively.

In all the cases, it is observable how the best performance is achieved by the greedy algorithm and the dLambda algorithm. The PRR algorithm without dynamic predictions also performs worse than round robin until slice 5 in the first case of Figure 4.3, suggesting that some jobs in the test set with that particular slicing reappear much later in the timeline.
The tests on this dataset show how much good predictions can improve the performance of the algorithms in terms of total completion time, converging to the optimal with the 1-consistent scheduler. It is interesting to note how the algorithms' performance differs from the tests presented in [20], where SPJF degrades really badly in performance. The NCS algorithm performs relatively better when comparing its performance with that reported in [14], where it constantly does worse than RR in the online learning experiment. Here, with the online training set version, we beat RR but still didn't capitalize on the excellent predictions.

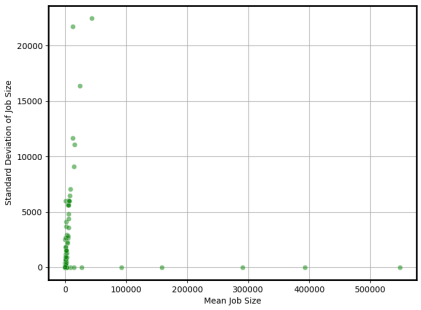
4.5.2 DAS2 & AuverGrid, bad predictions and robustness

With the DAS2 & AuverGrid datasets, the situation changes dramatically.
In AuverGrid (Figure 4.4a), there are some problems when performing predictions. There are a lot of jobs with minuscule sizes and only three classes. This makes algorithms without some RR implementation extremely lacking, as the self-reliant algorithm can rapidly "sweep" those minuscule jobs. Since we are dealing with

the unweighted version of the total completion time problem, those jobs are as important as the larger ones and should be dealt with immediately. The scatter plots of the classes give a general overview of the situation regarding mean and standard deviation.



(a) Mean versus standard deviation inside the classes of the jobs in AuverGrid.



(b) Mean versus standard deviation inside the classes of the jobs in DAS2.

In DAS2 (Figure 4.4b), the classes present a high relative standard deviation, further hindering the predictions. This is really damaging to the performance since the prediction is shared throughout the class. If the similarity between the jobs is low, then we may get results similar to those of random scheduling.
From the plots of the experiments, we can observe how the different algorithms deal with bad predictions.

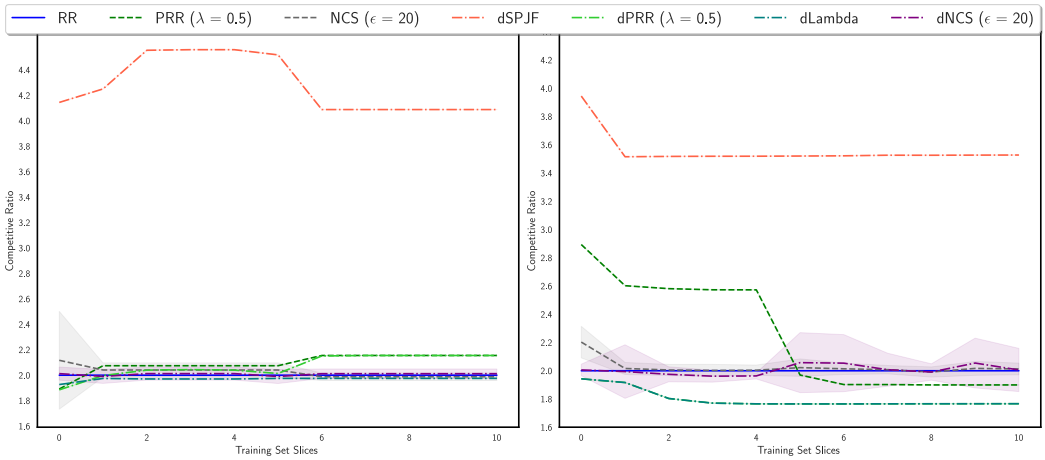


Figure 4.5. Experiments with NCS algorithm, 20k jobs in the test set, up to 40k in the training set. On the left is the experiment with the AuverGrid dataset, and on the right is DAS2.

In the DAS2 part of the experiment (Figure 4.5), most algorithms performed worse than the baseline RR. However, dLambda achieved a competitive ratio of around 1.8, outperforming RR.
When using the AuverGrid dataset, most algorithms achieved competitive ratios similar to RR, except for the greedy algorithm, which shows how poor the predictions

were.

