Combinatorial Challenges and Algorithms in New Energy Aware Scheduling Problems

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy by

Hsiang-Hsuan Liu

June 2017

Contents

Illustrations

List of Figures

List of Tables

Notations

The following notations and abbreviations are found throughout this thesis:

$\boldsymbol{\mathcal{J}}$ A set of input jobs $\mathcal{J} = \{J_1, J_2, J_3, \cdots\}$.
$\boldsymbol{w(J)}$ Time duration (width) of job J.
$\boldsymbol{h(J)}$ Power request (height) of job J.
$\boldsymbol{I(J)}$ Feasible timeslots of J. The set of available timeslots where job J can be executed. If $I(J)$ is a contiguous interval, we call it the feasible interval of J.
$\boldsymbol{r(J)}$ The earliest time when job J can be executed.
$\boldsymbol{d(J)}$ The latest time by then job J has to be finished.

$\boldsymbol{\ell oad(S, t)}$ The load at timeslot t in the S schedule.
$\mathbf{cost}(\boldsymbol{S})$ The total cost of the schedule S.
$\boldsymbol{st(\mathcal{A}, J)}$ The start time of job J in the S schedule.
$\boldsymbol{et(\mathcal{A}, J)}$ The end time of job J in the S schedule.

DVS The dynamic voltage/speed scaling problem
BINPACKING The bin packing problem
PARTITION The partition problem
$\boldsymbol{\mathcal{AVR}}$ The AVR algorithm of the DVS problem.
$\boldsymbol{\mathcal{BKP}}$ The BKP algorithm of the DVS problem.

Preface

A significant part of this thesis is based on three peer-reviewed papers that have been published in international conferences and a journal. The four papers have been adapted for the purpose of this thesis and expanded to contain work that was omitted from the conference versions of the papers. The extended versions of the papers are also either under submission for journal publication or in preparation for submission. An additional chapter presents work that has not yet been published that is related to one of the above papers.

Specifically, Sections 4.1 and 4.2 of the thesis are based on the paper entitled "Scheduling for Electricity Cost in Smart Grid", co-authored with Mihai Burcea, Wing-Kai Hon, Prudence W.H. Wong and David K. Y. Yau. The paper has been published in Proceedings of the 7th Annual International Conference on Combinatorial Optimization and Applications. The journal version is published in the Journal of Scheduling 2016.

Sections 4.3, 4.4 and Chapter 5 of the thesis are based on the paper entitled "Optimal Nonpreemptive Scheduling in a Smart Grid Model", co-authored with Fu-Hong Liu and Prudence W.H. Wong. The paper has been published in Proceedings of the 27th International Symposium on Algorithms and Computation.

Finally Chapter 6 represents a continuation of the work in Chapters 4 and 5 that focuses on other optimization problems. We show how to solve them by adapting our techniques and prove that our online algorithm can solve the machine minimization problem with an asymptotically optimal competitive ratio. In this chapter we also show that our exact algorithm can be adapt to solve other demand response management problems.

Abstract

In this thesis, we study the theoretical approach on energy-efficient scheduling problems arising in demand response management in the modern electrical smart grid. Consumers send in power requests with flexible feasible timeslots during which their requests can be served. The grid controller, upon receiving power requests, schedules each request within the specified interval. The electricity cost is measured by a convex function of the load in each timeslot. The objective is to schedule all requests with the minimum total electricity cost.

We study the smart grid scheduling problem in different models. For the offline model, we prove the problem is NP-hard for the general case. We propose a polynomial time algorithm for special input where jobs have unit power request and unit time duration. By adapting the polynomial time algorithm for unit-size jobs, we propose an approximation algorithm for more general input. On the other hand, we also present an exact algorithm to find the optimal schedule for the problem with general input.

For the online model, we propose an online algorithm for jobs with jobs with arbitrary power request, arbitrary time duration, and arbitrary contiguous feasible intervals. We also show a lower bound of the competitive ratio for the smart grid scheduling problem with unit height and arbitrary width. For special cases, we design different online algorithms with better competitive ratios.

Finally, we look at other optimization problems and show how to solve them by adapting our techniques. We prove that our online algorithm can solve the machine minimization problem with an asymptotically optimal competitive ratio. We also show that our exact algorithm can be adapted to solve other demand response management problems.

Acknowledgements

This project is a long journey. This journey would not be completed without the help of people around me. In this small section of acknowledgments, I would like to use this opportunity to show my gratitude.

I would like to express my immense gratitude to my supervisor, Prudence W.H. Wong. It is my pleasure of working with her. She has widened my view and shown me the beauty of problem-solving, critical thinking, and conciseness. This thesis would not have been possible without her wise mentorship and guidance. I am also grateful for Prudence's kind help and suggestions in many things throughout these years.

I also want to thank my supervisor in NTHU, Wing-Kai Hon, for his great support. Under his guidance and encouragement I can pursue many different research topics which I am interested in.

A great aid in evaluating my progress throughout the years was provided by my advisors, Paul Spirakis and Michele Zito. I am very greatfull for their suggestions on improving and expanding the work in this thesis.

Furthermore, I want to thank Ton Kloks, with whom I started the systematic research. Through him I can take a glimpse at how a decent researcher would be. I also want to thank my collaborators for material in this thesis, Mihai Burcea, Fu-Hong Liu, and David K. Y. Yau. I enjoyed the discussion with them very much and learned a lot from them.

I would like to thank my examiners, George Mertzios and Paul Spirakis. I am very grateful for their thoroughness and advice in revising this thesis to its final version.

Finally, I am going to thank Fu-Hong Liu for his constant support, encouragement, and unconditional love. To me, he is the light in the darkest time.

Chapter 1

Introduction

This thesis is a theoretical study on energy-efficient scheduling problems arising in "demand response management" in the modern electrical smart grid [25, 33, 38, 59, 84]. The electrical smart grid is one of the major challenges in the 21st century [23, 78, 79]. The smart grid [26, 62] is a power grid system that makes power generation, distribution, and consumption more efficient through information and communication technologies against the traditional power system. By the ability of communication, the smart grid management system is able to provide advanced management, improve energy efficiency and reduce cost [25].

There are many important issues in the research on smart grid [25]. For instance, the infrastructure of smart grid in which the energy can be monitored, the communication technology, the privacy protection, etc. Also, there are various management objectives like improving energy efficiency, balancing demand and supply, cost reduction, utility maximization, reducing energy consumption, price stabilization, etc.

This thesis focuses on the *demand response management* [11, 44, 58, 61, 64, 72] of smart grid. We consider the research problem as the following. Consumers send in power requests with a set of flexible feasible time intervals during which their requests can be served. The grid controller, upon receiving power requests, schedules each request within the specified interval. The electricity cost is measured by a convex function of the load in each timeslot. The objective is to schedule all requests with the minimum total electricity cost. We consider both offline and online settings and aim to minimize the total cost in the worst case.

1.1 Smart Grid Scheduling and Demand Respond Management

Unlike the traditional power grid where the power generator is centralized and the power needs to be transmitted over long distance to users, the smart grid allows distributed generation and uses the bi-directional flow of electricity. By using information and communication technologies in an automated fashion, the smart grid is able to improve the efficiency and reliability of production and distribution of electricity. The cost of generating electrical power has several factors such as fuel cost, heat rate, waste disposal. Different power sources have different generating cost. Usually the power company will use the cheaper power sources before using the more expensive one. If too many tasks are issued at the same time, the total power demand at the time might exceed the amount which the cheapest power source can supply. Hence a more expensive source have to be employed, and it increases the cost. Research initiatives in the area include [40, 57, 70, 77]. *Peak* demand hours happen only for a short duration, yet make existing electrical grid less efficient. It has been noted in [12] that in the US power grid, 10% of all generation assets and 25% of distribution infrastructure are required for less than 400 hours per year, roughly 5% of the time [79].

Demand response management. By communicating between producers and consumers and making decisions about when and how much the power should be generated, the smart grid can improve the efficiency of electricity generators. According to the information revealed by the *demand profile*, which is a curve of demand/electrical load over time, power company can plan how much power they need to generate to satisfy the requests from the consumers at any time. *Demand response management* is changing the demand profile to match the supply better by shifting requests to different time. It can reduce the peak load and avoid emergency to shift the demands of users from on-peak hours to off-peak hours [11, 44, 58, 61, 64, 72].

The electricity grid supports demand response mechanism and obtains energy efficiency by organizing consumption of electricity in response to supply conditions. It is demonstrated in [59] that demand response is of remarkable advantage to consumers, utilities, environment, and society. From the viewpoint of the system operator, effective demand load management brings down the cost of operating the grid, while it reduces electricity prices for users. It is also beneficial for utility providers to keep aggregate power demand as flat as possible since this lowers the cost as well as energy generation and distribution [58]. Considering the whole environment, demand response has the ability to minimize the cost for generating electricity and significantly reduces carbon emissions [59]. Demand response management is not only advantageous to the supplier but also to the consumers as well. It is common that electricity supplier charges according to the generation cost, i.e., the higher the generation cost the higher the electricity price. Therefore, it is to the consumers' advantage to reduce electricity consumption at a high price and hence reduce the electricity bill [72].

The smart grid operator and consumers communicate through smart metering devices [46, 62]. A consumer sends in a power request with the power requirement (cf. height of request), required duration of service (cf. width of request), and the time interval that this request can be served (giving some flexibility). For example, a consumer may want the dishwasher to operate for two hours during the periods from 10 a.m. to 12 noon or 2 p.m. to 5 p.m., and the washing machine to operate for one hour during the periods from 9 a.m. to 1 p.m. (see Figure 1.1). The grid operator upon receiving requests has to schedule them in their respective time intervals using the minimum electricity cost. The *load* of the grid at each timeslot is the sum of the power requirements of all requests allocated to that timeslot. The *electricity cost* is modeled by a convex function on the load, in particular, we consider the cost to be the α-th power of the load, where $\alpha > 1$ is an arbitrary real number. In practice, α is a small constant, e.g., $\alpha = 2$ [21, 73].

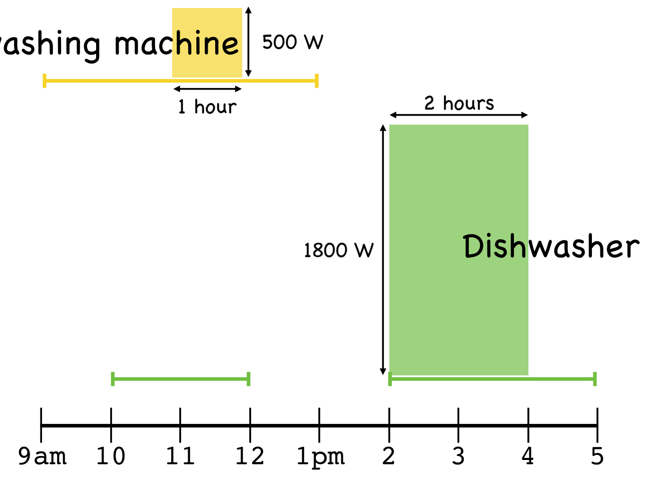


Figure 1.1: An illustration of demand response management.

1.2 Our contribution

Previously, Koutsopoulos and Tassiulas [45] have formulated a similar problem to our problem where the cost is an arbitrary convex function of the load. They claimed that this problem is NP-hard by proving the smart gird scheduling problem with minimizing peak object is NP-hard first, and proposed algorithms to minimize the total cost over the time horizon. For the offline setting, the authors gave an exact algorithm for the preemptive case and claimed that the non-preemptive case is NP-hard. For the online setting, the authors proposed a stochastic model and gave two strategies to minimize the long-term average cost. Their main contribution, *Controlled Release strategy*, is based on referencing a threshold power level. Given the Poisson distribution of jobs and the cost function, the threshold power level can be decided by running experiments. By deriving a lower bound, the authors proved that as the lengths of feasible intervals increase to infinity, there exists an optimal threshold such that the Controlled Release strategy guarantees asymptotically optimal expectation of total cost under the stochastic model.

Our contribution. We focus on the worst case analysis of the smart grid scheduling problem. Moreover, we consider the case where there is no knowledge of the distribution of release times, deadlines, widths and heights. We first show that this problem is strongly NP-hard, even for very restricted input set or preemptive case. However, we find that for special input where each job has unit power request and unit time duration, the smart grid scheduling problem can be solved efficiently. We propose a polynomial time offline algorithm that gives an optimal solution and show that the time complexity of the algorithm is $O(n^2\tau)$, where n is the number of jobs and τ is the number of timeslots. We further show that if the feasible timeslots for each job to be served form a contiguous interval, we can improve the time complexity to $O(n \log \tau + \min(n, \tau) n \log n)$. By employing an existing algorithm for the discrete dynamic voltage/speed scaling problem (see Section 3.2.2), we show that the time complexity can be further improved to $O(n \log n)$.

For more general input set where jobs have arbitrary widths, arbitrary heights, and arbitrary contiguous feasible intervals, we use special graphs to represent the jobs and the important notion maximal cliques to partition the time horizon into disjoint windows. The special corresponding intersection graph is an *interval graph*, which has special properties. These properties direct us to a dynamic programming approach to find the optimal schedule. We propose two exact algorithms; both are fixed-parameter algorithms. By these two fixed-parameter algorithms, we show that the smart grid scheduling problem is fixed-parameter tractable with respect to the maximum width of jobs, and the maximum number of overlapped feasible intervals. That is, when these parameters are constant, the grid problem is no longer NP-hard.

For the general input, we also propose a $36^\alpha \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)^\alpha \cdot (1 + \lceil \log \frac{h_{\max}}{h_{\min}} \rceil)^\alpha$-approximation algorithm by making use of the optimal scheduling algorithm for unit-size jobs, where $w_{\max}$, $w_{\min}$, $h_{\max}$, and $h_{\min}$ are the maximum time duration, minimum time duration, maximum power request, and minimum power request of the input jobs. Comparing with the exact algorithm, the approximation algorithm is more efficient.

For the smart grid problem in online model, we propose the first online algorithm for the general input with worst case competitive ratio, which is polylogarithmic in the max-min ratio of the duration of jobs. Our online algorithms are based on identifying a relationship with the dynamic speed/voltage scaling (DVS) problem. We first propose a $2^\alpha(8e^\alpha + 1)$-competitive algorithm for jobs with unit time duration and a $12^\alpha(8e^\alpha + 1)$-compeititve algorithm for jobs with uniform time duration. It means that when α is a constant, both algorithms are constant competitive. By generalizing the result, we present a $36^\alpha \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)^\alpha \cdot (8e^\alpha + 1)$-competitive algorithm for general input. The interesting thing is, our online algorithm and exact algorithms depend on the variation of the job widths but not the variation of the job heights.

On the other hand, we prove that for any deterministic online algorithm, the competitive ratio is at least $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^\alpha$. The lower bound on the competitive ratio is proven by an adversary where the input is a set jobs with uniform power request. For special cases of feasible intervals, there are online algorithms which perform better. For jobs with unit time duration, common feasible interval, we propose a 2^α-competitive algorithm. For jobs with uniform power request, common release time and common deadline, we propose a $2^{2\alpha}$-competitive algorithm. For jobs with agreeable deadlines (which means the jobs released later would have later deadlines) and uniform power request, we prove that a next-fit approach is $(\frac{(8\alpha)^\alpha}{2} + 2^\alpha)$-competitive. All these online algorithms for special input are $O(1)$-competitive when α is a constant. The results are listed in Table 1.1 (K_w is defined as $1 + \lceil \frac{w_{\max}}{w_{\min}} \rceil$, where $w_{\max}$ and $w_{\min}$ are the maximum and minimum widths among all jobs; similarly, K_h is defined as $1 + \lceil \frac{h_{\max}}{h_{\min}} \rceil$, where $h_{\max}$ and $h_{\min}$ are the maximum and minimum heights among all jobs.).

The techniques we use to solve the smart grid scheduling problem are adaptable for other optimization problems. By adapting our online algorithm, we can solve the online machine minimization problem and the peak minimization problem in smart grid optimally in an asymptotic sense. We also elaborate how to use our fixed parameter algorithm to solve the machine minimization problem.

Table 1.1: Our contribution on minimizing total cost in smart grid

Model	Width	Height	Performance	Chapter
Offline exact	Unit	Uniform	Contiguous feasible interval $O(n \log \tau + \min\{n, \tau\} n \log n)$ time	4.2
			$O(n^2\tau)$ time	4.2
			Contiguous feasible interval $O(n \log n)$ time	4.2.4
	Arbitrary	Arbitrary	Three parameters ${w_{\max}}^{2m} \cdot (W_{\max} + 1)^{4m} \cdot O(n^2)$ time	4.3.3
			Two parameters $(4m \cdot {w_{\max}}^2)^{2m} \cdot O(n^2)$ time	4.3.4
Offline approx.	Arbitrary	Arbitrary	$(36K_wK_h)^\alpha$-approximation	4.4
Online	Unit	Uniform	$\min\{\frac{(4\alpha)^\alpha}{2} + 1, 2^\alpha(8(e + e^2)^\alpha + 1)\}$-compet.	5.3.1
	Unit	Arbitrary	$2^\alpha(\min\{\frac{(2\alpha)^\alpha}{2}, 8(e + e^2)^\alpha\} + 1)$-compet.	5.1.1
			Common feasible interval 2^α-competitive	5.3.4
	Uniform	Arbitrary	$12^\alpha \cdot (\min\{\frac{(2\alpha)^\alpha}{2}, 8(e + e^2)^\alpha\} + 1)$-compet.	5.1.2
	Arbitrary	Uniform	Agreeable deadlines $(\frac{(8\alpha)^\alpha}{2} + 2^\alpha)$-competitive	5.3.2
			Common feasible interval $2^{2\alpha}$-competitive	5.3.3
	Arbitrary	Arbitrary	$(36K_w)^\alpha(\min\{\frac{(2\alpha)^\alpha}{2}, 8(e + e^2)^\alpha\} + 1)$-compet.	5.1.3

The differences between our results and the results in [45]. The main difference is that, in the online model, our goal is to guarantee the worst case performance for the case where and the jobs without knowledge of release times, deadlines, widths, and heights, while in [45], the authors established a stochastic model of the jobs and tried to minimize the longterm average cost. Also, in our work we focus on the cost function which is a power function of the load at any time, whereas in [45] the cost function can be an arbitrary convex function.

For the NP-hardness, the authors in [45] showed it by reducing the bin packing problem to a smart grid problem where jobs have heights and the objective is to minimize the maximum power consumption over time. By claiming that minimizing the maximum power consumption in the time horizon is equivalent to minimizing the total convex cost in the horizon, the authors claimed that the smart grid problem is NP-hard. Differently, we prove that the smart grid problem is NP-hard by reducing the 3-partition problem directly to the smart grid problem where the objective is to minimize the total cost.

1.3 Organization of the Thesis

The thesis is dedicated to the design and analysis of offline and online algorithms for the smart grid scheduling problem. The work in this thesis is mainly based on the following publications:

1. Mihai Burcea, Wing-Kai Hon, Hsiang-Hsuan Liu, Prudence W. H. Wong, David K. Y. Yau: Scheduling for Electricity Cost in Smart Grid. *The 7th Annual International Conference on Combinatorial Optimization and Applications (COCOA)*, 2013: 306-317 ([9])

2. Wing-Kai Hon, Hsiang-Hsuan Liu, Prudence W.H. Wong: Online Nonpreemptive Scheduling for Electricity Cost in Smart Grid. *The 12th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP)*, 2015: 193–195 ([35])

3. Mihai Burcea, Wing-Kai Hon, Hsiang Hsuan Liu, Prudence W. H. Wong, David K. Y. Yau: Scheduling for electricity cost in a smart grid. *Journal of Scheduling* 19(6): 687-699 (2016) ([10])

4. Fu-Hong Liu, Hsiang-Hsuan Liu, Prudence W.H. Wong: Optimal Nonpreemptive Scheduling in a Smart Grid Model. *The 27th International Symposium on Algorithms and Computation (ISAAC)*, 2016: 53:1-53:13 ([54])

Chapter 2 gives preliminaries about algorithms, tractability, and offline/online models. We also mention some commonly used terminologies in scheduling area. We formally define the smart grid scheduling discussed in this thesis.

Chapter 3 looks into a detailed background and history of the smart grid scheduling problem. We also elaborate relating scheduling problems and algorithms, including dynamic voltage speed/voltage scaling problem (DVS), bin packing problem, machine minimization problem, and load balancing problem. We compare the smart grid scheduling problem and these more classical scheduling problems; we also make contrast and show the difficulties of adapting the existing solutions. In this chapter, we also discuss graph algorithms and interval graphs, a special graph class. The graph algorithms give us directions to solve the smart grid scheduling problem with special input, and hence we can show that the grid problem with special input is polynomial-time solvable. On the other hand, the special properties of interval graphs give us a way of tickling the gird problem with more general input.

Chapter 4 discusses offline smart grid scheduling problem. First, we prove that the smart grid scheduling problem is NP-hard even when the input is very restricted or preemption is allowed. However, we show that for a special input set with unit-size jobs (that is, each job has unit power request and unit time duration), the optimal schedule can be found in polynomial time. The main idea is using a graph structure to capture all possible assignments of the current input jobs. The results were presented in [9] and the journal version [10]. For the unit-size jobs, we also suggest a faster algorithm by employing the results from the discrete DVS problem (Section 4.2.4.) This result is new and has not published in proceedings. For the smart grid scheduling problem with more general input, we give both approximation algorithms and exact algorithms. A simple approximation algorithm for jobs with unit time duration was presented in [10], and we generalize it to deal with the general input in this thesis (Section 4.4). The approximation algorithms use a classification technique and the polynomial time algorithm for the unit-size jobs. On the other hand, the exact algorithms (Section 4.3) are based on the observation of interval graphs. By the exact algorithms, we also prove the smart grid scheduling problem is fixed-parameter tractable with respect to the maximum width of jobs and the maximum number of jobs with overlapped feasible intervals. The results about the exact solution for general input were presented in [54].

Chapter 5 considers the smart grid scheduling problem under the online model. We present an online algorithm for general input with competitive ratio $36^\alpha \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)^\alpha \cdot (\min\{\frac{(2\alpha)^\alpha}{2}, 8(e(1+e))^\alpha\} + 1)$ (Section 5.1). The online algorithm is based on the one for unit power request input and uniform power request. We also prove that for any deterministic online algorithm, the competitive ratio is at least $(\frac{\alpha}{3})^\alpha$ for any constant α and $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^\alpha$ for arbitrary α. The results were presented in [54]. Furthermore, we investigate online algorithms for special input. We show that when input jobs have restricted width, height, or feasible intervals, there are online strategies with better performance (Section 5.3). The results about online algorithms for special input were presented in [35] and some of the results are further improved in this thesis.

Chapter 6 contains other work which has not been published in proceedings. We show how to use the techniques in this thesis to solve other problems like peak minimization problem in smart grid model and machine minimization problem. We also discuss the smart grid scheduling problem under limited power environment. For the approach to finding an exact solution using interval graphs properties, we investigate some other problems which are able to solve using this framework.

Chapter 7 gives concluding remarks. We also propose future directions for the work.

Chapter 2

Preliminaries and Definitions

In this thesis, we mainly investigate the theoretical approach to a smart grid scheduling problem. In the smart grid scheduling problem, each job associates with time duration, power request, and feasible timeslots where it can be executed. The aim is to execute the jobs within their feasible timeslots with the minimum sum of cost at every timeslot, which is convex in the power request at the timeslot. Our aim is to study the smart grid scheduling algorithms which guarantee worst case performance. We consider different models. In the offline model, the algorithms know the whole set of input jobs in advance while in the online model jobs arrive in an online manner and the algorithms have to make decisions without knowledge of the future input.

In this chapter, we give preliminaries about algorithms, tractability, and offline algorithms including approximation algorithm and exact algorithm in Section 2.1. In Section 2.2, we give an introduction to online algorithms and competitive analysis. We also introduce some commonly used terminologies in scheduling area in Section 2.3 which is widely used throughout this thesis. In Section 2.4, we formally define the smart grid scheduling problem discussed in this thesis.

2.1 Offline algorithms and class NP

Offline algorithms are algorithms with complete knowledge of the input for the problem in advance. For offline algorithms, one of the measurements of performance of an algorithm is its *time complexity*. A problem is said to be in class P if it is solvable in polynomial (in the size of the input) time. A problem is in the class NP if any of its yes instance (that is, an instance such that the answer of the decision problem is "yes") is verifiable in polynomial time. The problems in class P are also in class NP since they are naturally polynomial time verifiable for any yes instance. A problem is *NP-hard* if all problems in class NP can be reduced to it in polynomial time, even though the problem itself may not be in NP. An NP-hard problem is *NP-complete* if it is in NP.

There are many problems that have been proved to be in class NP. Some very classical ones are partition problem, bin packing problem, the decision version of traveling salesman problem, the decision version of the 0/1-knapsack problem, etc. [30]. Most of the optimization problems, including the smart grid scheduling problem, are NP-hard. We introduce the partition problem and bin packing problem:

The PARTITION problem. Given a set of integers $A = \{a_1, a_2, \cdots, a_n\}$, decide if there exists a subset of A such that the sum of integers in the subset is equal to half of the summation of all integers.

The BINPACKING problem. Given a set of items $A = \{a_1, a_2, \cdots, a_n\}$ and infinite supply of bins each with capacity c, decided if there exists a number of B bins to pack the items such that the total size of the items in a bin does not exceed the bin capacity.

A problem might have numerical parameters. For example, the magnitudes of the integers in the input to the PARTITION problem and the size of items and the capacity of bins in the BINPACKING problem. The numerical parameters are part of the input to the problem. For any NP-complete problem, there is no polynomial (in the input size) time algorithm unless P=NP. If there exists an algorithm for the NP-hard/complete problem whose running time is pseudo-polynomial in the input, the problem is said to be *weakly NP-hard/complete* (*NP-hard/complete in the weak sense*). On the other hand, a problem is *strongly NP-hard/complete* (*NP-hard/complete in the strong sense*), if it remains NP-hard/complete even when all the numerical parameters are polynomial in the input size. The BINPACKING problem is strongly NP-complete, while the PARTITION problem is weakly NP-complete.

2.1.1 Approximation algorithms

To solve NP-complete or NP-hard problems efficiently, the optimality might be sacrificed. In other words, there are trade-offs between the optimality and the efficiency. An algorithm which can find nearly optimal solutions within polynomial time may be good enough. *Approximation algorithms* are offline algorithms finding nearly optimal solutions.

We use *approximation ratio* to measure the performance of approximation algorithms. The approximation ratio of an algorithm is the worst case of its cost divided by the optimal cost for all feasible input. Consider a minimization problem and let $\mathcal{A}$ be an approximation algorithm. We denote $A(I)$ as the cost of the output of algorithm A with input I. We say that $\mathcal{A}$ is c-approximate if for any legal input $\mathcal{I}$, we have $\text{cost}(\mathcal{A}(\mathcal{I})) \leq c \cdot \text{cost}(\mathcal{O}(\mathcal{I})) + b$, where $\mathcal{O}$ is the optimal solution and b is a non-negative constant.

2.1.2 Fixed-parameter algorithms

Exact algorithms seek an optimal solution for optimization problems. The running time of these algorithms could be very large, especially for NP-hard problems. It is unknown if an NP-complete problem can be solved in polynomial time in the input size of the problem. In parameterized complexity theory, the complexity of a problem is not only measured regarding the input size, but also in terms of parameters. Generally speaking, an algorithm is a *fixed-parameter algorithm* if it solves a problem with input size n and a set of parameters $\{p_1, p_2, \cdots\}$ in $f(p_1, p_2, \cdots) \cdot O(g(n))$ time for some function f and some polynomial function g.

A problem is *fixed-parameter tractable* if it admits a fixed-parameter algorithm. The idea is that, by restricting parameters which are allowed to have exponential growth running time, we may get knowledge of how these parameters or characters of the problem influence the complexity. Hence, by studying fixed-parameter algorithms we might know better about which parameters make the decision-making so difficult. On the other hand, the parameters might be assumed to be small, and the time complexity of the fixed-parameter algorithm would be small if the parameters are small. That is, we can claim that if the parameters are small, the problem can be solved efficiently.

For a problem, there might be many different sets of parameters. Different ways of parameterizing a problem give different insights into the complexity of the problem, and there is no parameterization which is better than others. For example, in the CNF-SATISFIABILITY problem, some possible parameters are clause size, the number of variables, the number of clauses, etc. [66].

2.2 Online algorithms

In *online computation*, an algorithm has to make decisions based on past events and without information about future. Once a decision is made, it cannot be changed. Such algorithms are called the *online algorithms*. The difficulty of designing online algorithms is that each decision is made without knowing the whole picture of the input, and the impact of each decision influences the cost or performance of the final solution. In Section 5, we consider online algorithms, where the job information is only revealed at the time the job is released; the algorithm has to decide which jobs to run at the current time without future information and decisions made cannot be changed later.

There are two types of online models, *online time model* and *online list model*. In the online time model, the information of each job is known at the time it is available. Hence the available time of a job is equal to its release time and the job released earlier would be available earlier. In the online list model, the jobs are released according to an ordering. The next job in the ordering will be released once the previously released one is processed. The ordering of releasing does not depend on the earliest time when the jobs are available.

Similarly to the approximation ratio, we use *competitive ratio* to measure the performance of an online algorithm. The competitive ratio of an algorithm is the worst case of its cost divided by the optimal cost for all feasible input. We let $A(I)$ denote the cost of the output of algorithm A with input I. Consider a minimization problem and an online algorithm $\mathcal{A}$. We say that $\mathcal{A}$ is c-competitive if for all feasible input sequence $\mathcal{I}$, its cost $\text{cost}(\mathcal{A}(\mathcal{I})) \leq c \cdot \text{cost}(\mathcal{O}(\mathcal{I})) + b$, where $\mathcal{O}$ is the optimal solution and b is a constant at least 0. Notice that $\mathcal{O}$ is the optimal offline solution, which knows the whole input in advance. It is worth to mention that any online algorithm is also an approximation and the approximation ratio of the algorithm is exactly its competitive ratio.

The game with adversaries. When analyzing online algorithms, it can be viewed as a game between an online player who runs an online algorithm and an *adversary* who can create inputs. In this thesis, we consider *adaptive adversaries.* An adaptive adversary knows the action of the online player so far, and is able to design the next input according to this knowledge. In the game between an online algorithm and an adversary, the adversary tries to construct the worst possible input for the online algorithm such that the competitive ratio is maximized. In other words, the adversary works to make the competitive ratio as high as possible.

Consider a sequence $\mathcal{I}$ of input generated by an adversary and an online algorithm $\mathcal{A}$. If $\text{cost}(\mathcal{A}(\mathcal{I})) \geq c \cdot \text{cost}(\mathcal{O}(\mathcal{I}))$ for all instance $\mathcal{I}$, it means that the competitive ratio of $\mathcal{A}$ is at least c. On the other hand, if we can prove that for any online algorithms $\mathcal{A}$, there is an input $\mathcal{I}$ such that the competitive ratio is at least c, it means c is the lower bound of the competitive ratio of the optimization problem.

2.3 Scheduling

Scheduling is one of the intensively studied optimization problems. We refer to a survey by Leung [48]. Generally speaking, scheduling is about making decisions to allocate resources to activities with the objective of optimizing one or more measurement of performance. In this thesis, the scheduling problem is to allocate execution timeslots (resources) to jobs (activities) such that the total cost in generating power is minimized.

In the following, we introduce some terminologies widely used in scheduling area.

Preemptive and non-preemptive scheduling. In *preemptive* scheduling, jobs are allowed to stop temporarily and resume later. On the other hand, in *non-preemptive* scheduling, once a job starts executing, the execution will not stop until the task is finished. In this thesis, we consider non-preemptive scheduling.

The earliest deadline first (EDF) principle. The EDF principle is a natural strategy of scheduling. It defines the ordering of job executions according to their deadlines. At any time, the EDF principle chooses the job with the minimum deadline to be executed.

Special inputs. There are some special inputs discussed in scheduling problems. A *clique* instance is a set of jobs all containing at least one common timeslot. That is, a clique instance is a set of jobs $J_1, J_2, \cdots$, with at least one timeslot $t \in I(J_i)$ for all i, where $I(J_i)$ is the feasible interval of job J_i.

The jobs with *agreeable deadlines* means that for any two jobs J_1 and J_2, if J_1 is released no later than J_2, then the deadline of J_1 must be no later than the deadline of J_2.

More formally, for any two jobs J_1 and J_2 with their own feasible intervals $[r(J_1), d(J_1))$ and $[r(J_2), d(J_2))$, respectively, $r(J_1) \leq r(J_2)$ implies $d(J_1) \leq d(J_2)$.

A *laminar* instance is another well-know special input set. In laminar case, any two jobs J_1 and J_2 are either disjoint or one's feasible interval is completely inside the the other's feasible interval. That is, either $[r(J_1), d(J_1)) \cap [r(J_2), d(J_2)) = \emptyset$, $[r(J_1), d(J_1)) \subseteq [r(J_2), d(J_2))$, or $[r(J_1), d(J_1)) \supseteq [r(J_2), d(J_2))$.

2.4 Smart Grid Scheduling Problem and Definitions

In this thesis, we consider a smart grid scheduling problem. The input to the smart grid scheduling problem is a set of jobs with power requests, time durations, and a set of feasible timeslots in which it can be scheduled. The goal is to serve all jobs within their feasible timeslots without preemption such that the total electricity cost is minimized. It can be seen as deciding the time to start each job. According to the schedule, there is a profile of power needs to be generated at each time. That is, the total power request at each timeslot which is determined by the schedule. The cost needed for generating the power is a convex function of the power request at any time [45, 65]. We want to find a schedule such that the total electricity cost of the consequent power profile is as small as possible.

The input. We denote by $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ a set of input jobs. Each job J_j comes with *width* $w(J_j)$, representing the time duration required by J_j, and *height* $h(J_j)$, representing the power required by J_j. The *work* of job J_j, $p(J_j)$, is defined as the product of its power request and its time duration. That is, $p(J_j) = w(J_j) \times h(J_j)$. The time is labeled from 1 to $\tau + 1$ and we call the unit time $[t, t+1)$ *timeslot* t. That is, the time is divided into integral timeslots $\mathcal{T} = \{1, 2, 3, ..., \tau\}$. Each job J_j also associates with *feasible timeslots* $I(J_j) \subseteq \mathcal{T}$, representing the set of timeslots when J_j can be executed. We say that J_j is *available* during $I(J_j)$. If the feasible timeslots $I(J_j)$ of J_j form a contiguous interval, we call it the *feasible interval* of J_j. A feasible interval $I(J_j)$ is represented by $[r(J_j), d(J_j))$ where *release time* $r(J_j)$ is the earliest time J_j can be executed and J_j should be finished before its *deadline* $d(J_j)$. We consider events (release time, deadlines, feasible timeslots) occurring at integral time and assume $w(J_j)$, $h(J_j)$, $r(J_j)$, and $d(J_j)$, are integers.

Figure 2.1 shows illustrations of jobs. We consider each job J_j as a solid and unsplittable rectangle which can be shifted inside its feasible timeslots. An input job set would be a set of this kind of rectangles. Moreover, we assume that the input is *feasible*, that is, there exists a schedule such that each of the jobs can be assigned in its feasible timeslots. In other words, for every job J_j, there exist $t \in I(J_j)$ such that $[t, t + w(J_j)) \subseteq I(J_j)$. The rectangles can be stacked up, which means the corresponding jobs are executed at some same timeslots.

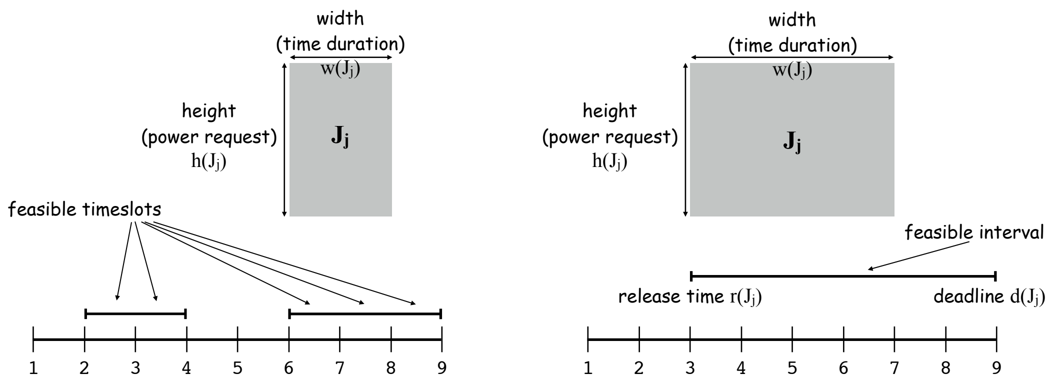


Figure 2.1: Illustrations of a job in the GRID problem.

Feasible schedule. A *feasible* schedule S has to assign for each job J a *start time* $st(S, J) \in I(J)$ meaning that J runs from $[st(S, J), et(S, J))$, where for each $t \in [st(S, J), et(S, J))$, $t \in I(J)$. Note that this means preemption is not allowed. The *load* of S at time t, denoted by $\ell oad(S, t)$ is the sum of the height (power request) of all jobs running at t, i.e., $\ell oad(S, t) = \sum_{J:t\in[st(S,J),et(S,J))} h(J)$. We drop S and use $\ell oad(t)$ when the context is clear. Given an interval $\mathcal{I}$, the work of a schedule S within $\mathcal{I}$ is defined as $\sum_{t\in\mathcal{I}} \ell oad(S, t)$. For any algorithm $\mathcal{A}$, we use $\mathcal{A}(\mathcal{J})$ to denote the schedule of $\mathcal{A}$ on $\mathcal{J}$. We denote by $\mathcal{O}$ the optimal algorithm.

The GRID algorithm. We consider the smart grid scheduling problem where the cost of a schedule S is the sum of the α-th power of the load over all time, for some $\alpha > 1$, i.e., $\text{cost}(S) = \sum_t (\ell oad(S, t))^\alpha$. For a set of timeslots $\mathcal{I}$ (not necessarily contiguous), we denote by $\text{cost}(S, \mathcal{I}) = \sum_{t\in\mathcal{I}} (\ell oad(S, t))^\alpha$. The objective is to find an assignment of all jobs in $\mathcal{J}$ to feasible timeslots such that the total cost is minimized. We call this the GRID problem.

In classical scheduling problems, the objectives usually concern about time. For example, minimizing flow time is to minimize the sum of the time from releasing a job to the time it is finished; minimizing makespan is to minimize the completion time of the last completed jobs. Unlike the classical scheduling problems, in the smart grid scheduling problem, the cost of a certain job dependents on not only its power request and time duration, but also the parameters of the jobs with overlapping execution intervals with it. Because of the convexity of the cost function, it would be much more expensive to execute jobs at the same time than to execute jobs without execution timeslots overlapping. In other words, it is better to schedule the jobs evenly over time.

Chapter 3

Literature Review

Koutsopoulos and Tassiulas [45] has formulated a similar problem to our problem and the objective is to minimize the long term cost given a distribution of the input jobs. They show that the problem is NP-hard. Feng et al. [27] have claimed that a simple greedy algorithm is 2-competitive for the unit case and $\alpha = 2$. However, as shown in [55], there is indeed a counter example that the greedy algorithm is at least 3-competitive and so the precise competitiveness of the greedy algorithm is still unknown. This implies that it is still an open question to derive competitive online algorithms for the problem. Salinas et al. [72] considered a multi-objective problem to minimize energy consumption cost and maximize some utility. A closely related problem is to manage the load by changing the price of electricity over time [11, 24, 61, 63]. Heuristics have also been developed for demand side management [58]. Other aspects of smart grid have also been considered, e.g., communication [12, 49, 53, 56], security [56, 60]. Reviews of smart grid can be found in [25, 33, 38, 59, 84].

The combinatorial problem we defined in this thesis has analogy to the traditional load balancing problem [4] and machine minimization problem [13, 16, 17, 71] but the main differences are the objectives. In the load balancing problem, the objective is to minimize the peak load among all machines [4]; in the machine minimization problem, the objective is to minimize the maximum number of machines needed [13, 16, 17, 71]. Minimizing peak load has also been looked at in the context of smart grid [1, 41, 76, 82, 83], some of which further consider allowing reshaping of the jobs [1, 41]. Our problem also has a resemblance to the dynamic speed scaling problem [2, 7, 81] and our algorithms employ some techniques there.

We first review the previous results of the grid problem with the objective to minimize the peak power request (Section 3.1.1), which has been widely studied. Then we move on to the GRID problem, that is, the smart grid scheduling problem with the objective to minimize the total cost (Section 3.1.2), which is the problem we discuss in this thesis. We also elaborate relating scheduling problems and algorithms, including dynamic voltage speed/voltage scaling problem (DVS) (Section 3.2), machine minimization problem (Section 3.3.1), bin packing problem (Section 3.3.2), and load balancing problem (Section 3.3.3). We aim to understand the relation between the GRID problem and these more classical scheduling problems. In this chapter, we also discuss graph algorithms and interval graphs, a special graph class. The graph algorithms give us directions to solve the smart grid scheduling problem with special input, and hence we can show that the grid problem with special input is polynomial-time solvable (Section 3.4). On the other hand, the special properties of interval graphs give us a way of tickling the grid problem with more general input (Section 3.4.3).

3.1 Previous Work on Smart Grid Scheduling

In this section, we first review the previous results in the smart grid scheduling problems. Before the research in the grid problem with the objective of minimizing the total cost, the grid problem with the objective of minimizing the peak power request is much widely investigated. We first review the results of the grid problem with the objective of minimizing the peak power request, then move on to the smart grid problem with the objective of minimizing the total cost, which is the problem we discuss in this thesis.

3.1.1 Minimizing peak power demand

The problem of minimizing peak power demand over time has been studied before [68, 69, 76, 82, 83]. The peak minimization problem is similar to the GRID problem but the objective is to find an assignment of all jobs to feasible timeslots such that the peak load along the time is minimized. That is, minimize $\max_t\{\ell oad(t)\}$. We call it the $\mathsf{GRID}_{\mathsf{peak}}$ problem.

Tang et al. [76] studied the $\mathsf{GRID}_{\mathsf{peak}}$ problem and considered a special case that the jobs have common feasible interval. That is, each job has same release time and same deadline. They proved that the $\mathsf{GRID}_{\mathsf{peak}}$ problem is NP-hard even for the common feasible interval special case. They further proposed an offline greedy strategy and proved that it is 7.82-approximate [76, 83].

The authors [76] also discussed another related problem, the *Delay Minimization Problem*, in which the maximum power supply is given. The Delay Minimization Problem is to schedule the jobs such that the power is no more than the maximum power supply at any time and the objective is to minimize the maximum finish time among all jobs. In other words, if the available power per time unit is limited, how to finish all the requests as soon as possible. For this problem with special input where all jobs have common feasible interval, the online greedy strategy is 2-competitive if there exists a feasible solution.

For the non-preemptive Peak Demand Minimization problem, Yaw et al. [83] further proved that it is even NP-hard to approximate within a factor of $\frac{3}{2} - \epsilon$ (this holds even for special input where each job has common feasible interval). The proof is by reducing from Bin-Packing problem. The authors proposed a 4-approximation algorithm for the instance where jobs have same release time and same deadline. The basic idea is grouping jobs by height, splitting the bin recursively, and schedule each group in the different area. For input job set with *agreeable deadlines* (that is, jobs have later release time must have later deadline), the authors also proposed a $O(\log \frac{w_{\max}}{w_{\min}})$-approximation algorithm by modifying their algorithm for common feasible interval instance. For general input where jobs have arbitrary release time and arbitrary deadline, the authors proposed a heuristic and experimental results. The heuristic is basically scheduling the *less flexible jobs* (that is, the jobs with higher $\frac{w(J)}{d(J)-r(J)}$ ratio) first.

Ranjan et al. [68] studied the preemptive case of the $\mathsf{GRID}_{\mathsf{peak}}$ problem. They showed that the $\mathsf{GRID}_{\mathsf{peak}}$ problem is NP-hard even when preemptive is allowed. The authors further proved that the next-fit decreasing height heuristic is 2-approximate when jobs (with same feasible interval) are all preemptive and 3-approximate when some of the jobs are non-preemptive. In their another paper [69], the authors further improved the approximation ratio upper bound to $1 + 1.7OPT$ by first-fit decreasing height strategy for instances with both preemptive and non-preemptive jobs.

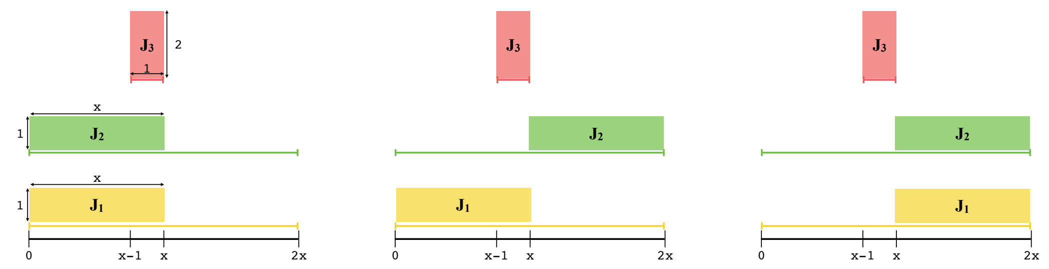
Yaw et al. [82] investigated an exact algorithm to find an optimal schedule of the non-preemptive $\mathsf{GRID}_{\mathsf{peak}}$ problem and gave experimental results. They showed that the problem is fixed-parameter tractable.

3.1.2 Minimizing total cost over time

In some cases, minimizing peak power demand may not be good enough. Although the peak is minimized, there might be many timeslots with high power demand and hence the total cost is still high. We focus on another objective in the smart grid scheduling problems which is to minimize the total electricity cost. There were some results in minimizing total cost [27, 45, 65]. We first elaborate the relation between the two objectives, minimizing total cost and minimizing peak demand.

Relating to minimizing peak demand. For the same input instance, the objective of minimizing peak or minimizing total cost might leads to different optimal schedule. That is, minimizing the peak demand does not necessarily minimize the total cost, and vice versa. Example 3.1 shows an instance and schedules with respect to minimizing peak and minimizing total cost. The example shows that a schedule with minimized peak may have a higher cost. In other way round, a schedule with minimized cost may have higher peak demand.

Example 3.1. *Consider positive integer x and $\mathcal{J} = \{J_1, J_2, J_3\}$ where $w(J_1) = w(J_2) = x$, $h(J_1) = h(J_2) = 1$, $w(J_3) = 1$, $h(J_3) = 2$, $I(J_1) = I(J_2) = [0, 2x)$ and $I(J_3) = [x-1, x)$ (see Figure 3.1a). Figure 3.1b shows a schedule S_c with minimum total cost, which is $3^\alpha + 2x - 1$. Figure 3.1c shows a schedule S_p with minimum peak, which has cost $(x+1) \cdot 2^\alpha$. It is easy to see that the peak in S_c is 3, which is higher than the peak in S_p, while the cost in S_c is lower than S_p when $x > \frac{3^\alpha - 2^\alpha - 1}{2^\alpha - 2}$.*



(a) Illustration of the three jobs in $\mathcal{J}$ (b) Schedule S_c with minimum total cost (c) Schedule S_p with minimum peak

Figure 3.1: An illustration of Example 3.1.

Intuitively, when α is big enough, the cost at the peak hour will dominate the total cost. In fact, it was shown in [55] that there is a polynomial time reduction of the decision version of the min-peak problem to that of the min-cost problem for a large enough α:

Lemma 3.1 ([55]). *A grid scheduling problem with objective to minimize the maximum power request can be reduced to a grid scheduling problem with objective to minimize the total cost by setting $\alpha > (\tau - 1)(2 \sum_{J \in \mathcal{J}} h(J) + 1)$, and the solution of min-cost problem under this setting is a solution of the corresponding min-max problem.*

Previously there were some results in minimizing total cost in smart grid model [27, 45, 65].

Koutsopoulos and Tassiulas [45] studied a similar problem to the GRID problem. Comparing to our problem, their cost function can be an arbitrary convex function while our cost function is an α-power function of load. Moreover, they studied stochastic model and aimed to minimize the expectation of long-term cost while we aim to minimize the total cost and guarantee the worst case performance.

The authors [45] claimed that for instance where jobs can be preempted, the GRID problem is equivalent to a load balancing problem (and hence NP-hard) and proposed an iterative load balancing algorithm to find an optimal schedule of preemptive jobs. For non-preemptive case of the GRID problem, the authors claimed that it is NP-hard.

For the online setting, the authors devised a stochastic model and focused on minimizing the long-term average cost. The authors proposed two strategies and proved one of them to be asymptotically optimal by deriving a lower bound for the performance of all policies.

Instead of minimizing the long-term average cost, Feng et al. [27] investigated the worst case competitive ratio of the GRID problem under online list setting. In the model time is divided into integral timeslots, each job has unit power request and unit time duration and is released with arbitrary feasible timeslots, which can be non-contiguous. Moreover, the authors consider the case where the cost function is quadratic (that is, $\alpha = 2$). They investigated the greedy strategy and claimed that it is 2-competitive.

However, Liu et al. [55] proved that greedy strategy is no better than 3-competitive when $\alpha = 2$ by showing an adversary and hence the precise competitiveness of the greedy algorithm is still unknown.

Narayanaswamy et al. [65] considered a more practical model that other than the power generated by the generator (with quadratic cost function), there is also a renewable resource. The renewable power resource (for example, wind power) varies over time and can only be predicted accurately before short period of time. The generator has to decide how much power to generate such that the generated power together with the renewable power satisfy the time-varying power demand. The objective is to generate sufficient amount of power with minimum *regret*, which is the difference between the online and the optimal costs. The authors applied recent work in online optimization and proved the algorithm is asymptotically good by deriving bounds in terms of the generator parameters.

Another flourishingly studied problem about optimizing power demand with convex cost function is Dynamic Voltage/Speed Scaling problem. We will discuss in details in Section 3.2

3.2 Dynamic Voltage/Speed Scaling Problem

The combinatorial problem we defined in this thesis has an analogy to the dynamic voltage/speed scaling problem. In this section, we review some famous algorithms for the dynamic voltage/speed scaling problem. In Chapter 5 we will further elaborate how to solve the GRID problem by adapting these algorithms. We also introduce different variations of the dynamic voltage/speed scaling problem and investigate the similarities and differences between these problems and the GRID problem. It gives us fascinating views on how the constraints/properties of the energy-efficient optimization problems affect on the complexities and strategies design.

The theoretical research of dynamic voltage/speed scaling (DVS) problem was raised by Yao, Demers, and Shenker in 1995 [81]. The authors proposed a model of job scheduling aiming at minimizing energy consumption. In their model, job requests J are given with their *work loads* $p(J)$, *release times* $r(J)$ and *deadlines* $r(J)$. For each job J, its work $p(J)$ should be finished within $[r(J), d(J))$ and preemption is allowed. A processor can run at speed $s \in [0, \infty)$ and consumes energy in a rate of s^{α}, for some $\alpha > 1$. The processor with speed s means that it can finish s units of work per unit of time. At any time the scheduler has to decide both which job to be executed and the processor speed to execute the job. The objective is to minimize the total energy consumption. There are different variations of the DVS problem. We also introduce the discrete dynamic DVS problem and the non-preemptive DVS problem.

3.2.1 Algorithms for the DVS Problem

In this section, we introduce various algorithms for the DVS problem under offline or online setting.

The YDS algorithm [81]. The authors proposed an $O(n^2 \log n)$ time algorithm for the offline DVS problem where n is the number of jobs. The key characterization of an optimal schedule is based on the notion of *critical intervals*. A critical interval is a time interval where a group of jobs must be scheduled at maximum, constant speed in any optimal schedule. The offline algorithm is iteratively picking critical intervals by finding intervals having most work has to be done within it. More formally, define *intensity* of interval I as the summation of work of jobs with feasible interval completely inside I divided by the length of $|I|$, it is easy to see that the intensity of I is a lower bound of the average speed in I. The YDS algorithm iteratively finds the interval I with the highest intensity, assigns the speed as the intensity and runs the jobs completely inside the interval.

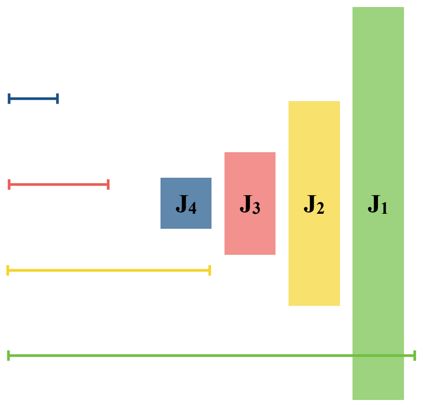
It is worth to mention that for the objective minimizing the maximum processor speed, YDS is also optimal [6].

The $\mathcal{AVR}$ algorithm [81]. For online DVS problem, Yao et al. [81] proposed the average rate algorithm ($\mathcal{AVR}$). The strategy is based on the *density* of each job J, which is defined as $\frac{p(J)}{d(J)-r(J)}$. In other words, consider an input with a single job, the density of the job is the processor speed in the optimal schedule. At any time t, $\mathcal{AVR}$ set the processor speed as the summation of the density of the jobs with feasible interval crossing t and the ordering of jobs execution follows the earliest deadline first (EDF) principle. In other words, in $\mathcal{AVR}$, the schedule and processor speed of each job are considered independently. The final schedule can be seen as stacking up of all independent schedules; at any time, the processor speed is the sum of processor speed in every independent schedule (Figure 3.2c) and the job to be executed is chosen according to the EDF principle (Figure 3.2d). The authors claimed that if the energy consumption function is in the form $P(s) = s^2$, the competitive ratio of $\mathcal{AVR}$ is between 4 to 8. Bansal, Kimbrel, and Pruhs [6] later proved that the AVR algorithm is $O(1)$-competitive if the energy consumption function is in the form $P(s(t)) = (s(t))^\alpha$.

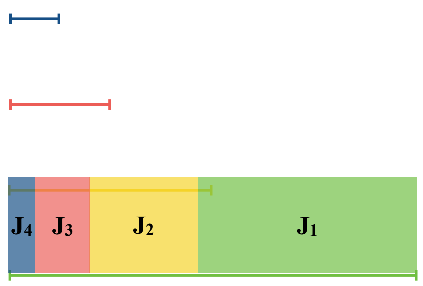
Lemma 3.2 ([6])**.** *The AVR algorithm is* $\frac{(2\alpha)^\alpha}{2}$*-competitive if the energy consumption function is in the form* $P(s(t)) = (s(t))^\alpha$*.*

Notice that $\mathcal{AVR}$ does not perform well with objective minimizing maximum speed. The following example shows an adversary on which $\mathcal{AVR}$ is $\Omega(n)$-competitive.

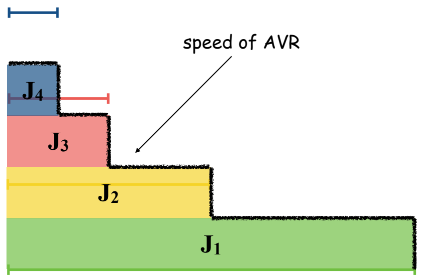
Example 3.2. *There are* n *jobs* $J_1, J_2, \cdots, J_n$*. For job* J_j*, its release time* $r(J_j) = 0$*, deadline* $d(J_j) = 2^{j-1}$*, and work load* $\ell_j = 2^{j-1}$*. The YDS algorithm runs at constant processor speed* $2 - 2^{1-n}$ *all over the time horizon while AVR runs at speed* n *at time interval* $[0, 1)$ *and speed* $n - i$ *at time* $t \in [2^{i-1}, 2^i)$ *for* $i \geq 1$*. The maximum speed of YDS is less than* 2 *while the maximum speed of AVR is* n*. Figure 3.2 shows an input set where* $n = 4$*.*



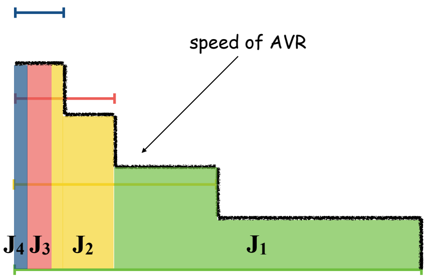
(a) A input jobs set of DVS



(b) YDS schedule



(c) How AVR speed is decided



(d) AVR schedule

Figure 3.2: Illustration of Example 3.2.

Time complexity of $\mathcal{AVR}$. At any time t, it needs $O(n)$ time to calculate the speed needed at t.

The $\mathcal{BKP}$ algorithm [6]. The $\mathcal{BKP}$ algorithm is proposed by Bansal, Kimbrel, and Pruhs. At time t, $\mathcal{BKP}$ considers all possible *windows* with specific proportion before and after t. More formally, the window, which is a time interval $[t_1, t_2)$ containing t, must have the relation that $|t_2 - t_1| : |t_2 - t| = e : 1$. The $\mathcal{BKP}$ algorithm considers all windows satisfying the property and finds the one with highest average released work load (by time t) completely inside it. That is, let $\ell(t, t_1, t_2) = \sum_{t \geq r(J_j) \wedge [r(J_j), d(J_j)) \subseteq [t_1, t_2)} \ell_j$ denote the total work load of jobs which have been released by time t and are completely inside the window $[t_1, t_2)$. Notice that the window $[t_1, t_2)$ should satisfy the property stated above. At any time t, $\mathcal{BKP}$ finds the window $[t_1, t_2)$ which has maximum value of $\frac{\ell(t,t_1,t_2)}{|t_2 - t_1|}$ and decides the processor speed to be at $e \times \frac{\ell(t,t_1,t_2)}{|t_2 - t_1|}$. The ordering of jobs execution follows the earliest deadline first (EDF) principle. The authors proved that $\mathcal{BKP}$ is $O(1)$-competitive with respective to both total energy consumption and maximum speed:

Lemma 3.3 ([6])**.** *The $\mathcal{BKP}$ algorithm is $2(\frac{\alpha}{\alpha-1})^\alpha e^\alpha$-competitive with respect to energy and e-competitive with respect to maximum speed.*

If $\alpha \geq 2$, the competitive ratio of $\mathcal{BKP}$ is at most $8e^{\alpha}$. Note that $\mathcal{BKP}$ has better competitive ratio than AVR when $\alpha \geq 5$. And also, $\mathcal{YDS}$, $\mathcal{AVR}$, and $\mathcal{BKP}$ are all independent of α.

Time complexity of $\mathcal{BKP}$. At any time t, consider the window $[t_1, t_2)$ with maximum $\frac{\ell(t,t_1,t_2)}{|t_2-t_1|}$ value, at least one of t_1 and t_2 is the release time or deadline of some jobs, otherwise $\frac{\ell(t,t_1,t_2)}{|t_2-t_1|}$ is not maximum. Hence the window $[t_1, t_2)$ can be found in $O(n_t^2)$ time where n_t is the number of jobs released by t.

3.2.2 Discrete dynamic voltage/speed scheduling

Unlike the DVS problem where the speed of the processor can be arbitrary, in the Discrete DVS problem, the processor speed is restricted to a given set of speeds. In the end of this section, we will see how to related the DVS problems to the GRID problem. The speed restriction of the Discrete DVS problem gives us an inspiration: since in the GRID problem, the power requests of jobs are fixed and not any real value of power request is needed (for example, if all jobs have integral power requests, then the non-integral speeds seem to be redundant), hence, maybe the Discrete DVS problem is closer to the GRID problem. In this section we introduce the results of the Discrete DVS problem. In Section 4.2.4, we will show in details how to relate the Discrete DVS problem to the GRID problem and elaborate a polynomial time algorithm for the GRID problem with special input by adapting an algorithm for the Discrete DVS problem.

Ishihara and Yasuura addressed the discrete dynamic voltage/speed scheduling problem [39]. In the original (continuous speed) DVS problem, the processor can be run at arbitrary real-valued speed. However, in real world the processor can only run at a speed selected from a given finite set of d speed levels $S = \{s_1, s_2, \cdots, s_d\}$ where $s_1 > s_2 > \cdots > s_d$. The DVS problem with discrete speeds constraint is called *discrete dynamic voltage/speed scheduling problem.* Under the discrete speeds constraints, the schedule generated by YDS algorithm may not be valid for the Discrete DVS problem since the processor speed decided by YDS may not be available. Kwon and Kim proposed a $O(n^2 \log n)$-time algorithm (where n is the number of jobs) to find a schedule with optimal energy consumption which also satisfies the discrete speed constraints [47]. The algorithm is based on the YDS algorithm for original DVS problem, which is probably invalid (for the Discrete DVS problem). If the speed s of YDS schedule in time interval $[t_1, t_2)$ is not available in the speed set S, the authors transform the YDS schedule to a valid schedule for the Discrete DVS problem by replacing the speed in $[t_1, t_2)$ by s_i and s_{i+1} (which are both in S) where $s_i > s > s_{i+1}$.

By directly characterizing the optimal schedule of the Discrete DVS problem, Li et al. [52] proposed an optimal algorithm with running time $O(n \log \max\{d, n\})$ where n is the number of jobs and d is the number of available speeds. The authors also revisited the continuous DVS problem and improved the running time from $O(n^2 \log n)$ to $O(n^2)$.

Also, the computation lower bound of the discrete DVS problem is proved to be at least $\Omega(n \log n)$ in the algebraic decision tree model.

For the online version of discrete DVS, Li proposed an $(\frac{2^{\alpha-1}(\alpha-1)^{\alpha-1}(\delta^\alpha-1)^\alpha}{(\delta-1)(\delta^\alpha-\delta)^{\alpha-1}} + 1)$-competitive online algorithm, where δ is the maximum ratio between any pair of adjacent non-zero speed levels [50]. That is, $\delta = \max_{s_i>0} \frac{s_i}{s_{i+1}}$. The algorithm is by transforming AVR to an online heuristic.

3.2.3 Non-preemptive dynamic voltage/speed scheduling

The DVS problems we discussed above all allow preemption. One of the main differences between the DVS problem and the GRID problem is that the preemption of jobs is allowed in the DVS problem while the preemption of jobs is not allowed in the GRID problem. It brings up a question: is the GRID problem more related to the non-preemptive DVS problem than to the DVS problem?

Although the DVS problem is polynomial time solvable for both single machine and multi-machine [5], the non-preemptive DVS is NP-hard. It can be proved by reducing from the PARTITION problem [3]. Moreover, it can be proved to be strongly NP-hard by reducing from the 3-PARTITION problem [3]. However, for instance with agreeable deadlines, the non-preemptive DVS problem is in P [3].

For jobs with equal work loads, the non-preemptive DVS problem is polynomial time solvable. Huang and Ott [37] proved it by proposing an $O(n^4)$ time algorithm based on dynamic programming.

Bampis et al. [5] proposed a $(1+\frac{\ell_{\max}}{\ell_{\min}})^\alpha$-approximation algorithm for the general case, where $\ell_{\max}$ and $\ell_{\min}$ are maximum work load and minimum work load, respectively. The algorithm is by transforming the YDS schedule (which is preemptive) to a non-preemptive one. By this result, there is a 2^α-approximation algorithm for uniform-workload jobs.

Antoniadis and Huang [3] proposed an approximation algorithm with approximation ratio $2^{5\alpha-4}$, which is not related to the work loads. This algorithm is based on a $2^{4\alpha-3}$-approximation algorithm for laminar case jobs.

In Table 3.1, we summarize the current results of different variations of the DVS problem under offline or online model.

Relating to the GRID problem. DVS and GRID are both energy-aware scheduling problems. The cost functions in the two problems are in the same convex form. However, the differences between these two problems are crucial and make the design and analysis of algorithms different. One of the main differences is, in the DVS problem, jobs are characterized by work load, which can be finished with any speed (and even different speeds at different time). In contrast, in the GRID problem, the jobs are characterized by fixed power request and time duration, which means each job has to be executed with specific power for a specific duration of time. Not to mention that preemption is allowed

Table 3.1: Summarize of results of the DVS problem.

	offline (running time)	online (competitve ratio)
DVS	$O(n^2 \log n)$ [52]	AVR: $2^{\alpha-1}\alpha^\alpha$ [6] BKP: $2(\frac{\alpha}{\alpha-1})^\alpha e^\alpha$ [6]
discrete DVS	$O(n \log \max\{d, n\})$ [52]	$\frac{2^{\alpha-1}(\alpha-1)^{\alpha-1}(\delta^\alpha-1)^\alpha}{(\delta-1)(\delta^\alpha-\delta)^{\alpha-1}} + 1$ [50]
non-preemptive DVS	strongly NP-hard $O(2^{5\alpha-4})$-approximate [3]	

in the original DVS problem while preemption is not allowed in the GRID problem. For visualizing the DVS problem, a job can be seen as arbitrary "shape" which can be stretched, rotated or split, as long as the total area of the shape remains the same as its work load.

For GRID problem with unit-size jobs (that is, the jobs have unit time duration and unit power request), preemption is not necessary since the jobs are only released or expired at the beginning of timeslots (which means the necessary preemptions must happen at the beginning of timeslots). Hence for GRID problem with unit-size jobs, it is relatively easy to adapt the algorithms of DVS.

We can transform an input set where all jobs have contiguous feasible intervals of GRID problem to an input set of DVS problem by the following: for each job J, the corresponding job in the DVS problem has work load $w(J) \cdot h(J)$, release time $r(J)$, and deadline $d(J)$. Notice that for special input where jobs sizes are unit, the consequent jobs are with unit work loads.

Since the power requests of jobs in GRID problem with unit-size jobs are exactly 1, the power demand at each timeslots in any schedule is integral. Recall the discrete DVS problem, by setting the speeds set $S = \{1, 2, 3, \cdots, n\}$ where n is the number of jobs, a discrete DVS schedule of the input jobs is also a GRID schedule. We will give details of this approach in Section 4.2.4.

For more general input of GRID problem, it is hard to adapt the algorithms for DVS problems. It is difficult even for unit-width jobs although there is no need for preemption. The first difficulty is that in the case where jobs with arbitrary heights and unit width, setting speeds set as all integers is not reasonable. Given a discrete DVS schedule, it may be impossible to transform the schedule into a GRID schedule since the speed at a timeslot may not be sum of heights of any subset of jobs available at that timeslot (it is just like the *McNugget number* problem). It makes it hard to upper bound the approximation/competitive ratio.

Although it is difficult to use the DVS algorithms to solve the GRID problem, the cost of optimal schedule in the DVS problem is a lower bound of the cost of optimal schedule in the GRID problem for the corresponding input sets. More formally, given any input set of the GRID problem, $\mathcal{J}_{\mathsf{GRID}}$, and the transformed input set of DVS problem,

$\mathcal{J}_{\mathsf{DVS}}$, $\text{cost}(\mathcal{O}_{\mathsf{GRID}}(\mathcal{J}_{\mathsf{GRID}})) \leq \text{cost}(\mathcal{O}_{\mathsf{GRID}}(\mathcal{J}_{\mathsf{GRID}}))$, where $\mathcal{O}_{\mathsf{GRID}}$ and $\mathcal{O}_{\mathsf{DVS}}$ are the optimal algorithm of GRID and DVS, respectively.

Comparing the non-preemptive DVS problem with the GRID problem, there is an example shows that given a job set $\mathcal{J}_{\mathsf{GRID}}$ and the corresponding $\mathcal{J}_{\mathsf{DVS}}$, $\text{cost}(\mathcal{O}_{\mathsf{DVS}}(\mathcal{J}_{\mathsf{DVS}}))$ may not necessarily lower than $\text{cost}(\mathcal{O}_{\mathsf{GRID}}(\mathcal{J}_{\mathsf{GRID}}))$, where $\mathcal{O}_{\mathsf{DVS}}$ here is the optimal algorithm for non-preemptive DVS.

Example 3.3. *There are two jobs. One has release time* 0*, deadline* 3*, width* 3 *and height* 1*. The other has release time* 1*, deadline* 2*, width* 1 *and height* 1*. Both jobs can only schedule at their release time in* GRID *since their widths are the same as the lengths of their feasible intervals. The optimal cost of* GRID *is* $1^\alpha + 2^\alpha + 1^\alpha = 2^\alpha + 2$*. Whereas the optimal cost of non-preemptive* DVS *is* $2^\alpha + 2^\alpha = 2 \cdot 2^\alpha$*. This is because the schedule uses speed* 2 *and runs the longer job with time interval with length* 1.5 *and the shorter job with time interval with length* 0.5*. The optimal cost of* GRID *is lower when* $\alpha > 1$*.*

Therefore, it is unclear how we may use the results on non-preemptive DVS problem and so we would stick with the preemptive DVS algorithms in this thesis.

3.3 Related Scheduling Problems

The energy-efficient optimization problem we consider in this thesis has analogies to the traditional optimization problems such as the Machine Minimization problem, the Load Balancing problem, and the Bin Packing problem. In this section, we investigate these problems and some of the techniques are adapted to solve the GRID problem in Section 5.3. Furthermore, in Chapter 6 we show that our technique can be adapted to solve the Machine Minimization problem.

3.3.1 Machine minimization

The *Machine Minimization problem* is a well-studied optimization problem. The non-preemptive machine minimization problem is a special case of the smart grid scheduling problem with objective of minimizing the peak. In this section, we investigate the Machine Minimization problem. In Chapter 6 we show that our technique can be adapted for this problem and we propose a competitive algorithm for the Machine Minimization problem.

The formal definition of the Machine Minimization problem is as the following. Given a set of jobs with processing times p, release times r, and deadlines d and infinite number of machines, each job has to be scheduled on a machine such that in its execution interval (which is inside the interval between its release time and deadline) there is no other jobs assigned on the same machine. The goal is to minimize the number of opened machines.

The machine minimization problem has been studied intensively for both preemptive and non-preemptive settings. The preemptive machine minimization problem is polynomial time solvable [36]. On the other hand, the non-preemptive case is NP-hard. Cieliebak et al. [18] showed that there is no polynomial time non-preemptive machine minimization problem with approximation ratio less than 2, unless P=NP. They also proved that there is an $(\lceil \frac{2\delta_{\max}}{p_{\min}} \rceil + 1)$-approximation algorithm where $p_{\min}$ is the minimum processing time among all jobs and $\delta_{\max}$ is the maximum *slack*, which is defined as $d - r - p$ for each job. For non-preemptive machine minimization, there are results with respect to different parameters. Chuzhoy et al. [15] proposed an $O(m)$-approximation algorithm where m is the optimal solution. It can also be shown to be $O(\sqrt{\frac{\log n}{\log \log n}})$-approximate where n is the number of jobs.

For the online version of preemptive machine minimization, Phillips et al. [67] showed that the *least laxity first* (LLF) strategy is $O(\log \frac{p_{\max}}{p_{\min}})$-competitive, where $p_{\max}$ and $p_{\min}$ are the maximum processing time and the minimum processing time, respectively. They also showed that there exists no $\frac{5}{4}$-competitive algorithm. For other parameters, Chen et al. [13] proposed an $O(\log m)$-competitive algorithm, where m is the number of machines used by an optimal schedule.

Saha [71] showed that any online algorithm for the non-preemptive machine minimization problem has competitive ratio at least $\log_3 \frac{p_{\max}}{p_{\min}}$. The adversary is like the following. There are n jobs, each job J_j has processing time $p(J_j) = 3^{n-j}$ and the length between its release time and deadline, $|d(J_j) - r(J_j)| = 3 \cdot p(J_j)$. The first job J_1 is released at time 1 and for $1 < j < n$, J_{j+1} is released at the time when J_j is executed. It is easy to see that the online algorithm uses $\log_3 \frac{p_{\max}}{p_{\min}}$ machines while an optimal schedule only needs one machine. In addition, it can be interpreted as the online algorithm using n machines and it leads to a competitive lower bound n.

Saha also proposed an $O(\log \frac{p_{\max}}{p_{\min}})$-competitive online algorithm in [71]. The algorithm basically classified jobs by their processing times and applies the constant approximation non-preemptive machine minimization algorithm proposed by Chuzhoy and Codenotti [20] for each class. Unfortunately, the authors [14] have retreated the results as they have identified a mistake in the analysis which invalidates the claimed approximation ratio and as a result the best approximation ratio for the problem remains $O(\sqrt{\frac{\log n}{\log \log n}})$. Hence it remains unknown if there existed an asymptotically optimal algorithm for the non-preemptive machine minimization problem. To answer the question, we propose a $O(1 + \lceil \log \frac{p_{\max}}{p_{\min}} \rceil)$-competitive algorithm in Section 6.1.

There are some constant competitive algorithms for the machine minimization problem with special input set. For the online preemptive machine minimization problem, Chen et al. [13] proposed a 96-competitive algorithm for laminar jobs and a 176-competitive algorithm for jobs with agreeable deadlines. For non-preemptive machine minimization, Devour et al. [19] proposed a e-competitive algorithm for jobs with unit processing times. And for jobs with equal deadlines, the authors proposed a 16-competitive algorithm. They also showed that for jobs with unit processing time, the competitive ratio lower bound is e, and for laminar jobs, the competitive ratio lower bound is n (this came from the adversary we mentioned above.)

On the other hand, there are also constant approximation algorithms for special input. Cieliebak et al. [17] proposed a $\frac{23+3\sqrt{5}}{2}$-approximation algorithm for *clique* instance, that is, all feasible intervals overlap at a timeslot.

Relating to the GRID problem. The non-preemptive machine minimization problem is a special case of the smart grid scheduling problem on minimizing the peak power request. The input jobs of machine minimization problem can be seen as input jobs of smart grid peak minimization problem such that each job has unit height and the width is the processing time. The number of machines used in a machine minimization problem is exactly the peak power in the corresponding GRID problem. Hence, the 2-approximation algorithm for jobs with same release times and the 6-approximation algorithms for jobs with equal widths in the smart grid peak minimization problem in [82] imply that in the machine minimization problem, there is a 2-approximation algorithm for jobs with same release times and a 6-approximation algorithms for jobs with uniform processing times.

In Section 6.1, we show how to use the techniques in this paper to solve the online machine minimization problem. We propose an asymptotically optimal online algorithm for the machine minimization problem. We also give a $2e$-competitive algorithm for jobs with unit processing times, and it leads to a $6e$-competitive algorithm for jobs with uniform processing times.

3.3.2 Bin Packing problem

The Bin Packing problem has similar flavor with the smart grid problem in which the object is to minimize the peak power request. In the Bin Packing problem, the objective is to minimize the opened bins such that all the items are packed; in the $\mathsf{GRID}_{\mathsf{peak}}$ problem, the objective is to schedule all the jobs such that the maximum load is minimized. In this section we review the Bin Packing problem and discuss the similarities and differences with the $\mathsf{GRID}_{\mathsf{peak}}$ problem. In Section 5.3.3, we adapt the technique in analyzing the First-Fit strategy for the Bin Packing problem to an analysis for First-Fit strategy for the GRID problem where the objective is to minimize the total cost.

Bin Packing is a classical NP-hard problem [30]. The problem is defined as the following. Given a set of items with real-valued *size* in $(0, 1]$ and an infinite number of bins with capacity 1, the objective is to pack the items into minimum number of bins such that within each bins, the total size of items packed in this bin does not exceed the bin capacity.

First-fit (FF) is a basic strategy that each item is put into the first bin which is capable of accommodating it. If all bins have the remaining capacity with size smaller than the size of the item, a new bin is created and the item is assigned to it. Notice that there is no specific order on items. Hence FF can be used as an online strategy. FF has been proved to use no more than $1.7\mathcal{O} + 1$ bins if the optimal solution uses $\mathcal{O}$ bins [29].

Strip packing problem. There are different variations of the packing problem. We introduce strip packing, which is also discussed in some research about smart grid scheduling with objective of minimizing the maximum power request. In the strip packing problem, there are a set of two-dimensional items with width and height both bounded by 1 and a single bin with width 1 and infinite height. The goal is to pack the items in the bin without overlapping or rotation such that the height of the final strip (that is, the minimum rectangle area in the bin which can cover all items) is minimized.

Relating to the GRID problem. The strip packing problem is similar to the smart gird peak-minimization problem. Like the jobs in the grid problem, the items in the packing problem are rectangles and cannot be rotated, and the objectives are both minimizing the maximum height of the consequence placement. However, in the smart grid peak-minimization problem, there are constraints on when jobs can be scheduled (that is, the release time and deadline constraint) while in the strip packing problem the items can be placed arbitrarily.

Although the strip packing problem cannot capture the time constraint in the smart grid peak-minimization problem, strip packing problem is a special case of the $\mathsf{GRID}_{\mathsf{peak}}$ problem where all jobs have same release times and same deadlines.

For the traditional Bin Packing problem, it also has similar flavor with the $\mathsf{GRID}_{\mathsf{peak}}$ problem with special input. Consider the jobs J in the grid problem with equal release time r, equal deadline d, unit height and arbitrary width $w(J)$. It can be seen as a Bin Packing problem where each bin has capacity $d - r$ and each of the items has size the same as the width of its corresponding job. The peak demand in the smart grid schedule is exactly the number of bins used in the Bin Packing algorithm. In [83], the inapproximability of the smart gird peak-minimization problem is proved by reducing from the Bin Packing problem.

In Section 5.3.3, we prove that First-Fit (FF) is $2^{2\alpha}$-competitive for the GRID problem (where the objective is minimizing the total cost), with input jobs having same release times, same deadlines, unit heights and arbitrary widths. The proof idea is from the analysis of first-fit strategy in Bin Packing problem.

3.3.3 Load Balancing problem

The Load Balancing problem is another classical optimization problem. There is a close resemblance between the Load Balancing problem and the smart grid problem with minimizing peak power request objective. The min-peak smart grid problem aims to minimize the peak power request, and the Load Balancing problem aims to assign jobs to the machines such that the heaviest load among all machines is minimized. In Section 5.3.4, we show how to adapt the technique in analyzing the greedy strategy for the Load Balancing problem to an analysis for the greedy strategy for the GRID problem with special input.

In the Load Balancing problem, there is a sequence of jobs with loads have to be assigned to one of a finite set of machines. The goal is to balance the summation of loads of jobs assigned to each machine such that the maximum load among all machines is minimized. There are different machine models. For *identical machines*, the load for each job is fixed no matter which machine it is assigned to. For *related machines*, it can be interpreted as each machine has different speed or power. The incurred load of a job on different machine differs by the speed of the machine; a machine with higher speed has smaller incurred load. For *unrelated machines*, the load of each job on different machines are arbitrary (but is given). The Load Balancing problem can be proved to be NP-hard by reducing from the PARTITION problem.

The GREEDY strategy (or Best-Fit) assigns the incoming job to the machine such that after assigning the job, the maximum load among all machines is minimized. Graham [31] proved that GREEDY is exactly $(2 - \frac{1}{m})$-competitive in identical machines model, where m is the number of machines. On the other hand, GREEDY is $O(\log m)$-competitive in the related machine model.

Restricted machines model. There is a special variation of the Load Balancing problem where the machines are identical and each job can be assigned to a subset of machines. For different jobs, the available subset of machines can be different. The GREEDY strategy is proved to be $(\lceil \log_2 m \rceil + 1)$-competitive while the competitive ratio lower bound is $\lceil \log(m+1) \rceil$ [8].

Relating to the GRID problem. The identical machines Load Balancing problem also has a similar flavor with the smart gird peak-minimization problem with a special input set. Consider a set of jobs in the grid problem where all jobs have same release times, same deadlines, unit widths and arbitrary heights. The input set can be transformed into an input set of the Load Balancing problem where each job has workload equals to the height of its corresponding job in the grid problem. Each machine can be seen as a timeslot and the jobs assigned to the machine can be seen as the jobs in the smart grid problem assigned to the corresponding timeslot. By the transformation, the peak power demand in the grid problem is exactly the maximum load among all machines.

Similarly, the related machines model might give us a direction for investigating the GRID problem with tariff constraints. That is, different timeslots may have different cost functions.

In the restricted machines model, the subset of machines associating with each job can be interpreted as the feasible timeslots of a job in the smart gird problem. However, this approach is adaptable only for jobs with unit widths.

In Section 5.3.4, we show that greedy strategy is 2^{α}-competitive for GRID problem for unit width job set with same release time and same deadline.

3.4 Related Graph Algorithms

The flow problem and the matching problem are two classical optimization problems in graph theory and are usually applied to design and analyze the other optimization problems. In this section, we investigate how to use the flow or matching algorithms to solve the GRID problem. More specifically, we consider the GRID problem with special input where jobs have unit height and unit width. We also introduce a special class of graphs, interval graphs. The special properties of interval graphs give us a way of tickling the GRID problem with more general input.

3.4.1 Flow problem

The minimum cost maximum flow problem with convex functions. Given a flow network, that is, a directed graph. There are source s and sink t, both are vertices. In the flow network, each edge has capacity, flow and cost. In the minimum cost maximum flow problem we want to find a path from s to t with the maximum flow but having the minimum cost [22, 80].

We can solve the unit-size scheduling problem by using the minimum cost maximum flow algorithm with convex functions. We can reduce the unit-size scheduling problem to the following min-cost max-flow problem. We have a graph $G = (V, E)$, where $V = \{s, t\} \cup \{J_1, J_2, ..., J_n\} \cup \{t_1, t_2, ..., t_\tau\}$. For any two vertices J_i and t_k, if time k is feasible for job J_i, there is an edge from vertex J_i to vertex t_k with capacity 1 and constant cost c. For vertices J_i where $1 \leq i \leq n$, there are edges (s, J_i) with capacity 1 and constant cost c. For vertices t_k where $1 \leq k \leq \tau$, there are edges (t_k, t) with capacity n and cost is convex function of the flow on the edge.

There are already some work about convex cost function [74, 75]. Sokkalingam et al. [74] proposed an $O(m(m + n \log n) \min\{\log(nU), m \log n\})$-time algorithm for convex cost function, where m is the number of edges, n is the number of vertices, and U is the maximum edge capacity. According to the reduction described above, it takes $O(n\tau(n\tau + n \log n) \min\{\log n, n\tau \log n\})$ time to solve our scheduling problem.

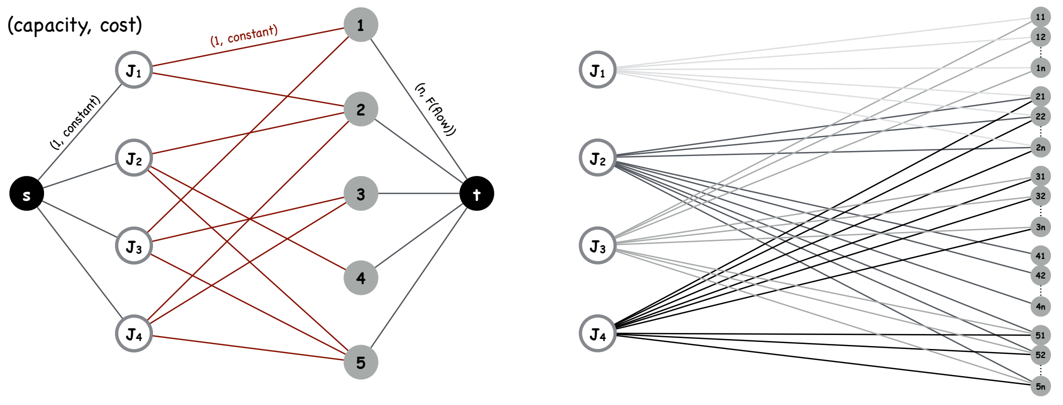
3.4.2 Matching problem

The maximum-cardinality minimum-weight matching on a bipartite graph. Given a bipartite graph $G = (U, V, E)$, where U and V are two parts of the graph and E is set of edges in G. For each edge (u, v) where $u \in U$ and $v \in V$, there is a weight $w(u, v)$. The maximum-cardinality minimum-weight matching problem on the graph G is finding the matching in G where have the maximum number of edges but the summation of the weights on these edges is minimum.

Given an instance of unit-size scheduling problem, we can reduce it to a maximum-cardinality minimum-weight matching instance as following. We have a bipartite graph $G = (U, V, E)$. Part $U = \{J_1, J_2, ..., J_n\}$ (each vertex J_i is corresponding to job J_i).

Part V has $n\tau$ vertices where $V = \{t_{11}, t_{12}, ..., t_{1n}, t_{21}, t_{22}, ..., t_{2n}, ..., t_{\tau 1}, t_{\tau 2}, ..., t_{\tau n}\}$. If time T is feasible for job J_i, there is an edge (J_i, t_{Tk}) for all $1 \leq k \leq n$. For each edge that has one end point t_{ij}, its weight is $f(j) - f(j-1)$, where the function f is the convex cost function in the scheduling problem. There are some algorithms for solving the weighted matching problem [34, 42]. However, the running time would be huge if we solve the scheduling problem by reducing it to a matching problem.

Example 3.4. *Given jobs $\{J_1, J_2, J_3, J_4\}$. The feasible (not contiguous) timeslots of J_1, J_2, J_3, J_4 are $\{1, 2\}$, $\{2, 4, 5\}$, $\{1, 3, 5\}$, and $\{2, 3, 5\}$, respectively. Figure 3.3 are illustrations about the graph algorithms. Figure 3.3a shows the minimum cost maximum flow problem transformed from the scheduling problem and Figure 3.3b shows the maximum-cardinality minimum-weight matching problem transformed from the scheduling problem.*



(a) (b)

Figure 3.3: Illustration of Example 3.4.

3.4.3 Interval graphs

Interval graphs is a class of graphs which is widely discussed in scheduling problems.

Definition 3.4. A graph G is an *interval graph* if it is the intersection graph of a collection of intervals on the real line. That is, there is an interval I_x for every vertex x in G such that two vertices x and y are adjacent in G if and only if $I_x \cap I_y \neq \emptyset$.

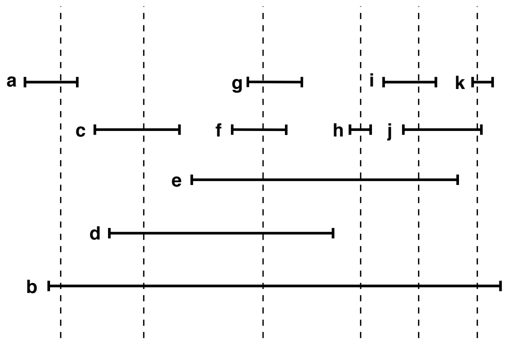
Because of the nature of the scheduling problems, the interval graphs can capture some properties in the scheduling problems. For example, if each the tasks in a scheduling problems has to be served in a time interval, the interval graph of these intervals encodes the competition for resources between the jobs. That is, two tasks have conflicts if and only if there is an edge between the corresponding vertices in the interval graph.

The class of interval graphs is a subset of the perfect graphs. There are many useful properties of interval graphs. Here we introduce a very important property of interval graphs:

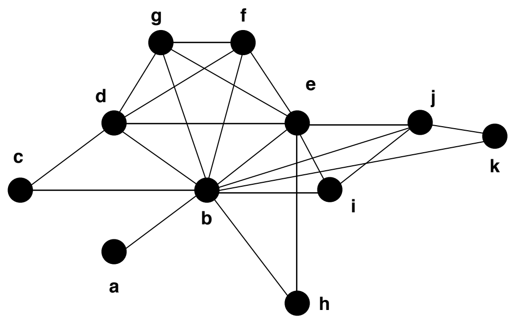
Lemma 3.5 ([32]). *An interval graph G has a* consecutive clique arrangement*, that is, there is a linear ordering $[M_1, M_2, \cdots, M_t]$ of the maximal cliques in G such that for every vertex x, the maximal cliques that contain x form a subsequence.*

We give an example of a interval graph corresponding to a set of intervals and illustrate its consecutive clique arrangement:

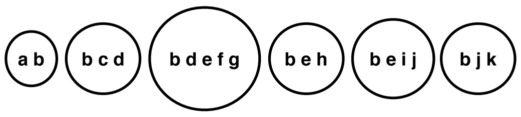
Example 3.5. *Figure 3.4 is an example of a set of jobs, its corresponding interval graph and the corresponding maximal cliques. Figure 3.4a shows a set of jobs, where the horizontal line segments are the feasible intervals of jobs. The vertical dotted lines indicate the maximal cliques. Figure 3.4b shows an interval graph of the corresponding job set. And Figure 3.4c is a set of all the maximal cliques in the interval graph. The cliques are put in such a way that any vertex appears consecutively if there is two or more of it.*



(a) An input instance and the windows.



(b) The corresponding interval graph.



(c) The consecutive cliques.

Figure 3.4: An input instance and the corresponding interval graph.

The consecutive clique arrangement property gives us a direction of dynamic programming. And hence many of the graph problems can be solved in polynomial time on interval graphs. For example, the maximum independent set problem, which is NP-hard for general graphs, is linear-time solvable for interval graphs [43]. By the consecutive clique arrangement property, we propose an exact algorithm for finding the optimal schedule of the GRID problem. The details are in Section 4.3.

3.5 Summary

In this chapter, we reviewed previous results in the smart grid scheduling problems arising in demand response management. We also investigate the similarities and differences between some classical optimization problems and the GRID or $\text{GRID}_{\text{peak}}$ problems. The DVS problem has a similar flavor to the GRID problem in the form of the cost function and suggests a lower bound on the GRID schedule. Interestingly, although the non-preemptive DVS problem has more similar form with the GRID problem since preemption is not allowed in the GRID problem, the optimal non-preemptive DVS schedule does not guarantee a lower bound of the GRID schedule. It is still unclear how to relate the non-preemptive DVS problem to the GRID problem.

The Machine Minimization problem is a special case of the smart grid scheduling problem with objective minimizing the peak power request. We will continue our investigation of these problems in Chapter 6.

For other classical optimization problems, we compare the Bin Packing problem, Load Balancing problem, flow problem and matching problem. By reducing to the flow problem or the matching problem, we showed that the GRID problem with unit-size input jobs is polynomial time solvable. On the other hand, the Bin Packing problem and Load Balancing problem give us directions to solve the GRID problem with other special inputs. The discussion will be continued in Section 5.3.

At the end of this chapter, we introduce a special class of graphs, interval graphs, which plays an important role in our exact algorithms, which would be introduced in Section 4.3.

Chapter 4

Offline Algorithms for The GRID Problem

In this chapter we consider the offline setting of the GRID problem where all widths, heights, and feasible timeslots of the jobs are known in advance. In Section 4.1, we prove that the GRID problem is NP-hard even when either the widths or heights of the jobs are of unit size. However, the GRID problem can be solved in polynomial time when both the width and height of each job are both of unit size. We propose a polynomial time algorithm for finding the optimal schedule for such input in Section 4.2. The basic idea of this polynomial time algorithm is to use a “feasibility graph” to capture all possible assignments of the jobs. By detailed analysis we show that the optimality of the current schedule can be known by a simple condition check and hence we can maintain and query the feasibility graph efficiently.

The feasibility graph algorithm works when each job has unit width and unit height. Yet, in Section 4.4 we use it to approximately solve a more general case where jobs have arbitrary width and height and contiguous feasible interval. The basic idea is to classify the jobs by their widths and heights, treating each class of jobs as unit size and scheduling each class independently.

For a general input where each job has arbitrary width, arbitrary height and contiguous feasible interval, we also investigate the exact algorithms (Section 4.3). We propose an exponential time algorithm to find an optimal schedule for the general case. The algorithm is based on dynamic programming by using special properties of interval graphs. Also, we show that the GRID problem is fixed parameter tractable since the exact algorithms are parameterized exact algorithms.

4.1 NP-hardness

Koutsopoulos and Tassiulas [44] claimed that the general GRID problem with any convex cost function is NP-hard. The proof is basically by reducing the BINPACKING problem to a smart grid problem where jobs have heights and the objective is to minimize the maximum power consumption over time. By claiming that minimizing the maximum power consumption in the time horizon is equivalent to minimizing the total convex cost in the horizon, the authors claimed that the GRID problem is NP-hard.

For completeness, we prove that the GRID problem is strongly NP-hard by reducing a strongly NP-hard problem directly to the GRID problem where the objective is to minimize the total cost.

In the following we define the decision version of GRID problem. Note that the input job parameters (width, height, and feasible timeslots) and the cost of a schedule follow the formal problem definition mentioned in Section 2.4.

The GRID problem (decision version). Given a set of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ and a non-negative real number K, decide whether there exists a feasible schedule for $\mathcal{J}$ such that the total cost is no more than K.

We first introduce a helpful observation of the schedules of the GRID problem by the convexity of the cost function (recall that the work of a job J is defined as $w(J) \cdot h(J)$):

Observation 1. In the GRID problem, consider an interval $\mathcal{I}$ with length ℓ and the total work P for all jobs which have to be finished within $\mathcal{I}$, (i) for any feasible schedule S, the cost $\text{cost}(S) \geq (\frac{P}{\ell})^\alpha \cdot \ell$; and (ii) if in a schedule S, there exists a timeslot t with load $> \frac{P}{\ell}$, then $\text{cost}(S) > (\frac{P}{\ell})^\alpha \cdot \ell$.

Note that the first part of Observation 1 shows a trivial lower bound of any feasible schedule of the GRID problem. The second part is easy to see from the first part.

We prove that the GRID problem with special input is NP-hard, hence the general GRID problem is NP-hard. First we consider the special case where each job has unit height and common contiguous feasible interval. We prove that the GRID problem is strongly NP-hard by reducing from the 3-PARTITION problem.

The 3-PARTITION problem. Given a set of $n = 3{\cdot}m$ numbers $A = \{a_1, a_2, \cdots, a_n\}$ and number $b = \frac{\sum_{a_i \in A} a_i}{m}$, decided if there exists a way to partition A into m disjoint subsets $S_1, S_2, \cdots, S_m$ such that in each S_j there are exactly 3 elements and the sum of the elements is exactly b.

The reduction works as follows. For each a_i in the input set in the 3-PARTITION problem, we construct a job J_i such that $w(J_i) = a_i$. For each job J_i, its release time is 0 and its deadline is b. That is, the feasible intervals of the jobs are common. Also, we let $K = b \cdot m^\alpha$.

It is easy to see that there exists a partition such that the sum of the numbers in each subset are equal if and only if there is an optimal schedule with total cost $b \cdot m^\alpha$. If there is a partition $S_1, S_2, \cdots, S_m$ in the 3-PARTITION problem, then the corresponding jobs in the GRID problem can be scheduled in m "levels" evenly. That is, the corresponding jobs in each S_j can be scheduled without overlapping to each other (and form a level). The m level of jobs can be stacked up and the resulting schedule has load m at any timeslot. Since the feasible intervals are common and the length is b, the total cost of the schedule is $b \cdot m^\alpha$. On the other hand, if there is no such partition in the 3-PARTITION problem, at least one of the jobs in the GRID problem has to be scheduled at the $m+1$ level (that is, there is at least one timeslot with load $> m$ in the schedule). By Observation 1, the total cost is greater than $K = b \cdot m^{\alpha}$.

Theorem 4.1. *The* GRID *problem is strongly NP-hard even when all jobs have common feasible interval and the heights of jobs are all of unit size.*

We prove in the following that the GRID problem is also strongly NP-hard when all jobs have unit width. We also prove it by reducing from the 3-PARTITION problem. The reduction works as follows. For each a_i in the input set in the 3-PARTITION problem, we construct a job J_i such that $h(J_i) = a_i$. For each job J_i, its release time is 0 and its deadline is m. That is, the feasible intervals of the jobs are common and with size m. Also, we let $K = m \cdot b^{\alpha}$.

We prove that there exists a partition $S_1, S_2, \cdots, S_m$ such that the sum of numbers in each subset is equal in the 3-PARTITION problem if and only if there is a schedule with total cost $m \cdot b^{\alpha}$. If there is a partition $S_1, S_2, \cdots, S_m$ in the PARTITION problem, the jobs in the GRID problem can be scheduled evenly in the m timeslots $1, 2, \cdots, m$, that is, the corresponding jobs in S_j can be all scheduled at timeslot j for all $1 \leq j \leq m$. The total cost of this schedule is $m \cdot b^{\alpha}$. On the other hand, if there is no such partition in the 3-PARTITION problem, then for any schedule, there must be at least one timeslot with load higher than b. By Observation 1, the total cost of the schedule is greater than $K = m \cdot b^{\alpha}$.