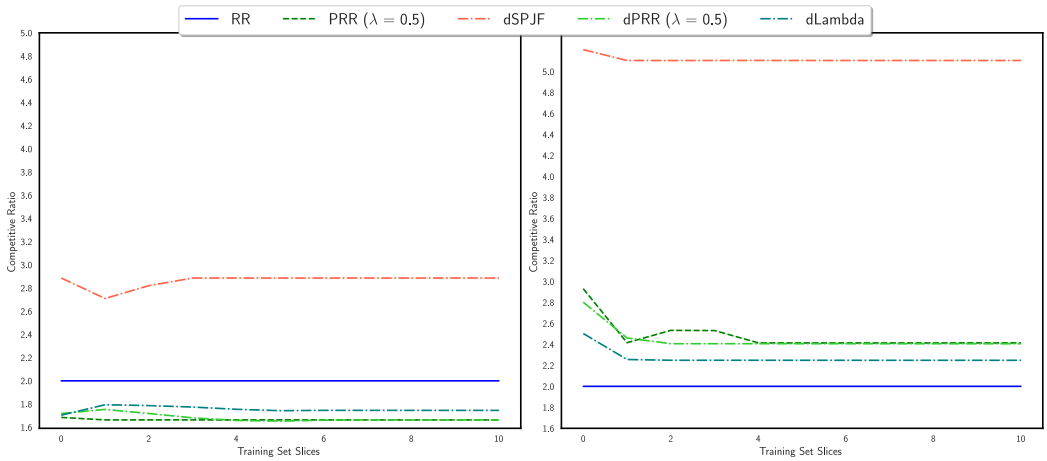
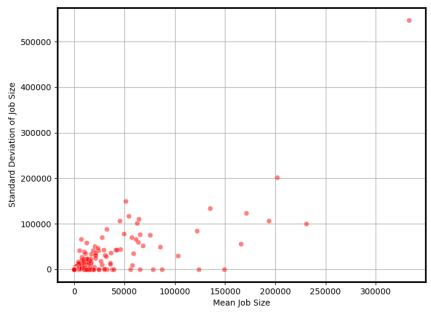


Figure 4.6. Experiments without the NCS algorithm, 100k jobs in the test set for AuverGrid and 200k for DAS2. On the left is the experiment with the AuverGrid dataset, and on the right is DAS2.

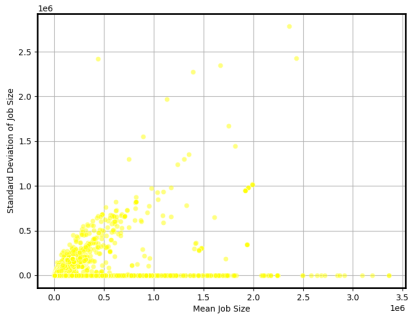
The algorithms made inaccurate predictions in the DAS2 test without NCS (Figure 4.6). RR achieved the best competitive ratio, 2, followed by the dLambda algorithm, which keeps proving its robustness throughout the tests.
The performance in AuverGrid is slightly anomalous, where the greedy algorithm is performing badly, dLambda is doing slightly better than RR, and the two versions of PRR have the best competitive ratios.

4.5.3 Grid5000 & SHARCNet, average predictions

In the datasets Grid5000 and SHARCNet, the predictions are of average quality. The plots of the classes follow a structure similar to the one of Google in Figure 4.1, but more sparse and slightly higher standard deviations.



(a) Mean versus standard deviation inside the classes of the jobs in Grid5000.



(b) Mean versus standard deviation inside the classes of the jobs in SHARCNet.

We perform the tests on the datasets with the NCS algorithm .