Theorem 4.2. *The* GRID *problem is strongly NP-hard even when all jobs have common feasible interval and the widths of jobs are all of unit size.*

Next we prove that the GRID problem is strongly NP-hard even when preemption is allowed. The preemptive GRID problem, $\mathsf{GRID}_{\mathsf{prmp}}$, is defined as the following. Each input job J has width $w(J)$, height $h(J)$, and contiguous feasible interval $[r(J), d(J))$. A feasible schedule of job J is a subset of timeslots in $[r(J), d(J))$ with cardinality $w(J)$. That is, the job J can be executed at exact $w(J)$ timeslots which are not necessarily contiguous, and at each timeslot the power request of J is exactly $h(J)$.

Theorem 4.3. *The problem* $\mathsf{GRID}_{\mathsf{prmp}}$ *is strongly NP-hard.*

Proof. Consider the $\mathsf{GRID}_{\mathsf{prmp}}$ problem with special input where jobs have unit time duration and arbitrary power request, there is no need for preemption. Hence this problem is the same as the problem GRID with special input where jobs have unit time duration. By Theorem 4.2, this special GRID problem is NP-hard. Therefore, the $\mathsf{GRID}_{\mathsf{prmp}}$ problem is NP-hard since one of the special case is NP-hard. □

4.2 Unit Case

In this section we consider the GRID problem with a special input set $\mathcal{J}$ where each job has unit power request and unit time duration. That is, for each $J \in \mathcal{J}$, $h(J) = 1$ and

$w(J) = 1$. Each job J associates with feasible timeslots $I(J) \subseteq \mathcal{T}$ in which the job can be executed. For example, the feasible timeslot of a job can be $\{1, 2, 4, 5\}$. We prove the smart grid scheduling problem is polynomial time solvable with this special input.

4.2.1 Feasibility graph Algorithm

We solve the GRID problem with a special input set $\mathcal{J}$ where each job has unit power request and unit time duration by using *feasibility graph* to represent alternative assignments. The basic idea is similar to the residual network in network flows; after scheduling a job, we can look for improvement via this feasibility graph. We show that each time a job is scheduled, the optimality can be maintained in polynomial time. For the analysis, we compare our schedule with an optimal schedule via the notion of *agreement graph*, which captures the differences of our schedule and an optimal schedule. We then show that we can transform our schedule stepwise to improve the agreement with the optimal schedule, without increasing the cost, thus proving the optimality of our algorithm.

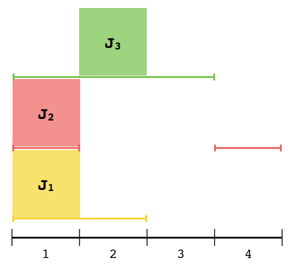
Feasibility graph. Given a particular job assignment S, we define a *feasibility graph* $G_{\mathrm{f}} = \{V_{\mathrm{f}}, E_{\mathrm{f}}\}$ to be a directed multi-graph that shows the potential allocation of each job in alternative assignments. Recall that the time is divided into set of timeslots $|\mathcal{T}| = \{1, 2, \cdots, \tau\}$, there are $|\mathcal{T}|$ vertices in $V_{\mathrm{f}} = \{v_1, v_2, \cdots, v_\tau\}$ and each vertex $v_t \in V_{\mathrm{f}}$ corresponds to the timeslot $t \in \mathcal{T}$. Moreover, each vertex v_t associates with a positive weight $\omega(v_t)$, denotes the load of the corresponding timeslot t, i.e., $\omega(v_t) = \ell oad(t)$. On the other hand, E_{f} captures all other possible assignments of each assigned job. If job J_j is assigned to timeslot t in S, then for all $t' \in I(J_j) \backslash \{t\}$ we add a directed arc $(v_t, v_{t'})$ to E_{f} with J_j as its label.

A feasibility graph G_{f} corresponds to an assignment S. If a job J is assigned to timelsot t in S, we also say that J is assigned to v_t for short.

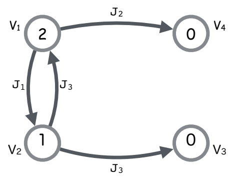
Legal-path in a feasibility graph. A path $[v_s, \cdots, v_t]$ in a feasibility graph G_{f} is a *legal-path* if and only if the weight of the starting point v_s is at least 2 more than the weight of the ending point v_t, i.e., $\ell oad(s) - \ell oad(t) \geq 2$.

Example 4.1 is an example of the feasibility graph corresponds to an input set and a legal-path in the feasibility graph. We will explain later that if there is a legal-path in the feasibility graph G_{f}, the corresponding job assignment is not optimal.

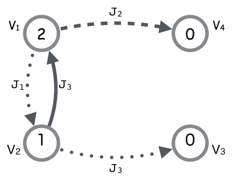
Example 4.1. *We now give an example of the notions feasibility graph and legal-path. Let* $\mathcal{J} = \{J_1, J_2, J_3\}$, $\mathcal{T} = \{1, 2, 3, 4\}$, $I(J_1) = \{1, 2\}$, $I(J_2) = \{1, 4\}$, *and* $I(J_3) = \{1, 2, 3\}$. *Figure 4.1a shows a schedule where* J_1 *and* J_2 *are both assigned to timeslot* 1, *and* J_3 *is assigned to timeslot* 2. *Figure 4.1b shows the feasibility graph* G_f *for an job assignment* S. *Figure 4.1c shows two legal-paths with respect to* S, $[v_1, v_4]$ *and* $[v_1, v_2, v_3]$.



(a) An illustration of the assignment S.



(b) The feasibility graph for S.



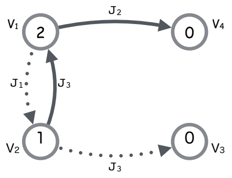
(c) Two legal-paths for S.

Figure 4.1: The feasibility graph and two legal-paths with respect to Example 4.1

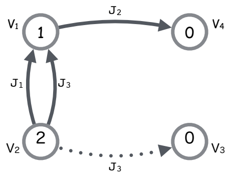
Observation 2. Moving J from timeslot s to timeslot t may result in decrease/increase/same cost depending on the loads of s and t. More formally, the overall energy cost (i) decreases if $\ell oad(s) > \ell oad(t)+1$, (ii) remains the same if $\ell oad(s) = \ell oad(t)+1$, and (iii) increases if $\ell oad(s) < \ell oad(t))+1$.

Shifting. By Observation 2, the existence of a legal-path implies that the corresponding assignment is not optimal and we can execute a "shift" and decrease the total cost of the assignment. Given a path $P = [u_1, u_2, \cdots, u_n]$ where $(u_i, u_{i+1}) \in E_f$ for each i, a *shift* moves each job corresponding to an arc (u_i, u_{i+1}) along P from the original assigned timeslot u_i to the timeslot u_{i+1}. More precisely, if the path contains an arc (u_i, u_{i+1}) with J_j as its label, then job J_j is moved from u_i to u_{i+1}. If there are more than one arc between u_i and u_{i+1}, we break ties arbitrarily. Note that after shifting, the load of vertices in P remains unchanged except u_1 and u_n. Moreover, $\ell oad(u_1)$ decreases by one and $\ell oad(u_n)$ increases by one.

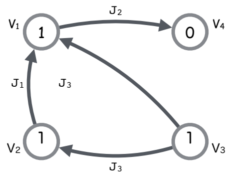
Figure 4.2 shows an illustration of shifting a legal-path referring to Example 4.1. Originally, the jobs J_1, J_2, and J_3 are assigned at timeslot 1, 1, and 2, respectively. The shifting is along the legal-path $[v_1, v_2, v_3]$.



(a) The feasibility graph G_f. Legal-path $[v_1,v_2,v_3]$.



(b) G_f after shifting job J_1 to timeslot 2.



(c) G_f after shifting job J_3 timeslot 3.

Figure 4.2: Example of shifting along a legal-path referring to Example 4.1.

From Observation 2 that such a shift along a legal-path decreases the cost, implying that the original assignment is not optimal. On the other hand, when there is no legal-path, it is not as straightforward to show that the assignment is optimal. Nevertheless, we will prove this is the case in Lemma 4.10.

Feasibility graph algorithm $\mathcal{A}_{\mathrm{FG}}$. We propose a polynomial time offline algorithm that minimizes the total cost for the GRID problem. The algorithm arranges the jobs in $\mathcal{J}$ in arbitrary order, and works in stages. In each stage, one new job joins and is considered. First the new job is greedily assigned to one of its feasible timeslots with lowest load. Then the corresponding feasibility graph is updated and shifting is executed along a legal-path (if there exists at least one legal-path.)

More formally, at any Stage j we have three steps (also see Algorithm 1):

(1) Assign J_j to a feasible timeslot with minimum load, breaking ties arbitrarily;

(2) Suppose J_j is assigned to timeslot r. We update the feasibility graph G_{f} to reflect this assignment in the following way. If applicable, we add arcs from v_r labelled by J_j to any other feasible timeslots (vertices) of J_j;

(3) If there exists any legal-path in G_{f} from r to any other vertex t, the algorithm executes a shift along the legal-path (Figure 4.2 shows an illustration). At the end, the algorithm updates the feasibility graph G_{f} to reflect this shift.

Note that when searching for legal-paths (Step (3)), we focus on those starting from v_r, where r is the only timeslot of which the load is changed. We need to prove that if there is any legal-paths after assigning the new job to r, there must be at least one which is start from v_r or they existed before the assigning step (Lemma 4.4). On the other hand, if there is no legal-paths before assigning the new job, the assignment is optimal or there must be at least one legal-path starting from v_r (Lemma 4.10).

Algorithm 1 The feasibility graph algorithm $\mathcal{A}_{\mathrm{FG}}$

```
Input: a set of jobs $\mathcal{J} = J_1, J_2, \cdots, J_n$
Output: an optimal schedule of $\mathcal{J}$
for $i$ from 1 to $n$ do
    $r \leftarrow \arg\min_{t \in I(J_i)} \ell oad(\mathcal{A}_{\mathrm{FG}}, t)$
    Assign $J_i$ to $r$
    Update the feasibility graph according to the assignment
    if there exists a legal-path $P$ starting from $r$ then
        Shift the jobs along $P$
        Update the feasibility graph according to the shifting
return the schedule corresponding to the final feasibility graph
```

Additional notations. To ease the discussion, in the remainder of this section, we use $\ell oad'_j(t)$ to represent the load of timeslot t after assigning J_j (but before the shift), $\ell oad_j(t)$ to represent the load of timeslot t at the end of Stage j, and $\ell oad'_j(s,t)$ and $\ell oad_j(s,t)$ to represent $\ell oad'_j(s) - \ell oad'_j(t)$ and $\ell oad_j(s) - \ell oad_j(t)$, respectively.

4.2.2 Correctness

We assume that at the beginning of each stage, there is no legal-paths in G_{f} and the current assignment is optimal for all assigned jobs:

Invariants. We show that the algorithm maintains the following two invariants. At the end of each stage:

(I1) There is no legal-path in the resulting feasibility graph;

(I2) The assignment is optimal for the jobs considered so far.

We want to prove that given the invariants hold at the beginning of each stage, after assigning and shifting the new job the invariants still hold.

Framework. Consider any stage of $\mathcal{A}_{\text{FG}}$. After Step (2), there may be a legal-path in the resulting feasibility graph G_{f}. In Lemma 4.4, we show that if a legal-path exists in G_{f} after assigning J_j to timeslot r, there is at least one legal-path starting from r. Suppose the algorithm chooses the legal-path $[r, \ldots, t]$ and executes the shift along this path in Step (3). In Lemma 4.7, we show that if there is no legal-path in the feasibility graph G_{f} before assigning a job, then after assigning a job and executing the corresponding shift by the algorithm, the resulting feasibility graph has no legal-paths. Therefore, Step (3) of the algorithm needs to be applied only once and there will be no legal-path left, implying that Invariant (I1) holds. In Lemma 4.10, we show that if there is no legal-path in a feasibility graph G_{f}, the corresponding assignment is optimal and hence Invariant (I2) holds.

We begin by proving Lemmas 4.4 and 4.5. Lemma 4.5 is a technical lemma used in the proof of Lemma 4.7.

In Step (2), we only search legal-paths starting from r, hence we need to prove that there is at least one legal-path after assigning the new job to r or there is no legal-paths.

Note that in G_{f}, there is no legal-paths before assigning j (Invariant (I1)). Hence, for any two vertices u and v in G_{f}, $\ell oad_j(u, v) \geq 2$ only if u, v are disconnected. In other words, $\ell oad_j(u, v) < 2$ for any pair of connected u and v.

Lemma 4.4. *Suppose that before assigning job J_j to timeslot r the feasibility graph G_f has no legal-path. If there is any legal-path after assigning J_j, there is at least one legal-path starting from r.*

Proof. Assume that there is a legal-path $[s, \cdots, t]$ after assigning J_j to timeslot r, so that $\ell oad'_j(s, t) \geq 2$. If $r = s$, we have obtained a desired legal-path. Otherwise, $r \neq s$, there are two cases:

Case 1. Vertices s and t are connected in G_{f} before assigning J_j. Since $r \neq s$, $\ell oad_{j-1}(s) = \ell oad'_j(s)$ and $\ell oad_{j-1}(t) \leq \ell oad'_j(t)$ (the latter inequality comes from the fact that r may be equal to t). This implies $\ell oad_{j-1}(s, t) \geq \ell oad'_j(s, t) \geq 2$, which contradicts the precondition that there is no legal-path before assigning J_j. Thus, Case 1 cannot occur.

Case 2. Vertices s and t are disconnected in G_{f} before assigning J_j. Since (s, t) are connected after assigning J_j, it must be the case that assigning J_j to timeslot r adds some new arc (r, w) (with J_j as its label) to G_{f}, which connects an existing $[s, \cdots, r]$ path and an existing $[w, \cdots, t]$ path. We know that $\ell oad_{j-1}(s) - \ell oad_{j-1}(r) \leq 1$ because there is no legal-path before assigning J_j. Also, $\ell oad'_j(s) = \ell oad_{j-1}(s)$, $\ell oad'_j(t) = \ell oad_{j-1}(t)$, and $\ell oad'_j(r) = \ell oad_{j-1}(r) + 1$ because the new job J_j is assigned to r, with $r \neq s$. Hence, $\ell oad'_j(s) \leq \ell oad'_j(r)$. It implies that $\ell oad'_j(r, t) \geq \ell oad'_j(s, t) \geq 2$, so that the $[r, \cdots, t]$ subpath is also a legal-path. □

In Step (3), we only choose one legal-path and execute the corresponding shifting. That is, only one job is moved from r and after the shifting this stage is finished. We need to prove that shifting one arbitrary legal-path is sufficient. First we prove that for a legal-path starting from r, it is not a legal-path after shifting.

Lemma 4.5. *If before assigning a job the feasibility graph G_f does not have a legal-path, then after assigning one more job there will be no legal-paths where the load of the starting point is at least* 3 *more than the load of the ending point. In other words, the load difference corresponding to any new legal-path, if it exists, is exactly* 2.

Proof. We prove that if there was no legal-paths, by assigning a single job is not capable to cause a legal-path which needs two shiftings.

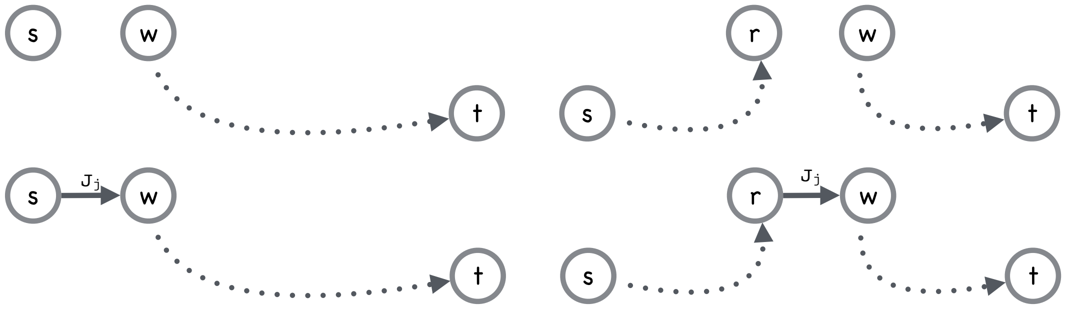
Assume on the contrary that there is a legal-path $[s, \cdots, t]$ with $\ell oad'_j(s, t) \geq 3$. There are two cases:

Case 1. Before assigning job J_j, s and t are connected. Assigning a job at timeslot r increases by one on the load difference of any path starting from r. On the other hand, the load difference of any path ending at r is decreased by one. Recall that $\ell oad'_j(s, t) \geq 3$. There are three situations: (i) Timeslot s is r, implying $\ell oad_{j-1}(s, t) \geq 2$; (ii) Timeslot t is r, which implies $\ell oad_{j-1}(s, t) \geq 4$; (iii) r, s, and t are three different timeslots, which implies $\ell oad_{j-1}(s, t) \geq 3$. Each of these cases contradicts the fact that G_{f} has no legal-path before assigning J_j.

Case 2. Before assigning job J_j, s and t are disconnected. That is, assigning job J_j creates a new path (legal-path) $[s, \cdots, t]$ with $\ell oad'_j(s, t) \geq 3$. There are two sub-cases (see Figure 4.3 for an illustration):

Case 2-1. Job J_j is assigned to timeslot s, which means s and r are the same timeslot. Since $[s, \cdots, t]$ becomes a new legal-path after assigning job J_j, it must be the case that assigning J_j to timeslot s adds some new arc (s, w) in G_{f} that connects s with an existing $[w, \cdots, t]$ path. The arc (s, w) means that job J_j can be assigned to timeslot s or w. By our algorithm, $\ell oad(w) \geq \ell oad(s)$. Furthermore, we have $\ell oad_{j-1}(s) = \ell oad'_j(s) - 1$, $\ell oad_{j-1}(t) = \ell oad'_j(t)$, and $\ell oad'_j(w) = \ell oad_{j-1}(w) \geq \ell oad_{j-1}(s)$. Hence, $\ell oad_{j-1}(w, t) = \ell oad_{j-1}(w) - \ell oad_{j-1}(t) \geq \ell oad_{j-1}(s) - \ell oad_{j-1}(t)$. According to our assumption, $\ell oad'_j(s, t) \geq 3$, thus $\ell oad_{j-1}(s, t) \geq 2$. It leads to that $\ell oad_{j-1}(w, t) \geq 2$, which contradicts the fact that there is no legal-paths before assigning J_j.

Case 2-2. The job J_j is assigned to timeslot r with $r \neq s$. Since (s, t) becomes a new legal-path after assigning job J_j, it must be the case that there is some new arc (r, w) added in G_f that connects an existing $[s, \cdots, r]$ path with an existing $[w, \cdots, t]$ path. Because there is no legal-path before assigning job J_j, $\ell oad_{j-1}(s, r) \leq 1$ and $\ell oad_{j-1}(w, t) \leq 1$. According to our assumption, $\ell oad'_j(s, t) \geq 3$; this implies $\ell oad_{j-1}(s, t) \geq 3$, so that $\ell oad_{j-1}(r) - \ell oad_{j-1}(w) \geq 1$. The latter inequality contradicts the fact that J_j is assigned to a feasible timeslot with minimum load. □



(a) Case 2-1 before and after assigning job J_j to timeslot s.

(b) Case 2-2 before and after assigning job J_j to timeslot r.

Figure 4.3: The two sub-cases of Case 2 in the proof of Lemma 4.5 (the dotted arcs are used to represent paths).

Now we are able to prove that after shifting along an arbitrary legal-path starting from r (Step (3)), there is no more legal-path if there is none at the beginning of this stage.

To ease the discussion, we introduce two parameters $IN(v)$ and $OUT(v)$ for any vertex v in G_f and give some properties. We define $IN_j(r)$ to be the set of vertices w such that a $[w, \cdots, r]$ path exists before assigning J_j, and $OUT_j(r)$ to be the set of vertices w such that an $[r, \cdots, w]$ path exists before assigning J_j. We assume that $r \in IN_j(r)$ and $r \in OUT_j(r)$ for the ease of later discussion. Similarly, we define $IN''_j(r)$ and $IN''_j(r)$ to be the set of vertices w such that a $[w, \cdots, r]$ path exists after assigning J_j and after shifting. respectively. Note that $IN''_j(r) = IN_{j+1}(r)$ and $OUT''_j(r) = OUT_{j+1}(r)$. Given a set R of vertices, let $IN_j(R) = \bigcup_{r \in R} IN_j(r)$ and $OUT_j(R) = \bigcup_{r \in R} OUT_j(r)$. The notation $IN''_j(R)$ and $OUT''_j(R)$ are defined analogously. The subscript of j can be ignore when the context is clear.

Figure 4.4 shows an illustration of a feasibility graph and the $IN(r)$ and $OUT(r)$ for a vertex r. The green vertices are those in $IN(r)$, which have at least one path to r; the yellow vertices are those in $OUT(r)$, which have at least one path from r. There are three vertices (including r) with colors green and yellow. They are the vertices having paths from r and paths to r. That is, these bi-color vertices are in $IN(r) \cap OUT(r)$.

Suppose that there were no legal-paths in G_f after Stage $j - 1$, but there is a new legal-path in G_f after assigning J_j. By Lemma 4.4, there must be one such legal-path

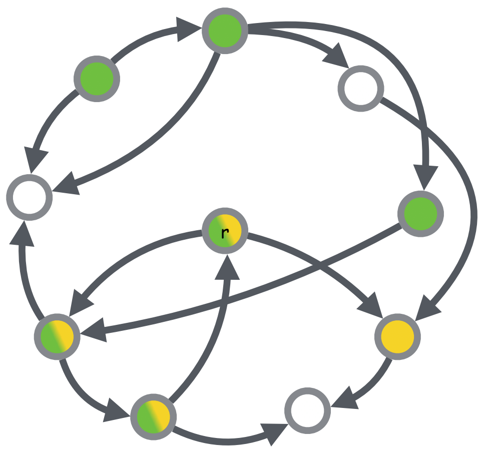


Figure 4.4: $IN(r)$ (green vertices) and $OUT(r)$ (yellow vertices). The three vertices including r with colors green and yellow are in both $IN(r)$ and $OUT(r)$.

$[s, \cdots, t]$ where s is the timeslot assigned to J_j, and without loss of generality, let the path be the one that is selected by our algorithm to perform the corresponding shift. Let the ordering of the vertices in the legal-path be $[s, v_1, v_2, \cdots, v_k, t]$, and P denote the set of these vertices.

Briefly speaking, we upper bound the load of a vertex in $IN''_j(P)$, and lower bound the load of a vertex in $OUT''_j(P)$, as any legal-path that may exist after the shift must start from a vertex in $IN''_j(P)$ and end at a vertex in $OUT''_j(P)$. Based on the bounds, we shall argue that there are no legal-paths as the load difference of any path after the shift will be at most 1. Recall that after the shift, only the load of t is increased by one, the load of s is decreased by one, whereas the load of any other vertex remains unchanged.

We observe the change of $IN(v)$ and $OUT(v)$ for vertices v on the legal-path after assigning J_j on vertex s (Lemma 4.6). First let us see what happened in the feasibility graph during Stage j. In Stage j, job J_j is assigned to s and there is a sequence of shifting along the legal-path $P = [s, v_1, v_2, \cdots, v_k, t]$. After J_j is assigned to s, the only difference in the feasibility graph is that there are some new arcs from s to other feasible vertices of J_j. If there was no (s, v_1) arc before assigning J_j, each vertex in $P \setminus \{s\}$ might get some new paths entering in it and leaving from vertices in $IN_j(s)$. However, consider vertices in P altogether, there are no new vertices in $IN_j(P)$ after assigning J_j. That is, if we denote the $IN'_j(P)$ (and $OUT'_j(P)$) as the vertices with path leaving (or entering) them and entering (or leaving) at least one of vertices in P after assigning but before shifting, $IN'_j(P) = IN_j(P)$. On the other hand, because of the new arcs between s and vertices in $I(J_j)$, there might be some new vertices having path leaving s and entering it after assigning J_j. Hence $OUT'_j(P) = OUT_j(P) \cup OUT_j(I(J_j))$.

After shifting, only the edges labeled by shifted jobs change. More specifically, only those edges labeled by shifted jobs change. There is no new jobs getting involved so

$IN''_j(P) = IN'_j(P) = IN_j(P)$ and $OUT''_j(P) = OUT'_j(P) = OUT_j(P) \cup OUT_j(I(J_j))$. Then we look at the $IN''_j(P')$ more carefully. Recall that when s and v_1 was not connected before assigning J_j, $IN'_j(v_1) = IN_j(v_1) \cup IN_j(s)$. However, after shifting, (s, v_1) arc is reversed and therefore all vertices in $IN'_j(v_1) \setminus IN_j(v_1)$ is not connected to vertices in P' anymore. Hence, $IN''_j(P') \subseteq IN_j(P')$ in general.

Lemma 4.6. *After assigning J_j at s, consider a legal-path $P = [s, v_1, v_2, \cdots, v_k, t]$ and its subpath $P' = P \setminus \{s\} = [v_1, v_2, \cdots, v_k, t]$. After shifting along P,*

a. $IN''_j(P') \subseteq IN_j(P')$ and $IN''_j(P) \subseteq IN_j(P)$

b. $OUT''_j(P) \subseteq OUT_j(P) \cup OUT_j(I(J_j))$.

Proof. (a) Let z be a vertex in $IN''(P')$ but not in $IN(P')$. Take the shortest path from z to some vertex in P' after the shift. Then all the intermediate vertices of such a path are not from P'. However, the jobs assigned to those intermediate vertices are unchanged, so that such a path also exists before the shift, and z is in $IN(P')$. A contradiction occurs. The second part can be proved similarly.

(b) Similar to (a), let z be a vertex in $OUT''(P)$ but not in $OUT(P) \cup OUT(I(J_j))$. Take the shortest path that goes to z starting from some vertex in P after the shift. Then all the intermediate vertices of such a path are not from P. If such a path does not involve vertices from $I(J_j)$, then this path must exist before the shift, so that z is in $OUT(P)$. Else, z is in $OUT(I(J_j))$. A contradiction occurs. □

Now we can show that if there were no legal-paths at the beginning of each stage, there are no legal-paths at the end of each stage. The key is, since every shifted job is assigned at the legal-path, edges not incident to the vertices on the legal-path will not change. Hence, if after assigning J_j and shifting along a legal-path there is still a legal-path, the legal-path must go through at least one changed edge. Otherwise, the legal-path existed before Stage j and it contradicts to the fact that there were no legal-paths at the beginning of Stage j. To put it in other words, the legal-path existing at the end of Stage j must contain at least one shifted job. These shifted jobs are assigned at vertices on the legal-path, hence the potential legal-path must start from a vertex in $IN''_j(P)$ and and end at a vertex in $OUT''_j(P)$. By Lemma 4.6, we can bound the load of vertices in $IN''_j(P \setminus \{s\})$, $IN''_j(P)$, and $OUT''_j(P)$ by the load of vertices in $IN_j(P \setminus \{s\})$, $IN_j(P)$, $OUT_j(P)$, and $OUT_j(I(J_j))$.

Lemma 4.7. *Suppose that G_f is a feasibility graph with no legal-paths. Then after assigning a job and executing the corresponding shift by the algorithm, the resulting feasibility graph has no legal-paths.*

Proof. Assume that J_j is assigned to s and $\ell oad_{j-1}(s) = x$. Suppose that there is a legal-path $P = [s, v_1, v_2, \cdots, v_k, t]$. For any vertex r in $I(J_j) \setminus \{s\}$, $\ell oad_{j-1}(r) \geq x$, since $s \in I(J_j)$ has the minimum load. This implies that the load for any vertex in $OUT_j(I(J_j))$ is at least $x-1$, or there was a legal-path leaving a vertex in $I(J_j)$ before assigning J_j. On the other hand, by Lemma 4.5, $\ell oad_{j-1}(t) = x-1$ since there is no legal-path before assigning J_j but there is one after assigning J_j. Hence, for all vertices $r \in IN_j(t)$, $\ell oad_{j-1}(r) \leq x$ or there was a legal-path from r to t before assigning J_j. Therefore, for each vertex v_h, $IN_j(v_h) \leq x$ because of v_h is on path P, and $IN_j(s) \leq x$ if there was an arc (s, v_1) at the beginning of Stage j. That is, for each vertex $r \in IN_j(P)$, $\ell oad_{j-1}(r) \leq x$ if (s, v_1) arc existed at the beginning of Stage j. On the other hand, we can only say that for each vertex $r \in IN_j(P \setminus \{s\})$, $\ell oad_{j-1}(r) \leq x$ if there was no (s, v_1)-arc before assigning J_j.

Concerning the legal-path P, there are two cases:

Case 1. There was an arc from s to v_1 in the feasibility graph G_f before assigning J_j. The load of any vertex in $OUT_j(P)$ is at least $x-1$ or there was a legal-path leaving s and entering the vertex with load less than $x-1$ before assigning J_j. Hence, after the shift, the load of any vertex in $IN''_j(P)$ is at most x, and the load of any vertex in $OUT''_j(P)$ is at least $x-1$, so no legal-paths will exist.

Case 2. There were no arcs from s to v_1 in the feasibility graph G_f before assigning J_j. In this case, J_j must be involved in the shift, so that the jobs assigned to s after the shift will be the same as if J_j was not assigned. Consequently, if there is still a legal-path after the shift, the starting vertex must be from $IN''(P \setminus \{s\})$ (since there is no arc (s, v_1) anymore), while the ending vertex must be from $OUT''(P)$ (note that there is an arc (v_1, s).)

Since $v_1 \in I(J_j)$, $\ell oad_{j-1}(v_1) \geq x$. Also, $\ell oad_{j-1}(v_1) \leq x$ or there was a legal-path from v_1 to t. Hence, $\ell oad_{j-1}(v_1) = x = \ell oad_{j-1}(s)$ and the load of any vertex in $OUT_j(P)$ is at least $x-1$, since there was no legal-path leaving s or v_1 before assigning J_j. In conclusion, after the shift, the load of any vertex in $IN''_j(P \setminus \{s\})$ is at most x, and the load of any vertex in $OUT''_j(P)$ is at least $x-1$, so no legal-paths will exist. □

We now prove in Lemma 4.10 (the other key lemma for the correctness) that non-existence of legal-paths implies the assignment is optimal. First we introduce a notation:

Agreement graph G_a. We define an *agreement graph* $G_a(S, \mathcal{O})$ which is a directed multi-graph that measures the difference between a job assignment solution S and an optimal assignment $\mathcal{O}$. In $G_a(S, \mathcal{O})$ each timeslot is represented by a vertex and the number inside the vertex denotes the load of the timeslot in S. For each job J_j such that J_j is assigned to different timeslots in S and $\mathcal{O}$, we add an arc from t to t', where t and t' are the timeslots that J_j is assigned to by S and $\mathcal{O}$, respectively. The arc (t, t') is labelled by the tuple $(J_j, +/-/=)$. The second value in the tuple is "+" or "−" if moving job J_j from timeslot t to timeslot t' causes the total cost of assignment S to increase or decrease, respectively. The value is "=" if moving the job does not cause any change in the total cost of assignment S.

The rough ideas about the non-existence of legal-paths implying the optimality of the assignment are as follows. Consider an optimal assignment $\mathcal{O}$ (satisfying some constraints as to be defined). In Lemma 4.9, we show that there is a sequence of agreement graphs $G_a(S_1, \mathcal{O})$, $G_a(S_2, \mathcal{O})$, $\cdots$, $G_a(S_k, \mathcal{O})$ where the cost is non-increasing every step, $S_1 = S$ is the original assignment of jobs given by our algorithm, and $S_k = \mathcal{O}$ is an optimal assignment. We prove Lemma 4.10 by contradiction, assuming there is no legal-path in the feasibility graph G_f but the assignment S is not optimal. We then consider the sequence of agreement graphs given in Lemma 4.9 and show that either there is no agreement graph in the sequence involving strict decrease of overall cost (which means S is already optimal) or that there is a legal-path in the feasibility graph G_f, leading to a contradiction.

Note that Lemma 4.9 considers an optimal assignment $\mathcal{O}$ such that $G_a(S, \mathcal{O})$ is acyclic. The existence of acyclic $\mathcal{O}$ is proved in Lemma 4.8.

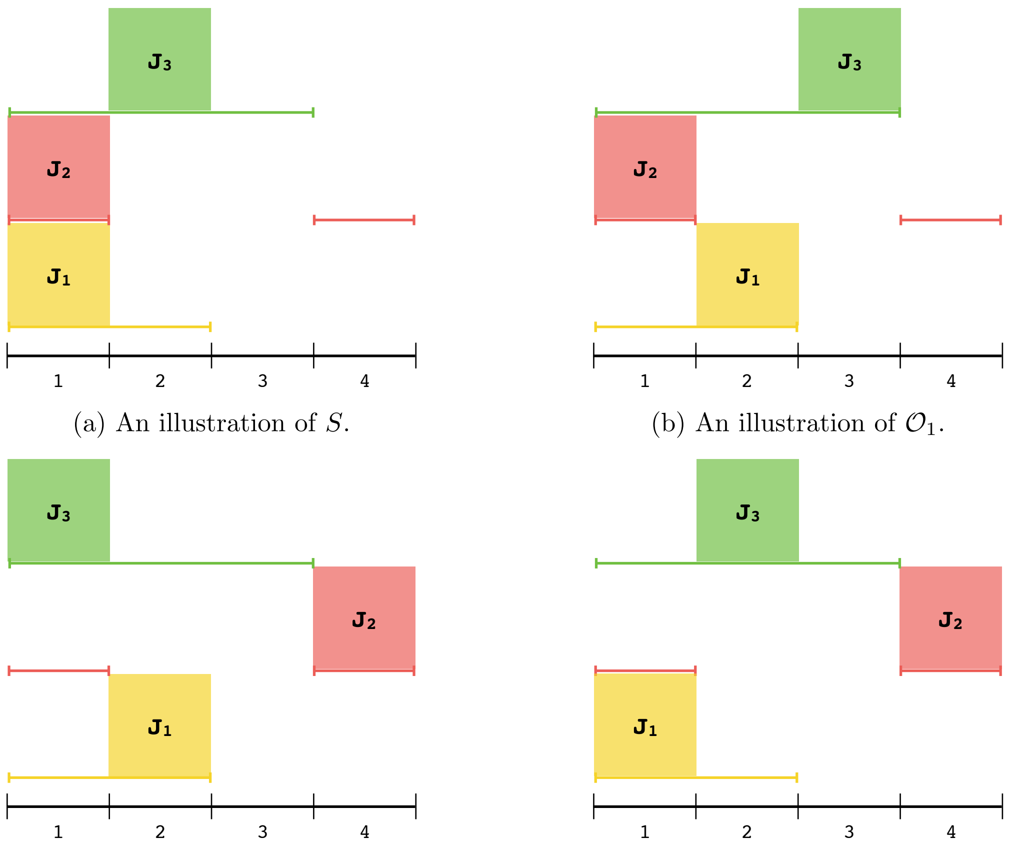
Lemma 4.8. *There exists an optimal assignment $\mathcal{O}$ such that $G_a(S, \mathcal{O})$ is acyclic.*

Proof. Consider an optimal assignment $\mathcal{O}'$ such that $G_a(S, \mathcal{O}')$ contains directed cycles. We show that the assignment $\mathcal{O}'$ can be transformed into an optimal assignment $\mathcal{O}$ such that $G_a(S, \mathcal{O})$ is acyclic. Recall that each timeslot is represented by a vertex in $G_a(S, \mathcal{O}')$ and an arc from vertex s to vertex t labelled by a tuple $(J_j,\ +/-/=)$ means that J_j is assigned to timeslot s in assignment S and timeslot t in $\mathcal{O}'$. For every cycle (s, t) such that $s = t$ in $G_a(S, \mathcal{O}')$, we show that the load of any vertex does not change after executing all the moves in the cycle. This implies that the total cost of $\mathcal{O}'$ remains the same after removing all cycles from $G_a(S, \mathcal{O}')$.

We consider a cycle that contains the vertices $[s, v_1, v_2, \cdots , v_k, t]$, for $s = t$. There are arcs from s to v_1, v_1 to v_2, and so on, until the last arc from v_k to $t = s$. An arc denotes the moving of a distinct job each step. As we move one job from s to v_1, $\ell oad(s)$ decreases by one and $\ell oad(v_1)$ increases by one. However, $\ell oad(v_1)$ returns to the original value as we move the respective job from vertex v_1 to v_2. Thus, $\ell oad(v_i)$, for $1 \leq i < k$ remains unchanged. As we move the last job from vertex v_k to $t = s$, both $\ell oad(v_k)$ and $\ell oad(s)$ return to their original value. Clearly, the load of all vertices remains the same even for cycles of size 2. Thus, the cost of $\mathcal{O}'$ remains the same after removing all cycles from $G_a(S, \mathcal{O}')$ and we denote the corresponding agreement graph by $G_a(S, \mathcal{O})$. □

We have an example to illustrate the previous proof; we refer to Example 4.1 and Figure 4.1. Consider two optimal solutions in Example 4.1: $\mathcal{O}_1$ and $\mathcal{O}_2$ (Figure 4.5b and 4.5c). In $\mathcal{O}_1$, J_1, J_2, and J_3 are assigned to v_2, v_1, and v_3, respectively. And in $\mathcal{O}_2$, J_1, J_2, and J_3 are assigned to v_2, v_4, and v_1, respectively.

Refer to the schedule S, $\mathcal{O}_1$, and $\mathcal{O}_2$ in Figure 4.5, Figure 4.6 shows agreement graphs $G_a(S, \mathcal{O}_1)$ and $G_a(S, \mathcal{O}_2)$ for assignment S where J_1, J_2, and J_3 are assigned to v_1, v_1, and v_2, respectively (Figure 4.5a). In Figure 4.6b, the agreement graph $G_a(S, \mathcal{O}_2)$ contains a cycle, yet an alternative optimal assignment $\mathcal{O}_2'$ exists such that $G_a(S, \mathcal{O}_2')$



(a) An illustration of S. (b) An illustration of $\mathcal{O}_1$.

(c) An illustration of $\mathcal{O}_2$. (d) An acyclic alternative of $\mathcal{O}_2$.

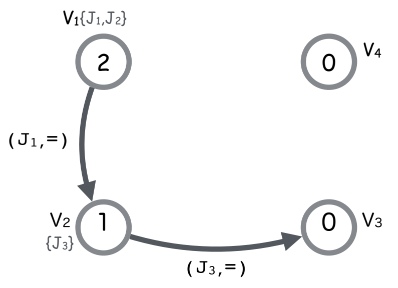
Figure 4.5: Illustration of assignments in Example 4.1.

contains no cycles, as depicted in Figure 4.6. Figure 4.5d shows $\mathcal{O}_2'$, an alternative of $\mathcal{O}_2$ without cycles, generated by the instructions mentioned in proof of Lemma 4.8.

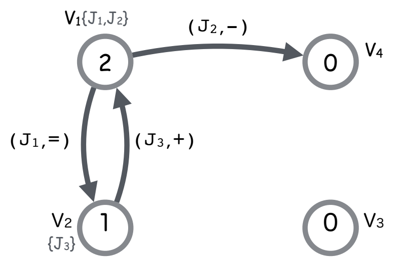
Lemma 4.9. *Suppose S is not optimal and $\mathcal{O}$ is an optimal assignment such that the agreement graph $G_a(S, \mathcal{O})$ is acyclic. Then we can have a sequence of agreement graphs $G_a(S_1, \mathcal{O})$, $G_a(S_2, \mathcal{O}), \cdots, G_a(S_k, \mathcal{O})$ such that $S_1 = S$, $S_k = \mathcal{O}$, and the cost is non-increasing every step.*

Proof. Consider the agreement graph $G_a(S_i, \mathcal{O})$, for $i \geq 1$, starting from $S_1 = S$. In each step, from $G_a(S_i, \mathcal{O})$ to $G_a(S_{i+1}, \mathcal{O})$, one arc is removed. For $i \geq 1$, we consider in $G_a(S_i, \mathcal{O})$ any arc labelled with either a "$-$" or an "$=$" and we execute the move corresponding to this arc. Through this move, we remove one arc, and thus we do not introduce any new arcs. However, the $+/-/=$ label of other arcs may change. If the resulting graph $G_a(S_{i+1}, \mathcal{O})$ does not contain any more "$-$" or "$=$" arcs, we stop. Otherwise, we repeat the process.

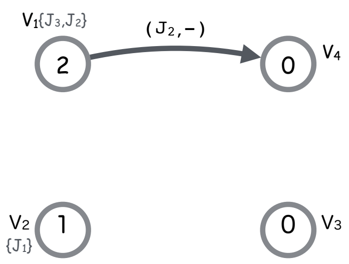
Note that the cost is non-increasing in every step since we only perform those move which is labeled with "$-$" or an "$=$". By the time we stop, if the resulting graph, say, $G_a(S_h, \mathcal{O})$, does not contain any more arcs, we have obtained the desired sequence of agreement graphs. Otherwise, we are left only with "$+$" labelled arcs in $G_a(S_h, \mathcal{O})$;



(a) The agreement graph $G_a(S, \mathcal{O}_1)$.



(b) The agreement graph $G_a(S, \mathcal{O}_2)$.



(c) The agreement graph $G_a(S, \mathcal{O}'_2)$.

Figure 4.6: Two agreement graphs for the same assignment S and two different optimal schedules $\mathcal{O}_1$ and $\mathcal{O}_2$ in Figure 4.5.

however, in the following, we shall show that such a case cannot happen, thus completing the proof of the lemma.

Firstly, $\text{cost}(S_h) \geq \text{cost}(\mathcal{O})$ since $\mathcal{O}$ is an optimal assignment. Next, by Lemma 4.8, the agreement graph $G_a(S_1, \mathcal{O})$ is acyclic and the resulting graph $G_a(S_h, \mathcal{O})$ by removing all "−" and "=" labelled arcs is also acyclic. Thus, in $G_a(S_h, \mathcal{O})$, there must exist at least one vertex with in-degree 0 and one vertex with out-degree 0. We look at all such $[v_1, \cdots, v_i]$ paths in $G_a(S_h, \mathcal{O})$, where v_1 has in-degree 0, v_i has out-degree 0, and $v_1 \neq v_i$. For any such $[v_1, \cdots, v_i]$ path, we show that by executing all moves of the path (i) the overall cost is increasing, and (ii) the labels of all arcs not contained in the $[v_1, \cdots, v_i]$ path remain "+". After executing all moves of the path, all arcs of the $[v_1, \cdots, v_i]$ path are removed.

(i) Suppose the vertices of the path are $[v_1, v_2, \cdots, v_i]$ and $\ell oad(v_1) = x$. As all arcs in (v_1, v_i) are labelled with "+" (i.e., the cost is increasing), $\ell oad(v_j) \geq \ell oad(v_{j-1})$, for $j > 1$. By executing all moves in the path, $\ell oad(v_1) = x - 1$, $\ell oad(v_j)$ is unchanged, for $1 < j < i$, and $\ell oad(v_i)$ is increased by one. Thus, the overall cost is increasing.

(ii) We show that the labels of all arcs not contained in the $[v_1, \cdots, v_i]$ path remain "+". There may be out-going arcs from v_1 to other vertices not in the $[v_1, \cdots, v_i]$ path initially labelled by "+". Before executing all the moves in the $[v_1, \cdots, v_i]$ path, the load of all these vertices with out-going arcs from v_1 is at least x as we assume $\ell oad(v_1) = x$. After the move, $\ell oad(v_1) = x - 1$ and out-going arcs from v_1 point to vertices with load at least x. Thus, an arc from v_1 to any other vertex denotes a further increase in the cost and the labels of the arcs do not change. For vertices v_j, for $1 < j < i$, the load of v_j remains unchanged and thus the labels of the arcs incoming to or outgoing from v_j remain the same. For v_i, there may be incoming arcs. Suppose $\ell oad(v_i) = y$ before executing all the moves in the $[v_1, \cdots, v_i]$ path. Then the load of all other vertices pointing to v_i is at most y and the arcs are labelled by "+". After executing all the moves in the $[v_1, \cdots, v_i]$ path, $\ell oad(v_i) = y + 1$, and thus any subsequent moves from vertices pointing to v_i cause further increases in the cost, i.e., the labels do not change.

Thus, the overall cost is increasing. We repeat this process until there are no more such $[v_1, \cdots, v_i]$ paths. We end up with $\text{cost}(S_k) > \text{cost}(\mathcal{O})$, which contradicts the fact that $\text{cost}(S_k) = \text{cost}(\mathcal{O})$ as $S_k = \mathcal{O}$. Thus, the case where we are left only with "+" labelled arcs in $G_a(S_h, \mathcal{O})$ cannot happen, and the lemma follows. □

Lemma 4.10. *If there is no legal-path in the feasibility graph G_f, the corresponding assignment is optimal.*

Proof. Suppose by contradiction there is no legal-path in the feasibility graph G_f, but the corresponding assignment A is not optimal. Let A^*, $A_1 = A, A_2, \cdots, A_k = A^*$ be the assignments as defined in Lemma 4.9. Note that each arc in the agreement graph $G_a(A_1, A^*)$ corresponds to an arc in the feasibility graph G_f (since G_f captures all possible moves of $A = A_1$). Because the sequence of agreement graphs in Lemma 4.9 only involves removing arcs, each arc in all of $G_a(A_i, A^*)$ corresponds to an arc in G_f.

Suppose $G_a(A_j, A^*)$ is the first agreement graph in which a "−" labelled arc is considered between some timeslots t_α and t_β. If there is no such arc, then A is already an optimal solution (since the sequence will be both non-increasing by Lemma 4.9 and non-decreasing as no "−" labelled arc is involved). Otherwise, if there is such an arc in $G_a(A_j, A^*)$, we show that there must have existed a legal-path in the feasibility graph G_f, leading to a contradiction. We denote by $\ell oad(A_i, t)$ the load of timeslot t in the agreement graph $G_a(A_i, A^*)$. Suppose $\ell oad(A_j, t_\alpha) = x$, then $\ell oad(A_j, t_\beta) \leq x - 2$ as the overall energy cost would be decreasing by moving a job from t_α to t_β. If $\ell oad(A_1, t_\alpha) = x$ and $\ell oad(A_1, t_\beta) \leq x - 2$ in the original assignment, then there is a legal-path in G_f, which is a contradiction. Otherwise, we claim that there are some timeslots u_{i_y} and v_{k_z} such that $\ell oad(A_1, u_{i_y}) \geq x$ and $\ell oad(A_1, v_{k_z}) \leq x - 2$, and there is a path from u_{i_y} to v_{k_z} in G_f. This forms a legal-path in G_f, leading to a contradiction.

To prove the claim, we first consider finding u_{i_y}. We first set $i_0 = j$ and $u_{i_0} = t_\alpha$. If $\ell oad(A_1, u_{i_0}) \geq x$, we are done. Else, since $\ell oad(A_j, u_{i_0}) = x$ and $\ell oad(A_1, u_{i_0}) < x$, there must be some job that is moved to u_{i_0} before A_j. Let $i_1 < i_0$ be the latest step such that a job is assigned to u_{i_0} and the job is moved from u_{i_1}. Note that since this move corresponds to an arc with label "=", $\ell oad(A_{i_1}, u_{i_1}) = x$ and $\ell oad(A_{i_1}, u_{i_0}) = x - 1$.

If $\ell oad(A_1, u_{i_1}) \geq x$, we are done. Otherwise, we can repeat the above argument to find u_{i_2} and so on. The process must stop at some step $i_y < i_0$ where $\ell oad(A_1, u_{i_y}) \geq x$. Similarly, we set $k_0 = j$ and $v_{k_0} = t_\beta$, so that we can find a step $k_z < k_0$ such that $\ell oad(A_1, v_{k_z}) \leq x - 2$. Recall that since each arc in $G_a(A_1, A^*)$ corresponds to an arc in the feasibility graph G_{f} and in all subsequent agreement graphs we only remove arcs, there is a path from u_{i_y} and v_{k_z} in G_{f}. Therefore, we have found a legal-path from u_{i_y} to v_{k_z} in G_{f}. □

Theorem 4.11. *Algorithm $\mathcal{A}_{FG}$ finds an optimal assignment.*

4.2.3 Time Complexity

We analyze the time complexity of our algorithm and show that this can be improved when the feasible timeslots associated with each job form a contiguous interval.

Noncontiguous Feasible Timeslots

In this section, we consider the case where jobs have noncontiguous feasible timeslots and show that the time complexity is $O(n^2\tau)$.

Input set. In this case, each job J_j associates with feasible timeslots $I(J_j)$. Assume that there are τ_j timeslots in $I(J_j)$ and let $\tau' = \sum_j \tau_j$, the input size of job J_j is $O(\log n + \tau_j \log \tau')$. Hence, the total input size is $O(n \log n + \tau' \log \tau')$.

Analysis. Now we show that the feasibility graph approach can be done in polynomial time.

Theorem 4.12. *We can find the optimal schedule in $O(n^2\tau)$ time.*

Proof. We assign jobs one by one. Each round when we assign the job J to timeslot t, we add arcs (t, w) labelled by J for all vertices w that $w \in I(J)$ in the feasibility graph. By Lemma 4.4, there is a legal-path starting from t if there is a legal-path after assigning J to timeslot t. When J is assigned to t, we start breadth-first search (BFS) at t. By Lemma 4.5, if there is a node w which can be reached by the search and the number of jobs assigned to w is two less than the number of jobs assigned to t, it means that there is a legal-path $[t, \cdots, w]$. Then we shift the jobs according to the $[t, \cdots, w]$ legal-path. After shifting there will be no legal-paths anymore by Lemma 4.7. Finally we update the arcs of the vertices on the legal-path in the feasibility graph.

Adding J to the feasibility graph needs $O(|I(J)|)$ time. Because $|I(J)|$ is at most the total number of timeslots in T, $|I(J)| = O(\tau)$ where τ is the number of timeslots. The BFS takes $O(\tau + n\tau)$ time because there are at most $n\tau$ arcs in the feasibility graph. If a legal-path exists after assigning J and its length is l, the shifting needs $O(l)$ time, which is $O(\tau)$ because there are at most τ vertices in the legal-path. After the shift, the final step to update the arcs of the vertices on the legal-path takes at most $O(n\tau)$ time because there are at most $n\tau$ arcs in the feasibility graph. The total time for assigning n jobs is thus bounded by $O(n^2\tau)$. □

Contiguous Intervals

In this section, we consider the special case where each job $J \in \mathcal{J}$ is associated with an interval of contiguous timeslots $I(J) = [r(J), d(J))$, for positive integers $r(J) < d(J)$. We show that we can improve the time complexity to $O(n \log \tau + \min\{n, \tau\} n \log n)$.

Input set. In this setting, each job J_j associates with a feasible interval $[r(J_j), d(J_j))$ where $r(J_j), d(J_j) \in [1, \tau]$. The input size of job J_j is $O(\log n + 2 \log \tau)$. Hence, the total input size is $O(n \log n + n \log \tau)$.

Framework. Recall that our algorithm first assigns a job J to a feasible timeslot s with minimum load and then executes a shift if there is a legal-path in the resulting feasibility graph. When the feasible timeslots of a job form a contiguous interval, we will use several data structures to help finding a legal-path. In particular, we first find a path from s such that the end point has the minimum load. If this path is a legal-path, we execute the shift, otherwise, there is no legal-path from s. To find such a path, we exploit the notion of l-reachable intervals (to be defined). To compute reachable intervals, we maintain two heaps for each timeslot t to store $r(J_i)$ and $d(J_i)$ of jobs J_i which are assigned to t. To find the timeslot with minimum load in a reachable interval, we use a data structure that supports dynamic range minimum query (RMQ). Before we give the detailed analysis, we first define a few notions on a feasibility graph.

Reachable interval. For every timeslot t, the *l-reachable interval* of t, denoted by $\mathcal{R}_t^{(l)}$, is defined to be the set of timeslots s such that there is a path from t to s with length at most l. We define $\mathcal{R}_t^{(0)} = \{t\}$ and $\mathcal{R}_t^{(-1)} = \emptyset$. Note that $\mathcal{R}_t^{(1)} = \cup_{st(J)=t} I(J)$ (that is, the union of feasible intervals for all jobs J which are assigned at t). We call $\mathcal{R}_t^{(1)}$ the *directly reachable interval* of t. The set $\mathcal{R}_t^{(1)}$ is a contiguous interval containing t since the feasible timeslots of each job form a contiguous interval and the feasible timeslots of any job that is assigned to t must contain t.

Notice that $\mathcal{R}_t^{(l+1)}$ is the union of the directly reachable intervals of each timeslot in $\mathcal{R}_t^{(l)}$. For any $s \in \mathcal{R}_t^{(l)}$, as observed above, the directly rearchable interval of the timeslot s must contain s and thus is a contiguous interval overlapping $\mathcal{R}_t^{(l)}$. Therefore, $\mathcal{R}_t^{(l+1)}$ forms a contiguous interval and $\mathcal{R}_t^{(l)} \subseteq \mathcal{R}_t^{(l+1)}$ for $l \geq 0$. We denote the interval $\mathcal{R}_t^{(l)}$ as $[\alpha_t^{(l)}, \beta_t^{(l)}]$. In particular, for directly reachable intervals, $\alpha_t^{(1)} = \min_{st(J)=t} r(J)$ and $\beta_t^{(1)} = \max_{st(J)=t} d(J)$.

Depth. We note that if $\mathcal{R}_t^{(l+1)}$ is the same as $\mathcal{R}_t^{(l)}$, then for any $k \geq l$, $\mathcal{R}_t^{(k)}$ is also the same as $\mathcal{R}_t^{(l)}$. We define the *depth* D_t of t to be the smallest integer l such that $\mathcal{R}_t^{(l+1)} = \mathcal{R}_t^{(l)}$. Note that the depth D_t is the longest length of the shortest paths starting from t to any other connected vertex in the feasibility graph. Furthermore, the D_t-reachable interval of t is the set of all timeslots such that there is a path from t.

Path-finder-jobs. Consider any $1 \leq l \leq D_i$. The definition of D_t implies that $\mathcal{R}_t^{(l)} \supsetneq \mathcal{R}_t^{(l-1)}$. We define the *left-path-finder-job* (*right-path-finder-job* resp.) of $\mathcal{R}_t^{(l)}$ as the job J (with smallest job index) such that $st(J) \in \mathcal{R}_t^{(l-1)}$ and $r(J) = \alpha_i^{(l)}$ ($d(J) = \beta_i^{(l)}$ resp.). We denote them by $\text{lpfj}(\mathcal{R}_t^{(l)})$ and $\text{rpfj}(\mathcal{R}_t^{(l)})$, respectively. Then we have the following property about path-finder-jobs, which then leads to a bound on D_t in

Property 4.2. Figure 4.7 shows an illustration of reachable intervals and the path-finder jobs of an assignment.

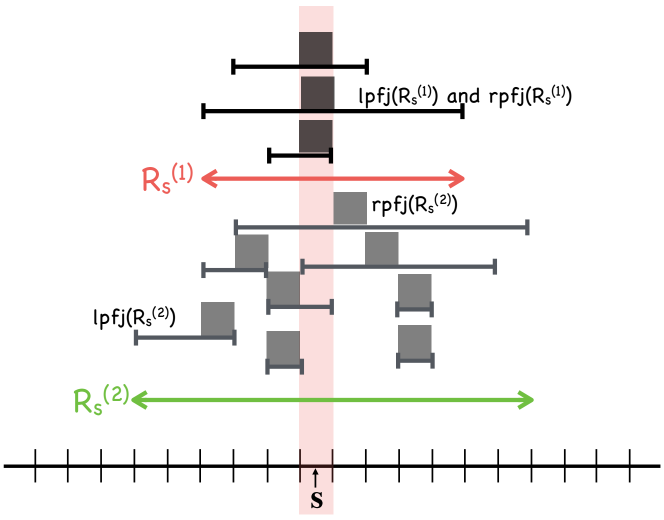


Figure 4.7: An illustration of reachable intervals and the path-finder jobs.

Algorithm 2 The process of assigning job J_j

```
$H_{\min}(t)$ is a min-heap containing the release times of the jobs assigned at $t$
$H_{\max}(t)$ is a max-heap containing the deadlines of the jobs assigned at $t$
$l \leftarrow 0$
$s \leftarrow \arg\min_{t \in I(J_i)} \ell oad(\mathcal{A}_{\text{FG}}, t)$
$\alpha_s^{(l)} \leftarrow s$
$\beta_s^{(l)} \leftarrow s$
while $\mathcal{R}_s^{(l-1)} \neq \mathcal{R}_s^{(l)}$ do
    $l \leftarrow l + 1$
    $\alpha_s^{(l)} \leftarrow \alpha_s^{(l-1)}$
    $\beta_s^{(l)} \leftarrow \beta_s^{(l-1)}$
    for $t \in \mathcal{R}_s^{(l)} \setminus R_s^{(l-1)}$ do
        if $\alpha_s^{(l)} >$ the min value in $H_{\min}(t)$ then
            $\text{lpfj}(\mathcal{R}_t^{(l)}) \leftarrow \arg\min_{st(\mathcal{A}_{\text{FG}},i)=t} r(J_i)$
            $\alpha_s^{(l)} \leftarrow$ the min value in $H_{\min}(t)$
        if $\beta_s^{(l)} <$ the max value in $H_{\max}(t)$ then
            $\text{rpfj}(\mathcal{R}_t^{(l)}) \leftarrow \arg\max_{st(\mathcal{A}_{\text{FG}},i)=t} d(J_i)$
            $\beta_s^{(l)} \leftarrow$ the max value in $H_{\max}(t)$
$t \leftarrow \arg\min_{t' \in [\alpha_s^{(D_t)}, \beta_s^{(D_t)})} \ell oad(\mathcal{A}_{\text{FG}}, t')$
Rebuild the legal-path from $s$ to $t$
Shift the jobs along the legal-path
```

Property 4.1. *For* $1 \leq l \leq D_t$, $\text{lpfj}(\mathcal{R}_t^{(l)})$ *or* $\text{rpfj}(\mathcal{R}_t^{(l)})$ *is assigned to a timeslot in* $\mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$.

Proof. By the definition of D_t, $\mathcal{R}_t^{(l)} \supsetneq \mathcal{R}_t^{(l-1)}$, implying that $\alpha_t^{(l)} < \alpha_t^{(l-1)}$ or $\beta_t^{(l)} > \beta_t^{(l-1)}$. Consider the former case. Let $J = \text{lpfj}(\mathcal{R}_t^{(l)})$; i.e., $r(J) = \alpha_t^{(l)}$. We claim that

$st(J) \in \mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$; otherwise, $st(J) \in \mathcal{R}_t^{(l-2)}$ implying $\alpha_t^{(l-1)} \leq r(J)$, which is contradicting to $r(J) = \alpha_t^{(l)}$ and $\alpha_t^{(l)} < \alpha_t^{(l-1)}$. Using a similar argument, the latter case implies that rpfj$(\mathcal{R}_t^{(l)})$ is assigned to a timeslot in $\mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$. Combining the two cases, the property holds. □

Property 4.2. *For every timeslot t,* (i) $D_t \leq \min\{n, \tau\}$*;* (ii) *the number of timeslots in $\mathcal{R}_t^{(D_t)}$ that have jobs assigned to them is at most* $\min\{n, \tau\}$*.*

Proof. (i) We observe that $D_t \leq \tau$ because $\mathcal{R}_t^{(l)} \supsetneq \mathcal{R}_t^{(l-1)}$ for $1 \leq l \leq D_t$. On the other hand, by Property 4.1, there is at least one job assigned to a timeslot in $\mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$ for every $1 \leq l \leq D_t$ and each job is only assigned to one timeslot, therefore, $D_t \leq n$. (ii) is trivial. □

Analysis. To compute reachable intervals, we have to know the minimum of $r(J)$ and maximum of $d(J)$ of jobs J assigned to each timeslot. We use two heaps for each timeslot to keep this information: a min-heap (max-heap resp.) keeps the starting timeslot $r(J)$ (ending timeslot $d(J)$ resp.) of all jobs assigned to the timeslot. Using these two heaps we can compute the directly reachable interval of any timeslot in $O(1)$ time. When a job is assigned to or moved away from a timeslot, the corresponding heaps have to be updated and each such update takes $O(\log n)$ time since the size of the heap is bounded by the total number of jobs. By Property 4.2 (i), the update time for each newly assigned job is bounded by $O(\min\{n, \tau\} \log n)$.

Lemma 4.13. *For each timeslot t, we can compute D_t-reachable intervals in* $O(\min\{n, \tau\})$*-time.*

Proof. As described above, we can compute directly reachable interval in $O(1)$ time. By Property 4.1, to compute $\mathcal{R}_t^{(l)}$, we only need to consider timeslots in $\mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$ by checking the corresponding heaps; hence, each timeslot in the D_t-reachable interval needs to be considered in the computation of one l-reachable interval only. The number of timeslots in the D_t-reachable interval could be τ. However, we only need to consider those "occupied" timeslots that have jobs assigned to them. We can keep links among occupied timeslots by a doubly linked list. Each occupied timeslot s is linked to two nearest occupied timeslots $s_l < t$ and $s_r > t$. In this way, we can skip non-occupied timeslots and only check occupied timeslots in $\mathcal{R}_t^{(l-1)} \setminus \mathcal{R}_t^{(l-2)}$ when we compute $\mathcal{R}_t^{(l)}$. By Property 4.2 (ii), the number of occupied timeslots in the D_t-reachable interval is at most $\min\{n, \tau\}$ and the overall computation takes $O(\min\{n, \tau\})$ time. □

Using Lemma 4.13, we can analyze the overall time complexity which takes into account also the time taken to update various data structures.

Theorem 4.14. *We can find the optimal schedule in* $O(n \log \tau + \min\{n, \tau\} n \log n)$*-time for the case where the feasible timeslots associated with each job form a contiguous interval.*

Proof. For each newly considered job which is assigned to s, we first compute the D_s-reachable interval $\mathcal{R}_s^{(D_s)}$ in $O(\min\{n, \tau\})$ time. We then check the existence of a legal-path from s by examining the timeslot t in $\mathcal{R}_s^{(D_s)}$ with the minimum load. If the load $\ell oad(s) - \ell oad(t) \geq 2$, then there is a legal-path, otherwise, there is no legal-path. This timeslot t can be found by using a simple balanced binary tree structure that supports dynamic range minimum query (RMQ). We store the load of the timeslots 1, 2, $\cdots, \tau$ in the leaves from left to right. Each internal node maintains the minimum load in its subtree. Using this data structure, we can return the minimum load in any time interval $[x, y]$ in $O(\log \tau)$ time. The value from the root to a leaf needs to be modified when the load of a leaf is changed, and such update takes $O(\log \tau)$ time. When a new job is assigned and a possible shift takes place, at most two timeslots have their load changed. Therefore, the update of the dynamic RMQ structure takes $O(\log \tau)$ time for each job assigned.

If there exists a legal-path from s to t, we construct a legal-path with length at most D_s as follows (see Algorithm 3). Suppose $t \in \mathcal{R}_s^{(k)} \setminus \mathcal{R}_s^{(k-1)}$ for some $l \leq D_s$. We construct a legal-path $[s = v_0, v_1, \cdots, v_k = t]$ in a bottom-up fashion. If v_k is on the right extension of $\mathcal{R}_s^{(k-1)}$, i.e., $v_k > \beta_s^{(k-1)}$, then we set v_{k-1} to be the timeslot that rpfj$(\mathcal{R}_s^{(k)})$ is assigned to; otherwise, i.e., $v_k < \alpha_s^{(k-1)}$, we set v_{k-1} to be the timeslot that lpfj$(\mathcal{R}_s^{(k)})$ is assigned to. By the definition of path-finder-jobs, v_{k-1} has jobs assigned to it and one of these jobs (rpfj$(\mathcal{R}_s^{(k)})$ or lpfj$(\mathcal{R}_s^{(k)})$ accordingly) has a feasible interval covering v_k, hence the arc (v_{k-1}, v_k) exists in the feasibility graph and the arc is labelled by rpfj$(\mathcal{R}_s^{(k)})$ or lpfj$(\mathcal{R}_s^{(k)})$ correspondingly. Inductively, we can define v_{j-1} from v_j, for $j = k, k-1, \cdots, 1$, until we reach s.

Given the legal-path found, we execute a shift along the path. We need to update the heaps of at most D_s timeslots, thus taking $O(\min\{n, \tau\} \log n)$ time, by Property 4.2 (i). The doubly linked list in the proof of Lemma 4.13 can be updated in $O(1)$ time when a job is added or removed from a timeslot, and hence updating at most D_s timeslots takes $O(\min\{n, \tau\})$ time.

In summary the time taken for assigning a new job (including update of data structures) is bounded by $O(\log \tau + \min\{n, \tau\} \log n)$. Therefore, the overall time complexity for assigning n jobs is $O(n \log \tau + \min\{n, \tau\} n \log n)$ and the theorem follows. □

Figure 4.8 shows an illustration of legal path from s to t. The shifting is according to the description in the proof of Theorem 4.14.

4.2.4 Using a Discrete DVS Algorithm

In this section we aim to demonstrate how to solve the GRID problem where the jobs have unit size and contiguous feasible interval by reducing to the Discrete DVS problem. It helps us to understand the relationship between the GRID problem and the DVS problem better.

Algorithm 3 The process of rebuilding the $[s, \cdots, t]$ legal-path and shifting after assigning J_j

```
**Input:** $s, t \in \mathcal{R}_s^{(k)} \setminus \mathcal{R}_s^{(k-1)}, \text{lpfj}(\mathcal{R}_s^{(0)}), \text{lpfj}(\mathcal{R}_s^{(1)}), \cdots, \text{lpfj}(\mathcal{R}_s^{(D_t)}), \text{rpfj}(\mathcal{R}_s^{(0)}), \text{rpfj}(\mathcal{R}_s^{(1)}), \cdots, \text{rpfj}(\mathcal{R}_s^{(D_t)}), \alpha_s^{(0)}, \alpha_s^{(1)}, \cdots, \alpha_s^{(D_t)}, \beta_s^{(0)}, \beta_s^{(1)}, \cdots, \beta_s^{(D_t)}$
$t_k \leftarrow t$
for i = k to 2 do
    if $t_i < \alpha_s^{(i-1)}$ then
        $J_{(i)} \leftarrow \text{lpfj}(\mathcal{R}_s^{(i)})$
        $t_{i-1} \leftarrow st(\mathcal{A}_{\text{FG}}, \text{lpfj}(\mathcal{R}_s^{(i)}))$
    else
        $J_{(i)} \leftarrow \text{rpfj}(\mathcal{R}_s^{(i)})$
        $t_{i-1} \leftarrow st(\mathcal{A}_{\text{FG}}, \text{rpfj}(\mathcal{R}_s^{(i)}))$
if $t_1 < s$ then
    $J_{(1)} \leftarrow \text{lpfj}(\mathcal{R}_s^{(1)})$
else
    $J_{(1)} \leftarrow \text{rpfj}(\mathcal{R}_s^{(1)})$
for i=1 to k do
    Shift job $J_{(i)}$ to $t_i$
```

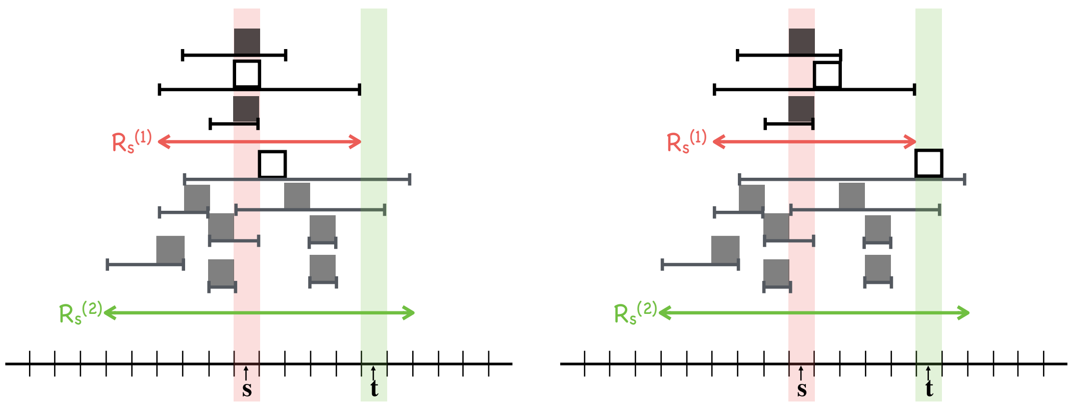


Figure 4.8: An illustration of legal path from s to t and the shifting.

In Section 3.2, we introduced the DVS problem and elaborated the difference between the GRID problem and the DVS problem. Simply speaking, the main differences of DVS problem to the GRID problem include (i) jobs in DVS can be preempted while preemption is not allowed in the GRID problem; (ii) as processor speed in DVS can scale, a job can be executed for varying time duration as long as the total work is completed while in the GRID problem a job must be executed for a fixed duration given as input; (iii) the work requirement $p(J)$ of a job J in DVS can be seen as $w(J) \times h(J)$ for the corresponding job in GRID.

More importantly, the DVS problem can be solved in polynomial time while the GRID problem is NP-hard except for the case where jobs have unit size. We investigate the GRID problem for unit-size jobs by relating to the DVS problem. More specifically, we relate the GRID problem to the discrete DVS problem. As mentioned in Section 3.2.2, unlike in DVS problem the speed of processor can be arbitrary, in Discrete DVS problem, the processor speed is restricted to a given set of speeds. We will use a polynomial time algorithm for Discrete DVS problem proposed by Li et al. [52] to solve the GRID problem.

The formal problem definition of Discrete DVS is as the following [51]. There are n jobs in the input job set $\mathcal{J}$. Each job J has three parameters: arrival time $r(J)$, deadline $d(J)$, and required work to be finished $p(J)$. The work $p(J)$ should be finished between $r(J)$ and $d(J)$ and preemption is allowed. At any time t, the processor speed $s(t)$ can be chosen from d given speed levels $s_1 > s_2 > \cdots > s_d$. Speed s means that the processor can do s units of work per unit of time. For example, if a job J with work $p(J)$ is executed at speed s, it needs $\frac{p(J)}{s}$ unit of time to finish the job. A schedule S of $\mathcal{J}$ needs to decide the speed and the jobs to be executed at any time t. A schedule S is feasible if all jobs are finished before their deadlines. The cost at time t is a convex function of the speed at t, denoted by $P(s(t))$. The total energy consumption by a schedule S is $E(S) = \int_{t=0}^{\infty} P(s(t))\mathrm{d}t$. The goal is to find a feasible schedule that minimize the total energy consumption.

The $O(n \log \max\{d, n\})$-time algorithm for Discrete DVS problem [52]. Li [52] proposed a $O(n \log \max\{d, n\})$-time algorithm to solve the Discrete DVS problem where n is the number of jobs and d is the number of allowed speeds. The basic idea of the algorithm is as follows. The time horizon $\mathcal{T}$ is partitioned into $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_d$ such that each $\mathcal{T}_i$ is a collection of time intervals and in each $\mathcal{T}_i$, the speed of each timeslot in the optimal schedule for continuous DVS problem is within $[s_i, s_{i+1})$. According to the $\mathcal{T}_i$s, the jobs in $\mathcal{J}$ can be partitioned into $\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_d$. Each $\mathcal{J}_i$ is corresponding to $\mathcal{T}_i$. The jobs in $\mathcal{J}_i$ have to be scheduled within $\mathcal{T}_i$ for each i using speed s_i or s_{i+1}. Finally, the authors proposed (s_i, s_{i+1})-schedule algorithm to feasibly schedule jobs in $\mathcal{J}_i$. The (s_i, s_{i+1})-schedule algorithm is based on the schedule using constant speed s_{i+1} (which may not be feasible.) By consulting the schedule using constant speed s_i, the final schedule is feasible and optimal.

By the following procedure we can transform an input job set $\mathcal{J}$ of GRID problem into an input job set $\mathcal{J}^*$ of Discrete DVS problem. For any job $J \in \mathcal{J}$, its corresponding job J^* has $r(J^*) = r(J)$, $d(J^*) = d(J)$, and $p(J) = w(J) \cdot h(J)$. Also, we set the set of allowed speeds $S = \{n, n-1, n-2, \cdots, 2, 1\}$. Note that the number of available speed $d = n$.

The output of Li's algorithm is a schedule S which is a pair of functions $(s(t), job(t))$ defined as the processor speed and the job to be executed at time t. However, it is not easy to transform the schedule to a schedule for the GRID problem. It is because that in the Discrete DVS schedule, a job can be execute at any speed in S. Moreover, at each timeslot, the schedule of Discrete DVS problem might have different speed. Hence we design our own (s_i, s_{i+1})-schedule algorithm instead using the one in [51].

Our (s_i, s_{i+1})-schedule algorithm (Algorithm 4). Recall that Li's algorithm partitions $\mathcal{J}$ and $\mathcal{T}$ into $\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_n$ and corresponding $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n$. For each $\mathcal{J}_i$

over $\mathcal{T}_i$, it can be feasibly scheduled by speed s_i and s_{i+1}. Our (s_i, s_{i+1})-schedule algorithm works for each pair of $\mathcal{J}_i$ and $\mathcal{T}_i$ as follows. For each timeslot t in $\mathcal{T}_i$, assign s_{i+1} available jobs in $\mathcal{J}_i$ at t by the EDF principle. At any timeslot t, if there are k jobs missing their deadlines, we set the "overflow" number of t, $o_t = k$ and ignore those jobs. When all timeslots in $\mathcal{T}_i$ are done and o_t is know for each $t \in \mathcal{T}_i$, we maintain a counter n_o which is 0 in the very beginning and schedule the jobs in a reverse sequence. That is, we go through the timeslot in $\mathcal{T}_i$ from right to left (instead of from left to right) and assign jobs in the *latest release time first* principle. At timeslot t, we add o_t to n_o. If $n_o > 0$, we schedule s_i jobs at t by the latest release time first principle and minus one from n_o; otherwise, we schedule s_{i+1} jobs at t by the latest release time first principle.

Algorithm 4 Our (s_i, s_{i+1})-schedule algorithm

```
**Input:** $\mathcal{T}_i = \{t_1, t_2, \cdots, t_{\tau_i}\}$, $\mathcal{J}_i = \{J_{(1)}, J_{(2)}, \cdots, J_{(n')}\}$, where the jobs are sorted by their deadlines
$n_o \leftarrow 0$
**for** $j = 1$ to $\tau_i$ **do**
    Assign $s_{i+1}$ available jobs to $t_j$
    $o_{t_j} \leftarrow$ the number of jobs which has deadline at $t$ and has not been assigned yet
Reorder the jobs in $\mathcal{J}_i$ such that the jobs are sorted by their release time
**for** $j = \tau_i$ to 1 **do**
    $n_o \leftarrow n_o + o_{t_j}$
    **if** $n_o > 0$ **then**
        Assign $s_i$ available jobs at $t_j$ by the latest release time first principle
        $n_o \leftarrow n_o - 1$
    **else**
        Assign $s_{i+1}$ available jobs at $t_j$ by the latest release time first principle
```

It is easy to see that the total number of assigned jobs will be equal to the number of jobs in $\mathcal{J}_i$. At each timeslot t, if $n_o > 0$, there must be s_{i+1} available jobs, otherwise, there should not be n_o overflowed jobs in the time interval $[t, t + n_o)$. In our (s_i, s_{i+1})-scheduling algorithm, we perform two scheduling with constant speed, while there are one in Li's algorithm. Hence the time complexity of our algorithm is at most twice of Li's algorithm. Also, in [52], Lemma 5.4, the authors proved that all (s_i, s_{i+1})-schedule have the same cost. We have the following lemma.

Lemma 4.15. *There exists an $O(n \log n)$-time algorithm to solve the* GRID *problem where there are input n jobs and each job has unit width, unit height and arbitrary contiguous feasible interval.*

Proof. According to [52], the $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_d$ and $\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_d$ can be found in $O(n \log n)$ time.

Now we prove that our (s_i, s_{i+1})-schedule can be done in $O(n_i \log n_i)$ time where $n_i = |\mathcal{J}_i|$ for each i. The sorting in the (s_i, s_{i+1})-schedule need $O(n_i \log n_i)$ time. By the definition of $\mathcal{T}_i$, $\tau_i \leq n_i$ for all $i > 0$. Also, each job is exactly considered twice, one for each round. Hence the time needs for our (s_i, s_{i+1})-schedule is $(n_i \log n_i)$.

Finally, the total time for all (s_i, s_{i+1})-schedules is $O(n_1 \log n_1 + n_2 \log n_2 + \cdots) = O((n_1 + n_2 + \cdots) \log n) = O(n \log n)$. □

4.3 Exact Algorithms for Jobs with Arbitrary Widths and Heights

In parameterized complexity theory, the complexity of a problem is not only measured in terms of the input size, but also in terms of parameters. The theory focuses on situations where the parameters can be assumed to be small, and the time complexity is exponential mainly because of these small parameters. The problems having such small parameters are captured by the concept "fixed-parameter tractability". An algorithm with parameters $p_1, p_2, \cdots$ is said to be an *fixed parameter algorithm* if it runs in $f(p_1, p_2, \cdots) \cdot O(g(N))$ time for any function f and any polynomial function g, where N is the size of input. A parameterized problem is *fixed-parameter tractable* if it can be solved by a fixed parameter algorithm. In this section, we show that the general case of GRID problem, jobs with arbitrary contiguous feasible intervals (that is, arbitrary release times and arbitrary deadlines), widths and heights, is fixed-parameter tractable with respect to a few small parameters. Table 4.1 summarizes our exact algorithms for different parameters.

Parameters	**Time complexity**
$w_{\max}, m, W_{\max}$	${w_{\max}}^{2m} \cdot (W_{\max} + 1)^{4m} \cdot O(n^2)$
$w_{\max}, m$	$(4m \cdot {w_{\max}}^2)^{2m} \cdot O(n^2)$

Table 4.1: Summary of our exact algorithms (n is the number of jobs; $w_{\max}$ is the maximum width of jobs; m is the maximum size of cliques; $W_{\max}$ is the maximum length of windows; k is the number of windows).

4.3.1 Key notions

We design two fixed parameter algorithms that are based on a dynamic programming fashion. Roughly speaking, we divide the timeline into k contiguous windows in a specific way, where each window W_i represents a time interval $[b_i, b_{i+1})$ for $1 \leq i \leq k$. The algorithm visits all windows accordingly from the left to the right and maintains a candidate set of schedules for the visited windows that no optimal solution is deleted from the set.

In the first fixed parameter algorithm, the parameters of the algorithm are the maximum width of jobs, the maximum number of overlapped feasible intervals and the maximum size of windows, where the latter two can be parameterized if we interpret the input job set as an "interval graph". In the second algorithm, we further drop out the last parameter. All these parameters do not increase necessarily as the number of jobs grows, and can be assumed to be small in practice. For example, a width of a job is a requested amount of time to run an appliance, and the running time is usually a few hours, which is small when we make a timeslot to be an hour. And the number of overlapped feasible intervals is at most the number of appliances.

Interval graph. A graph $G = (V, E)$ is an *interval graph* if it captures the intersection relation for some set of intervals on the real line. Formally, for each $v \in V$, we can associate v to an interval I_v such that (u, v) is in E if and only if $I_u \cap I_v \neq \emptyset$. It has been shown in [28, 32] that an interval graph has a "consecutive clique arrangement", i.e., its maximal cliques in an interval graph can be linearly ordered in a way that for every vertex v in the graph, the maximal cliques containing v occur consecutively in the linear order.

For any instance of the GRID problem, we can transform it into an interval graph $G = (V, E)$: For each job J with interval $I(J)$, we create a vertex $v(J) \in V$ and an edge is added between $v(J)$ and $v(J')$ if and only if $I(J)$ intersects $I(J')$. We can then obtain a set of maximal cliques in linear order, C_1, C_2, $\cdots$, C_k, by sweeping a vertical line from the left to the right, where k denotes the number of maximal cliques thus obtained. The parameter of our algorithm, the maximum number of overlapped feasible intervals, is just the maximum size of these maximal cliques.

Figure 4.9 shows a set of input jobs and the corresponding interval graph. The maximal cliques in this interval graph are $C_1 = \{v(J_1), v(J_2)\}$, $C_2 = \{v(J_2), v(J_3), v(J_4)\}$, $C_3 = \{v(J_2), v(J_4), v(J_5), v(J_6), v(J_7)\}$, $C_4 = \{v(J_2), v(J_5), v(J_8)\}$, $C_5 = \{v(J_2), v(J_9), v(J_{10})\}$, and $C_6 = \{v(J_2), v(J_{10}), v(J_{11})\}$.

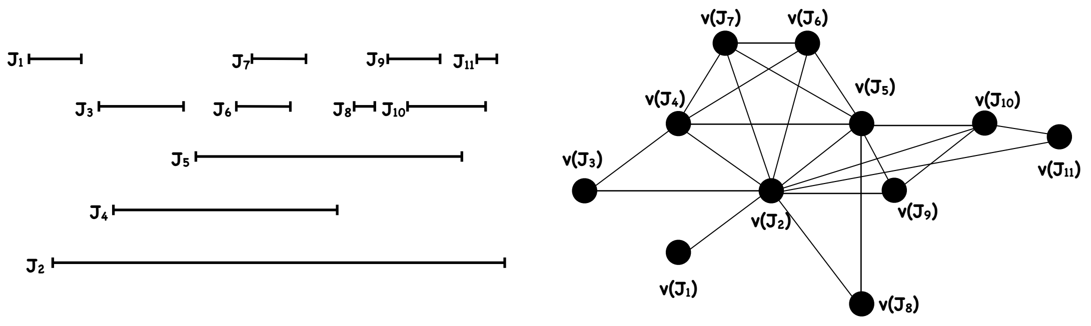


Figure 4.9: A set of input jobs and the corresponding interval graph.

Boundaries and windows. Based on the maximal cliques described above, we define some "windows" W_1, W_2, $\cdots$, W_k with "boundaries" b_1, b_2, $\cdots$, b_{k+1} as follows. We first give the definition of boundaries for the first algorithm. This definition will be generalized in section 4.3.4 for the second algorithm. For $1 \leq i \leq k$, the i-th *boundary* b_i is defined as the earliest release time of jobs in clique C_i but not in cliques before C_i (see Figure 4.11), precisely, $b_i = \min\{t \mid t = r(J_j) \text{ and } J_j \in C_i \setminus (\cup_{s=1}^{i-1} C_s)\}$. The rightmost boundary b_{k+1} is defined as the latest deadline among all jobs. With the boundaries, we partition the timeslots into contiguous intervals called *windows*. The i-th window W_i is defined as $[b_i, b_{i+1})$.

Figure 4.11 shows the boundaries and windows corresponding to the input in Figure 4.9. It is easy to see that each window W_i is corresponding to the clique C_i. That is, the jobs J with $I(J) \cap [b_i, b_{i+1})$ forms clique C_i. For example, the jobs passing W_3 are J_2, J_4, J_5, J_6, and J_7, which are the jobs in C_3. Moreover, each job is contained within at least one window. If there are multiple windows which contain a job J, the windows must form a contiguous sequence. That is, the windows containing certain job J are $W_i, W_{i+1}, W_{i+2}, \cdots, W_j$ for some $i \leq j$. For example, the job J_5 is contained in W_3, W_4, and W_5.

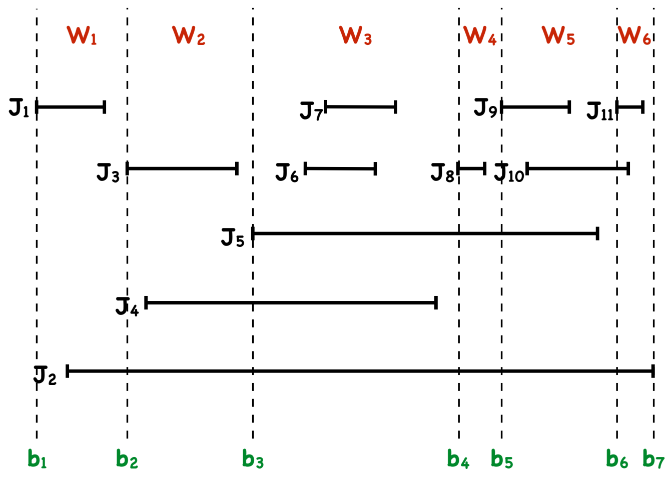


Figure 4.10: The windows corresponding to the input in Figure 4.9.

4.3.2 Framework of the algorithms

We propose two exact algorithms, both of which runs in k stages corresponding to each of the k windows. We maintain a table T_{left} that stores all "valid" configurations of jobs in all the windows that have been considered so far. A configuration of a job corresponds to an execution segment. And a row in the table consists of the configurations of all the jobs. In addition, for each window W_i, we compute a table T_{right_i} to store all possible configurations of start and end time of jobs available in W_i. The configurations in T_{right_i} would then be "concatenated" to some configurations in T_{left} that are "compatible" with each other. These merged configurations will be filtered to remove those non-optimal ones. The remaining configurations will become the new T_{left} for the next window. To describe the details of the algorithm, we explain several notions below. We denote by W_{left} the union of the windows corresponding to T_{left}. More formally, in the i-th stage, $W_{\text{left}} = \cup_{j<i} W_j$. And we use T_{right} to denote T_{right_i} when the context is clear.

Configurations. A *configuration* $F_i(J)$ of job J in window W_i is an "execution segment", denoted by $[st_i(J), et_i(J))$ contained completely by W_i. An execution segment can be seen as a part of the whole execution interval. That is, assume the execution interval of job J in the schedule is $[st(J), et(J))$, $[st_i(J), et_i(J)) = [st(J), et(J)) \cap W_i$. For a collection C of jobs, we use $F_i(C)$ to denote the set of configurations of all jobs in C, and $F_{\text{left}}(J)$ and $F_{\text{left}}(C)$ for the counterparts corresponding to T_{left}. The cost of $F_i(C)$ is the cost corresponding to the execution segments in $F_i(C)$. That is, $\text{cost}(F_i(C)) = \sum_{t \in W_i} (\sum_{J \in C : t \in F_i(J)} h(J_j))^\alpha$.

We want to enumerate all possible configurations in each window W_i. That is, for each job J, $st_i(J)$ can be chosen from $\{b_i, b_i+1, \cdots, b_{i+1}-1\} \cup \{b_i - 1, b_{i+1}\}$ and $et_i(J)$ can be chosen from $\{b_i+1, b_i+2, \cdots, b_{i+1}\} \cup \{b_i, b_{i+1}+1\}$. We say that $st_i(J) \in W_i$ or $et_i(J) \in W_i$ if $st_i(J) \in [b_i, b_{i+1})$ or $et_i(J) \in (b_i, b_{i+1}]$ respectively. And J is executed in W_i if both $st_i(J) \in W_i$ and $et_i(J) \in W_i$ hold. There are three other special configurations (the last three configurations in Figure 4.11):

- Setting $st_i(J) = b_i - 1$ and $et_i(J) = b_i$ means J is executed completely before W_i.
- Setting $st_i(J) = b_{i+1}$ and $et_i(J) = b_{i+1} + 1$ means J starts execution after W_i.
- Setting $st_i(J) = b_i - 1$ and $et_i(J) = b_{i+1} + 1$ means J starts execution before W_i, crosses the whole window W_i, and ends execution after W_i.

Figure 4.11 shows an illustration of all possible configurations of J_5 in window W_4.

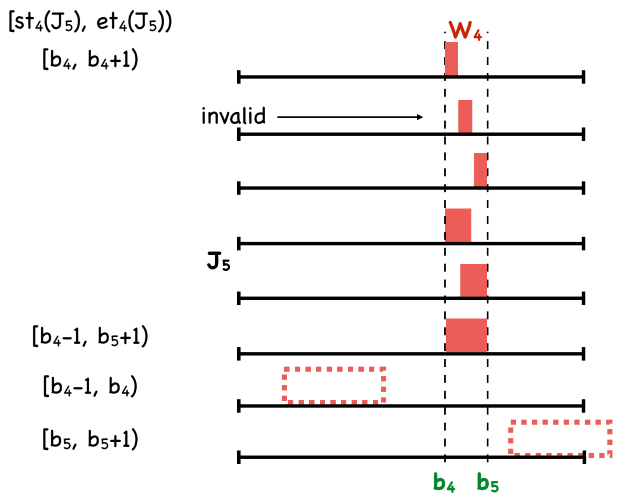


Figure 4.11: Configurations $F_4(J_5)$.

Validity. To make sure that the final schedule is a feasible one, we have to make sure that the configurations are valid. A configuration $F_i(J)$ is *invalid* if one of the following conditions hold:

1. $st_i(J) \geq et_i(J)$;
2. $et_i(J) > st_i(J) + w(J_j)$ meaning that the length of execution segment of J is larger than the width of J;

3. $(et_i(J) < st_i(J) + w(J_j)) \wedge (st_i(J) \geq b_i) \wedge (et_i(J) \leq b_{i+1})$ meaning that the length of execution segment of J is smaller than the width of J;

4. $(st_i(J) < r(J_j)) \wedge (st_i(J) < b_{i+1})$ meaning that the start time of J is earlier than the release time of J;

5. $(et_i(J) > d(J_j)) \wedge (et_i(J) > b_i)$ meaning that the end time of J exceeds the deadline of J.

Note that for $F_{\text{left}}(J)$, the validity is defined on the boundaries b_1 (instead of b_i) and b_{i+1}. And for T_{left}, $F_{\text{left}}(J)$ is also invalid if $st_{\text{left}}(J) = b_1 - 1$ since there is no window on the left of W_{left}. Similarly, $F_k(J)$ is invalid if $et_k(J) = b_{k+1} + 1$. A configuration $F_i(C)$ is *invalid* if there exists $J \in C$ such that $F_i(J)$ is invalid.

Compatibility. For each job, the configurations of it in different windows should be consistent, which means the configurations should not state contradicted schedules. We say that the configurations are *compatible* if they do not contradict to each other. For job J, the two configurations $F_{\text{left}}(J)$ and $F_i(J)$ are compatible if:

1. The configuration $F_{\text{left}}(J)$ states that J is executed in W_{left}, and the configuration $F_i(J)$ states that J is executed before W_i.

2. The configuration $F_{\text{left}}(J)$ states that J starts execution in W_{left} and ends execution after W_{left}, and the configuration $F_i(J)$ states that J starts execution before W_i and ends execution either in W_i or after W_i.

3. The configuration $F_{\text{left}}(J)$ states that J is executed completely after W_{left}, and the configuration $F_i(J)$ states that J does not start before W_i.

Concatenating configurations. To concatenate two configurations $F_{\text{left}}(J)$ and $F_i(J)$, we create a new $F_{\text{left}}(J)$ by the following setting based on the three types of compatible configurations described in the previous paragraph: for type (i), $st_{\text{left}}(J)$ and $et_{\text{left}}(J)$ leave unchanged; for type (ii), $st_{\text{left}}(J)$ leaves unchanged and set $et_{\text{left}}(J) \leftarrow et_i(J)$; and for type (iii), set $st_{\text{left}}(J) \leftarrow st_i(J)$ and $et_{\text{left}}(J) \leftarrow et_i(J)$. *Concatenating* $F_{\text{left}}(C)$ and $F_i(C)$ is to concatenate the configurations of each job in C. The corresponding cost is simply adding the cost of the two configurations.

Note that the concatenation of two valid configurations might not be valid, hence we have check the validity after concatenation. Figure 4.12 shows examples of concatenation. The black block in the middle indicates the width and height of the job J_5. The example in green is a valid configuration after concatenation while the example in red is an invalid one since the width of the concatenated job dose not fit the width of J_5.

Uncertainty and identity. A configuration $F_i(J)$ is *uncertain* if $et_i(J) = b_{i+1} + 1$ meaning that the end time of J is not determined yet, and we are not sure at the i-th stage whether $F_i(J)$ will be valid after concatenating $F_i(J)$ and $F_{i+1}(J)$. Note that the certain jobs will not affect the jobs released later since the later-released jobs have release time

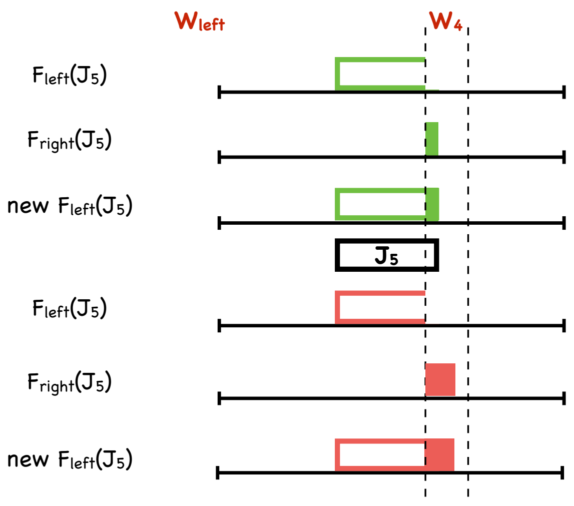


Figure 4.12: Valid and invalid concatinations of $F_{\text{left}}(J_5)$ and $F_{\text{right}}(J_5)$.

later than the end time of the certain jobs. Figure 4.13 shows an example of certain and uncertain job configurations. The red blocks mean that the configurations are uncertain, that is, the end times of the red jobs are not determined yet. In Stage i, consider any configuration, the jobs in $\bigcap_{k>i} C_k$ will not have execution intervals overlapping with the certain jobs.

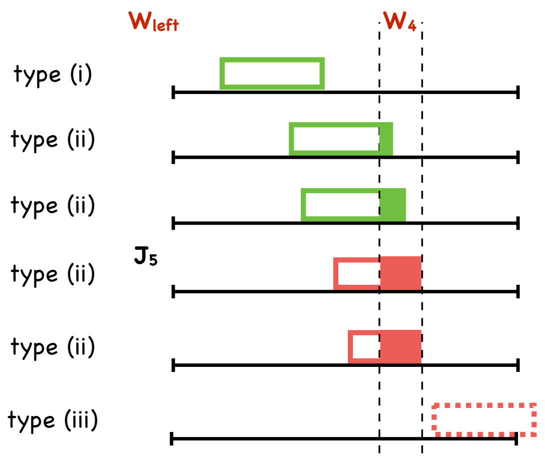
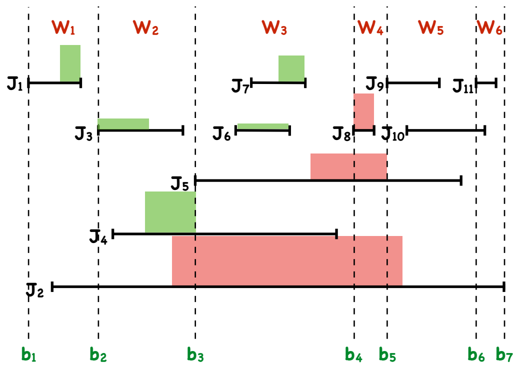


Figure 4.13: Concatinating $F_{\text{left}}(J_5)$ and $F_4(J_5)$.

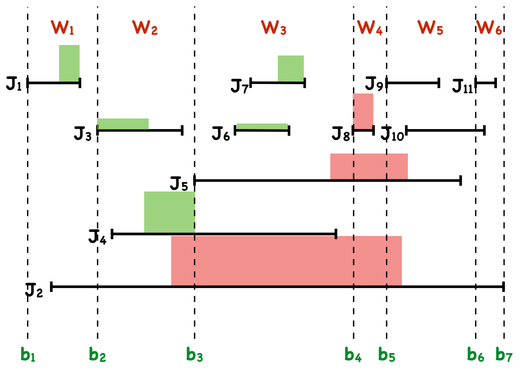
Two configurations $F_i(C)$ and $F'_i(C)$ are *identical* if (i) $F_i(J)$ is uncertain if and only if $F'_i(J)$ is uncertain for all job $J \in C$; and (ii) the start time of $F_i(J)$ is equal to the start time of $F'_i(J)$ for all uncertain configuration $F_i(J)$ and $J \in C$. That is, we only consider the differences among the start times of those jobs with uncertain configurations when we distinguish two configurations of a set of jobs.