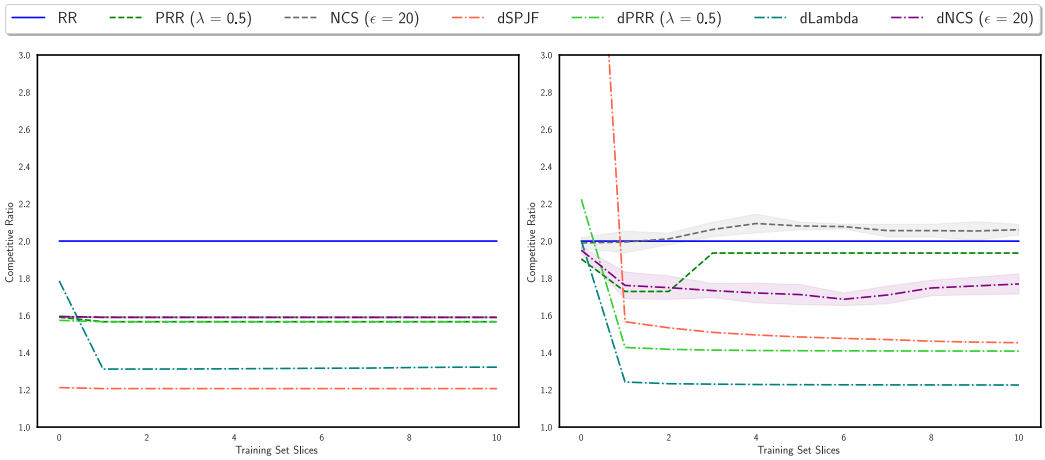


Figure 4.8. Experiments with NCS algorithm, 20k jobs in the test set, up to 40k in the training set. The dataset on the left is Grid5000, while the one on the right is SHARCNet.

In the Grid5000 test (Figure 4.8 on the left), NCS's performance remains relatively constant for both online and offline training sets. This behavior is similar to the findings in [14] in terms of indifference with respect to learning, although NCS performs much better in terms of competitive ratio. Additionally, there is little to no standard deviation across the ten runs.
In the test with SHARCNet (Figure 4.8 on the right), the NCS algorithm has an average performance. We can see that the predictions are bad at first, but they improve with the following training set slices. The dLambda algorithm achieves the best competitive ratio.

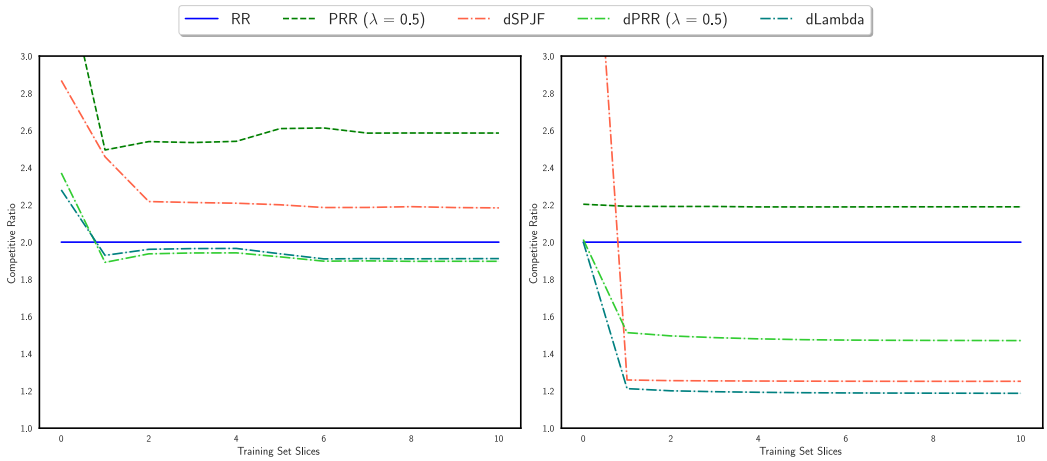


Figure 4.9. Experiments without NCS algorithm, 200k jobs in the test set for Grid5000 (left) and 50k for SHARCNet (right).

In the test without NCS (Figure 4.9 on the left), dLambda and PRR with an online training set perform optimally in the Grid5000 dataset. This happens regardless of the somewhat bad predictions signaled by SPJF's performance.
In the test with SHARCNet (Figure 4.9 on the right), dLambda shows optimal performance, surpassing the greedy algorithm's performance. It can also be observed that the offline predictions here are not enough to guarantee a good schedule.

Chapter 5

Conclusion

This thesis has explored the integration of machine learning predictions with online algorithms to enhance their performance in scheduling problems using real-world datasets from academic and industrial environments. The primary objective was to address the challenge of improving decision-making in online scheduling and how the theoretical bounds in competitive ratio correlate to actual performance. Our investigation was exclusively interested in minimizing the total completion time of jobs.

The study began with a comprehensive review of existing online algorithms and their competitive ratios, highlighting their strengths and limitations. This review provided a foundational understanding of the current landscape of online scheduling algorithms. Building on this foundation, we introduced a new variation of an online scheduling algorithm. We provided a worst-case bound for the performance of this algorithm. Through extensive simulations, we evaluated the proposed algorithms. The results showed that our algorithm variation outperforms existing methods in various scenarios, both in the case of good and bad predictions, and that in real-world scenarios, the performance of algorithms with greedier approaches fare better than in theoretical settings with artificial errors.

Future research could further refine prediction models to enhance their accuracy and reliability. Another interesting aspect is applying the simulations on past knowledge in other algorithms to estimate prediction performance and adjust possible pre-set hyper-parameters. To further create credibility for those online algorithms, real-world experimentation with jobs requiring real processing power is crucial.

Bibliography

[1] Grid workloads archive. `http://gwa.ewi.tudelft.nl/tools/`. Accessed: 2024-05-19.

[2] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1):203–218, 2000.

[3] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 571–580, New York, NY, USA, 1992. Association for Computing Machinery.

[4] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: investigating unfairness. *SIGMETRICS Perform. Eval. Rev.*, 29(1):279–290, jun 2001.

[5] Shai Ben-David, Allan Borodin, Richard Karp, Gábor Tardos, and Avi Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2–14, 01 1994.

[6] Michael Bender and William Kuszmaul. *Randomized Cup Game Algorithms Against Strong Adversaries*, pages 2059–2077. 01 2021.

[7] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, 1998.

[8] Pavel Čížek, Wolfgang Härdle, Rafał Weron, and Wolfgang Härdle. *Statistical tools for finance and insurance.* Springer, 2011.

[9] Google. Google cluster data v2. `https://github.com/google/cluster-data`, 2011. Accessed: 2024-05-19.

[10] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, aug 1997.

[11] Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. Non-clairvoyant scheduling with predictions. *ACM Trans. Parallel Comput.*, 10(4), dec 2023.

[12] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. Chapter 9 sequencing and scheduling: Algorithms and

complexity. In *Logistics of Production and Inventory*, volume 4 of *Handbooks in Operations Research and Management Science*, pages 445–522. Elsevier, 1993.

[13] Bi Li, Wenxuan Xie, Wenjun Zeng, and Wenyu Liu. Learning to update for object tracking with recurrent meta-learner. *IEEE Transactions on Image Processing*, 28(7):3624–3635, 2019.

[14] Alexander Lindermayr and Nicole Megow. Permutation predictions for non-clairvoyant scheduling, 2022.

[15] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice, 2020.

[16] Andrés Muñoz Medina and Sergei Vassilvitskii. Revenue optimization with approximate bid predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1856–1864, Red Hook, NY, USA, 2017. Curran Associates Inc.

[17] John Morton and Clayton Chrusch. Gwa-t-10 (sharcnet), n.d. Accessed: 2024-05-20.

[18] Rajeev Motwani, Steven Phillips, and Eric Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[19] Ramtin Pedarsani, Mohammad Ali Maddah-Ali, and Urs Niesen. Online coded caching. *IEEE/ACM Transactions on Networking*, 24(2):836–845, 2016.

[20] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ml predictions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[21] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[22] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, feb 1985.

[23] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64(9):1007–1016, 2004.

[24] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. 12 2013.

[25] AuverGrid Team. Gwa-t-4 (auvergrid), n.d. Accessed: 2024-05-20.

[26] DAS-2 Team. Gwa-t-1 (das-2), n.d. Accessed: 2024-05-20.

[27] Grid'5000 Team. Gwa-t-2 (grid'5000), n.d. Accessed: 2024-05-20.

[28] Mengyu Wang, Shuyong Zhu, and Yujun Zhang. A heuristic online algorithm for routing in large-scale deterministic networks. In *2023 24st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 189–194, 2023.

[29] Alexander Wei and Fred Zhang. Optimal robustness-consistency trade-offs for learning-augmented online algorithms. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8042–8053. Curran Associates, Inc., 2020.

[30] Jake Ryland Williams and Haoran Zhao. Reducing the need for backpropagation and discovering better optima with explicit optimizations of neural networks, 2023.

Acknowledgments

First and foremost, I would like to extend my heartfelt thanks to my supervisor, Alessandro Panconesi, for suggesting this topic and to PhD candidate Mirko Giacchini for his support during the experiments and in writing this thesis. I thank my family for their unwavering support in my studies and journey. I am grateful to Alex, who has always been by my side and a supporter of all my decisions. Lastly, I thank my friends for the carefree moments and the happiness they give me.