Figure 4.14 shows six configurations of jobs in $\bigcup_{i=1}^{4} C_i$. The configurations in 4.14a and 4.14b are not identical since the job J_5 is uncertain in 4.14b but not in 4.14a. The configurations in 4.14c and 4.14e are identical since the configuration of uncertain jobs

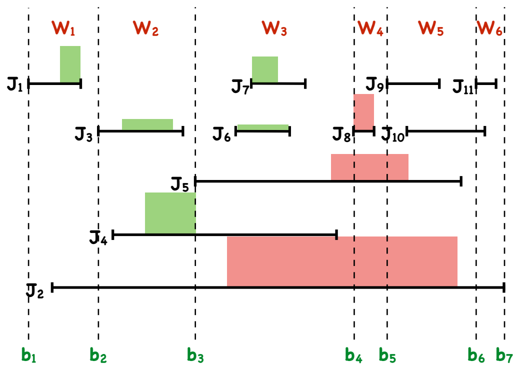
(J_2 and J_5) are the same (that is, the jobs different configurations have same start time). Note that the jobs J_3, J_4, and J_7 are not uncertain jobs. Although these jobs have different execution times in different configurations, the two configurations are still considered as identical. The configurations in 4.14d and 4.14f are identical since there is no uncertain jobs. The configurations in 4.14b and 4.14c are not identical since the uncertain job J_2 has different start time each of the configurations.



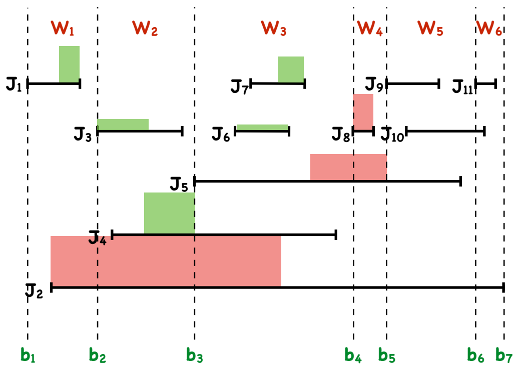
(a)



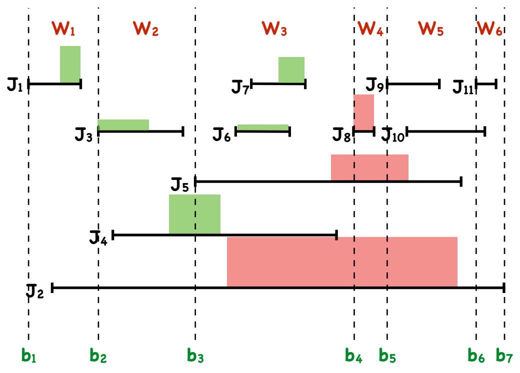
(b)



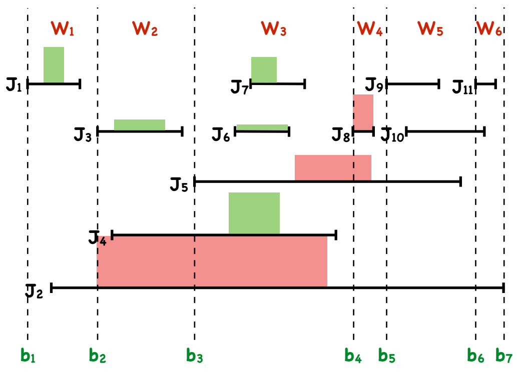
(c)



(d)



(e)



(f)

Figure 4.14: Illustrations of configurations.

The linear property of the consecutive clique arrangement of interval graphs gives a direction to design a dynamic programming algorithm, which breaks down a problem into overlapped subproblems until the subproblems are simple enough to be solved. Given a configuration a configuration $F_i(C_i)$, considered as fixed, we break down the problem to the subproblem that find the "best" schedule of the jobs $J \in \bigcup_{k=1}^{i-1} C_k \setminus C_i$ such that the cost (together with $F_i(C_i)$) is the least. This schedule thus obtained would be adopted later when we consider the best schedule of the subsequent cliques.

Figure 4.15 shows four different configurations $F_4(C_4)$ and the corresponding best schedules. The red blocks are jobs in C_4, and their execution intervals form a configuration in $F_4(C_4)$. The green blocks represent the jobs with feasible intervals complete before W_4, that is, the jobs are in $\bigcup_{k=1}^{3} C_k \setminus C_4$. Since the green jobs' feasible intervals end before W_4, given a configuration $F_4(C_4)$ (that is, the red jobs), we can schedule the green jobs such that the cost within $[b_1, b_{i+1})$ is minimized. Note that the configurations of the red jobs are given and the red jobs are considered as fixed (that is, not shiftable) while the green jobs can be shifted. The best schedule of the green jobs means a schedule of green jobs such that the cost (of the green jobs together with those fixed red jobs) is minimized. Also note that different configurations lead to different schedule of the green jobs and different cost. Hence not all these schedules are the optimal with respect to all the jobs seen so far. For example, the top-right subgraph in Figure 4.15 has lower cost than the other three, while all four schedules are best for the given configurations.

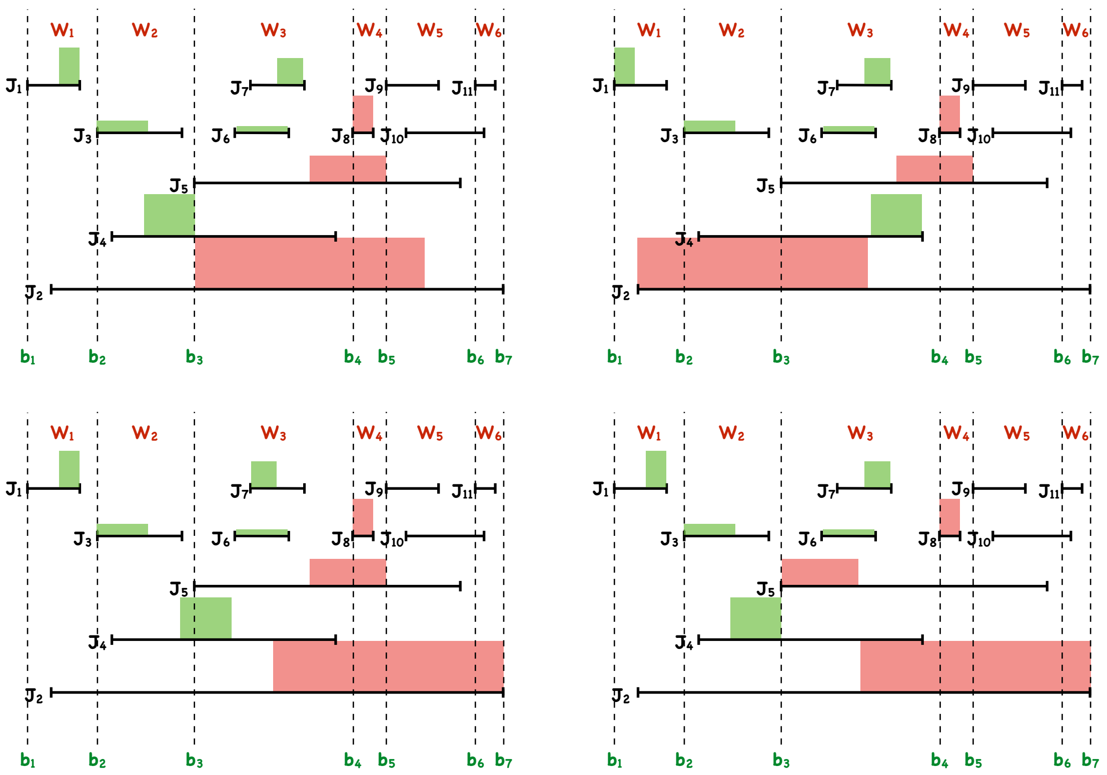


Figure 4.15: Different $F_4(C_4)$ and the corresponding optimal schedules.

4.3.3 An algorithm with three parameters

Algorithm $\mathcal{E}$. The algorithm consists of three components: ListConfigurations, ConcatenateTables and FilterTable. In the algorithm, we first transform the input job set $\mathcal{J}$ to an interval graph, and obtain the maximal cliques C_i for $1 \leq i \leq k$ and the corresponding windows W_i. We start with T_{left} containing the only configuration, which initially sets $st_0(J) = b_1$ and $et_0(J) = b_1+1$ for all jobs $J \in \mathcal{J}$. That is, the configuration treats all the jobs to be not yet executed. Then we visit the windows from the left to the right.

ListConfigurations: For window W_i and jobs in C_i, we construct T_{right} storing all configurations of $J \in C_i$. We enumerate all $st_i(J) \in \{b_i, b_i+1, \cdots, b_{i+1}-1\} \cup \{b_i-1, b_{i+1}\}$ and $et_i(J) \in \{b_i + 1, b_i + 2, \cdots, b_{i+1}\} \cup \{b_i, b_{i+1} + 1\}$ for each job $J \in C_i$, list all the combinations of all the jobs J with all of its start times and end times, and store the results in T_{right} in the way that one row is for one configuration $F_i(C_i)$. In another words, T_{right} stores all the combinations of possible execution segments of J in W_i for all $J \in C_i$. Recall that the configurations where jobs with release time later than W_i are considered to execute after W_i and the jobs with deadline earlier than W_i are considered to execute before W_i. For each configuration $F_i(C_i)$, we further store its cost contribution in W_i, $\text{cost}(F_i(C_i))$. That is, we only count the cost caused by the jobs' execution segments within W_i. We also check each of the configurations and delete those invalid ones.

ConcatenateTables: We then concatenate compatible configurations in T_{left} and T_{right}. The resulting table is the new T_{left}. More specifically, for each configuration $F_{\text{left}}(C)$ in T_{left} and each configuration $F_{\text{right}}(C)$ in T_{right}, we concatenate $F_{\text{left}}(C)$ and $F_{\text{right}}(C)$ if they are compatible, and store the result to a new row in T_{left}. Note that in the ListConfigurations step, the configurations in T_{left} and T_{right} are all valid. However, the concatenation of two valid configurations might not be valid (see Figure 4.12), hence we also check the validity of each of the configurations in the new T_{left} and delete those invalid ones.

FilterTable: After concatenation, we filter non-optimal configurations. We classify all the configurations in (the new) T_{left} into groups such that the configurations in a group are identical and no two configurations from different groups are identical. For each group, we only leave the configuration with the lowest cost (break tie arbitrarily) and remove the others in the group (see Figure 4.15). In the current T_{left}, no two configurations are identical.

After processing all windows, the only configuration in the final T_{left} is returned as the solution. Algorithm 5 is the pseudocode of this algorithm.

Lemma 4.16. *Algorithm $\mathcal{E}$ outputs an optimal solution.*

Proof. In each stage, we list all possible configurations. A configuration is deleted only when it is invalid or it is identical to another configuration with lower cost. It is easy

Algorithm 5 The fixed parameter algorithm $\mathcal{E}$

```
Input: a set of job $\mathcal{J}$
Output: an optimal configuration of $\mathcal{J}$
$\{(W_i, C_i)\}_{i=1}^{k} \leftarrow$ the windows and their corresponding cliques of $\mathcal{J}$
$T_{\text{left}} \leftarrow$ a configuration that sets all jobs $J_j \in \mathcal{J}$ to be not yet executed
for $i$ from 1 to $k$ do
    $T_{\text{right}} \leftarrow$ ListConfigurations($W_i$, $C_i$)
    $T_{\text{left}} \leftarrow$ ConcatenateTables($T_{\text{left}}$, $T_{\text{right}}$)
    $T_{\text{left}} \leftarrow$ FilterTable($T_{\text{left}}$)
return any configuration in $T_{\text{left}}$
```

to see an invalid configuration cannot be optimal since it will not be a feasible schedule. So we focus on the other case. Given two identical configurations $F_{\text{left}}(C)$ and $F'_{\text{left}}(C)$ with $\text{cost}(F_{\text{left}}(C)) < \text{cost}(F'_{\text{left}}(C))$, we show that $F'_{\text{left}}(C)$ cannot be optimal. Suppose there is an optimal solution F^* containing $F'_{\text{left}}(C)$, which means each execution segment $F'_{\text{left}}(J)$ in $F'_{\text{left}}(C)$ is completely contained by the corresponding execution interval of J in F^*. Since $F_{\text{left}}(C)$ and $F'_{\text{left}}(C)$ are identical, the start times of J are the same in the two configurations for all uncertain jobs J. In W_{left}, this means the uncertain jobs do not make the costs of the two configurations to be different, and the jobs $\mathcal{J}_c$ that are not uncertain do. Note that $\mathcal{J}_c$ is consisted of the jobs with their end times being determined. This means we can replace the configurations of $\mathcal{J}_c$ in $F'_{\text{left}}(C)$ by the configurations of $\mathcal{J}_c$ in $F_{\text{left}}(C)$ and this action will not affect the procedures in the algorithm thereafter. However, this also results in a solution of lower cost and contradicts the assumption that F^* is optimal. Thus $F'_{\text{left}}(C)$ cannot be optimal. Therefore, none of the deleted configuration can be part of an optimal schedule. That is, no optimal schedule would be removed through out the whole process. □

Theorem 4.17. *Algorithm $\mathcal{E}$ computes an optimal solution in $O(k \cdot {w_{\max}}^{2m} \cdot (W_{\max} + 1)^{4m} \cdot n)$ time, where n is the number of jobs, $w_{\max}$ is the maximum width of jobs, m is the maximum size of cliques, $W_{\max}$ is the maximum length of windows, and k is the number of windows.*

Proof. We first compute the time complexities for the three components of the algorithm, and then compute the total time complexity. For ListConfigurations, there are at most $(W_{\max} + 1)^{2m}$ configurations in the outputted table T_{right}, since there are at most $W_{\max} + 1$ possible start times and end times respectively and at most m jobs that should be considered in the current window. For each configuration, it takes $O(n)$ time for construction and validity checking. It also takes $O(nW_{\max})$ to compute the cost of a configuration. So, the time complexity for ListConfigurations is

$$O((W_{\max} + 1)^{2m} \cdot nW_{\max}) = O((W_{\max} + 1)^{2m+1} \cdot n) \ .$$

Before computing the time complexities of the other components, we focus on the number of configurations of T_{left} at the end of each iteration in the algorithm. Since T_{left}

is filtered to have no identical configurations, the number of configurations can be upper bounded. This number depends on the number of different start times of uncertain jobs. There are at most m uncertain jobs, and for each such job, the number of start times is at most $w_{\max}$. Note that the end times of these jobs are all set to be later than the current window and will not affect the number of configurations. So the number of configurations of T_{left} at the end of each iteration is at most ${w_{\max}}^m$.

For ConcatenateTables, there are at most ${w_{\max}}^m \cdot (W_{\max}+1)^{2m}$ configurations in the outputted table T_{left}. This is because for each configuration in the input T_{left}, we need to compare it with all the configurations in T_{right} for compatibility checking. For each configuration, it takes $O(n)$ time for compatibility checking, concatenation and validity checking. Thus the time complexity for ConcatenateTables is $O({w_{\max}}^m \cdot (W_{\max}+1)^{2m} \cdot n)$.

For FilterTable, the number of configurations in the outputted table T_{left} is at most the number of configurations outputted by ConcatenateTables. Also, the number of groups is at most its number of configurations. Thus it takes

$$O([{w_{\max}}^m \cdot (W_{\max}+1)^{2m}]^2 \cdot n) = O({w_{\max}}^{2m} \cdot (W_{\max}+1)^{4m} \cdot n)$$

time for classification. And it takes $O({w_{\max}}^m \cdot (W_{\max}+1)^{2m})$ time for deletion. So the time complexity for FilterTable is $O({w_{\max}}^{2m} \cdot (W_{\max}+1)^{4m} \cdot n)$. Since there are k iterations, the total time complexity is $O(k \cdot {w_{\max}}^{2m} \cdot (W_{\max}+1)^{4m} \cdot n)$. $\square$

In the worst case, there are at most $O(n)$ windows. So algorithm $\mathcal{E}$ also runs in $f(w_{\max}, m, W_{\max}) \cdot O(n^2)$ time where $f(w_{\max}, m, W_{\max}) = {w_{\max}}^{2m} \cdot (W_{\max}+1)^{4m}$.

Corollary 4.18. GRID *problem is fixed parameter tractable with respect to the maximum width of jobs, the maximum number of overlapped feasible intervals, and the maximum length of windows.*

4.3.4 An algorithm with two parameters

This section describes how to drop out the parameter $W_{\max}$ in the previous algorithm by generalizing the definitions of windows and boundaries.

At the beginning of Algorithm $\mathcal{E}$, we transform a set of jobs to its corresponding interval graph and obtain a sequence of windows by the set of maximal cliques in the interval graph. We require in the algorithm that all the cliques should be maximal. However, the algorithm is still optimal and has parameterized bound of time complexity if we divide a maximal clique into multiple non-maximal cliques in a specific way. Given a maximal clique C_i and its corresponding window W_i, we divide W_i into a set of contiguous windows $W_{i_1}, W_{i_2}, \ldots$ such that $W_i = \cup_j W_{i_j}$. Note that the set of jobs C_{i_j} corresponding to W_{i_j} is a clique in the interval graph since C_i is a clique and $C_{i_j} \subseteq C_i$. In this way, the number of jobs in the window W_{i_j} is still at most m. Furthermore, since this window division does not affect the proof of lemma 4.16, the algorithm is still optimal. Thus we have the following observation.

Observation 3. Algorithm $\mathcal{E}$ outputs an optimal solution if it receives a set of contiguous windows containing all the jobs such that each window represents a clique (not necessarily maximal) in the interval graph of the input jobs. And we have the number of jobs in each window is at most the maximum number of overlapped feasible intervals.

To drop out the parameter $W_{\max}$ in the previous algorithm, we divide windows into smaller ones such that the number of configurations in a window can be bounded by $w_{\max}$ and m. In the new algorithm, we set the locations of boundaries at the release times and deadlines of all the jobs and construct the windows bases on these boundaries. In this way, there is no job being released or attaining its deadline in the middle of a window, and all the jobs in the window can be put anywhere in the window. Thus the number of used timeslots is at most $m \cdot w_{\max} + 2(w_{\max} - 1)$. This is because in the worst case, all jobs in a window are scheduled such that no job overlaps to another and these jobs consume at most $m \cdot w_{\max}$ timeslots. In addition, we need to consider the cases that a job's start time is earlier than the window or its deadline is later than the window. Both cases consume at most $w_{\max} - 1$ timeslots respectively. Note that this window division results in a set of windows that their sizes are smaller than their original counterparts, and thus observation 3 can be applied. Based on this new window division, we have the following algorithm.

Algorithm $\mathcal{E}^+$. This algorithm is similar to algorithm $\mathcal{E}$ except the definitions of boundaries and the component ListConfigurations. Given a set of jobs $\mathcal{J}$, the algorithm uses the set of boundaries $\{r(J_j) \mid J_j \in \mathcal{J}\} \cup \{d(J_j) \mid J_j \in \mathcal{J}\}$ to construct the windows and obtain the corresponding cliques. Let k denotes by the number of windows. There are k stages for the algorithm. At the i-th stage, the algorithm runs ListConfigurations, ConcatenateTables and FilterTable accordingly as algorithm 5 does. It finally outputs the only configuration in T_{left}. For the component ListConfigurations, we only consider to schedule jobs on the timeslots used instead of enumerating all possibilities of start times and end times. The algorithm tries all $m \cdot w_{\max}$ timeslots (the worst case described in the previous paragraph) as the start time of a job, and also the $2(w_{\max} - 1)$ schedules that a job is partially executed in the window. In addition, the component shall includes the cases that either a job is completely executed before the window, it is completely executed after the window, or it crosses the window.

Theorem 4.19. *Algorithm $\mathcal{E}^+$ computes an optimal solution in $f(w_{\max}, m) \cdot O(n^2)$ time, where n is the number of jobs, $w_{\max}$ is the maximum width of jobs, m is the maximum size of cliques, and $f(w_{\max}, m) = (4m \cdot {w_{\max}}^2)^{2m}$.*

Proof. As in the proof of theorem 4.17, we compute the running time of the three components and then the total time complexity. For the component ListConfigurations, there are at most $(m \cdot w_{\max} + 2(w_{\max} - 1) + 3)^m$ outputted configurations, since there are at most $m \cdot w_{\max} + 2(w_{\max} - 1) + 3$ schedules for a job (see the description in the previous paragraph) and at most m jobs in a window. It takes $O(n(m \cdot w_{\max} + 2(w_{\max} - 1))) \leq$

$O(n \cdot m \cdot w_{\max})$ time to compute the cost for each configuration. Thus the time complexity for ListConfigurations is at most

$$O((m \cdot w_{\max} + 2(w_{\max} - 1) + 3)^m \cdot (n \cdot m \cdot w_{\max})) \leq O((4m \cdot w_{\max})^{m+1} \cdot n) \ .$$

The time complexities of ConcatenateTables and FilterTable are similar to that in the proof of theorem 4.17 except the number of outputted configurations. For ConcatenateTables and FilterTable, both the number of outputted configurations are at most ${w_{\max}}^m \cdot (4m \cdot w_{\max})^m$. Thus their running time are at most $O({w_{\max}}^{2m} \cdot (4m \cdot w_{\max})^{2m} \cdot n)$. Since there are $k = O(n)$ iterations, the total time complexity of the algorithm is at most

$$O((4m \cdot {w_{\max}}^2)^{2m} \cdot n^2) = f(w_{\max}, m) \cdot O(n^2) \ . \qquad \square$$

Corollary 4.20. GRID *problem is fixed parameter tractable with respect to the maximum width of jobs, and the maximum number of overlapped feasible intervals.*

The time complexity of the exact algorithm $O((4m \cdot {w_{\max}}^2)^{2m} \cdot n^2)$ seems to be very big. However, in the real world, m is the number of devices and $w_{\max}$, which is the longest processing time, can be actually small. Hence the time complexity is considerably small if we consider a small scale environment (for example, within a house).

4.4 An $(36(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)(1 + \lceil \log \frac{h_{\max}}{h_{\min}} \rceil))^{\alpha}$-Approximation Algorithm for General Case

We have shown in Section 4.2 that for input jobs with unit height and unit width, the GRID problem can be solved in polynomial time. In this section we are going to propose an approximation algorithm for GRID problem with a more general input where each job has arbitrary size (that is, arbitrary width and arbitrary height) and contiguous feasible interval by using the polynomial time algorithm. The idea is to classify the jobs by their heights and weights such that in each class of jobs where jobs heights (and widths) are bounded within a factor of two. For each class, the jobs are treated as unit jobs and we run the polynomial time algorithm independently. We show that such a transformation only increases the cost modestly and establish the approximation ratio of our algorithm.

In Section 4.4.1, we propose an approximation algorithm for uniform input (that is, jobs with uniform widths and uniform heights) by using the polynomial time optimal algorithm for unit case. In Section 4.4.2, we use the approximate algorithm in Section 4.4.1 to schedule a general input where jobs have arbitrary widths and arbitrary heights.

4.4.1 Uniform Widths and Uniform Heights Jobs

In this section, we consider job set $\mathcal{J}$ where each job has uniform width and uniform height. The idea of handling uniform width and uniform height jobs is to treat them as if they were unit width and unit height, however, this would mean that jobs may have release times or deadlines at non-integral time. To remedy this, we define a procedure AlignFI to align the feasible intervals (precisely, release times and deadlines) to the new time unit of duration w. Let $\mathcal{J}$ be a uniform width job set. We classify the jobs in $\mathcal{J}$ by the length of their feasible intervals. A job J is said to be *tight* if $|I(J)| \leq 2w$; otherwise, it is *loose*. Intuitively, the tight jobs are less flexible. In fact, for each tight job, there must be at least one timeslot in its feasible interval which has non-zero load in any feasible schedule. The in-flexibility guarantees that any strategy for tight jobs will not be too bad comparing to the optimal schedule. Let $\mathcal{J}_{\mathrm{T}}$ and $\mathcal{J}_{\mathrm{L}}$ denote the disjoint subsets of tight and loose jobs of $\mathcal{J}$, respectively. We have different strategies for tight and loose jobs. As to be shown, an arbitrary feasible schedule of tight jobs is good enough comparing to the optimal schedule. On the other hand, we modify every loose job via Procedure AlignFI such that its release time and deadline are both multiple of w. Then, we can treat the loose jobs as unit input and run $\mathcal{A}_{\mathrm{FG}}$.

Procedure AlignFI**.** Given a loose job set $\mathcal{J}_{\mathrm{L}}$ in which $w(J) = w$ and $|I(J)| > 2w$ for all $J \in \mathcal{J}_{\mathrm{L}}$. We define the procedure AlignFI to transform each loose job $J \in \mathcal{J}_{\mathrm{L}}$ into a job J' with release time and deadline "aligned" as follows. We denote the resulting job set by $\mathcal{J}'$.

- $r(J') \leftarrow \min_{i\geq 0}\{i \cdot w \mid i \cdot w \geq r(J)\}$;
- $d(J') \leftarrow \max_{i\geq 0}\{i \cdot w \mid i \cdot w \leq d(J)\}$.

Figure 4.16 shows an illustration of three feasible intervals $I(J_j)$ and corresponding $I(J'_j)$ generated by AlignFI where $w = 4$.

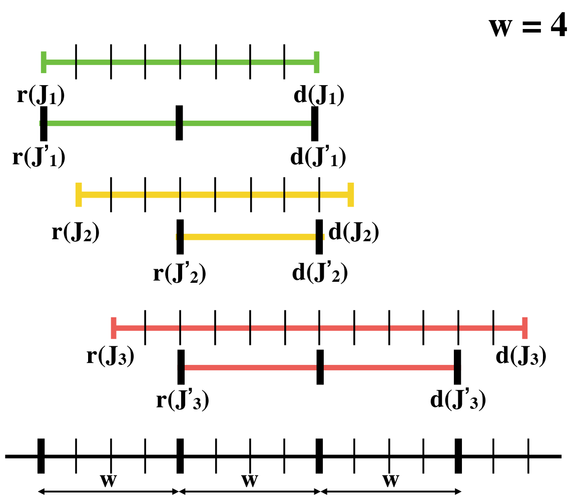


Figure 4.16: An illustration of Procedure AlignFI.

Observation 4 asserts that after performing AlignFI, the job set $\mathcal{J}'$ is a feasible input. Moreover, for every $J \in \mathcal{J}_{\mathrm{L}}$, we give a range where the feasible interval of the corresponding J' would be and a range of the length of $I(J')$.

Observation 4. For any job $J \in \mathcal{J}_{\mathrm{L}}$ and the corresponding J' generated by AlignFI,

(i) $d(J') > r(J')$ and $|I(J')| \geq w$;

(ii) $I(J') \subseteq I(J)$;

(iii) $\frac{1}{3}|I(J)| < |I(J')| \leq |I(J)|$.

Proof. (i) Consider any job $J \in \mathcal{J}$ and the corresponding J'. If $r(J)$ is a multiple of w, $d(J) \geq r(J) + 2w$ since J is a loose job. Hence $d(J') > d(J) - w \geq r(J) + 2w - w = r(J') + w$. On the other hand, if $r(J)$ is not a multiple of w, that is, $(i-1)w < r(J) < iw$ for some integer i, then $d(J) > (i+1)w$ since $|I(J)| \geq 2w$. Thus, $d(J') \geq (i+1)w$, $r(J') = iw$, and $|I(J')| \geq w$.

(ii) It is clear to see that $r(J) \leq r(J') < r(J) + w$. Also, $d(J) - r(J) \geq 2w$, thus $r(J') \in I(J)$. Similarly, $d(J') \in I(J)$. Hence $I(J') \subseteq I(J)$.

(iii) According to (i), $d(J') - r(J') \geq w$. Because $r(J') - r(J) < w$ and $d(J) - d(J') < w$, $|I(J)| - |I(J')| < 2w \leq 2|I(J')|$. □

Notice that after AlignFI, the release time and deadline of each loose job are aligned to timeslot $i_1 \cdot w$ and $i_2 \cdot w$ for some integers $i_1 < i_2$. Furthermore, after AlignFI all jobs are released at time which is a multiple of w. Also, by Observation 4 (ii), any feasible schedule of $\mathcal{J}'$ is feasible for $\mathcal{J}_{\mathrm{L}}$. Hence the job set $\mathcal{J}'$ can be treated as job set with unit size, where each unit has duration w instead of 1 and power request h instead of 1.

As a consequence of altering the feasible intervals, we introduce two additional procedures that convert associated schedules. Given a schedule S for a loose job set $\mathcal{J}_{\mathrm{L}}$, AlignSch converts it to a schedule S' for the corresponding job set $\mathcal{J}'$ where each job is treated as unit input. The other procedure FreeSch takes a schedule S' for a job set $\mathcal{J}'$ and converts it to a schedule S for $\mathcal{J}_{\mathrm{L}}$.

Since any feasible schedule of $\mathcal{J}'$ is feasible for $\mathcal{J}_{\mathrm{L}}$, the Transformation FreeSch is straightforward:

Transformation FreeSch**.** FreeSch transform S' into S.

- $st(S, J) \leftarrow st(S', J')$;
- $et(S, J) \leftarrow et(S', J')$.

The feasibility of S' can be proved easily by Observation 4 (iii). And the cost of S is the same as S'.

Lemma 4.21. *Using* FreeSch*, we have cost*$(S) =$ *cost*(S')*.*

Proof. It is easy to see that $\ell oad(S, t) = \ell oad(S', t)$ for all t. Hence $\mathrm{cost}(S) = \mathrm{cost}(S')$. □

The other direction, AlignSch, is trickier. Apparently, the start time of J in a feasible schedule can be at any time which is not necessary a multiple of w. Not to mention that $I(J)$ might greater than $I(J')$ and the execution interval of J could be partially outside the $I(J')$ (see Figure 4.17).

The idea of AlignSch is that, if the execution interval of J is not at a multiple of w time, we shift it to the right to the first multiple of w it can reach. It also fixes the case where the start time of J is outside the $I(J')$ (note that in this case $st(S, J) < r(J')$). It could be possible that shifting the executional interval of J to the right does not give a feasible schedule of J'. Consider the case that the start time of J is inside $I(J')$ but the whole execution interval of J is not completely inside $I(J')$. More formally, it happens when $d(J') - w < st(S, J) < d(J')$. In this case, if we shift the execution interval to the right to the first multiple of w, which is exactly $d(J')$, the resulting schedule is not feasible for J'.

Transformation AlignSch**.** AlignSch transforms S into S' by shifting the execution interval of every job $J \in \mathcal{J}_{\mathrm{L}}$.

- $st(S', J') \leftarrow \min\{d(J') - w, \min_{i \geq 0}\{i \cdot w \mid i \cdot w \geq st(S, J)\}\}$;
- $et(S', J') \leftarrow st(S', J') + w$.

Figure 4.17 shows an illustration of the Transformation AlignSch. Simply speaking, the Transformation AlignSch shifts the execution interval of J to the right to align at the first multiple of w on the right-hand side. If it would cause infeasibility (for example, job J_2 in Figure 4.17), the execution interval is shifted to the left instead and align at the first multiple of w on the left-hand side. Note that either shifting to the right or shifting to the left, the distance of shifting is no more than w.

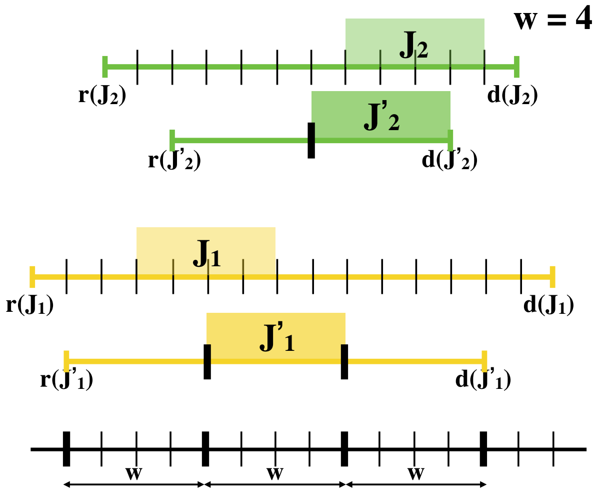


Figure 4.17: An illustration of Transformation AlignSch.

The following properties can be proved directly from the Transformation AlignSch:

Observation 5. Consider any schedule S for $\mathcal{J}_{\mathrm{L}}$ and the schedule S' for $\mathcal{J}'$ constructed by AlignSch. The following properties hold:

(i) $[st(S',J'), et(S',J')) \cap [st(S,J), et(S,J)) \neq \emptyset$; and

(ii) S' is a feasible schedule for $\mathcal{J}'$.

Proof. (i) In Transformation AlignSch, the execution interval of J can be shift to right or left. No matter which side it is shifted to, the distance of shifting is no more than w or contradiction occurs.

(ii) By the first step in AlignSch, $r(J') \leq st(S',J') \leq d(J') - w$. Hence the whole execution interval $[st(S',J'), et(S',J'))$ is inside $I(J')$ □

In the following lemma we show that after AlignSch, the load at any timeslot in the schedule S' can be captured by the loads of constant number of timeslots in S. The key idea is that the execution intervals in S which after AlignSch is passing t must not be too far from t.

Lemma 4.22. *Consider any schedule S for $\mathcal{J}_{\mathrm{L}}$ and the schedule S' for $\mathcal{J}'$ constructed by* AlignSch. *At any timeslot t, $\ell oad(S',t) \leq \ell oad(S,t) + \ell oad(S,t-(w-1)) + \ell oad(S,t+(w-1))$.*

Proof. Consider that for any timeslot t, the execution intervals of jobs, which are possible to be shifted such that the resulting execution intervals passing t, are inside an interval with length $4w-3$ centers around t. Formally, by Observation 5 (i), if $t \in [st(S',J'), et(S',J'))$, the execution interval of the corresponding J, $[st(S,J), et(S,J)) \subseteq [st(S',J')-(w-1), et(S',J')+(w-1)) \subseteq [t-2(w-1), t+2(w-1))$. In other words, if $st(S,J) < t-2(w-1)$ or $et(S,J) > t+2(w-1)$, the resulting execution interval after AlignSch will not cover t.

Consider all jobs with execution interval completely inside $[t-2(w-1), t+2(w-1))$, they can be partitioned into two types:

- $t-2(w-1) \leq st(S,J) < t-(w-1)$ or $t+(w-1) < et(S,J) \leq t+2(w-1)$
- $[st(S,J), et(S,J)) \subseteq [t-(w-1), t+(w-1))$

The first type of jobs have original execution intervals not covering t and after AlignSch, the new execution intervals might cover t. Since the jobs widths are w, the original execution intervals of these jobs must cover either timeslot $t-(w-1)$ or timeslot $t+(w-1)$. On the other hand, the second type of jobs have original execution intervals covering t. Hence, we can upper bound $\ell oad(S',t)$ by sampling the load of S at timeslots t, $t-(w-1)$, and $t+(w-1)$. That is, $\ell oad(S',t) \leq \ell oad(S,t) + \ell oad(S,t-(w-1)) + \ell oad(S,t+(w-1))$. □

Figure 4.18 shows an illustration for Lemma 4.22. The gray solid blocks represent execution intervals of jobs J in $\mathcal{J}_{\mathrm{L}}$ and the corresponding blocks without filling color show the union of possible execution intervals of the corresponding J'. It is easy to see that jobs with execution intervals too far from t (jobs J_2 and J_6) cannot have execution intervals of their corresponding J' over timeslot t. On the other hand, consider those jobs J' with execution interval over t, their corresponding J have execution interval in S over timeslot $t-(w-1)$ (job J_1), $t+(w-1)$ (jobs J_4 and J_5), or t itself (jobs J_3 and J_4). Note that the execution interval of J_4 passing both t and $t+(w+1)$. Hence what we calculate is an upper bound of $\ell oad(S', t)$.

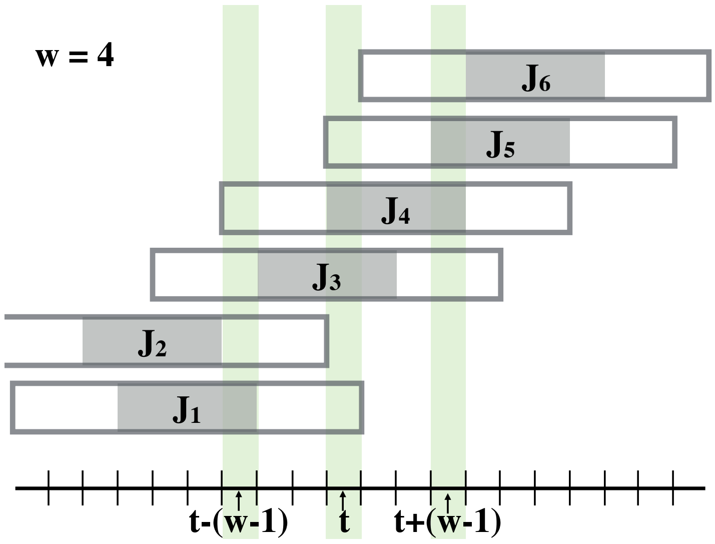


Figure 4.18: An illustration for Lemma 4.22.

Since we can upper bound the loads of timeslots in S' by the loads of timeslots in S, we can bound the cost of S':

Corollary 4.23. *Using* AlignSch *to generate S' given S, we have cost$(S') \leq 3^\alpha \cdot$ cost(S).*

Proof. By Lemma 4.22, $\text{cost}(S') = \sum_t \ell oad(S',t)^\alpha \leq \sum_t (3 \cdot \ell oad(S,t))^\alpha = 3^\alpha \cdot \text{cost}(S)$. □

By bounding the cost affected by performing AlignSch, we can bound the optimal cost of S' by the optimal cost of the corresponding S:

Lemma 4.24. *Given a loose job set $\mathcal{J}_L$ and the corresponding $\mathcal{J}'$ generated by* AlignFI*, cost$(\mathcal{O}(\mathcal{J}')) \leq 3^\alpha \cdot$ cost$(\mathcal{O}(\mathcal{J}_L))$.*

Proof. Given $\mathcal{O}(\mathcal{J}_L)$, there exists a schedule $S(\mathcal{J}')$ generated by AlignSch for the corresponding $\mathcal{J}'$. By Corollary 4.23, $\text{cost}(S(\mathcal{J}')) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_L))$. Hence, $\text{cost}(\mathcal{O}(\mathcal{J}')) \leq \text{cost}(S(\mathcal{J}')) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_L))$. □

Approximation algorithm $\mathcal{A}_u$ for uniform size input (Algorithm 6). The algorithm takes a job set $\mathcal{J}$ with uniform width w and uniform height h as input and schedules the jobs in $\mathcal{J}$ as follows. Let $\mathcal{J}_T$ be the set of tight jobs in $\mathcal{J}$ and $\mathcal{J}_L$ be the set of loose jobs in $\mathcal{J}$. Then the resulting schedule is transformed into a scheduled for jobs in $\mathcal{J}_T$ by FreeSch.

1. For any tight job $J \in \mathcal{J}_\mathrm{T}$, schedule J to start at $r(J)$.
2. Loose jobs in $\mathcal{J}_\mathrm{L}$ are converted into $\mathcal{J}'$ by AlignFI. For $\mathcal{J}'$, we run Algorithm $\mathcal{A}_\mathrm{FG}$, which is elaborated in Section 4.2.1.

The running time of Algorithm $\mathcal{A}_u$ is exactly the running time of $\mathcal{A}_\mathrm{FG}$ since for each job it only takes constant extra time to transform it to a suitable input for $\mathcal{A}_\mathrm{FG}$.

Algorithm 6 The approximation algorithm $\mathcal{A}_u$

```
Input: a set of job $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ where $w(J_j) = w$ for all $J_j$.
$\mathcal{A}_\mathrm{FG}$ is the offline algorithm for unit case
$\mathcal{J}' \leftarrow \emptyset$
for each job $J_j$ do
    if $|I(J_j)| < 2w$ then
        $st(\mathcal{A}_u, J_j) \leftarrow r(J_j)$
    else
        $w(J'_j) \leftarrow w(J_j)$
        $h(J'_j) \leftarrow h(J_j)$
        $r(J'_j) \leftarrow w\lceil \frac{r(J_j)}{w} \rceil$
        $d(J'_j) \leftarrow w\lfloor \frac{d(J_j)}{w} \rfloor$
        Add $J'_j$ into $\mathcal{J}'$
for each job $J'_j \in \mathcal{J}'$ do
    $st(\mathcal{A}_u, J_j) \leftarrow st(\mathcal{A}_\mathrm{FG}(\mathcal{J}'), J'_j)$
return the schedule $\mathcal{A}_u$
```

Analysis of Algorithm $\mathcal{A}_u$. We analyze the tight jobs and loose jobs separately. We first give an observation about the optimal cost of a job set and the one of its subset.

Observation 6. For any to job sets $\mathcal{J}_x \subseteq \mathcal{J}_y$, $\mathrm{cost}(\mathcal{O}(\mathcal{J}_x)) \leq \mathrm{cost}(\mathcal{O}(\mathcal{J}_y))$.

Proof. Assume on the contrary that $\mathrm{cost}(\mathcal{O}(\mathcal{J}_y)) < \mathrm{cost}(\mathcal{O}(\mathcal{J}_x))$, we can generate a schedule $S(\mathcal{J}_x)$ by removing jobs from $\mathcal{O}(\mathcal{J}_y)$ which are not in $\mathcal{J}_x$. It follows that $\mathrm{cost}(S(\mathcal{J}_x)) \leq \mathrm{cost}(\mathcal{O}(\mathcal{J}_y)) < \mathrm{cost}(\mathcal{O}(\mathcal{J}_x))$, which is contradicting to the fact that $\mathcal{O}(\mathcal{J}_x)$ is optimal for $\mathcal{J}_x$. □

In the following we prove that since the tight jobs are "inflexible", any feasible schedule is good enough comparing to the optimal.

Lemma 4.25. *Given an arbitrary feasible schedule $S(\mathcal{J}_T)$ where $\mathcal{J}_T$ is a set of tight jobs, cost$(S(\mathcal{J}_T)) \leq 3^\alpha \cdot$ cost$(\mathcal{O}(\mathcal{J}_T))$.*

Proof. We first *extend* jobs $J \in \mathcal{J}_\mathrm{T}$ to J^* as the following: $r(J^*) = r(J)$, $d(J^*) = d(J)$, $w(J^*) = d(J) - r(J)$, and $h(J^*) = h(J)$. That is, every job has its width as the length of its feasible interval. We denote the resulting job set by $\mathcal{J}^*$. It is easy to see that because of each job in $\mathcal{J}^*$ are not shiftable, there is only one feasible schedule for $\mathcal{J}^*$ and it is optimal.

Similar to the claim in the proof of Observation 6, $\text{cost}(S(\mathcal{J}_\text{T})) \leq \text{cost}(\mathcal{O}(\mathcal{J}^*))$ since we can get $S(\mathcal{J}_\text{T})$ by shaving $\mathcal{O}(\mathcal{J}^*)$.

Because for each job in $\mathcal{J}_\text{T}$, the length of its feasible interval is at most $2w - 1$, we can bound the load at any time t of $\mathcal{O}(\mathcal{J}^*)$ by the loads of constant number of timeslots in $S(\mathcal{J}_\text{T})$. Assume that at timeslot t an extended job J^* is executed. That is, $t \in [r(J^*), d(J^*))$ since J^* is not shiftable. Consider the job J corresponding to J^*, the execution interval of J in any feasible schedule must contains either timeslot $t-(w-1)$, $t+(w-1)$, or t (see Figure 4.19). Hence we can upper bound the load at any time t in $\mathcal{O}(\mathcal{J}^*)$: $\ell oad(\mathcal{O}(\mathcal{J}^*), t) \leq \ell oad(\mathcal{O}(\mathcal{J}_\text{T}), t-(w-1)) + \ell oad(\mathcal{O}(\mathcal{J}_\text{T}), t+(w-1)) + \ell oad(\mathcal{O}(\mathcal{J}_\text{T}), t)$. Similar to the proof of Corollary 4.23, $\text{cost}(\mathcal{O}(\mathcal{J}^*)) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{T}))$. To sum up, $\text{cost}(S(\mathcal{J}_\text{T})) \leq \text{cost}(\mathcal{O}(\mathcal{J}^*)) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{T}))$. □

Figure 4.19 shows an illustration for proof of Lemma 4.25 where $w = 5$. The gray solid blocks represent the execution interval of each job J_i; the corresponding blocks with bold edge and without filling color represent the execution interval of corresponding J_j^*, which is exactly the feasible interval of J_j. Since the length of any feasible interval is at most $2w - 1$ and each job has width w, any job J_j^* with execution interval passing t must have execution interval of corresponding J passing at least one of the timeslots t (jobs J_3, J_5, and J_6), $t-(w-1)$ (jobs J_1 and J_2), or $t+(w-1)$ (jobs J_3 and J_4).

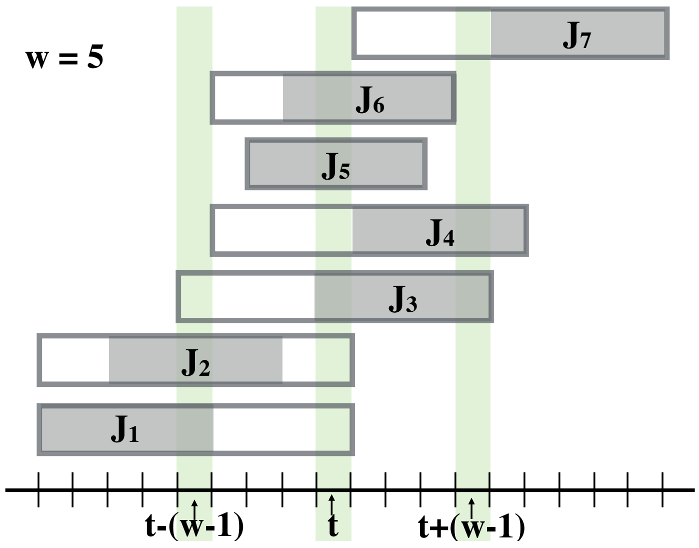


Figure 4.19: An illustration for Lemma 4.25.

In $\mathcal{A}_u$, the start time of each tight job is exactly its release time, hence it is feasible and we have the following corollary:

Corollary 4.26. *cost*$(\mathcal{A}_u(\mathcal{J}_T)) \leq 3^\alpha \cdot$ *cost*$(\mathcal{O}(\mathcal{J}_T))$.

For loose jobs, we convert them such that their release times and deadlines are at multiple of w time. Then we run $\mathcal{A}_\text{FG}$ on the resulting jobs. In the following we prove that these procedure guarantee a good ratio.

Lemma 4.27. *cost*$(\mathcal{A}_u(\mathcal{J}_L)) \leq 3^\alpha \cdot$ *cost*$(\mathcal{O}(\mathcal{J}_L))$.

Proof. By Lemma 4.21, $\text{cost}(\mathcal{A}_u(\mathcal{J}_\text{L})) = \text{cost}(\mathcal{A}_\text{FG}(\mathcal{J}'))$. As mentioned in Section 4.2, $\text{cost}(\mathcal{A}_\text{FG}(\mathcal{J}')) = \text{cost}(\mathcal{O}(\mathcal{J}'))$. By Lemma 4.24, $\text{cost}(\mathcal{O}(\mathcal{J}')) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{L}))$ given $\mathcal{J}_\text{L}$ and the corresponding $\mathcal{J}'$. To sum up, $\text{cost}(\mathcal{A}_u(\mathcal{J}_\text{L})) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{L}))$ □

By Corollary 4.26 and Lemma 4.27, we can bound the over all approximation ratio of $\mathcal{A}_u$ and show that the approximation ratio of $\mathcal{A}_u$ is constant.

Theorem 4.28. *cost*$(\mathcal{A}_u(\mathcal{J})) \leq 6^\alpha \cdot \mathcal{O}(\mathcal{J})$.

Proof. In the schedule computed by $\mathcal{A}_u$, the load at t is from both tight jobs and loose jobs. We can upper bound the cost of $\mathcal{A}_u$ over all jobs by summation of $\text{cost}(\mathcal{A}_u(\mathcal{J}_\text{T}))$, the cost of $\mathcal{A}_u$ over tight jobs, and $\text{cost}(\mathcal{A}_u(\mathcal{J}_\text{L}))$, the cost of $\mathcal{A}_u$ over loose jobs, with a constant blow-up. More formally, $\text{cost}(\mathcal{A}_u(\mathcal{J})) = \sum_t (\ell oad(\mathcal{A}_u(\mathcal{J}_\text{T}), t) + \ell oad(\mathcal{A}_u(\mathcal{J}_\text{L}), t))^\alpha \leq 2^{\alpha-1} \cdot (\sum_t \ell oad(\mathcal{A}_u(\mathcal{J}_\text{T}), t)^\alpha + \sum_t \ell oad(\mathcal{A}_u(\mathcal{J}_\text{L}), t)^\alpha) = 2^{\alpha-1} \cdot (\text{cost}(\mathcal{A}_u(\mathcal{J}_\text{T})) + \text{cost}(\mathcal{A}_u(\mathcal{J}_\text{L})))$. By Corollary 4.26 and Lemma 4.27, $\text{cost}(\mathcal{A}_u(\mathcal{J})) \leq 2^{\alpha-1} \cdot (3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{T})) + 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_\text{L})))$. By Observation 6, $\text{cost}(\mathcal{A}_u(\mathcal{J})) \leq 2^{\alpha-1} \cdot (3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J})) + 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))) = 6^\alpha \cdot \mathcal{O}(\mathcal{J})$. □

4.4.2 General Input

In this section we present an algorithm $\mathcal{A}_g$ for jobs with arbitrary width and arbitrary height. We first transform job set $\mathcal{J}$ to a "nice" job set $\mathcal{J}^*$ (to be defined) and run the algorithm $\mathcal{A}_u$ for uniform width and uniform height jobs introduced in Section 4.4.1. We show that such a transformation only increases the cost modestly. Furthermore, we show that for any nice job set $\mathcal{J}^*$, we can bound $\text{cost}(\mathcal{A}_g(\mathcal{J}))$ by $\text{cost}(\mathcal{O}(\mathcal{J}^*))$ and in turn by $\text{cost}(\mathcal{O}(\mathcal{J}))$. Then we can establish the competitive ratio of $\mathcal{A}_g$.

Classes of jobs. Consider a job J, if $2^{p-1} < w(J) \leq 2^p$ and $2^{q-1} < h(J) \leq 2^q$, it is in class $C_{p,q}$. Since jobs heights and widths are both at least 1, $p, q \geq 0$. In class $C_{0,q}$, jobs have unit widths and in class $C_{p,0}$, jobs have unit heights. Let $w_{\max}$, $w_{\min}$, $h_{\max}$, and $h_{\min}$ denote the maximum width, minimum width, maximum height, and minimum height over all jobs, there are in total $(\lceil \log w_{\max} \rceil - \lceil \log w_{\min} \rceil + 1)(\lceil \log h_{\max} \rceil - \lceil \log h_{\min} \rceil + 1)$ classes. To simplify the notation, note that if the largest width (or height) and smallest width (or height) over all jobs are not the same, $\lceil \log w_{\max} \rceil - \lceil \log w_{\min} \rceil + 1 \leq 3\lceil \log \frac{w_{\max}}{w_{\min}} \rceil$. Otherwise, there is only one class regarding to jobs widths (or heights). Hence, there are at most $(1 + 3\lceil \log \frac{w_{\max}}{w_{\min}} \rceil)(1 + 3\lceil \log \frac{h_{\max}}{h_{\max}} \rceil)$ classes.

Nice job set and transformations. In this section we define that a job J is a *nice job* if $w(J) = 2^p$ and $h(J) = 2^q$ where p and q are both non-negative integers. A job set $\mathcal{J}^*$ is a *nice job set* if all jobs in $\mathcal{J}^*$ are nice jobs. In other words, if a nice job has width 2^p and height 2^q, it is in class $C_{p,q}$

We convert arbitrary job set $\mathcal{J}$ into a nice job set by the Procedure CONVERT.

Procedure CONVERT. Given an jobs set $\mathcal{J}$, we define a procedure CONVERT to transform each job $J \in \mathcal{J}$ into a nice job J^* by rounding up the widths and heights of the jobs to the next power of 2. The resulting job set is denoted by $\mathcal{J}^*$. Suppose that job J is in class $C_{p,q}$, we modify its width, height and deadline. The procedure is defined as follows.

- $w(J^*) \leftarrow 2^p$
- $h(J^*) \leftarrow 2^q$
- $r(J^*) \leftarrow r(J)$
- $d(J^*) \leftarrow r(J^*) + \max\{d(J) - r(J), w(J^*)\}$

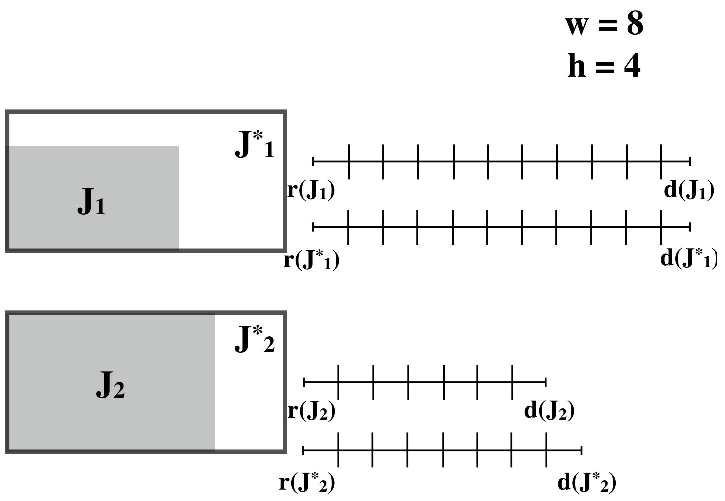


Figure 4.20: An illustration for Procedure CONVERT.

Figure 4.20 shows an illustration for Procedure CONVERT where $p = 3$ and $q = 2$. After CONVERT, the jobs in $C_{p,q}$ form a nice job set $\mathcal{J}^*_{p,q}$. It means that after CONVERT, for each $C_{p,q}$, the jobs have same widths and same heights. The modification of width and height is to transform J into a nice job by rounding up its width and height. After rounding up to the next power of 2, the width may greater than the length of the original feasible and it makes the job itself infeasible (for example, the job J_2 in Figure 4.20). To make sure the resulting job is a part of feasible input, we also modify the deadline if needed. The following observation is about the properties of J and the corresponding J^* generated by CONVERT. The following observation can be proved directly from the procedure CONVERT.

Observation 7. For any job J and its corresponding J^* after performing CONVERT,

(i) $|I(J^*)| \geq w(J^*)$, that is, $\mathcal{J}^*$ is a feasible input set;

(ii) $I(J) \neq I(J^*)$ if and only if $|I(J)| < w(J^*)$; and

(iii) $I(J) \subseteq I(J^*)$.

We then define two procedures that transform schedules related to nice job sets. RelaxSch takes a schedule S for a job set $\mathcal{J}$ and coverts it to a schedule S^* for the corresponding nice job set $\mathcal{J}^*$. On the other hand, ShrinkSch takes a schedule S^* for a nice job set $\mathcal{J}^*$ and converts it to a schedule S for $\mathcal{J}$.

Transformation RelaxSch**.** RelaxSch transfors feasible S into S^* by moving the start and end time of every job J.

- $st(S^*, J^*) \leftarrow \min\{d(J^*) - w(J^*), st(S, J)\}$
- $et(S^*, J^*) \leftarrow st(S^*, J^*) + w(J^*)$

The following observation asserts that the resulting schedule S^* is feasible for $\mathcal{J}^*$.

Observation 8. Consider any schedule S for $\mathcal{J}$ and the schedule S^* constructed by RelaxSch for the corresponding $\mathcal{J}^*$. We have
(i) $[st(S, J), et(S, J)) \subseteq [st(S^*, J^*), et(S^*, J^*))$; and
(ii) $[st(S^*, J^*), et(S^*, J^*)) \subseteq [r(J^*), d(J^*))$ for all $J^* \in \mathcal{J}^*$; in other words, S^* is feasible for $\mathcal{J}^*$.

Proof. (i) By the first step, $st(S^*, J^*) \leq st(S, J)$. Assume that $st(S^*, J^*) = st(S, J)$, $et(S^*, J^*) \geq et(S, J)$ since $et(S^*, J^*) = st(S^*, J^*) + w(J^*) \geq st(S, J) + w(J) = et(S, J)$. On the other hand, Assume that $st(S^*, J^*) = d(J^*) - w(J^*)$. In this case the execution $et(S^*, J^*) = d(J^*)$. By the Procedure Convert, $d(J^*) \geq d(J) \geq et(S, J)$ since S is feasible.

(ii) We prove that $st(S^*, J^*) \geq r(J^*)$ and $et(S^*, J^*) \leq d(J^*)$. Assume that $st(S^*, J^*) = d(J^*) - w(J^*)$. By the Procedure Convert, $d(J^*) - r(J^*) \geq w(J^*)$. Hence $st(S^*, J^*) \geq r(J^*)$. In this case, $et(S^*, J^*) = st(S^*, J^*) + w(J^*) = d(J^*)$.

On the other hand, assume that $st(S^*, J^*) = st(S, J)$. In this case, $st(S, J) \leq d(J^*) - w(J^*)$. Hence, $et(S^*, J^*) = st(S^*, J^*) + w(J^*) = st(S, J) + w(J^*) \leq d(J^*)$. Also, it is easy to see that $st(S^*, J^*) \geq r(J^*)$ since $r(J^*) = r(J)$ and $\mathcal{J}$ is a feasible input. □

In the following we want to analyze the cost of S^* and show that it can be bounded by the cost of S.

According to the Procedure Convert and the Transformation RelaxSch, we can image that the S^* is the result of S by expanding each job such both vertically and horizontally. That is, the jobs widths and heights are both expanded by at most twice and hence affect the cost. For the vertical expanding, it is easy to analyze since the heights are not increased by more than two. However, the analysis is more complicated for the horizontally expanding. By Transformation RelaxSch, the horizontally expanding of a job execution interval can be extending to the left or extending to the right. Once the execution interval of a job is extended, the job's height affects the loads of each timeslots which is in the new execution interval but not in the original one. For example, assume that a job J is in class $C_{p,q}$, there might be at most 2^{p-1} timeslots have $w(J)$-higher load in S^* than in S. In other words, consider timeslot t, the jobs executed at timeslots $t-2^{p-1}+1, t-2^{p-1}+2, t-2^{p-1}+3, \cdot, t, t+1, t+2, \cdot, t+2^{p-1}-1$ in S could possibly extend to t and affect its load in S^*. It seems the cost of S^* might be $O(2^{p-1})$ times to the cost of S. However, in the following lemmas we show that by smartly sampling constant number of timeslots, the cost of S^* is bounded by a constant times the cost of S.

Lemma 4.29. *Let $\mathcal{J}_{p,q}$ and $\mathcal{J}^*_{p,q}$ denote the set of jobs in class $C_{p,q}$ and the nice jobs generated by* Convert. *Also we denote $S_{p,q}$ and $S^*_{p,q}$ as the schedule of $\mathcal{J}_{p,q}$ and the schedule generated by* RelaxSch. *For any time t, $\ell oad(S^*_{p,q}, t) \leq 2(\ell oad(S_{p,q}, t) + \ell oad(S_{p,q}, t-(2^{p-1}-1)) + \ell oad(S_{p,q}, t+(2^{p-1}-1)))$.*

Proof. We first prove that the jobs which can affect the load $\ell oad(S^*_{p,q}, t)$ are assigned in a range centering around t. Next, we claim that inside this range of timeslots, there are several special timeslots of which the load in $S_{p,q}$ are essential for estimating the cost of $\ell oad(S^*_{p,q}, t)$.

Consider any job $J \in C_{p,q}$ and its execution interval in $S_{p,q}$, $[st(S_{p,q}, J), et(S_{p,q}, J))$. According to RelaxSch, $[st(S_{p,q}, J), et(S_{p,q}, J)) \subseteq [st(S^*_{p,q}, J^*), et(S_{p,q}, J))$. Hence, after performing RelaxSch, the execution interval of the corresponding J^* is within $[st(S_{p,q}, J) - (2^{p-1}-1), et(S_{p,q}, J) + (2^{p-1}-1))$ since the amount of horizontal expanding is at most $2^{p-1}-1$. In other words, consider timeslot t, the jobs J^* with execution interval in $S^*_{p,q}$ which is over t, those nice jobs must have their corresponding J with execution interval in $S_{p,q}$ completely inside $[t-(2^p-1), t+(2^p-1))$. Since $2^{p-1} < w(J) \leq 2^p$, the execution interval of any J must over at least one of the following three timeslots: t, $t-(2^{p-1}-1)$, or $t+(2^{p-1}-1)$. Hence the $\ell oad(S^*_{p,q}, t)$ can be upper bounded by the three loads in $S_{p,q}$: $\ell oad(S_{p,q}, t)$, $\ell oad(S_{p,q}, t-(2^{p-1}-1))$, and $\ell oad(S_{p,q}, t+(2^{p-1}-1))$. Also, after rounding up the height, for a job J and its corresponding J^*, $h(J^*) \leq 2h(J)$. Hence $\ell oad(S^*_{p,q}, t) \leq 2\cdot(\ell oad(S_{p,q}, t) + \ell oad(S_{p,q}, t-(2^{p-1}-1)) + \ell oad(S_{p,q}, t+(2^{p-1}-1)))$ □

Figure 4.21 shows an illustration for proof of Lemma 4.25 where $p = 3$. The gray solid blocks represent the execution interval of each job J_j; the corresponding hollowed blocks represent the possible execution interval with load after round-up of corresponding J^*_j.

Since we can bound the load of $S^*_{p,q}$ at any time t by the loads of $S_{p,q}$, we can bound the cost of $S^*_{p,q}$ by the cost of $S_{p,q}$:

Lemma 4.30. *Using* RelaxSch, *we have $cost(S^*_{p,q}) \leq 6^\alpha \cdot cost(S_{p,q})$.*

Proof. By definition, $\text{cost}(S^*_{p,q}) = \sum_t \ell oad(S^*_{p,q}, t)^\alpha$. By Lemma 4.29, $\sum_t \ell oad(S^*_{p,q}, t)^\alpha \leq \sum_t (2(\ell oad(S_{p,q}, t) + \ell oad(S_{p,q}, t-(2^{p-1}-1)) + \ell oad(S_{p,q}, t+(2^{p-1}-1))))^\alpha$. By calculation we get $\sum_t \ell oad(S^*_{p,q}, t)^\alpha \leq 2^\alpha \sum_t 3^{\alpha-1}(\ell oad(S_{p,q}, t)^\alpha + \ell oad(S_{p,q}, t-(2^{p-1}-1))^\alpha + \ell oad(S_{p,q}, t+(2^{p-1}-1))^\alpha) \leq 2^\alpha \cdot 3^{\alpha-1} \cdot 3 \cdot \text{cost}(S_{p,q}) = 6^\alpha \cdot \text{cost}(S_{p,q})$. □

The other direction which convert a schedule for nice job set to a schedule for the corresponding job set is easier:

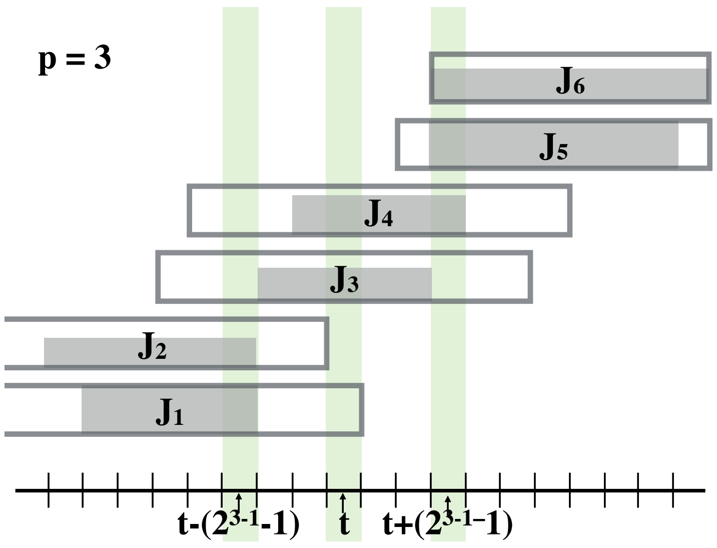


Figure 4.21: An illustration for Lemma 4.29.

Transformation ShrinkSch**.** ShrinkSch converts a schedule S^* for a nice job set $\mathcal{J}^*$ to a schedule S for the corresponding job set $\mathcal{J}$;

- $st(S, J) \leftarrow st(S^*, J^*)$.
- $et(S, J) \leftarrow st(S, J) + w(J)$. Since $w(J) \leq w(J^*)$, $et(S, J) \leq et(S^*, J^*)$.

The Transformation ShrinkSch shrink the execution interval of each nice job J^* to an execution interval of its corresponding J. By the following observation we show that the resulting execution interval is valid for J.

Observation 9. Consider any feasible schedule S^* for $\mathcal{J}^*$ and schedule S constructed by ShrinkSch for the corresponding $\mathcal{J}$, where $\mathcal{J}^*$ is the generated by Convert given $\mathcal{J}$. For any J^* and the corresponding J,
(i) $[st(S, J), et(S, J)) \subseteq [st(S^*, J^*), et(S^*, J^*))$;
(ii) $[st(S, J), et(S, J)) \subseteq [r(J), d(J))$.

Proof. (i) It is clear to see by the procedure of ShrinkSch that $st(S, J) = st(S^*, J^*)$ and $et(S, J) \leq et(S^*, J^*)$.

(ii) Since S^* is feasible, $[st(S^*, J^*), et(S^*, J^*)) \subseteq I(J^*)$. By Convert, $r(J^*) = r(J)$, so $st(S, J) \geq r(J)$. According to Convert, there are two possibility of $I(J^*)$, $I(J^*) = I(J)$ or $I(J^*) \neq I(J)$. Assume that $|I(J^*)| = |I(J)|$, it is easy to see that $[st(S, J), et(S, J)) \subseteq [st(S^*, J^*), et(S^*, J^*)) \subseteq I(J^*) \subseteq I(J)$. On the other hand, assume that $I(J) \neq I(J^*)$. In this case $[st(S^*, J^*), et(S^*, J^*)) = I(J^*)$, $r(J) = r(J^*)$, and $d(J) \geq r(J) + w(J)$. Hence $[st(S, J), et(S, J)) \subseteq [r(J), d(J))$. □

The following lemma shows that given a schedule $S_{p,q}$, the cost of the corresponding schedule $S^*_{p,q}$ generated by ShrinkSch can be bounded by the cost of $S_{p,q}$.

Lemma 4.31. *Using* ShrinkSch*, we have* $cost(S_{p,q}) \leq cost(S^*_{p,q})$.

Proof. According to ShrinkSch, the execution interval of each job J shrinks from the one of its corresponding J^*. Hence for any time t, $\ell oad(S_{p,q}, t) \leq \ell oad(S^*_{p,q}, t)$. By definition, $\text{cost}(S_{p,q}) \leq \text{cost}(S^*_{p,q})$. □

Now we have all elements we need and are ready to introduce the online algorithm for general input:

Approximation algorithm $\mathcal{A}_g$ for general input (Algorithm 7). When a job J is released, it is converted to J^* by Convert and classified into one of the classes $C_{p,q}$. Jobs in the same class after Convert are with uniform widths and uniform heights. For each class, jobs are scheduled by $\mathcal{A}_u$ independently of other classes. We then modify the execution interval of J^* to a execution interval of J by ShrinkSch.

Algorithm 7 The approximation algorithm $\mathcal{A}_g$

```
Input: a set of job $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$.
$\mathcal{A}_u$ is the approximation algorithm for uniform-width case
$\mathcal{J}^*_{ij} \leftarrow \emptyset$ for $1 \leq i \leq 1 + \lceil \log_2 w_{\max} \rceil$ and $1 \leq j \leq 1 + \lceil \log_2 h_{\max} \rceil$
for each job $J_k$ do
    $i \leftarrow \lceil \log_2 w(J_k) \rceil$
    $j \leftarrow \lceil \log_2 h(J_k) \rceil$
    $w(J^*_k) \leftarrow 2^i$
    $h(J^*_k) \leftarrow 2^j$
    $r(J^*_k) \leftarrow r(J_k)$
    $d(J^*_k) \leftarrow r(J_k) + \max\{d(J_k) - r(J_k), w(J^*_k)\}$
    Add $J^*_k$ into $\mathcal{J}^*_{ij}$
for each job $J^*_k \in \mathcal{J}^*_{ij}$ do
    $st(\mathcal{A}_g, J_k) \leftarrow st(\mathcal{A}_u(\mathcal{J}^*_{ij}), J^*_k)$
return the schedule $\mathcal{A}_g$
```

Analysis of Algorithm $\mathcal{A}_g$. Using Theorem 4.28, we know the ratio between the cost of $\mathcal{A}_g(\mathcal{J}_{p,q})$ and the cost of $\mathcal{O}(\mathcal{J}^*_{p,q})$ for each class $C_{p,q}$. It remains to find out the relation between the cost of $\mathcal{O}(\mathcal{J}^*_{p,q})$ and the cost of $\mathcal{O}(\mathcal{J}_{p,q})$.

Observation 10. Consider any job set $\mathcal{J}$, its corresponding nice job set $\mathcal{J}^*$ generated by Convert and the the corresponding job set of each class $\mathcal{J}_{p,q}$ and $\mathcal{J}^*_{p,q}$.

(i) $\text{cost}(\mathcal{O}(\mathcal{J}^*_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_{p,q}))$;

(ii) $\text{cost}(\mathcal{O}(\mathcal{J}^*_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$.

Proof. (i) Given $\mathcal{O}(\mathcal{J}_{p,q})$, there exists a schedule $S(\mathcal{J}^*_{p,q})$ generated by RelaxSch. By Lemma 4.30, $\text{cost}(S(\mathcal{J}^*_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_{p,q}))$. Hence, $\text{cost}(\mathcal{O}(\mathcal{J}^*_{p,q})) \leq \text{cost}(S(\mathcal{J}^*_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_{p,q}))$.

(ii) By Observation 6, $\text{cost}(\mathcal{O}(\mathcal{J}_{p,q})) \leq \text{cost}(\mathcal{O}(\mathcal{J}))$. Also by (i) $\text{cost}(\mathcal{O}(\mathcal{J}^*_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_{p,q})) \leq 6^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$, □

Now we are ready to compute the approximation ratio of algorithm $\mathcal{A}_g$:

Theorem 4.32. *For any job set $\mathcal{J}$, we have cost*$(\mathcal{A}_g(\mathcal{J})) \leq (36K_wK_h)^\alpha \cdot$ *cost*$(\mathcal{O}(\mathcal{J}))$, *where* $K_w = 1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil$ *and* $K_h = 1 + \lceil \log \frac{h_{\max}}{h_{\min}} \rceil$.

Proof. We first focus on the cost in one single class $C_{p,q}$. According to the algorithm $\mathcal{A}_g$, $\text{cost}(\mathcal{A}_g(\mathcal{J}_{p,q})) = \sum_t \ell oad(\mathcal{A}_g(\mathcal{J}_{p,q}), t)^\alpha \leq \sum_t \ell oad(\mathcal{A}_u(\mathcal{J}^*_{p,q}), t)^\alpha = \text{cost}(\mathcal{A}_u(\mathcal{J}^*_{p,q}))$. By Theorem 4.28, $\text{cost}(\mathcal{A}_u(\mathcal{J}^*_{p,q})) \leq 36^\alpha \cdot \mathcal{O}(\mathcal{J}^*_{p,q})$. By Observation 10, $\text{cost}(\mathcal{A}_u(\mathcal{J}^*_{p,q})) \leq 36^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$.

Now we turn to the total cost of the whole input set. By definition, $\text{cost}(\mathcal{A}_g(\mathcal{J})) = \sum_t \ell oad(\mathcal{A}_g(\mathcal{J}), t)^\alpha = \sum_t (\sum_{p,q} \ell oad(\mathcal{A}_g(\mathcal{J}_{p,q}), t))^\alpha$. By carefully calculation, $\text{cost}(\mathcal{A}_g(\mathcal{J})) \leq (K_wK_h)^{\alpha-1} \sum_t \sum_{p,q} \ell oad(\mathcal{A}_g(\mathcal{J}_{p,q}), t)^\alpha = (K_wK_h)^{\alpha-1} \sum_{p,q} \text{cost}(\mathcal{A}_g(\mathcal{J}_{p,q}))$. As the analysis stated in last paragraph, $\text{cost}(\mathcal{A}_g(\mathcal{J}_{p,q})) \leq 36^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. Hence $\text{cost}(\mathcal{A}_g(\mathcal{J})) \leq \sum_{p,q} (K_wK_h)^{\alpha-1} \cdot 36^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J})) = 36^\alpha \cdot (K_wK_h)^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. □

4.5 Summary

In this chapter we investigate the GRID problem under the offline model. First we show that the GRID problem is NP-hard even when all jobs have common feasible interval, and the power request or the time duration of the jobs are unit size. Moreover, the GRID problem is NP-hard even when preemption is allowed.

However, when all jobs have unit power request, unit time duration, and arbitrary feasible timeslots, the GRID problem is polynomial time solvable. We proposed a polynomial time algorithm for this special input by using a "feasibility graph" to capture all possible assignments. By relating to the Discrete DVS problem, we proposed another polynomial time algorithm to solve the GRID problem with unit power request and unit time duration and contiguous feasible intervals. It gives us a clearer view about the the GRID problem and the DVS problem.

For the general input where jobs have arbitrary widths, arbitrary heights, and arbitrary feasible intervals, we proposed two exact algorithms inspired by the interval graphs. We showed that the GRID problem is fixed-parameter tractable. We also proposed an approximation algorithm for the general input jobs by generalizing the optimal algorithm for unit case.

Chapter 5

Online Algorithms for The GRID Problem

This chapter investigates the GRID problem in the online model. We consider online algorithms, where the job information is only revealed at the time the job is released. The scheduling algorithm has to decide which jobs to run at the current time without future information and decisions made cannot be changed later. In particular, we consider non-preemptive algorithms where a job cannot be preempted to resume/restart later. Our goal is to minimize the worst case competitive ratio, i.e. the maximum ratio between the cost of the online algorithm and the cost of an offline optimal algorithm.

Our main contribution is proposing the first online algorithm to the online GRID problem for jobs with arbitrary power requests, arbitrary time durations, and arbitrary feasible intervals (which is the time interval between the release time and deadline of a job). The online algorithm is based on the classification on the time durations of jobs and the worst case competitive ratio is $(36(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil))^{\alpha} \cdot \Big(\min\{8(e + e^2), \frac{(2\alpha)^{\alpha}\alpha}{2}\} + 1 \Big)$ where $w_{\max}$ and $w_{\min}$ are the maximum and minimum time duration among the jobs, respectively. We also prove that for any deterministic online algorithm, the competitive ratio is at least $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^{\alpha}$.

Other than the GRID problem for jobs with arbitrary widths, arbitrary heights, and arbitrary feasible intervals, we also investigate the GRID problem with special input. We show that when input jobs have restricted release times or deadlines, there are online strategies with better performance (Section 5.3).

Our online algorithms are based on identifying a relationship with the dynamic speed (voltage) scaling (DVS) problem. The main challenge, even when jobs have uniform width or uniform height, is that in time intervals where the "workload" is low, the optimal DVS schedule may have much lower cost than the optimal GRID schedule because jobs in DVS schedules can effectively be stretched as flat as possible while jobs in GRID schedules have rigid duration and cannot be stretched. In such case, it is insufficient to simply compare with the optimal DVS schedule. Therefore, our analysis is divided into two parts: for high workload intervals, we compare with the optimal DVS schedule; and

Width	**Height**	**Ratio**
Unit	Uniform	$\min\{\frac{(4\alpha)^\alpha}{2}+1, 2^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)\}$-competitive
Unit	Arbitrary	$2^{\alpha+1}$-approximate
		same release time and same deadline: 2^α-competitive
		$2^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)$-competitive
Uniform	Arbitrary	$12^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)$-competitive
Arbitrary	Uniform	same release time and same deadline: $2^{2\alpha}$-competitive
		agreeable deadline: $(\frac{(8\alpha)^\alpha}{2}+2^\alpha)$-competitive
Arbitrary	Arbitrary	$(36K_w)^\alpha \cdot \Big(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1\Big)$-competitive

Table 5.1: Summary of our online or approximation algorithms.

for low workload intervals, we directly compare with the optimal GRID schedule via a lower bound on the total workload over these intervals. For jobs with arbitrary width, we adopt the natural approach of classification based on job width. We then align the "feasible interval" of each job in a more uniform way so that we can use the results on uniform width.

Table 5.1 summarizes our results of online algorithms for different input set ($K_w = 1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil$).

Relating to the speed scaling problem. The GRID problem resembles the dynamic speed scaling (DVS) problem [81] and we are going to refer to three algorithms for the DVS problem, namely, the $\mathcal{YDS}$ algorithm which gives an optimal algorithm for the DVS problem, the online algorithms called $\mathcal{BKP}$ and $\mathcal{AVR}$. We first recap the DVS problem and the associated algorithms. In the DVS problem, jobs J come with release time $r(J)$, deadline $d(J)$, and a work requirement $p(J)$. A processor can run at speed $s \in [0, \infty)$ and consumes energy in a rate of s^α, for some $\alpha > 1$. The objective is to complete all jobs by their deadlines using the minimum total energy. The main differences of DVS problem to the GRID problem include (i) jobs in DVS can be preempted while preemption is not allowed in our problem; (ii) as processor speed in DVS can scale, a job can be executed for varying time duration as long as the total work is completed while in our problem a job must be executed for a fixed duration given as input; (iii) the work requirement $p(J)$ of a job J in DVS can be seen as $w(J) \times h(J)$ for the corresponding job in GRID.

With the resemblance of the two problems, we make an observation about their optimal algorithms. Let $\mathcal{O}_D$ and $\mathcal{O}_G$ be the optimal algorithm for the DVS and GRID problem, respectively. Given a job set $\mathcal{J}_G$ for the GRID problem, we can convert it into a job set $\mathcal{J}_D$ for DVS by keeping the release time and deadline for each job and setting the work requirement of a job in $\mathcal{J}_D$ to the product of the width and height of the corresponding job in $\mathcal{J}_G$. Then we have the following observation.

Observation 11. Given any schedule S_G for $\mathcal{J}_G$, we can convert S_G into a feasible schedule S_D for $\mathcal{J}_D$ such that $\text{cost}(S_D(\mathcal{J}_D)) \leq \text{cost}(S_G(\mathcal{J}_G))$; implying that $\text{cost}(\mathcal{O}_D(\mathcal{J}_D)) \leq \text{cost}(\mathcal{O}_G(\mathcal{J}_G))$.

Proof. Consider any feasible schedule S_G. At timeslot t, suppose there are k jobs scheduled and their sum of heights is H. The schedule for S_D during timeslot t can be obtained by running the processor at speed H and the jobs time-share the processor in proportion to their height. This results in a feasible schedule with the same cost and the observation follows. □

It is known that the online algorithm $\mathcal{AVR}$ for the DVS problem is $\frac{(2\alpha)^\alpha}{2}$-competitive [81]. Basically, at any time t, $\mathcal{AVR}$ runs the processor at a speed which is the sum of the densities of jobs that are available at t. By Observation 11, we have the following corollary. Note that it is not always possible to convert a feasible schedule for the DVS problem to a feasible schedule for the GRID problem easily. Therefore, the corollary does not immediately solve the GRID problem but as to be shown it provides a way to analyze algorithms for GRID.

Corollary 5.1. *For any input $\mathcal{J}_G$ and the corresponding input $\mathcal{J}_D$, cost$(\mathcal{AVR}(\mathcal{J}_D)) \leq \frac{(2\alpha)^\alpha}{2} \cdot$ cost$(\mathcal{O}_G)$.*

The online algorithm $\mathcal{BKP}$ proposed by Bansal et al. [6] for DVS problem is $8e^\alpha$-competitive. Let $\ell oad(\mathcal{BKP}, t)$ denote the speed of $\mathcal{BKP}$ at time t. $\ell oad(\mathcal{BKP}, t) = \max_{t'>t} \frac{p(t,[et-(e-1)t',t'))}{t'-t}$ where $p(t, I)$ denotes the total work of jobs J with $I(J) \subseteq I$ and $r(J) \leq t$. That is, $\mathcal{BKP}$ chooses the interval $I^\star = [t_1, t_2)$ which has maximal released average load and $|[t_1, t_2)| : |[t, t2)| = e : 1$ and uses $e \cdot \frac{p(t,I^\star)}{|I^\star|}$ as speed at t. By Observation 11 we have the following corollary:

Corollary 5.2. *For any input $\mathcal{J}_G$ and the corresponding input $\mathcal{J}_D$, cost$(\mathcal{BKP}(\mathcal{J}_D)) \leq 8e^\alpha \cdot$ cost$(\mathcal{O}_G)$.*

Remark: One may consider the non-preemptive DVS problem as the reference of the GRID problem. However, given a job set $\mathcal{J}_G$ and the corresponding $\mathcal{J}_D$, $\text{cost}(\mathcal{O}_D(\mathcal{J}_D))$ may not necessarily lower than $\text{cost}(\mathcal{O}_G(\mathcal{J}_G))$, where $\mathcal{O}_D$ here is the optimal algorithm for non-preemptive DVS. There is an instance shows the optimal cost of GRID is smaller. The instance contains two jobs. One has release time 0 deadline 3, width 3 and height 1. The other has release time 1, deadline 2, width 1 and height 1. Both jobs can only schedule at their release time in GRID since their widths are the same as the lengths of their feasible intervals. The optimal cost of GRID is $1^\alpha + 2^\alpha + 1^\alpha = 2^\alpha + 2$. Whereas the optimal cost of non-preemptive DVS is $2^\alpha + 2^\alpha = 2 \cdot 2^\alpha$. This is because the schedule uses speed 2 and runs the longer job with time interval 1.5 and the shorter job with time interval 0.5. The optimal cost of GRID is lower when $\alpha > 1$. Therefore, it is unclear how we may use the results on non-preemptive DVS problem and so we would stick with the preemptive DVS algorithms.

In Section 5.1.3, we consider an algorithm that classifies jobs according to their widths. To ease discussion, we let $w_{\max}$ and $w_{\min}$ be the maximum and minimum width over all jobs, respectively. We further define the logarithm of the max-min ratio of width, denoted by k_w, to be $k_w = \log \frac{w_{\max}}{w_{\min}}$. We further define K_w to be $K_w = 1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil$, which is the number of classes (to be defined). Without loss of generality, we assume that $w_{\min} = 1$. We say that a job J is in *class* C_p if and only if $2^{p-1} < w(J) \leq 2^p$ for any $0 \leq p \leq K_w$.

5.1 General Case

To handle jobs of arbitrary width and height, we first study the case when jobs have unit width (that is, all jobs J have $w(J) = 1$). We present an online algorithm $\mathcal{V}$ which makes reference to an arbitrary feasible online algorithm for the DVS problem, denoted by $\mathcal{R}$. Using a similar technique in Section 4.4, we generalize this algorithm to solve the case when jobs have uniform width (i.e., all jobs J have the uniform width $w(J) = w > 1$) in Section 5.1.2 and the general case where jobs have arbitrary width in Section 5.1.3.

5.1.1 Unit width and arbitrary height

In this section, we consider jobs with unit width and arbitrary height. We present an online algorithm $\mathcal{V}$ which makes reference to an arbitrary feasible online algorithm for the DVS problem, denoted by $\mathcal{R}$. In particular, we require that the speed of $\mathcal{R}$ remains the same during any integral timeslot, i.e., in $[t, t+1)$ for all integers t. Note that when jobs have integral release times and deadlines, many known DVS algorithms satisfy this criteria, including $\mathcal{YDS}$ and $\mathcal{AVR}$. On the other hand, $\mathcal{BKP}$ does not satisfy this criteria and the speeds in $[t, t+1)$ might change. Nevertheless, by Lemma 5.3, we can modify $\mathcal{BKP}$ such that the criteria is satisfied and the performance remains well.

Lemma 5.3. *Let $s(t)$ denote the speed at t decided by $\mathcal{BKP}$. For any integral time t and a constant $0 < \Delta \leq 1$, $s(t + \Delta) \leq (1 + e) \cdot s(t)$ if the release times and deadlines of jobs are integral.*

Proof. Recall that the speed of $\mathcal{BKP}$ at time t is $s(t) = \max_{t'>t} e \cdot \frac{p(t,I)}{|I|}$ where $I = [t_1, t_2)$ and $|[t_1, t_2)| : |[t, t2)| = e : 1$. The proof idea is, consider the interval I chosen by $\mathcal{BKP}$ corresponding to $t + \Delta$, we can transform it into another interval I' which is one of the interval candidate for t. We show that the $e \cdot \frac{p(t,I')}{|I'|}$ is at least $\frac{1}{1+e}$ times of the speed of $\mathcal{BKP}$ at $t + \Delta$.

Assume that at time $t + \Delta$, $s(t + \Delta) = e \cdot \frac{p(t,I)}{|I|}$ where $I = [t_1, t_2)$. We can construct $I' = [t'_1, t_2)$ such that $|[t'_1, t_2)| : |[t, t_2)| = e : 1$ by setting $t'_1 = t_2 - e(t_2 - t)$. It is clear that $I \subset I'$ since the two intervals have same right endpoint and I' is longer than I. In fact, $|I'| = e(t_2 - t) = e(t_2 - (t + \Delta) + \Delta) = e(t_2 - (t + \Delta)) + e\Delta = |I| + e\Delta \leq |I| + e$. Moreover, for any interval candidate, the length must be at least 1 if the release times or deadlines of the jobs are integral. Otherwise, the interval contains no jobs and the speed is 0. Hence, $|I'| \leq (1+e)|I|$. By $\mathcal{BKP}$, $s(t) \geq e \cdot \frac{p(t,I')}{|I'|} = e \cdot \frac{p(t+\Delta,I')}{|I'|}$. The later equality holds since there is no job released between t and $t+\Delta$. Since $I \subset I'$ and $|I'| \leq (1+e)|I|$, $e \cdot \frac{p(t+\Delta,I')}{|I'|} \geq e \cdot \frac{p(t+\Delta,I)}{|I'|} \geq e \cdot \frac{p(t+\Delta,I)}{(1+e)|I|}$. Hence, $s(t) \geq \frac{1}{1+e} \cdot s(t+\Delta)$. $\square$

Lemma 5.3 implies that, although the speeds of $\mathcal{BKP}$ change within $[t, t+1)$, the speeds are bounded by $(1+e)$ times of the speed at t. Hence we can modify the $\mathcal{BKP}$ into $\mathcal{BKP}'$ as follows: at integral time t, the speed of $\mathcal{BKP}'$, $s(\mathcal{BKP}', t) = (1+e)s(t)$; at time $t' = t + \Delta$ where t is integral and $0 < \Delta < 1$, $s(\mathcal{BKP}', t') = s(\mathcal{BKP}', t)$. By the modification, the speed of $\mathcal{BKP}'$ remains the same during any integral timeslot, and $\text{cost}(\mathcal{BKP}') \leq (1+e)^\alpha \cdot \text{cost}(\mathcal{BKP})$. Similar to Corollary 5.1 and 5.4, we have the following corollary:

Corollary 5.4. *For any input $\mathcal{J}_G$ and the corresponding input $\mathcal{J}_D$, cost$(\mathcal{BKP}'(\mathcal{J}_D)) \leq 8(e+e^2)^\alpha \cdot$ cost$(\mathcal{O}_G)$.*

We can transform an input set of GRID problem to an input set of DVS problem by the following: for each job J, the corresponding job in the *dvs* problem has work load $p(J) \leftarrow w(J) \cdot h(J)$, release time $r(J)$, and deadline $d(J)$. Notice that for special input where jobs sizes are unit, the consequent jobs are with unit work loads. We simulate a copy of $\mathcal{R}$ on the converted job set and denote the speed used by $\mathcal{R}$ at t as $\ell oad(\mathcal{R}, t)$. Our algorithm makes reference to $\ell oad(\mathcal{R}, t)$ but not the jobs run by $\mathcal{R}$ at t.

Algorithm $\mathcal{V}$. For each timeslot t, we schedule jobs to start at t such that $\ell oad(\mathcal{V}, t)$ is at least $\ell oad(\mathcal{R}, t)$ or until all available jobs have been scheduled. Jobs are chosen in an EDF manner.

Analysis. We note that since $\mathcal{V}$ makes decision at integral time and jobs have unit width, each job is completed before any further scheduling decision is made. In other words, $\mathcal{V}$ is non-preemptive. To analyze the performance of $\mathcal{V}$, we first note that $\mathcal{V}$ gives a feasible schedule (Lemma 5.5), and then analyze its competitive ratio (Theorem 5.7).

Lemma 5.5. *$\mathcal{V}$ gives a feasible schedule.*

Proof. Let $\ell oad(S, \mathcal{I})$ denote the total work done by schedule S in $\mathcal{I}$. That is, $\ell oad(S, \mathcal{I}) = \sum_{t \in \mathcal{I}} \ell oad(S, \mathcal{I})$. According to the algorithm, for all $\mathcal{I}_t = [0, t)$, $\ell oad(\mathcal{V}, \mathcal{I}_t) \geq \ell oad(\mathcal{R}, \mathcal{I}_t)$.

Suppose on the contrary that $\mathcal{V}$ has a job J_m missing deadline at t. That is, $d(J_m) = t$ but J_m is not assigned before t. By the algorithm, for all $t' \in [0, t)$, $\ell oad(\mathcal{V}, t') \geq \ell oad(\mathcal{R}, t')$ unless there are less than $\ell oad(\mathcal{R}, t')$ available jobs at t' for $\mathcal{V}$. Let t_0 be the last timeslot in $[0, t)$ such that $\ell oad(\mathcal{V}, t_0) < \ell oad(\mathcal{R}, t_0)$, $r(J_m) > t_0$ since all jobs released at or before t_0 have been assigned. For all $t' \in (t_0, t)$, $\ell oad(\mathcal{V}, t') \geq \ell oad(\mathcal{R}, t')$. Also, all jobs J with $r(J) \leq t_0$ are finished by $t_0 + 1$ and jobs executed in (t_0, t) are those released after t_0. Consider set $\mathcal{J}_t$ of jobs with feasible interval completely inside $\mathcal{I} = (t_0, t)$ (note that $J_m \in \mathcal{J}_t$), $\ell oad(S, \mathcal{I}) \geq \sum_{J \in \mathcal{J}_t} h(J)$ for any feasible schedule S. Since $\mathcal{V}$ assigns jobs in EDF manner and is not feasible, $\ell oad(\mathcal{V}, \mathcal{I}) < \sum_{J \in \mathcal{J}_t} h(J)$. It follows that $\sum_{J \in \mathcal{J}_t} h(J) > \ell oad(\mathcal{V}, \mathcal{I}) \geq \ell oad(\mathcal{R}, \mathcal{I})$. It contradicts to the fact that $\mathcal{R}$ is feasible. Hence $\mathcal{V}$ finishes all jobs before their deadlines. $\square$

Let $h_{\max}(\mathcal{V},t)$ be the maximum height of jobs scheduled at t by $\mathcal{V}$; we set $h_{\max}(\mathcal{V},t) = 0$ if $\mathcal{V}$ assigns no job at t. We first classify each timeslot t into two types: (i) $h_{\max}(\mathcal{V},t) < \ell oad(\mathcal{R},t)$, and (ii) $h_{\max}(\mathcal{V},t) \geq \ell oad(\mathcal{R},t)$. We denote by $\mathcal{I}_1$ and $\mathcal{I}_2$ the union of all timeslots of Type (i) and (ii), respectively. Notice that $\mathcal{I}_1$ and $\mathcal{I}_2$ can be empty and the union of $\mathcal{I}_1$ and $\mathcal{I}_2$ covers the entire time line. The following lemma bounds the cost of $\mathcal{V}$ in each type of timeslots. Recall that $\text{cost}(S,\mathcal{I})$ denotes the cost of the schedule S over the interval $\mathcal{I}$ and $\text{cost}(S)$ denotes the cost of the entire schedule.

Lemma 5.6. *The cost of $\mathcal{V}$ satisfies the following properties.*
(i) $cost(\mathcal{V},\mathcal{I}_1) \leq 2^\alpha \cdot cost(\mathcal{R})$; and
(ii) $cost(\mathcal{V},\mathcal{I}_2) \leq 2^\alpha \cdot cost(\mathcal{O})$.

Proof. (i) By the algorithm, $\ell oad(\mathcal{V},t) < \ell oad(\mathcal{R},t) + h_{\max}(\mathcal{V},t) \leq 2 \cdot \ell oad(\mathcal{R},t)$ for $t \in \mathcal{I}_1$. It follows that $\text{cost}(\mathcal{V},\mathcal{I}_1) \leq 2^\alpha \cdot \sum_{t\in\mathcal{I}_1} \ell oad(\mathcal{R},t)^\alpha = 2^\alpha \cdot \text{cost}(\mathcal{R},\mathcal{I}_1) \leq 2^\alpha \cdot \text{cost}(\mathcal{R})$.

(ii) By convexity, $\text{cost}(\mathcal{O}) \geq \sum_{J\in\mathcal{J}} h(J)^\alpha$. It is easy to see that $\text{cost}(\mathcal{O}) \geq \sum_{t\in\mathcal{I}_2} h_{\max}(\mathcal{V},t)^\alpha$. According to the algorithm, $\ell oad(\mathcal{V},t) < \ell oad(\mathcal{R},t) + h_{\max}(\mathcal{V},t) \leq 2 \cdot h_{\max}(\mathcal{V},t)$ for $t \in \mathcal{I}_2$. Hence, $\text{cost}(\mathcal{V},\mathcal{I}_2) = \sum_{t\in\mathcal{I}_2} \ell oad(\mathcal{V},t)^\alpha \leq 2^\alpha \cdot \sum_{t\in\mathcal{I}_2} h_{\max}(\mathcal{V},t)^\alpha \leq 2^\alpha \cdot \text{cost}(\mathcal{O})$. □

Notice that $\text{cost}(\mathcal{V}) = \text{cost}(\mathcal{V},\mathcal{I}_1) + \text{cost}(\mathcal{V},\mathcal{I}_2)$ since $\mathcal{I}_1$ and $\mathcal{I}_2$ have no overlap. Together with Lemma 5.6 and Observation 11, we obtain the competitive ratio of $\mathcal{V}$ in the following theorem.

Theorem 5.7. *Algorithm $\mathcal{V}$ is $2^\alpha \cdot (R+1)$-competitive, where R is the competitive ratio of the reference* DVS *algorithm $\mathcal{R}$.*

As mentioned in Section 5.1.1 and Section 3.2, the $\mathcal{BKP}'$ algorithm is $8(e+e^2)^\alpha$-competitive and $\mathcal{AVR}$ algorithm is $\frac{(2\alpha)^\alpha}{2}$-competitive. On the other hand, $\mathcal{V}$ can take an offline DVS algorithm, e.g., the optimal $\mathcal{YDS}$ algorithm, as reference and returns an offline schedule. Therefore, we have the following corollary.

Corollary 5.8. *$\mathcal{V}$ is $2^\alpha \cdot (8(e+e^2)^\alpha + 1)$-competitive, $2^\alpha \cdot (\frac{(2\alpha)^\alpha}{2} + 1)$-competitive and $2^\alpha \cdot 2$-approximate when the algorithm $\mathcal{BKP}'$, $\mathcal{AVR}$ and $\mathcal{YDS}$ are referenced, respectively.*

5.1.2 Uniform width and arbitrary height

In this section, we consider jobs J with uniform width w and arbitrary height. We adapt the techniques in Section 4.4 and generalize the algorithm $\mathcal{V}$ to serve uniform width input. Let $\mathcal{J}^*$ be a uniform width job set and let $\mathcal{J}^*_{\text{T}}$ and $\mathcal{J}^*_{\text{L}}$ be the disjoint subsets of tight and loose jobs of $\mathcal{J}^*$, respectively. We design different strategies for tight and loose jobs. As to be shown, tight jobs can be handled easily by starting them at their release times. For any loose job, we modify it via Procedure AlignFI such that its release time and deadline is a multiple of w. The consequent job set $\mathcal{J}'$ can be treated as a set of unit-width jobs and the algorithm $\mathcal{V}$ can be performed on $\mathcal{J}'$.

Online algorithm $\mathcal{UV}$. The algorithm takes a job set $\mathcal{J}^*$ with uniform width w as input and schedules the jobs in $\mathcal{J}^*$ as follows. Let $\mathcal{J}^*_{\text{T}}$ be the set of tight jobs in $\mathcal{J}^*$ and $\mathcal{J}^*_{\text{L}}$ be the set of loose jobs in $\mathcal{J}^*$.

1. For any tight job $J^* \in \mathcal{J}^*_{\mathrm{T}}$, schedule J^* to start at $r(J^*)$.
2. Loose jobs in $\mathcal{J}^*_{\mathrm{L}}$ are converted to $\mathcal{J}'$ by AlignFI. For $\mathcal{J}'$, we run Algorithm $\mathcal{V}$, which is defined in Section 5.1.1, with $\mathcal{BKP}'$ as the reference DVS algorithm. Jobs are chosen in an earliest deadline first (EDF) manner.

Note that the decisions of $\mathcal{UV}$ can be made online.

Analysis of Algorithm $\mathcal{UV}$. We analyze the tight jobs and loose jobs separately. The analysis is mainly based on properties of AlignFI, AlignSch, and FreeSch which are proved in Section 4.4. However, since in Section 4.4 we generalized the algorithm $\mathcal{A}_{\mathrm{FG}}$ while in this section we generalize the algorithm $\mathcal{V}$, which is based on referencing the $\mathcal{BKP}$ algorithm, some parts of the analysis are different from the one in Section 4.4.

Lemma 5.9. *$cost(\mathcal{UV}(\mathcal{J}^*_T)) \le 3^\alpha \cdot cost(\mathcal{O}(\mathcal{J}^*))$.*

Proof. By Lemma 4.25, $\mathrm{cost}(\mathcal{UV}(\mathcal{J}_{\mathrm{T}})) \le 3^\alpha \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}_{\mathrm{T}})) \le 3^\alpha \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*))$. □

Lemma 5.10. *$cost(\mathcal{UV}(\mathcal{J}^*_L)) \le 6^\alpha \cdot (8(e+e^2)^\alpha + 1) \cdot cost(\mathcal{O}(\mathcal{J}^*))$.*

Proof. For $\mathcal{J}^*_{\mathrm{L}}$, we perform AlignFI and get $\mathcal{J}'$. We then run $\mathcal{V}$ on $\mathcal{J}'$ and get $\mathcal{V}(\mathcal{J}')$, which can be viewed as a schedule for unit width jobs. We get $S^*(\mathcal{J}^*_{\mathrm{L}}) = \mathcal{V}(\mathcal{J}')$ by FreeSch. Hence, $\mathrm{cost}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{L}})) = \sum_t \mathit{load}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{L}}), t)^\alpha = \sum_t \mathit{load}(S^*(\mathcal{J}^*_{\mathrm{L}}), t)^\alpha = \sum_t \mathit{load}(\mathcal{V}(\mathcal{J}'), t)^\alpha = \mathrm{cost}(\mathcal{V}(\mathcal{J}'))$. According to Corollary 5.8, $\mathrm{cost}(\mathcal{V}(\mathcal{J}')) \le 2^\alpha \cdot (8(e(1+e))^\alpha + 1) \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}'))$ by choosing $\mathcal{BKP}'$ as reference algorithm. Since $\mathcal{J}^*_{\mathrm{L}}$ is set of loose jobs with uniform width, $\mathrm{cost}(\mathcal{O}(\mathcal{J}')) \le 3^\alpha \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*_{\mathrm{L}})) \le 3^\alpha \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*))$ by Lemma 4.24 and Observation 6. Therefore, $\mathrm{cost}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{L}})) \le 2^\alpha \cdot (8 \cdot (e(1+e))^\alpha + 1) \cdot 3^\alpha \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*))$. □

Theorem 5.11. *$cost(\mathcal{UV}(\mathcal{J}^*)) \le 12^\alpha \cdot (8(e+e^2)^\alpha + 1) \cdot cost(\mathcal{O}(\mathcal{J}^*))$.*

Proof. By definition, $\mathrm{cost}(\mathcal{UV}(\mathcal{J}^*)) = \sum_t \mathit{load}(\mathcal{UV}(\mathcal{J}^*), t)^\alpha = \sum_t (\mathit{load}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{T}}), t) + \mathit{load}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{L}}), t))^\alpha \le 2^{\alpha-1} \cdot (\mathrm{cost}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{T}})) + \mathrm{cost}(\mathcal{UV}(\mathcal{J}^*_{\mathrm{L}})))$. By Lemma 5.9 and 5.10, $\mathrm{cost}(\mathcal{UV}(\mathcal{J}^*)) \le 2^{\alpha-1} \cdot (3^\alpha + 6^\alpha \cdot (8(e(1+e))^\alpha + 1)) \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*)) \le 2^\alpha \cdot 6^\alpha \cdot (8(e(1+e))^\alpha + 1) \cdot \mathrm{cost}(\mathcal{O}(\mathcal{J}^*))$. □

Similarly, if we choose $\mathcal{AVR}$ as the reference algorithm, we can get the following corollary:

Corollary 5.12. *$cost(\mathcal{UV}(\mathcal{J}^*)) \le 12^\alpha \cdot (\frac{(2\alpha)^\alpha}{2} + 1) \cdot cost(\mathcal{O}(\mathcal{J}^*))$.*

Corollary 5.13. *$cost(\mathcal{UV}(\mathcal{J}^*)) \le 12^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot cost(\mathcal{O}(\mathcal{J}^*))$.*

5.1.3 Arbitrary Input

In this section, we present an algorithm $\mathcal{A}$ for jobs with arbitrary width and height. We first transform job set $\mathcal{J}$ to a "nice" job set $\mathcal{J}^*$ (to be defined) and show that such a transformation only increases the cost modestly. Furthermore, we show that for any nice job set $\mathcal{J}^*$, we can bound $\text{cost}(\mathcal{A}(\mathcal{J}^*))$ by $\text{cost}(\mathcal{O}(\mathcal{J}^*))$ and in turn by $\text{cost}(\mathcal{O}(\mathcal{J}))$. Then we can establish the competitive ratio of $\mathcal{A}$.

Note that the definition of nice jobs in this section is different from the one in Section 4.4.2. Hence the analysis is different from the one we had before.

Nice job set and transformations. A job J is said to be a *nice job* if $w(J) = 2^p$, for some non-negative integer p and a job set $\mathcal{J}^*$ is said to be a *nice job set* if all its jobs are nice jobs. In other words, the nice job J is in class C_p.

Procedure Convert**.** Given a job set $\mathcal{J}$, we define the procedure Convert to transform each job $J \in \mathcal{J}$ into a nice job J^* as follows. We denote the resulting nice job set by $\mathcal{J}^*$. Suppose J is in class C_p. We modify its width, release time and deadline.

- $w(J^*) \leftarrow 2^p$;
- $r(J^*) \leftarrow r(J)$;
- $d(J^*) \leftarrow r(J^*) + \max\{d(J) - r(J), 2^p\}$.

Note that in Section 4.4, the Procedure Convert rounds up both the widths and the heights of the jobs, while in this section the Procedure Convert only rounds up the widths of the jobs. It is because that in Section 4.4, we adapt the optimal algorithm $\mathcal{A}_{\text{FG}}$ for jobs with unit width and unit height, while in this section we adapt the algorithm $\mathcal{V}$ for unit-width jobs.

Figure 5.1 shows an illustration for Procedure Convert where $p = 3$. After Convert, the jobs in C_p form a nice job set $\mathcal{J}^*$. It means that after Convert, for each C_p, the jobs have same widths. The modification of width is to transform J into a nice job by rounding up its width. After rounding up, the width may greater than the length of the original feasible and it makes the job itself infeasible (for example, the job J_2 in Figure 5.1). To make sure the resulting job is a part of feasible input, we also modify the deadline if needed.

The observation below follows directly from the definition.

Observation 12. For any job J and its nice job J^* transformed by Convert,
(i) $I(J) \subseteq I(J^*)$; and
(ii) $I(J) \neq I(J^*)$ if and only if $|I(J)| < 2^p$; in this case, $\text{den}(J) > \frac{1}{2}$ and $\text{den}(J^*) = 1$.

We then define two procedures that transform schedules related to nice job sets. RelaxSch takes a schedule S for a job set $\mathcal{J}$ and converts it to a schedule S^* for the corresponding nice job set $\mathcal{J}^*$. On the other hand, ShrinkSch takes a schedule S^* for a nice job set $\mathcal{J}^*$ and converts it to a schedule S for $\mathcal{J}$.

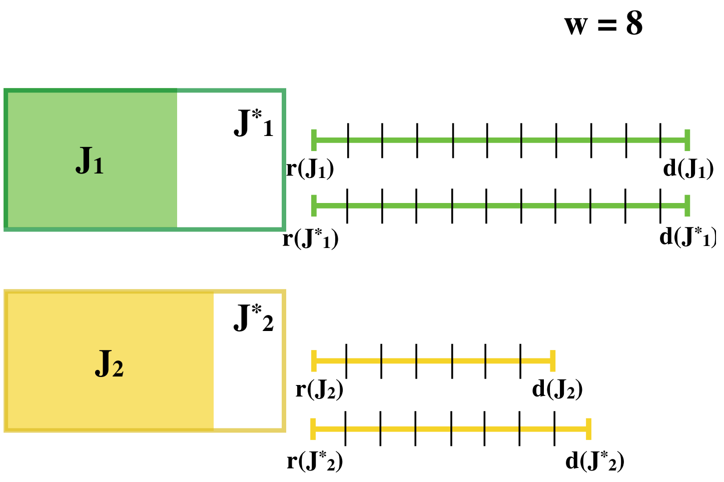


Figure 5.1: An illustration for Lemma 4.29.

Transformation RelaxSch**.** RelaxSch transforms S into S^* by moving the start and end time of every job J.

- $st(S^*, J^*) = \min\{d(J^*) - w(J^*), st(S, J)\}$
- $et(S^*, J^*) = st(S^*, J^*) + w(J^*)$.

Observation 13 asserts that the resulting schedule S^* is feasible for $\mathcal{J}^*$ while Lemmas 5.14 and 5.15 analyze the load and cost of the schedule.

Observation 13. Consider any schedule S for $\mathcal{J}$ and the schedule S^* constructed by RelaxSch for the corresponding $\mathcal{J}^*$. We have $[st(S^*, J^*), et(S^*, J^*)) \subseteq [r(J^*), d(J^*))$; in other words, S^* is a feasible schedule for $\mathcal{J}^*$.

To analyze the load of the schedule S^*, we consider partial schedule $S_p^* \subseteq S^*$ (resp. $S_p \subseteq S$) which is for all the jobs of $\mathcal{J}^*$ (resp. $\mathcal{J}$) in class C_p. Intuitively, the load of S_p^* at any time is at most the sum of the load of S_p at the current time and $2^{p-1} - 1$ timeslots before and after the current time.

Lemma 5.14. *At any time t, $\ell oad(S_p^*, t) \leq \ell oad(S_p, t) + \ell oad(S_p, t - (2^{p-1} - 1)) + \ell oad(S_p, t + (2^{p-1} - 1))$.*

Proof. We prove that for any job J, J^* contributes to $\ell oad(S^*, t)$ only if J contributes to either $\ell oad(S_p, t)$, $\ell oad(S_p, t - (2^{p-1} - 1))$, or $\ell oad(S_p, t + (2^{p-1} - 1))$. There are two cases that J does not contribute to $\ell oad(S_p, t - (2^{p-1} - 1))$ nor $\ell oad(S_p, t + (2^{p-1} - 1))$: (i) $et(S_p, J) < t - (2^{p-1} - 1)$ or $st(S_p, J) > t + (2^{p-1} - 1)$, and (ii) $[st(S_p, J), et(S_p, J)) \cap (t - (2^{p-1} - 1), t + (2^{p-1} - 1)) \neq \emptyset$.

Consider case (i), by Transformation RelaxSch, $et(S^*, J^*) \leq et(S_p, J) + (2^{p-1} - 1)$ and $st(S^*, J^*) \geq et(S_p, J) - (2^{p-1} - 1)$. Hence, $t \notin [st(S^*, J^*), et(S^*, J^*))$ if $et(S_p, J) < t - (2^{p-1} - 1)$ or $st(S_p, J) > t + (2^{p-1} - 1)$. That is, J^* does not contribute to $\ell oad(S_p^*, t)$.

Notice that if $et(S_p, J) = t-(2^{p-1}-1)$ or $st(S_p, J) = t+(2^{p-1}-1)$, J does not necessarily contribute to $\ell oad(S_p^*, t)$. We count the contribution for worst case analysis.

For case (ii), consider job J with $et(S_p, J)) \cap (t-(2^{p-1}-1), t+(2^{p-1}-1)) \neq \emptyset$, that is, $[st(S_p, J), et(S_p, J)) \subseteq (t-(2^p-1), t+(2^p-1))$. Since $2^{p-1} < w(J) \leq 2^p$, J contributes to at least of $\ell oad(S_p, t)$, $\ell oad(S_p, t-(2^{p-1}-1))$, or $\ell oad(S_p, t+(2^{p-1}-1))$, no matter if the execution interval of the corresponding J^* generated by RelaxSch contributes to timeslots $t-(2^{p-1}-1)$ or $t+(2^{p-1}-1)$ or not.

By case (i) and (ii), for any job J with $[st(S_p, J), et(S, J)) \cap [t-(2^{p-1}-1), t+(2^{p-1}-1)) = \emptyset$, its corresponding J^* does not contribute to $\ell oad(S_p^*, t)$. And for any job J with $[st(S_p, J), et(S_p, J)) \subseteq (t-(2^{p-1}-1), t+(2^{p-1}-1))$, J contributes to $\ell oad(S_p, t)$. Hence, by assuming all jobs at timeslot $t-(2^{p-1}-1)$ or $t+(2^{p-1}-1)$ contribute to $\ell oad(S_p^*, t)$, $\ell oad(S_p^*, t)$ is bounded by $\ell oad(S_p, t)+\ell oad(S_p, t-(2^{p-1}-1))+\ell oad(S_p, t+(2^{p-1}-1))$. $\square$

Figure 5.2 shows an illustration for Lemma 5.14. The solid rectangles represent the jobs $J \in \mathcal{J}$, and the hollowed rectangles represent the possible execution intervals of the corresponding J^* generated by RelaxSch. Jobs J_1, J_2, and J_3 have execution intervals completely ouside the interval $[t-(2^{p-1}-1), t+(2^{p-1}-1))$ (that is, the (i) case in the proof). It is easy to see that the execution intervals generated by RelaxSch are not capable for covering t since $w(J^*) < 2w(J)$ for all $J \in \mathcal{J}$. On the other hand, jobs J_4, J_5, and J_6 have execution intervals overlapping with the interval $[t-(2^{p-1}-1), t+(2^{p-1}-1))$ (that is, the (ii) case in the proof). For these jobs, the original execution interval must over one of these timeslots: t, $t-(2^{p-1}-1)$, or $t+(2^{p-1}-1)$.

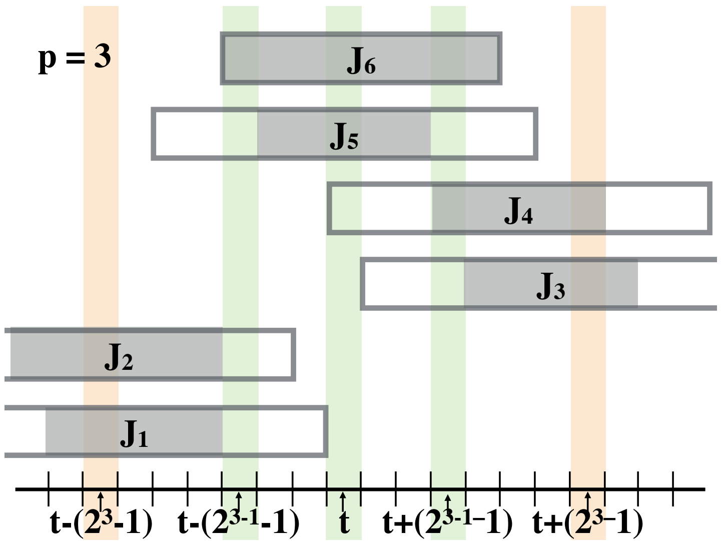


Figure 5.2: An illustration for Lemma 5.14.

Lemma 5.15. *Using* RelaxSch, *we have* $cost(S_p^*) \leq 3^\alpha \cdot cost(S_p)$.

Proof. By Lemma 5.14, $\text{cost}(S_p^*) = \sum_t \ell oad(S_p^*, t)^\alpha \leq \sum_t (\ell oad(S_p, t)+\ell oad(S_p, t-(2^{p-1}-1))+\ell oad(S_p, t+(2^{p-1}-1)))^\alpha \leq \sum_t (3 \cdot \ell oad(S_p, t))^\alpha = 3^\alpha \cdot \text{cost}(S_p)$. $\square$

Transformation ShrinkSch**.** On the other hand, ShrinkSch converts a schedule S^* for a nice job set $\mathcal{J}^*$ to a schedule S for the corresponding job set $\mathcal{J}$. We set

- $st(S, J) \leftarrow st(S^*, J^*)$;
- $et(S, J) \leftarrow st(S, J) + w(J)$, therefore, $et(S, J) \leq et(S^*, J^*)$.

Observation 14 asserts that the resulting schedule S is feasible for J and Lemma 5.16 analyzes the cost of the schedule.

Observation 14. Consider any schedule S^* for $\mathcal{J}^*$ and schedule S constructed by ShrinkSch for the corresponding $\mathcal{J}$. For any J^* and the corresponding J,
(i) $[st(S, J), et(S, J)] \subseteq [st(S^*, J^*), et(S^*, J^*)]$;
(ii) $[st(S, J), et(S, J)] \subseteq [r(J), d(J)]$.

By Observation 14, we have the following lemma.

Lemma 5.16. *Using* ShrinkSch*, we have cost*$(S_p) \leq$ *cost*(S_p^*).

The online algorithm $\mathcal{A}$. We are now ready to describe the algorithm $\mathcal{A}$ for an arbitrary job set $\mathcal{J}$. When a job J is released, it is converted to J^* by Convert and classified into one of the classes C_p. Jobs in the same class after Convert (being a uniform-width job set) are scheduled by $\mathcal{UV}$ independently of other classes. We then modify the execution time of J^* in $\mathcal{UV}$ to the execution time of J in $\mathcal{A}$ by Transformation ShrinkSch. Note that all these procedures can be done in an online fashion.

Using the results in Sections 5.1.2 and 5.1.3, we can compare the cost of $\mathcal{A}(\mathcal{J})$ with $\mathcal{O}(\mathcal{J}_p^*)$ for each class C_p (see Theorem 5.17). It remains to analyze the cost of $\mathcal{O}(\mathcal{J}_p^*)$ and $\mathcal{O}(\mathcal{J})$ in the next observation.

Observation 15. Consider any job set $\mathcal{J}$, its corresponding job set $\mathcal{J}^*$ and the corresponding job set of each class $\mathcal{J}_p$ and $\mathcal{J}_p^*$.
(i) $\text{cost}(\mathcal{O}(\mathcal{J}_p^*)) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_p))$;
(ii) $\text{cost}(\mathcal{O}(\mathcal{J}_p)) \leq \text{cost}(\mathcal{O}(\mathcal{J}))$.

Proof. (i) Given $\mathcal{O}(\mathcal{J}_p)$, there exists schedule $S(\mathcal{J}_p^*)$ generated by RelaxSch. By Lemma 5.15, $\text{cost}(S(\mathcal{J}_p^*)) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_p))$. Hence, $\text{cost}(\mathcal{O}(\mathcal{J}_p^*)) \leq \text{cost}(S(\mathcal{J}_p^*)) \leq 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_p))$.

(ii) Assume on the contrary that $\text{cost}(\mathcal{O}(\mathcal{J})) < \text{cost}(\mathcal{O}(\mathcal{J}_p))$, we can generate a schedule $S(\mathcal{J}_p)$ by removing jobs from $\mathcal{O}(\mathcal{J})$ which are not in $\mathcal{J}_p$. It follows that $\text{cost}(S(\mathcal{J}_p)) \leq \text{cost}(\mathcal{O}(\mathcal{J})) < \text{cost}(\mathcal{O}(\mathcal{J}_p))$, contradicting to the fact that $\mathcal{O}(\mathcal{J}_p)$ is optimal for $\mathcal{J}_p$. □

Theorem 5.17. *Denote* $K_w = 1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil$*, for any job set* $\mathcal{J}$*, we have cost*$(\mathcal{A}(\mathcal{J})) \leq (36K_w)^\alpha \cdot \left(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1\right) \cdot$ *cost*$(\mathcal{O}(\mathcal{J}))$.

Proof. By definition, $\text{cost}(\mathcal{A}(\mathcal{J})) = \sum_t \ell oad(\mathcal{A}(\mathcal{J}), t)^\alpha = \sum_t (\sum_{p=1}^{K_w} \ell oad(\mathcal{A}(\mathcal{J}_p), t))^\alpha$. The latter is at most ${K_w}^{\alpha-1} \sum_{p=1}^{K_w} \sum_t \ell oad(\mathcal{A}(\mathcal{J}_p), t)^\alpha$. For each group of jobs $\mathcal{J}_p$, we Convert it to $\mathcal{J}_p^*$, perform algorithm $\mathcal{UV}$ on it, and transform the schedule into a schedule for $\mathcal{J}_p$ by ShrinkSch. Hence, $\ell oad(\mathcal{A}(\mathcal{J}_p), t) \leq \ell oad(\mathcal{UV}(\mathcal{J}_p^*), t)$ for each t. It follows that $\text{cost}(\mathcal{A}(\mathcal{J})) \leq {K_w}^{\alpha-1} \sum_{p=1}^{K_w} \text{cost}(\mathcal{A}(\mathcal{J}_p)) \leq {K_w}^{\alpha-1} \cdot \sum_{p=1}^{K_w} \text{cost}(\mathcal{UV}(\mathcal{J}_p^*))$. By Corollary 5.13 and Observations 15, $\text{cost}(\mathcal{UV}(\mathcal{J}_p^*)) \leq 12^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot \text{cost}(\mathcal{O}(\mathcal{J}_p^*)) \leq 12^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot 3^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}_p)) \leq 36^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. Hence $\text{cost}(\mathcal{A}(\mathcal{J})) \leq 36^\alpha \cdot {K_w}^{\alpha-1} \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot \sum_{p=1}^{K_w} \text{cost}(\mathcal{O}(\mathcal{J})) = (36K_w)^\alpha \cdot (\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1) \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. □

Discussion about the classification factor. The logarithm in the competitive ratio comes from the number of classes defined in Section 5.1.3 since we partition the jobs by their width. In Class C_p, jobs J have width $2^{p-1} < w(J) \leq 2^p$ for $0 < p \leq \lceil \log K_w \rceil = \lceil \log_2 \lceil \frac{w_{\max}}{w_{\min}} \rceil \rceil$. For jobs in C_p, their widths are round up to 2^p. Due to the round up procedure, three timeslots load should be sampled in order to bound the load after Transformation RelaxSch (Lemma 5.14) and later it affects the competitive ratio by 3^α (Lemma 5.15, Observation 15 (i), and Theorem 5.17). The competitive ratio is also affected by $\log^\alpha K_w$ where $\log K_w$ is the number of classes.

Suppose we change the definition of classes such that class p includes jobs of width in the range $((1+\lambda)^{p-1}, (1+\lambda)^p]$ for some $\lambda > 0$ (which is not necessarily 2) and Procedure Convert such that the width of jobs in class C_p is round up to $(1+\lambda)^p$. Then, the number of classes becomes $\lceil \log_{1+\lambda} K_w \rceil$. When λ is big, the number of classes is less. However, the number of timeslots needs to be sample is bigger. More precisely, the competitive ratio depends on Lemma 5.14 that bounds the load at any timeslot by the load of three other timeslots. This number of timeslots is also affected by the definition of classes.

We define *penalty factor*, $F_p(\lambda)$, as $spl(\lambda) \cdot cls(\lambda)$ where spl is the number of timeslot to be sampled to bound the load and cls is the number of classes with classification factor λ. Let κ_w denote $\log_2 \frac{w_{\max}}{w_{\min}}$, $cls(\lambda) = \lceil \frac{\kappa_w}{\log_2 \lambda} \rceil$. We have the following observation about $F_p(\lambda)$ and the competitive ratio: In summary, the following lemma states the competitive ratio for varying λ.

Lemma 5.18. *Let λ be the classification factor. Consider the case that the width of jobs is big enough.*

$$F_p(\lambda) = \begin{cases} 2 \cdot \lceil \frac{\kappa_w}{\log_2 \lambda} \rceil & \textit{if } 0 < \lambda \leq 0.5 \\ 3 \cdot \lceil \frac{\kappa_w}{\log_2 \lambda} \rceil & \textit{if } 0.5 < \lambda \leq 1 \\ (2\lambda - 1) \cdot \lceil \frac{\kappa_w}{\log_2 \lambda} \rceil & \textit{if } \lambda > 1 \end{cases}$$

Proof. If we λ is bigger, it needs to sample more timeslot in order to bound the load after round up (Lemma 5.14.) When the classification factor is between 1 and 1.5, the number of timeslots it needs to bound the load of a timeslot after round up is 2, which is the minimum number of timeslots we need to sample among all cases. However, there are more classes and it brings even bigger penalty cost. □

Lemma 5.19. *For* $0 < \lambda \leq 0.5$*,* $0.5 < \lambda \leq 1$ *and* $\lambda > 1$*, the competitive ratio of our algorithm becomes* $(12 \times 2\lceil \log_{1+\lambda} K_w \rceil)^\alpha(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)$*,* $(12 \times 3\lceil \log_{1+\lambda} K_w \rceil)^\alpha(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)$*, and* $(12 \times (2\lambda+1)\lceil \log_{1+\lambda} K_w \rceil)^\alpha(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1)$*, respectively.*

Proof. The number of classes is $\lceil \log_{1+\lambda} K_w \rceil$, which replaces $\lceil \log K_w \rceil$ in Theorem 5.17. We note that this number decreases as λ increases. In Lemma 5.14, the load of S_p^* at any time t is bounded by the load of S_p at three timeslots when $\lambda = 1$. We observe that this property stays the same for $0.5 < \lambda \leq 1$. Using a similar argument, we can show that if λ is smaller and $0 < \lambda \leq 0.5$, then the number of timeslots involved becomes smaller and equals to 2. Furthermore, when $\lambda > 1$, the number of timeslots increases and equals to $2\lambda + 1$. □

We note the competitive ratio for $\lambda < 1$ is larger than that for $\lambda = 1$, and the best competitive ratio occurs when $1 < \lambda < 2$.

Corollary 5.20. *Use classification factor* 2.156*, we can get the minimum competitive ratio* $(2.9882 \cdot 12k_w)^\alpha \cdot \Big(\min\{8(e+e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\}+1\Big)$.

5.2 Lower Bound

In this section, we show a lower bound of competitive ratio for GRID problem with unit height and arbitrary width by designing an adaptive adversary for the problem. Furthermore, we consider the cost at any time t as $\ell oad(t)^\alpha$ with arbitrary $\alpha > 1$. This lower bound is immediately a lower bound for the general case of GRID problem.

The adversary constructs a set of jobs with a low cost of offline optimal schedule but a high cost of any online algorithm $\mathcal{A}$. It generates jobs one by one and assigns release times, deadlines and widths of jobs based on the previously generated jobs. The start times of jobs scheduled by algorithm $\mathcal{A}$ will be used for the job generations later. This ensures that algorithm $\mathcal{A}$ has to put a job on top of all existing jobs and results in a high energy cost for algorithm $\mathcal{A}$. Meanwhile, the adversary will choose an appropriate feasible interval for each job such that an optimal offline algorithm can schedule the job set with low energy cost. The following is the description of the adversary.

Adversary $\mathit{\Lambda}$ and job instance $\mathcal{J}$. Given an online algorithm $\mathcal{A}$, arbitrary $\alpha > 1$ and a large number x, adversary $\mathit{\Lambda}$ outputs a set of jobs $\mathcal{J}$ with $\lfloor \alpha \rfloor + 1$ jobs. Let J_i be the ith job of $\mathcal{J}$. The adversary first computes a width for each job before running algorithm $\mathcal{A}$. It sets $w(J_{\lfloor \alpha \rfloor}) = x$, $w(J_{\lfloor \alpha \rfloor + 1}) = x - 1$, and $w(J_i) = 3 \cdot w(J_{i-1}) + 1$ for $1 \leq i \leq \lfloor \alpha \rfloor - 1$. Then adversary $\mathit{\Lambda}$ computes a release time and deadline for each job through a interaction with algorithm $\mathcal{A}$. For the first job J_1, adversary $\mathit{\Lambda}$ chooses any release time and deadline such that $d(J_1) - r(J_1) \geq 3w(J_1)$. For the ith job $J_i \in \mathcal{J}$ for $2 \leq i \leq \lfloor\alpha\rfloor + 1$, adversary $\mathit{\Lambda}$ sets $r(J_i) = st(\mathcal{A}, J_{i-1}) + 1$ and $d(J_i) = et(\mathcal{A}, J_{i-1})$. This limits algorithm $\mathcal{A}$ to fewer choices of start times for scheduling a new job. A job can only be scheduled in the execution interval of the previous job by algorithm $\mathcal{A}$. On the other hand, no two jobs have the same release time. Algorithm $\mathcal{A}$ shall schedule the jobs accordingly from J_1 to $J_{\lfloor\alpha\rfloor+1}$ and one job at a time.

Let $w_{\max}$ and $w_{\min}$ denote by the maximum and minimum width of jobs respectively, and let $\mathcal{O}$ be an optimal offline algorithm for GRID problem. We have the following results.

Lemma 5.21. *cost*$(\mathcal{O}(\mathcal{J})) \leq x \cdot 3^{\lfloor\alpha\rfloor}$ *where x is a large number.*

Proof. By the setting of adversary $\mathit{\Lambda}$, we show that $\mathcal{O}$ can schedule all jobs in $\mathcal{J}$ without overlapping, and the cost of an optimal schedule is just the summation of widths of all the jobs.

For any job $J_i \in \mathcal{J}$ and $i \geq 2$, the length of its feasible interval is $d(J_i) - r(J_i) = et(\mathcal{A}, J_{i-1}) - (st(\mathcal{A}, J_{i-1}) + 1) = w(J_{i-1}) - 1 = 3w(J_i)$. This means no matter where we schedule a job, one of the lengths of $[r(J_i), st(J_i))$ and $[et(J_i), d(J_i))$ is at least $w(J_i)$, and algorithm $\mathcal{O}$ can schedule the remaining jobs in the interval with length at least $w(J_i)$ because the summation of widths of all the remaining jobs does not exceed $w(J_i)$. The remaining jobs will not overlap to J_i. Since this argument can be applied on all the jobs, this implies that all the jobs do not overlap with each other in an optimal schedule. Thus the cost of an optimal schedule is the summation of widths of all the jobs. More precisely,

$$\begin{aligned}\text{cost}(\mathcal{O}(\mathcal{J})) &= (x-1) + x + (3x+1) + (3(3x+1)+1) + \ldots + w_{\max} \\ &\leq 2x + 2\cdot 3x + 2\cdot 9x + \ldots + 2\cdot 3^{\lfloor\alpha\rfloor - 1}x \\ &= 2x \cdot \frac{3^{\lfloor\alpha\rfloor} - 1}{2} \leq x \cdot 3^{\lfloor\alpha\rfloor} \ . \end{aligned}$$ □

Theorem 5.22. *For any deterministic online algorithm $\mathcal{A}$ for* GRID *problem with unit height and arbitrary width, adversary $\mathit{\Lambda}$ constructs an instance $\mathcal{J}$ such that*

$$\frac{\mathit{cost}(\mathcal{A}(\mathcal{J}))}{\mathit{cost}(\mathcal{O}(\mathcal{J}))} \geq \left(\frac{1}{3}\log\frac{w_{\max}}{w_{\min}}\right)^{\alpha} \ .$$

Proof. We first give a lower bound of cost$(\mathcal{A}(\mathcal{J}))$ and then give the lower bound of the ratio by combining cost$(\mathcal{A}(\mathcal{J}))$ with Lemma 5.21.

By the setting of adversary $\mathit{\Lambda}$, all the jobs scheduled by algorithm $\mathcal{A}$ overlap to each other. For ease of the computation, we only consider the timeslots contained by the execution interval of the last job $J_{\lfloor\alpha\rfloor+1}$ when we compute the cost of $\mathcal{A}$. Thus cost$(\mathcal{A}(\mathcal{J})) \geq (x-1)\cdot(\lfloor\alpha\rfloor+1)^{\alpha}$. Now we use $w_{\max}$ and $w_{\min}$ to bound $\lfloor\alpha\rfloor + 1$.

According to Lemma 5.21, we have $w_{\max} \le \mathrm{cost}(\mathcal{O}(\mathcal{J})) \le x \cdot 3^{\lfloor \alpha \rfloor}$, and thus

$$\lfloor \alpha \rfloor \ge \log_3 \frac{w_{\max}}{x} \ge \log_3 \frac{w_{\max}}{3(x-1)} = \log_3 \frac{w_{\max}}{w_{\min}} - 1 \ .$$

Note that $x \le 3(x-1)$ if $x \ge 2$. Therefore, $\mathrm{cost}(\mathcal{A}(\mathcal{J})) \ge (x-1) \cdot \log^{\alpha} \frac{w_{\max}}{w_{\min}}$. Combining with Lemma 5.21, we have the lower bound of the competitive ratio

$$\frac{\mathrm{cost}(\mathcal{A}(\mathcal{J}))}{\mathrm{cost}(\mathcal{O}(\mathcal{J}))} \ge \frac{(x-1) \cdot \log^{\alpha} \frac{w_{\max}}{w_{\min}}}{x \cdot 3^{\lfloor \alpha \rfloor}} \ge \left(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}} \right)^{\alpha}$$

as x to be large enough. □

Corollary 5.23. *For any deterministic online algorithm for* GRID *problem, the competitive ratio is at least* $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^{\alpha}$.

5.3 Special Cases

In this section we focus on uniform-height jobs of height h and consider two special cases of the width. We investigate the "density" of jobs and relating it to the online algorithm $\mathcal{AVR}$ proposed by Yao et al. [81]. We first consider jobs with uniform-height and unit-width (Section 5.3.1) and secondly consider jobs with agreeable deadlines (Section 5.3.2). Note that the results in Section 5.1.3 are applicable to unit-height jobs but we can improve the competitive ratio further.

To ease the discussion, we refine a notation we defined before. For any algorithm $\mathcal{A}$ for a job set $\mathcal{J}$ and a time interval $\mathcal{I}$, we denote by $\mathcal{A}(\mathcal{J}, \mathcal{I})$ the schedule of $\mathcal{A}$ on $\mathcal{J}$ over the time interval $\mathcal{I}$.

A framework using $\mathcal{AVR}$. We define the *density* of J, denoted by $\mathrm{den}(J)$, to be $\frac{w(J)*h(J)}{d(J)-r(J)}$. Roughly speaking, the density signifies the average load required by the job over its feasible interval. We then define the *average load* at any time t as $\mathrm{avg}(t) = \sum_{J:t \in I(J)} \mathrm{den}(J)$. In our analysis, we have to distinguish timeslots with high and low average load with respect to h. Therefore, for any $h > 0$, we define $\mathcal{I}_{>h}$ and $\mathcal{I}_{\le h}$ to be set of timeslots where the average load $\mathrm{avg}(t)$ is larger than h and at most h, respectively. Note that $\mathcal{I}_{>h}$ and $\mathcal{I}_{\le h}$ do not need to be contiguous.

The main idea is to make reference to the online algorithm $\mathcal{AVR}$ and consider two types of intervals, $\mathcal{I}_{>h}$ where the average load is higher than h and $\mathcal{I}_{\le h}$ where the average load is at most h. For the former, we show that we can base on the competitive ratio of $\mathcal{AVR}$ directly; for the latter, our load could be much higher than that of $\mathcal{AVR}$ and in such case, we compare directly to the optimal algorithm. Combining the two cases, we have Lemma 5.24, which holds for any job set. In Sections 5.3.1 and 5.3.2, we show how we can use this lemma to obtain algorithms for the special cases. Notice that the number $\lceil \frac{\mathrm{avg}(t)}{h} \rceil$ is the minimum number of jobs needed to make the load at t at least $\mathrm{avg}(t)$.

Lemma 5.24. *Suppose we have an algorithm $\mathcal{A}$ for a any job set $\mathcal{J}$ such that (i) $\ell oad(\mathcal{A},t) \le ch \cdot \lceil \frac{avg(t)}{h} \rceil$ for all $t \in \mathcal{I}_{>h}$, and (ii) $\ell oad(\mathcal{A},t) \le c'h$ for all $t \in \mathcal{I}_{\le h}$. Then we have $cost(\mathcal{A}(\mathcal{J})) \le (\frac{(4c\alpha)^\alpha}{2} + c'^\alpha) \cdot cost(\mathcal{O}(\mathcal{J}))$.*

Proof. We denote the speed of $\mathcal{AVR}$ at t as $\ell oad(\mathcal{AVR},t)$. We are going to prove that $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{>h})) \le \frac{(4c\alpha)^\alpha}{2} \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$ and $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{\le h})) \le c'^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. Hence the total cost $\text{cost}(\mathcal{A}(\mathcal{J})) \le (\frac{(4c\alpha)^\alpha}{2} + c'^\alpha) \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$ since $\mathcal{I}_{>h}$ and $\mathcal{I}_{\le h}$ are disjoint.

By assumption of $\mathcal{A}$, $\ell oad(\mathcal{A},t) \le ch \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$ for all $t \in \mathcal{I}_{>h}$. Since $\text{avg}(t) > h$, $\lceil \frac{\text{avg}(t)}{h} \rceil < \frac{\text{avg}(t)}{h} + 1 < 2 \cdot \frac{\text{avg}(t)}{h}$. Hence $\ell oad(\mathcal{A},t) \le 2c \cdot \text{avg}(t)$. The total cost of $\mathcal{A}$ over $\mathcal{I}_{>h}$, $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{>h})) = \sum_{t \in \mathcal{I}_{>h}} \ell oad(\mathcal{A},t)^\alpha < \sum_{t \in \mathcal{I}_{>h}} (2c \cdot \text{avg}(t))^\alpha$. Recall that $\ell oad(\mathcal{AVR},t) = \text{avg}(t)$ for each t. As a result $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{>h})) \le (2c)^\alpha \cdot \text{cost}(\mathcal{AVR}(\mathcal{J},\mathcal{I}_{>h})) \le (2c)^\alpha \cdot \text{cost}(\mathcal{AVR}(\mathcal{J}))$. By Corollary 5.1, $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{>h})) \le (2c)^\alpha \cdot \frac{(2\alpha)^\alpha}{2} \cdot \text{cost}(\mathcal{O}(\mathcal{J})) = \frac{(4c\alpha)^\alpha}{2} \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$.

On the other hand, although we know that $\ell oad(\mathcal{A}(\mathcal{J}),t) \le c'h$ for each $t \in \mathcal{I}_{\le h}$. It may not work if we upper bound the $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{\le h}),t)$ simply by $(c'h)^\alpha$ since the size of $\mathcal{I}_{\le h}$ can be very big. Hence we need to calculate $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{\le h}))$ more carefully. Since only jobs which are available in $\mathcal{I}_{\le h}$ can be scheduled at $t \in \mathcal{I}_{\le h}$, $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{\le h})) \le \sum_{J:I(J)\cap\mathcal{I}_{\le h}\neq\emptyset} w(J)\cdot(c'h)^\alpha \le \sum_{J\in\mathcal{J}} w(J)\cdot(c'h)^\alpha$. By convexity, $\text{cost}(\mathcal{O}(\mathcal{J})) \ge \sum_{J\in\mathcal{J}} w(J)\cdot h(J)^\alpha = \sum_{J\in\mathcal{J}} w(J)\cdot h^\alpha$. Hence $\text{cost}(\mathcal{A}(\mathcal{J},\mathcal{I}_{\le h})) \le c'^\alpha \cdot \text{cost}(\mathcal{O}(\mathcal{J}))$. □

Since we have Lemma 5.24 as a framework, in the Section 5.3.1 and 5.3.2 we elaborate two online algorithms for special case and upper bound their competitive ratio using the framework.

5.3.1 Unit-width and uniform-height

In this section we consider job sets where all jobs have unit width and uniform height, i.e., $w(J) = 1$ and $h(J) = h$ for all $J \in \mathcal{J}$. Note that such case is a sub-case discussed in Section 5.1.1. Here we illustrate a different approach using the ideas above and describe the algorithm $\mathcal{UU}$ for this case. The competitive ratio of $\mathcal{UU}$ is better than that of Algorithm $\mathcal{V}$ in Section 5.1.1 when $\alpha < 3.2198$.

Algorithm $\mathcal{UU}$. At any time t, choose $\lceil \frac{\text{avg}(t)}{h} \rceil$ jobs according to the EDF rule and schedule them to start at t. If there are fewer jobs available, schedule all available jobs.

The feasibility of $\mathcal{UU}$ can be proved by comparing to $\mathcal{AVR}$:

Lemma 5.25. *The schedule construct by Algorithm $\mathcal{UU}$ is feasible for $\mathcal{J}$.*

Proof. We prove the feasibility of $\mathcal{UU}$ by comparing to $\mathcal{AVR}$. Simply speaking, by each timeslot t, we show that the amount of work done by $\mathcal{UU}$ is at least the one done by $\mathcal{AVR}$. Hence $\mathcal{UU}$ is feasible since $\mathcal{AVR}$ is feasible.

At any time t, the total work done by $\mathcal{AVR}$ in interval $[0,t)$ is $\sum_{t'<t} \text{avg}(t')$. On the other hand, the total work done by $\mathcal{UU}$ in the same interval is $\sum_{t'<t} h \cdot \lceil \frac{\text{avg}(t)}{h} \rceil \ge$

$\sum_{t'<t} h \cdot \frac{\text{avg}(t)}{h} = \sum_{t'<t} \text{avg}(t)$ if there are enough available jobs in this interval. If the number of available jobs is less than $\sum_{t'<t} h \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$, $\mathcal{UU}$ will execute all these jobs released by t. The work done by $\mathcal{UU}$ within interval $[0, t)$ is at least the work done by $\mathcal{AVR}$ in both cases. Hence, $\mathcal{UU}$ is feasible since $\mathcal{AVR}$ is feasible. □

The next theorem states the competitive ratio of $\mathcal{UU}$ by using Lemma 5.24.

Theorem 5.26. *Algorithm $\mathcal{UU}$ is $(\frac{(4\alpha)^\alpha}{2} + 1)$-competitive.*

Proof. We note that for each $t \in \mathcal{I}_{>h}$, $\ell oad(\mathcal{UU}, t) \leq h \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$. And for each $t \in \mathcal{I}_{\leq h}$, $\ell oad(\mathcal{UU}, t) \leq h$ since there are at most one job assigned at t. We set $c' = 1$ for $t \in \mathcal{I}_{\leq h}$ and set $c = 1$ for $t \in \mathcal{I}_{>h}$. By Lemma 5.24 competitive ratio of $\mathcal{UU}$ is $\frac{(4 \cdot 1 \cdot \alpha)^\alpha}{2} + 1^\alpha = \frac{(4\alpha)^\alpha}{2} + 1$. □

Remark. Note that by Theorem 5.7, with referencing to $\mathcal{AVR}$ or $\mathcal{BKP}$, $\mathcal{V}$ is $(\frac{(4\alpha)^\alpha}{2} + 2^\alpha)$-competitive or $2^\alpha(8e^\alpha + 1)$-competitive, respectively, for the unit input. Hence $\mathcal{UU}$ has better performance for unit input when $\alpha < 3.2198$.

5.3.2 Uniform-height, arbitrary widths and agreeable deadlines

In this section we consider jobs with arbitrary width, uniform height h and agreeable deadlines. That is, in the input sequence of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$, $h(J_i) = h$ for all i, and $r(J_i) \leq r(J_j)$ and $d(J_i) \leq d(J_j)$ for all $i < j$. We first note that simply scheduling $\lceil \frac{\text{avg}(t)}{h} \rceil$ number of jobs may not return a feasible schedule even for jobs with common feasible interval.

Example 5.1. *Consider four jobs, each job J_j with $r(J_j) = 0$, $d(J_j) = 5$, $h(J_j) = h$, $w(J_j) = 3$. Note that avg$(t) = 2.4 \cdot h$ for all t. If we schedule at most $\lceil \frac{avg(t)}{h} \rceil = 3$ jobs at any time, we can complete three jobs but the remaining job cannot be completed. To schedule all jobs feasibly, we need at least two timeslots where all jobs are being executed.*

The reason why scheduling $\lceil \frac{\text{avg}(t)}{h} \rceil$ number of jobs does not work is that in the GRID problem, jobs cannot be preempted. Hence even when the total amount of work which $\mathcal{AVR}$ is capable to do in the rest of timeslots (that is, time interval $[4, 5)$) is more than the total work which has not been scheduled in the GRID problem (that is, one remaining job with total work $3h$), the remaining jobs in the GRID problem cannot be feasibly scheduled. Note that Algorithm $\mathcal{UU}$ escapes from this problem since when all jobs are unit size, there is no need to preempt.

To schedule jobs with arbitrary width, uniform height h and agreeable deadlines, we first observe in Lemma 5.27 that for a set of jobs with total density at most h, it is feasible to schedule them such that the load at any time is at most h. Roughly speaking, we consider jobs in the order of releasing, and hence in EDF manner since the jobs have agreeable deadlines. We keep the current ending time of all jobs that have been considered. As a new job is released, if its release time is earlier than the current ending time, we set its start time to the current ending time (and increase the current ending time by the width of the new job); otherwise, we set its start time to be its release time. Lemma 5.27 asserts that such scheduling is feasible and maintains the load at any time to be at most h.

Using this observation, we then partition the jobs into "queues" each of which has sum of densities at most h. Each queue $\mathcal{Q}_i$ is scheduled independently and the resulting schedule is to "stack up" all these schedules. The queues are formed in a Next-Fit manner: (i) the current queue $\mathcal{Q}_q$ is kept "open" and a newly arrived job is added to the current queue if including it makes the total densities stays at most h; (ii) otherwise, the current queue is "closed" and a new queue $\mathcal{Q}_{q+1}$ is created as open.

Lemma 5.27. *Given any set of jobs of uniform-height h, arbitrary-width and agreeable deadlines. If the sum of densities of all these jobs is at most h, then it is feasible to schedule all of them using a maximum load h at any time. That is, there is no stacking up among these jobs.*

Proof. Suppose there are k jobs $J_1, J_2, \cdots, J_k$ such that $\sum_{i=1}^{k} \mathrm{den}(J_i) \leq h$. Without loss of generality, we assume that $d(J_i) \leq d(J_j)$ and $r(J_i) \leq r(J_j)$ for $1 \leq i < j \leq k$. We claim that by EDF principle, each job J_i can be finished by its deadline $d(J_i)$. More formally, we show that it is feasible to set $st(J_i)$ to $\max\{r(J_i), et(J_{i-1})\}$ and $et(J_i)$ to $st(J_i)+w(J_i)$ for all i (note that J_1 has the earliest release time and $st(J_1)$ is set to $r(J_i)$). By the algorithm, the execution intervals of jobs have no overlapping. Also, each job has uniform height h. Hence the load in the resulting schedule at any timeslot t is no more than h.

We prove that every job can be finished without preemption before its deadline by induction. We assume there is no gap between the jobs execution. That is, there is no idle time between the execution intervals of jobs. First we observe that $\mathrm{den}(J_1) \leq h$ since $\sum_{i=1}^{k} \mathrm{den}(J_i) \leq h$. It is feasible to set $[st(J_1), et(J_1))$ to $[r(J_1), r(J_1)+w(J_1))$ since the input is feasible. Next, we have to prove that J_i can be finished before $d(J_i)$ given all jobs $J_{i'}$ can be finished before $d(J_{i'})$ without preemption for all $i' < i$. There are two cases, $r(J_i) \geq et(J_{i-1})$ or $r(J_i) < et(J_{i-1})$. If $r(J_i) \geq et(J_{i-1})$, it is easy to see that J_i can be finished by $d(J_i)$ (by setting $st(J_i)$ to $r(J_i)$) since the input is feasible. On the other hand, if $r(J_i) < et(J_{i-1})$, since $\sum_{j=1}^{i} \mathrm{den}(J_j) \leq \sum_{j=1}^{k} \mathrm{den}(J_j)$, we know $\sum_{j=1}^{i} \frac{w(J_j)\cdot h}{d(J_j)-r(J_j)} \leq h$. Because the jobs have agreeable deadline, $d(J_i) \geq d(J_{i'})$ for all $i' \leq i$ and $r(J_1) \leq r(J_{i'})$ for all $i' \geq 1$. Therefore, $h \geq \sum_{j=1}^{i} \frac{w(J_j)\cdot h}{d(J_j)-r(J_j)} \geq \sum_{j=1}^{i} \frac{w(J_j)\cdot h}{d(J_i)-r(J_1)}$. That is, $\sum_{j=1}^{i} w(J_j) \leq d(J_i) - r(J_1)$. Hence $J_1, J_2, \cdots, J_i$ can be finished by $d(J_i)$. Moreover, since the jobs have agreeable deadline, there is no need to preempt jobs. □

We have shown that for uniform-height jobs with agreeable deadlines, there is a way to schedule a subset of jobs with a special property (that is, total density is bounded by h) feasibly and without overlapping. Now we want to introduce an online scheduling algorithm for the whole input set. The basic idea is to partition jobs into subset where the summation of jobs' densities is at most h in a next-fit manner (**InsertQueue**.) In each subset, jobs are scheduled independently from jobs in other subsets (**SetStartTime** and **ScheduleQueue**.)

Algorithm $\mathcal{AD}$. The algorithm consists of the following components: InsertQueue, SetStartTime and ScheduleQueue.
InsertQueue: We keep a counter q for the number of queues created and $\mathcal{Q}_q$ is the current queue. When a job J arrives, if $\text{den}(J) + \sum_{J' \in \mathcal{Q}_q} \text{den}(J') \leq h$, then job J_j is added to $\mathcal{Q}_q$; otherwise, job J is added to a new queue $\mathcal{Q}_{q+1}$ and we set $q \leftarrow q+1$.
SetStartTime: For the current queue, we keep a current ending time E, initially set to 0. When a new job J is added to the queue, if $r(J) \leq E$, we set $st(J) \leftarrow E$; otherwise, we set $st(J) \leftarrow r(J)$. We then update E to $st(J) + w(J)$.
ScheduleQueue: At any time t, schedule all jobs in all queues with start time set at t.

Note that in the InsertQueue procedure, a queue is no longer under consideration if it is not capable to accommodate the currently arrived job. In other words, we pack jobs into the queues in a next-fix manner. In the SetStartTime procedure, the scheduling strategy is the same as the one we used in proof of Lemma 5.27. Hence by Lemma 5.27, the schedule returns by $\mathcal{AD}$ is feasible.

We then analyze the load of $\mathcal{AD}$ and hence derive the competitive ratio. Recall that $\mathcal{I}_{>h}$ and $\mathcal{I}_{\leq h}$ are set of timeslots where the average load $\text{avg}(t)$ is larger than h and at most h, respectively.

Lemma 5.28. *Using $\mathcal{AD}$, we have (i) $\ell oad(\mathcal{AD}, t) \leq 2 \cdot h \cdot \lceil \frac{avg(t)}{h} \rceil$ for $t \in \mathcal{I}_{>h}$; (ii) $\ell oad(\mathcal{AD}, t) \leq 2h$ for $t \in \mathcal{I}_{\leq h}$.*

Proof. For any timeslot t, suppose there are k queues $(\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_k)$ which contains jobs available at t. By the algorithm, $\ell oad(\mathcal{UU}, t) \leq k \cdot h$. We first observe that because of the next-fix strategy, each queue $\mathcal{Q}_q$ contains a contiguous subsequence of input jobs $J_i, J_{i+1}, J_{i+2}, \cdots$. Let D_i denote the summation of densities of jobs which is available at t and is assigned to $\mathcal{Q}_i$. We prove (ii) first since it is simpler.

(ii) Consider $t \in \mathcal{I}_{\leq h}$. By definition, $\text{avg}(t) \leq h$. That is, the sum of densities of all available jobs at t is no more than h. By the InsertQueue procedure all jobs will be in at most two consecutive queues. Note that although all the jobs can be put into a single queue, it could be the case that some of the jobs are put into the previous queue and hence the jobs available at t are placed into two queues. Therefore, $\ell oad(\mathcal{AD}, t) \leq 2h$ for $t \in \mathcal{I}_{\leq h}$.

(i) If $t \in \mathcal{I}_{>h}$, according to our algorithm, $\ell oad(\mathcal{AD}, t) \leq k \cdot h$. We want to show that if k is big at t, $\text{avg}(t)$ is also big.

Consider the queues $\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_k$, since the jobs have agreeable deadlines and we use next-fit strategy for assigning jobs into queues, we first observe that for any job $J_{j(q)}$ which is the first job assigned to queue $\mathcal{Q}_q$, $\text{den}(J_{j(q)}) + D_{q-1} > h$. Otherwise, the job $J_{j(q)}$ would be assigned to the queue D_{q-1}. Note that it may not be true for $\text{den}(J_{j(2)}) + D_1$ because some of the jobs in $\mathcal{Q}_1$ may not be available at t. That is, D_1 may be less than $\sum_{J \in \mathcal{Q}_1} \text{den}(J)$. Therefore, if $k \geq 3$, $\text{avg}(t) = \sum_{i=1}^{k} D_i > \sum_{i=1}^{\lfloor \frac{k-1}{2} \rfloor} (D_{2i} + J_{j(2i+1)}) > \lfloor \frac{k-1}{2} \rfloor \cdot h$. It can be shown that $k \leq 2 \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$ since $\text{avg}(t) > 1$. Let $r = \frac{\text{avg}(t)}{h}$, since $t \in \mathcal{I}_{>h}$, $r > 1$. By deduction, $\lfloor \frac{k-1}{2} \rfloor < r \leq \lceil r \rceil$. We investigate the relation between k and r. Since $\lfloor \frac{k-1}{2} \rfloor$ is a natural number, by calculation, $\frac{k-1}{2} - 1 < \lfloor \frac{k-1}{2} \rfloor \leq \lceil r \rceil - 1$. Also, k is a natural number, so $k \leq 2\lceil r \rceil$. Therefore, $\ell oad(\mathcal{AD}, t) \leq k \cdot h < 2\lceil r \rceil \cdot h = 2 \cdot h \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$ for $t \in \mathcal{I}_{>h}$.

On the other hand, if $k = 2$, $\ell oad(\mathcal{UU}, t) \leq 2h$. Since $\text{avg}(t) > h$ (by definition of $\mathcal{I}_{>h}$), $\ell oad(\mathcal{UU}, t) < 2 \cdot \text{avg}(t) \leq 2 \cdot h \cdot \lceil \frac{\text{avg}(t)}{h} \rceil$. □

By Lemma 5.28 and Lemma 5.24, we have the competitive ratio of $\mathcal{AD}$ by setting $c = 2$ and $c' = 1$ in the following Theorem.

Theorem 5.29. *For jobs with uniform height, arbitrary width and agreeable deadlines, $\mathcal{AD}$ is $(\frac{(8\alpha)^\alpha}{2} + 2^\alpha)$-competitive.*

Remark. In Section 5.1.3, we showed that for general input, there is an online algorithm $\mathcal{A}$ to schedule the jobs with competitive ratio $36^\alpha \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)^\alpha \cdot (\min\{8(e + e^2)^\alpha, \frac{(2\alpha)^\alpha}{2}\} + 1)$. However, if we know the jobs released later would have later (or at least the same) deadlines, we can improve the competitive ratio. By generalizing algorithm $\mathcal{AD}$ similarly to the method introduced in Section 4.4, the competitive ratio for scheduling jobs with arbitrary widths, arbitrary heights, and agreeable deadlines is $2^\alpha \cdot (1 + \lceil \log \frac{h_{\max}}{h_{\min}} \rceil)^\alpha \cdot (\frac{(8\alpha)^\alpha}{2} + 2^\alpha)$.

5.3.3 Uniform height job set with common feasible intervals

In this section we consider jobs with arbitrary width, uniform height h and common feasible interval. That is, in the input sequence of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$, $h(J_i) = h$, $r(J_i) = 0$ and $d(J_i) = D$ for all i. Recall that this setting is NP-hard. However, we prove that a simple greedy algorithm is $2^{2\alpha}$-competitive, and hence $2^{2\alpha}$-approximate.

We introduce an greedy algorithm $\mathcal{C}_{uh}$ which schedule the jobs such that the increasing of current total cost is minimized. In other words, assume the highest load of the current schedule is ℓ, a released job J will be assigned to level $\ell + 1$ if and only if each level from 1 to ℓ cannot accommodate it. We consider the greedy strategy as a bin packing one. That is, each bin has capacity D and once a job J arrives, it is assigned to the first bin which has remaining capacity at least $w(J)$.

Algorithm $\mathcal{C}_{uh}$. The algorithm consists of the following components: InsertQueue, SetStartTime and ScheduleQueue.
InsertQueue: We keep a sequence of queues $\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \cdots$. When a job J arrives, it is assigned to $\mathcal{Q}_q$ with minimum q such that $\text{den}(J) + \sum_{J' \in \mathcal{Q}_q} \text{den}(J') \leq D$; otherwise, job J is added to a new queue.

SetStartTime: For each queue, we keep a current ending time E, initially set to 0. When a new job J is added to the queue, if $r(J) \leq E$, we set $st(J) \leftarrow E$; otherwise, we set $st(J) \leftarrow r(J)$. We then update E to $st(J) + w(J)$.
ScheduleQueue: At any time t, schedule all jobs in all queues with start time set at t.

It is clear to see that the algorithm $\mathcal{C}_{uh}$ is exactly the First-Fit strategy of BINPACKING. It has been proven that First-Fit is 2-competitive for BINPACKING. Therefore, for the GRID problem where the object is to minimize maximum power request, $\mathcal{C}_{uh}$ is 2-competitive. We prove that $\mathcal{C}_{uh}$ is $2^{2\alpha}$-competitive for the GRID problem objected to minimize the total cost by bounded the total work and hence the lower bound of any optimal schedule.

Theorem 5.30. *The algorithm $\mathcal{C}_{uh}$ is $2^{2\alpha}$-competitive for Grid problem for uniform height job set with same release time and same deadline.*

Proof. Let $\mathcal{O}_G$ and $\mathcal{O}_B$ denote the optimal schedules of the GRID and BINPACKING problem, respectively. Assume in the schedule generated by $\mathcal{C}_{uh}$, the highest load among timeslot $1, 2, \cdots, D$ is k. By the analysis of First-Fit strategy on bin-packing, $\mathcal{O}_B$ uses at least $\frac{k}{2}$ bins. Hence, the total work of all jobs is at least $\frac{k}{2} \cdot \frac{D}{2}$ (since the load of each bin in $\mathcal{O}_B$ might be less than D but must over $\frac{D}{2}$ or $\mathcal{O}_B$ is not optimal.) The best the optimal schedule in the smart grid problem $\mathcal{O}_G$ can do is to schedule the total load evenly on each timeslot. Hence, $\text{cost}(\mathcal{O}) \geq (\frac{\frac{kD}{4}}{D})^\alpha \cdot D = (\frac{k}{4})^\alpha \cdot D$. On the other hand, $\text{cost}(\mathcal{C}_{uh}) \leq k^\alpha \cdot D$. Therefore, the competitive ratio of $\mathcal{C}_{uh}$ is at most 4^α. □

5.3.4 Unit width job set with common feasible intervals

In this section we consider jobs with unit width, arbitrary height and common feasible interval. That is, in the input sequence of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$, $w(J_i) = w$, $r(J_i) = 0$ and $d(J_i) = D$ for all i. Recall that this setting is NP-hard (see Section 4.1. However, we prove that a simple greedy algorithm is 2^α-competitive, and hence 2^α-approximate.

We introduce an greedy algorithm $\mathcal{C}_{uw}$ which schedule the jobs such that the increasing of current total cost is minimized. In other words, a released job J will be assigned to the timeslot with lowest load. We consider this problem as a load balancing problem with D uniform machines. That is, each timeslot is considered as a machine and each job is considered as a task which has to be assigned to one machine (timeslot.)

Algorithm $\mathcal{C}_{uw}$. Consider timeslots $1, 2, 3, \cdots, D$. When a job J arrives, it is assigned to t with minimum load.

It has been proven that the greedy strategy is 2-competitive for uniform machines load balancing problem. Therefore, for the GRID problem where the object is to minimize maximum power request, $\mathcal{C}_{uw}$ is 2-competitive. Similar to Section 5.3.3, we prove that $\mathcal{C}_{uw}$ is 2^α-competitive for the GRID problem objected to minimize the total cost by bounded the total work and hence the lower bound of any optimal schedule.

We first observe the $\mathcal{C}_{uw}$ schedule and have the following bounds. For each timeslot t, let h_t be the height of the last job which is assigned to t in the $\mathcal{C}_{uw}$ schedule, and $S_t = \ell oad(\mathcal{C}_{uw}, t) - h_t$. Note that $h_t > 0$ and $S_t \geq 0$ since there are at least one job at t.

Lemma 5.31. *In the $\mathcal{C}_{uw}$ schedule, if there exists at least one timeslot with more than one job assigned to it, assume the last job J_n in $\mathcal{J}$ is assigned to timeslot x:*

(i) there must be at least two jobs at x;

(ii) $S_t \leq S_x$ for all $t \neq x$;

(iii) cost$(\mathcal{O}) \geq \sum_{t=1}^{D} h_t^\alpha$; and

(iv) cost$(\mathcal{O}) \geq D \cdot S_x^\alpha$.

Proof. First we observe that there is at least one job at each timeslot, or the last job assigned at the timeslot which has at least jobs would be assigned to the empty timeslot. Hence, $h_t > 0$ for all t. Let $\ell oad_j(t)$ denote the load of $\mathcal{C}_{uw}$ at timeslot t when the first j jobs are released and assigned.

(i) Assume by contrary that if there is only one job assigned at x. By the definition of tiemslot x, the job assigned at x is J_n. We let timeslot t' denote the timeslot with more than one jobs and J_i be the last job which is assigned to t'. Consider the load at t' when J_i arrived, that is, $\ell oad_{i=1}(t')$, it is clear that $\ell oad_{i-1}(t') > 0$ since there are at least two jobs would be assigned at t'. On the other hand, $\ell oad_{i-1}(x) \leq \ell oad(\mathcal{C}_{uw}, x) = 0$, hence $\ell oad_{i-1}(x) = 0 < \ell oad_{i-1}(t')$. It contradicts to how the algorithm works. Hence there must be at least two jobs assigned at x.

(ii) Consider the last job J_i which is assigned at timeslot t where $t \neq x$. Recall that J_n, the last job in $\mathcal{J}$, is assigned at timeslot x, hence $i < n$. By definition, $\ell oad_{i-1}(t) = S_t$. When J_i is released, the load at t, $S_t = \ell oad_{i-1}(t) \leq \ell oad_{i-1}(x)$, or J_i would be assigned at x. Also, since at least one of the jobs which would be assigned at x has not arrived (that is, J_n), $S_x \geq \ell oad_{i-1}(x)$. Therefore, $S_t \leq S_x$.

(iii) Note that the last jobs at each timeslot form a subset of the whole input set $\mathcal{J}$. By the convexity of the cost function, cost$(\mathcal{O}) = \sum_{j=1}^{n} h(J_j)^\alpha \geq \sum_{t=1}^{D} h_t^\alpha$. (iv) By (i), there must be at least two jobs assigned at x, hence $S_x > 0$. Consider the last job assigned to any timeslot $t \neq x$, it is assigned to t because by then, $\ell oad(\mathcal{C}_{uw}, t) = S_t \leq S_x$ or the job would be assigned at x (recall that timeslot x has the last job in the input set.) Furthermore, the total work of jobs is at least $(D-1)S_x + \ell oad(\mathcal{C}_{uw}, x) \geq (D-1)S_x + S_x \geq D \cdot S_x$. Therefore, cost$(\mathcal{O}) \geq D \cdot (\frac{DS_x}{D})^\alpha = D \cdot S_x^\alpha$. □

Now we are ready to analysis the competitive ratio of $\mathcal{C}_{uw}$.

Theorem 5.32. *The algorithm $\mathcal{C}_{uw}$ is 2^α-competitive for* GRID *problem for unit width job set with same release time and same deadline.*

Proof. First we observe that if there for each timeslot there is at most one job assigned at it, by the convexity of the cost function, cost$(\mathcal{C}_{uw})$ = cost$(\mathcal{O})$. Similarly, if there is exactly one job at each timeslot, cost$(\mathcal{C}_{uw})$ = cost$(\mathcal{O})$.

On the other hand, we consider the case where there exist at least one timeslot has more than one jobs. By definition, $\text{cost}(\mathcal{C}_{uw}) = \sum_{t=1}^{D} \ell oad(\mathcal{C}_{uw}, t)^{\alpha} = \sum_{t=1}^{D}(h_t + S_t)^{\alpha} \leq 2^{\alpha-1}(\sum_{t=1}^{D} h_t^{\alpha} + \sum_{t=1}^{D} S_t^{\alpha})$. By Lemma 5.31 (ii) and (iv), $\sum_{t=1}^{D} S_t^{\alpha} \leq D \cdot S_x^{\alpha} \leq \text{cost}(\mathcal{O})$. Also, by Lemma 5.31 (iii), $\sum_{t=1}^{D} h_t^{\alpha} \leq \text{cost}(\mathcal{O})$. Therefore $\text{cost}(\mathcal{C}_{uw}) \leq 2^{\alpha-1}(\text{cost}(\mathcal{O}) + \text{cost}(\mathcal{O})) = 2^{\alpha} \cdot \text{cost}(\mathcal{O})$.

□

Corollary 5.33. *For unit case with same release time and same deadline,* $\mathcal{C}_{uw} = \mathcal{O}$.

5.4 Summary

In this chapter we investigated the GRID problem in the online model. We proposed a $(36(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil))^{\alpha} \cdot (\min\{8(e + e^2)^{\alpha}, \frac{(2\alpha)^{\alpha}}{2}\} + 1)$-competitive online algorithm for the jobs with arbitrary widths, arbitrary heights, and arbitrary feasible intervals. We also prove that for any deterministic online algorithm, the competitive ratio is at least $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^{\alpha}$.

For special cases, we showed that there are better strategies. There is a $(\frac{(4\alpha)^{\alpha}}{2} + 1)$-competitive algorithm for the unit case by relating to the $\mathcal{AVR}$ algorithm for the DVS problem. For uniform height, arbitrary widths and agreeable deadlines jobs, we showed that a Next-Fit algorithm is $(\frac{(8\alpha)^{\alpha}}{2} + 1)$-competitive. For jobs with common intervals, we showed that the First-Fit algorithm is $2^{2\alpha}$-competitive for unit-height jobs, and the Best-Fit algorithm is 2^{α}-competitive for unit-width jobs.

Chapter 6

Extensions to Other Problems

In this chapter we extend our algorithms to solve other problems different from GRID. We find that the online algorithm introduced in Section 5.1 also performs well on other problems. We also find that the exact algorithm introduced in Section 4.3 can solve other demand response management problems with different objective functions, and hence we can prove these problems are all fixed-parameter tractable.

First we turn to the online smart grid scheduling problem, $\mathsf{GRID}_{\mathsf{peak}}$, objective to minimize the maximum power request over time. The $\mathsf{GRID}_{\mathsf{peak}}$ problem is quite similar to the classical Machine Minimization problem. In fact, the Machine Minimization problem is a special case of the smart grid scheduling problem objective to minimize the peak power request over time where jobs have unit height. Saha [71] has proven that no deterministic online algorithm for Machine Minimization can achieve an approximation factor better than $\log_3 \frac{w_{\max}}{w_{\min}}$. It also gives us a lower bound that no deterministic online algorithm for the $\mathsf{GRID}_{\mathsf{peak}}$ problem can achieve a competitive ratio better than $\log_3 \frac{w_{\max}}{w_{\min}}$.

In Section 6.1, we show that the online algorithm presented in Section 5.1 is asymptotically optimal for the $\mathsf{GRID}_{\mathsf{peak}}$ problem and the Machine Minimization problem. In Section 6.2, we show how to adapt the exact algorithm presented in Section 4.3 to solve the $\mathsf{GRID}_{\mathsf{peak}}$ problem. We further give an idea about solving the GRID problem with power limit, the $\mathsf{GRID}_{\mathsf{limited}}$ problem. That is, given a power limit L, is there a feasible schedule such that for each timeslot, the load cannot exceed L?

6.1 Online algorithm: Minimizing the peak power request

In this section, we investigate the online smart grid scheduling problem objective to the maximum power request over time horizon instead of the total cost. We prove that the online algorithm $\mathcal{A}$ proposed in Section 5.1 is also asymptotically optimal for the $\mathsf{GRID}_{\mathsf{peak}}$ problem objective to maximum power request.

The $\mathsf{GRID}_{\mathsf{peak}}$ problem. The input of the $\mathsf{GRID}_{\mathsf{peak}}$ problem is a set of jobs where each job has a power request, time duration, and a feasible intervals in which the job can be scheduled. The goal is to feasibly serve all jobs without preemption such that the maximum power request over time horizon is minimized.

The $\mathsf{GRID}_{\mathsf{peak}}$ problem has been proven to be NP-hard by Tang et al. [76]. Yaw et al. [83] proposed a 4-approximation algorithm for input jobs with common feasible interval and an $O(\log \frac{w_{\max}}{w_{\min}})$-approximation algorithm for jobs with agreeable deadlines.

In this section, we consider the $\mathsf{GRID}_{\mathsf{peak}}$ problem in the online model. Saha [71] proved that any online algorithm for the non-preemptive Machine Minimization problem has competitive ratio at least $\log_3 \frac{w_{\max}}{w_{\min}}$. Since the Machine Minimization problem is a special case of the $\mathsf{GRID}_{\mathsf{peak}}$ problem where jobs have unit height, the lower bound also holds for the $\mathsf{GRID}_{\mathsf{peak}}$ problem. We prove that the online algorithm $\mathcal{A}$ proposed in Section 5.1 is asymptotically optimal for the $\mathsf{GRID}_{\mathsf{peak}}$ problem. More specifically, we use the online strategy to solve the $\mathsf{GRID}_{\mathsf{peak}}$ problem by using the adapted $\mathcal{BKP}$ algorithm, $\mathcal{BKP}'$ (see Section 5.1.1), as the reference algorithm. Bansal et al. [6] proved the following lemma:

Lemma 6.1 ([**6**])**.** *The $\mathcal{BKP}$ algorithm is e-competitive with respect to maximum speed.*

According to Lemma 5.3, we have the following lemma:

Lemma 6.2. *The $\mathcal{BKP}'$ algorithm is $e(1+e)$-competitive with respect to maximum speed.*

Also, the $\mathcal{YDS}$ algorithm guarantees that maximum speed is minimized [6]. Let function $\text{peak}(S)$ denote the maximum power request of schedule S. More formally, $\text{peak}(S) = \max_t \ell oad(S,t)$. Similar to Observation 11, $\mathcal{YDS}$ gives a lower bound for the $\mathsf{GRID}_{\mathsf{peak}}$ problem.

Observation 16. Let $\mathcal{O}_D$ and $\mathcal{O}_G$ be the optimal schedule for the DVS and $\mathsf{GRID}_{\mathsf{peak}}$ problem, respectively. Given a job set $\mathcal{J}_G$ for the $\mathsf{GRID}_{\mathsf{peak}}$ problem and convert it into a job set $\mathcal{J}_D$ for the DVS problem (see Chapter 5), $\text{peak}(\mathcal{O}_D(\mathcal{J}_D)) \leq \text{peak}(\mathcal{O}_G(\mathcal{J}_G))$.

Analysis of the online algorithm $\mathcal{A}$. Now we analyze the online algorithm $\mathcal{A}$ and show that it is $O(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)$-competitive with respect to the maximum power request.

We first introduce an observation which is widely used throughout the analysis:

Observation 17. For two schedule S_1 and S_2, if the following relation between the loads holds $\ell oad(S_1,t) \leq \sum_{i=1}^{k} \ell oad(S_2, c_i(t))$, where $c_i(t)$ is a function of t, then $\text{peak}(S_1) \leq k \cdot \text{peak}(S_2)$.

Proof. By definition, $\text{peak}(S_1) = \max_t \ell oad(S_1,t)$. By the given condition, $\text{peak}(S_1) \leq \max_t \sum_{i=1}^{k} \ell oad(S_2, c_i(t)) \leq \sum_{i=1}^{k} \max_t \ell oad(S_2, c_i(t)) = \sum_{i=1}^{k} \text{peak}(S_2) = k \cdot \text{peak}(S_2)$. □

Recall that in Section 5.1, we solve the general case by generalizing the online algorithm for special input. The algorithm $\mathcal{V}$ can schedule unit-width jobs with promising performance by referencing to an arbitrary feasible online algorithm for the DVS problem. By generalizing $\mathcal{V}$, we have online algorithm $\mathcal{UV}$ for jobs with uniform width. By further generalizing $\mathcal{UV}$, we have online algorithm $\mathcal{A}$ for jobs with arbitrary widths and arbitrary heights.

We first prove that the algorithm $\mathcal{V}$ for unit-width jobs schedules the jobs with good performance by referencing to $\mathcal{BKP}'$. Recall that for each timeslot t, $\mathcal{V}$ schedules jobs to start at t such that $\ell oad(\mathcal{V}, t)$ is at least $\ell oad(\mathcal{BKP}', t) = (1 + e) \cdot \ell oad(\mathcal{BKP}, t)$ or until all available jobs have been scheduled. We prove that although $\ell oad(\mathcal{V}, t)$ might be higher than $\ell oad(\mathcal{BKP}', t)$, the peak of $\mathcal{V}$ is no more than $2e$ times of the peak of the optimal.

Let $h_{\max}(\mathcal{V}, t)$ be the maximum height of jobs scheduled at t by $\mathcal{V}$; we set $h_{\max}(\mathcal{V}, t) = 0$ if $\mathcal{V}$ assigns no job at t. We classify each timeslot t into two types: (i) $h_{\max}(\mathcal{V}, t) < \ell oad(\mathcal{BKP}', t)$, and (ii) $h_{\max}(\mathcal{V}, t) \geq \ell oad(\mathcal{BKP}', t)$. We denote by $\mathcal{I}_1$ and $\mathcal{I}_2$ the union of all timeslots of Type (i) and (ii), respectively. Notice that $\mathcal{I}_1$ and $\mathcal{I}_2$ can be empty and the union of $\mathcal{I}_1$ and $\mathcal{I}_2$ covers the entire time line. In Lemma 6.3 bounds the cost of $\mathcal{V}$ in each type of timeslots, and Theorem 6.4 asserts that $\mathcal{V}$ is $2(e + e^2)$-competitive.

Lemma 6.3. *For any job set $\mathcal{J}$ where for each job $J \in \mathcal{J}$, $w(J) = 1$, (i) peak$(\mathcal{V}(\mathcal{J}), \mathcal{I}_1) \leq 2(e + e^2) \cdot$ peak$(\mathcal{O}(\mathcal{J}))$; and (ii) peak$(\mathcal{V}(\mathcal{J}), \mathcal{I}_2) \leq 2 \cdot$ peak$(\mathcal{O}(\mathcal{J}))$.*

Proof. (i) For every timeslot $t \in \mathcal{I}_1$, $\ell oad(\mathcal{V}, t) < \ell oad(\mathcal{BKP}', t) + h_{\max}(\mathcal{V}, t) \leq 2 \cdot \ell oad(\mathcal{BKP}', t) \leq 2(1+e) \cdot \ell oad(\mathcal{BKP}, t)$. By definition, $\text{peak}(\mathcal{V}, \mathcal{I}_1) = \max_{t \in \mathcal{I}_1} \ell oad(\mathcal{V}, t) < 2(1 + e) \cdot \ell oad(\mathcal{BKP}, t)$. Therefore, $\text{peak}(\mathcal{V}, \mathcal{I}_1) \leq 2e(1 + e) \cdot \text{peak}(\mathcal{O})$ by Observation 16 and Lemma 6.1.

(ii) For every timeslot $t \in \mathcal{I}_2$, $\ell oad(\mathcal{V}, t) < \ell oad(\mathcal{BKP}', t) + h_{\max}(\mathcal{V}, t) \leq 2 \cdot h_{\max}(\mathcal{V}, t)$. In the optimal schedule, the job with height $h_{\max}(\mathcal{V}, t)$ has to be scheduled somewhere or the schedule is not feasible, so $\text{peak}(\mathcal{O}) \geq h_{\max}(\mathcal{V}, t)$. Hence, $\text{peak}(\mathcal{V}, \mathcal{I}_2) = \max_{t \in \mathcal{I}_2} \ell oad(\mathcal{V}, t) \leq 2h_{\max}(\mathcal{V}, t) \leq 2 \cdot \text{peak}(\mathcal{O})$. $\square$

Theorem 6.4. *For any job set $\mathcal{J}$ where for each job has unit width, peak$(\mathcal{V}(\mathcal{J})) \leq 2(e + e^2) \cdot$ peak$(\mathcal{O}(\mathcal{J}))$.*

Proof. Since $\mathcal{I}_1$ and $\mathcal{I}_2$ are disjoiont, $\text{peak}(\mathcal{V}) = \max\{\text{peak}(\mathcal{V}, \mathcal{I}_1), \text{peak}(\mathcal{V}, \mathcal{I}_2)\}$. By Lemma 6.3, $\text{peak}(\mathcal{V}) = \max\{2(e + e^2) \cdot \text{peak}(\mathcal{O}), 2 \cdot \text{peak}(\mathcal{O})\} = 2(e + e^2) \cdot \text{peak}(\mathcal{O})$. $\square$

Next, we show that $\mathcal{UV}$ is $(6(e + e^2) + 1)$-competitive (Theorem 6.9). We let $\mathcal{J}^*$ denote the input set where jobs have uniform width, $\mathcal{J}^*_{\mathrm{T}}$ denote the tight job set in $\mathcal{J}^*$ and $\mathcal{J}^*_{\mathrm{L}}$ denote the loose job set in $\mathcal{J}^*$. First we prove that any feasible schedule for tight jobs is 3-competitive because of the "inflexibility".

Lemma 6.5. *For any feasible schedule S, peak$(S(\mathcal{J}^*_T)) \leq 3 \cdot$ peak$(\mathcal{O}(\mathcal{J}^*))$.*

Proof. We prove it by showing that even if the execution intervals of jobs are considered as the whole feasible interval, the peak is not too much larger than the peak in the optimal schedule.

We first *extend* jobs $J \in \mathcal{J}^*_{\mathrm{T}}$ to J^+ as the following: $r(J^*) = r(J)$, $d(J^*) = d(J)$, $w(J^*) = d(J) - r(J)$, and $h(J^*) = h(J)$. That is, every job has its width as the length of its feasible interval. We denote the resulting job set by $\mathcal{J}^+$. It is easy to see that because of each job in $\mathcal{J}^+$ are not shiftable, there is only one feasible schedule for $\mathcal{J}^+$ and it is optimal. It is clear that $\text{peak}(S(\mathcal{J}^*_{\text{T}})) \leq \text{peak}(\mathcal{O}(\mathcal{J}^+))$.

Similar to Lemma 4.25, we can bound the load at any time t of $\mathcal{O}(\mathcal{J}^+)$ by the loads of constant number of timeslots in $S(\mathcal{J}^*_{\text{T}})$. Consider the job J corresponding to J^+, the execution interval of J in any feasible schedule must contains either timeslot $t-(w-1)$, $t+(w-1)$, or t. Hence we can upper bound the load at any time t in $\mathcal{O}(\mathcal{J}^+)$ as following: $\ell oad(\mathcal{O}(\mathcal{J}^+), t) \leq \ell oad(\mathcal{O}(\mathcal{J}^*_{\text{T}}), t-(w-1)) + \ell oad(\mathcal{O}(\mathcal{J}^*_{\text{T}}), t+(w-1)) + \ell oad(\mathcal{O}(\mathcal{J}^*_{\text{T}}), t)$. By Observation 17, $\text{peak}(S(\mathcal{J}^*_{\text{T}})) \leq \text{peak}(\mathcal{O}(\mathcal{J}^+)) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*_{\text{L}}))$. □

Recall that for the loose jobs set $\mathcal{J}^*_{\text{L}}$ where jobs have uniform width w, we transform it to $\mathcal{J}'$ by AlignFI into jobs with release times and deadlines being at $i \cdot w$ for certain integers i. The schedule $\mathcal{UV}(\mathcal{J}^*_{\text{L}})$ is transformed from the schedule $\mathcal{V}(\mathcal{J}')$ by FreeSch. In the following we prove that the Transformation FreeSch does not affect the performance (Lemma 6.6), and the Transformation AlignSch does not enlarge the gap between the peak of $\mathcal{UV}$ and the peek of $\mathcal{O}$ too much (Lemma 6.7). Hence, the $\mathcal{UV}(\mathcal{J}^*_{\text{L}})$ is $O(1)$-competitive (Lemma 6.8).

Lemma 6.6. *Given S', consider the S^* generated by* FreeSch, *peak*$(S^*) =$ *peak*(S').

Proof. It is easy to see that $\ell oad(S^*, t) = \ell oad(S', t)$ for all t. Hence $\text{peak}(S^*) = \text{peak}(S')$. □

According to Lemma 4.22, we know that for any schedule S^* for $\mathcal{J}^*_{\text{L}}$ and the schedule S' for $\mathcal{J}'$ constructed by AlignSch, at any timeslot t, $\ell oad(S', t) \leq \ell oad(S^*, t) + \ell oad(S^*, t-(w-1)) + \ell oad(S^*, t+(w-1))$. Therefore we have the following lemma:

Lemma 6.7. *Given a schedule S^* for a set of loose jobs $\mathcal{J}^*_L$ with uniform width and the schedule S' generated by* AlignSch *for the corresponding $\mathcal{J}'$ generated by Procedure* AlignFI, *peak*$(\mathcal{O}(\mathcal{J}')) \leq 3 \cdot$ *peak*$(\mathcal{O}(\mathcal{J}^*_L))$.

Proof. By definition, $\text{peak}(S') = \max_t \ell oad(S', t)$. By Lemma 4.22 and Observation 17, $\text{peak}(S') \leq \max_t(\ell oad(S', t) \leq \ell oad(S^*, t) + \ell oad(S^*, t-(w-1)) + \ell oad(S^*, t+(w-1)))$ $\leq 3 \cdot \text{peak}(S^*)$. Given $\mathcal{O}(\mathcal{J}^*_{\text{L}})$, there exists a schedule $S(\mathcal{J}')$ generated by AlignSch. Therefore, $\text{peak}(S(\mathcal{J}')) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*_{\text{L}}))$. Hence $\text{peak}(\mathcal{O}(\mathcal{J}')) \leq \text{peak}(S(\mathcal{J}')) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*_{\text{L}}))$. □

Lemma 6.8. *For loose jobs set $\mathcal{J}^*_L$ where jobs have uniform width, peak*$(\mathcal{UV}(\mathcal{J}^*_L)) \leq 6(e+e^2) \cdot$ *peak*$(\mathcal{O}(\mathcal{J}^*))$.

Proof. In $\mathcal{UV}$, the job set $\mathcal{J}^*_{\text{L}}$ is transformed into a job set $\mathcal{J}'$ by AlignFI and $\mathcal{V}$ is performed on $\mathcal{J}'$. Then, $\mathcal{UV}(\mathcal{J}^*_{\text{L}}) = \mathcal{V}(\mathcal{J}')$ by Transformation FreeSch. Hence, $\text{peak}(\mathcal{UV}(\mathcal{J}^*_{\text{L}})) = \text{peak}(\mathcal{V}(\mathcal{J}'))$. By Theorem 6.4, $\text{peak}(\mathcal{V}(\mathcal{J}')) \leq 2(e+e^2) \cdot \text{peak}(\mathcal{O}(\mathcal{J}'))$. By Lemma 6.7, $\text{peak}(\mathcal{O}(\mathcal{J}')) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*_{\text{L}}))$. Hence, $\text{peak}(\mathcal{UV}(\mathcal{J}^*_{\text{L}})) \leq 6(e+e^2) \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*))$. □

Now we are ready to prove that the algorithm $\mathcal{UV}$ performs well when the tight jobs and loose jobs are considered together:

Theorem 6.9. *peak*$(\mathcal{UV}(\mathcal{J}^*)) \leq (6(e+e^2)+1) \cdot$ *peak*$(\mathcal{O}(\mathcal{J}^*))$.

Proof. By definition, $\text{peak}(\mathcal{UV}(\mathcal{J}^*)) \leq \text{peak}(\mathcal{UV}(\mathcal{J}^*_{\text{T}})) + \text{peak}(\mathcal{UV}(\mathcal{J}^*_{\text{L}}))$. By Lemma 6.5 and 6.8, $\text{peak}(\mathcal{UV}(\mathcal{J}^*)) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*)) + 6(e+e^2) \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*)) = (6(e+e^2)+1) \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*))$. □

Finally we are going to bound the competitive ratio of $\mathcal{A}$. Recall that $\mathcal{A}(\mathcal{J})$ partition jobs into subsets $\mathcal{J}_p$ such that in each $\mathcal{J}_p$ jobs have bounded widths. For each $\mathcal{J}_p$, it is transformed into $\mathcal{J}^*_p$ and $\mathcal{UV}$ is performed. Then, $\mathcal{A}(\mathcal{J}_p) = \mathcal{UV}(\mathcal{J}_p)$ by Transformation ShrinkSch. We show that ShrinkSch does not increase the peak in Lemma 6.12. On the other hand, we show that the effect on considering jobs as nice jobs is bounded (Lemma 6.11). Let $\mathcal{J}_p$ and $\mathcal{J}^*_p$ denote the set of jobs in class C_p and the nice jobs generated by Convert. Also we denote S_p and S^*_p as the schedule of $\mathcal{J}_p$ and the schedule generated by RelaxSch. According to Lemma 4.29, we know that for any time t, $\ell oad(S^*_p, t) \leq \ell oad(S_p, t) + \ell oad(S_p, t-(2^{p-1}-1)) + \ell oad(S_p, t+(2^{p-1}-1))$. Hence we have the following lemma:

Lemma 6.10. *Given S_p and S^*_p generated by* RelaxSch*, peak*$(S^*_p) \leq 3 \cdot$ *peak*(S_p)

Proof. By Lemma 4.29 and Observation 17, $\text{peak}(S^*_p) = \max_t \ell oad(S^*_p, t) \leq \max_t(\ell oad(S_p, t) + \ell oad(S_p, t-(2^{p-1}-1)) + \ell oad(S_p, t+(2^{p-1}-1))) \leq 3 \cdot \text{peak}(S_p)$. □

Lemma 6.11. *Consider any job set $\mathcal{J}$, its corresponding $\mathcal{J}^*$ and the corresponding job set of each class $\mathcal{J}_p$ and $\mathcal{J}^*_p$, peak*$(\mathcal{O}(\mathcal{J}^*_p)) \leq 3 \cdot$ *peak*$(\mathcal{O}(\mathcal{J}))$.

Proof. Given $\mathcal{O}(\mathcal{J}_p)$, there exists schedule $S(\mathcal{J}^*_p)$ generated by RelaxSch. By Lemma 6.10, $\text{peak}(S(\mathcal{J}^*_p)) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}_p))$. Hence, $\text{peak}(\mathcal{O}(\mathcal{J}^*_p)) \leq \text{peak}(S(\mathcal{J}^*_p)) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}_p)) \leq 3 \cdot \text{peak}(\mathcal{O}(\mathcal{J}))$. □

Lemma 6.12. *Using* ShrinkSch*, peak*$(S_p) \leq$ *peak*(S^*_p)

Proof. It can be easily seen that for all t, $\ell oad(S_p, t) \leq \ell oad(S^*_p, t)$. Hence $\text{peak}(S_p) \leq \text{peak}(S^*_p)$. □

The following theorem asserts that $\mathcal{A}$ is $O(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil)$-competitive.

Theorem 6.13. *For any job set $\mathcal{J}$, peak*$(\mathcal{A}(\mathcal{J})) \leq (6(e+e^2)+1) \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil) \cdot$ *peak*$(\mathcal{O}(\mathcal{J}))$.

Proof. By definition, $\text{peak}(\mathcal{A}(\mathcal{J})) = \max_t \ell oad(\mathcal{A}(\mathcal{J}, t)) = \max_t \sum_{p=1}^{1+\lceil \log \frac{w_{\max}}{w_{\min}} \rceil} \ell oad(\mathcal{UV}(\mathcal{J}_p), t)$. By shuffling the terms, $\text{peak}(\mathcal{A}(\mathcal{J})) \leq \sum_{p=1}^{1+\lceil \log \frac{w_{\max}}{w_{\min}} \rceil} \text{peak}(\mathcal{UV}(\mathcal{J}_p))$. By Theorem 6.9, $\text{peak}(\mathcal{A}(\mathcal{J})) \leq \sum_{p=1}^{1+\lceil \log \frac{w_{\max}}{w_{\min}} \rceil} (6(e+e^2)+1) \cdot \text{peak}(\mathcal{O}(\mathcal{J}^*_p))$. By Lemma 6.11, $\text{peak}(\mathcal{A}(\mathcal{J})) \leq \sum_{p=1}^{1+\lceil \log \frac{w_{\max}}{w_{\min}} \rceil} (6(e+e^2)+1) \cdot \text{peak}(\mathcal{O}(\mathcal{J})) \leq (6(e+e^2)+1) \cdot (1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil) \cdot \text{peak}(\mathcal{O}(\mathcal{J}))$.

□

The Machine Minimization problem. The Machine Minimization problem is as follows. The inputs are a set of jobs J with processing times $p(J)$, release times $r(J)$, and deadlines $d(J)$ and infinite number of machines. Each machine can serve only one job at a time. Each job has to be scheduled on a machine such that in its execution interval (which is in the interval between its start time and end time) there is no other job assigned on the same machine. The goal is to minimize the number of used machines.

The Machine Minimization problem can be seen as a special case of the $\mathsf{GRID_{peak}}$ problem by considering the processing time as the time duration of the job. In the Machine Minimization problem, the jobs have no power request. Hence we can reduce an input set $\mathcal{J}_M$ of the machine minimization problem to an input set $\mathcal{J}_D$ of the $\mathsf{GRID_{peak}}$ problem as follows. For every job $J \in \mathcal{J}_M$, there is a corresponding job $J' \in \mathcal{J}_D$ with $w(J') \leftarrow p(J)$, $h(J') \leftarrow 1$, $r(J') \leftarrow r(J)$ and $d(J') \leftarrow d(J)$. By Theorem 6.13, we prove that there exists an asymptotically optimal online algorithm for the Machine Minimization problem.

6.2 The interval graph approach on other problems

In Section 4.3, we introduced an exact algorithm $\mathcal{E}$ using the *linear clique arrangement* property of the interval graphs. The linear property of the consecutive clique arrangement of interval graphs gives a direction to design a dynamic programming algorithm, which breaks down a problem into overlapped subproblems until the subproblems are simple enough to be solved. Recall the algorithm $\mathcal{E}$:

Algorithm $\mathcal{E}$ (also see Section 4.3). Basically, the jobs are considered as time intervals and the time horizon is chopped into "windows". We first transform the input job set $\mathcal{J}$ to an interval graph, and obtain the maximal cliques C_i for $1 \leq i \leq k$ and the corresponding windows W_i. The algorithm visits all windows accordingly from the left to the right. In Stage i, the i-th window is visited and the algorithm maintains a candidate set of schedules for the visited windows that no optimal solution is deleted from the set. In each Stage i, the algorithm consists of three procedures: ListConfigurations, ConcatenateTables and FilterTable.

The ListConfigurations procedure lists all possible configurations (i.e., execution segments) of the jobs in C_i within W_i. The invalid configurations will be deleted. The valid configurations together with their cost will be stored in a table.

The ConcatenateTables procedure concatenates the configurations in the current window W_i and the configurations in the windows which have been seen so far. If the execution interval after concatenation is not valid, it is deleted from the table. The cost of the new configuration is simply the sum of the two concatenated configurations.

The FilterTable procedure filters non-optimal configurations. The idea is, given a configuration of the jobs in C_i, there must be a best decision of the jobs in $\bigcup_{k=1}^{i-1} C_k \setminus C_i$ which has minimum cost within the intervals $[0, b_{i+1})$, where b_{i+1} is the right boundary of the window W_i. For each configuration, we only keep the (partial) schedule with the minimum cost.

After processing all the windows, the schedule with minimum cost can be found in the final table.

In the following of this section, we show that the $\mathcal{E}$ can be used to solve other demand response management problems by changing the objective function.

6.2.1 Minimizing the peak power request

To solve the $\mathsf{GRID}_{\mathsf{peak}}$ problem, we modify the algorithm $\mathcal{E}$ as the following. Recall that in Stage i, in the original ListConfigurations procedure, each valid configuration will be stored in the table together with their cost. We store the valid configurations together with the peak power request instead of the total cost. In the ConcatenateTables procedure, the peak power request of the new configuration is the bigger one of the two concatenated configurations. In the FilterTable procedure, we keep the partial schedule with minimum peak power request for each given configurations of jobs in C_i.

It is easy to see that we list all possible configurations. A configuration is deleted only when it is invalid or it is identical to another configuration with lower peak power request. Hence in the end we get an optimal schedule. It also shows that the $\mathsf{GRID}_{\mathsf{peak}}$ problem is fixed parameter tractable with respect to the maximum width of jobs and the maximum number of overlapped feasible intervals, and the maximum length of windows.

Corollary 6.14. *The* $\mathsf{GRID}_{\mathsf{peak}}$ *problem is fixed parameter tractable with respect to the maximum width of jobs, the maximum number of overlapped feasible intervals, and the maximum length of windows.*

Corollary 6.15. *The* $\mathsf{GRID}_{\mathsf{peak}}$ *problem is fixed parameter tractable with respect to the maximum width of jobs, and the maximum number of overlapped feasible intervals.*

6.2.2 Minimizing the total cost with limited power

In this section we consider the GRID problem with power limit L. More formally, the aim is to find a schedule which feasibly schedule all jobs such that for every timeslot, the power request is at most L and the total cost is minimized.

The $\mathsf{GRID}_{\mathsf{limited}}$ problem. The input of the $\mathsf{GRID}_{\mathsf{limited}}$ problem is a set of jobs where each job has a power request, a time duration, and a feasible intervals in which the job can be scheduled. Moreover, there is a power limit L such that the total power request at any timeslot cannot be more than L. The goal is to feasibly serve all jobs without preemption such that for all timeslot t, $\ell oad(t) \leq L$ and the total cost is minimized.

To solve the $\mathsf{GRID}_{\mathsf{limited}}$ problem, we modify the algorithm $\mathcal{E}$ by redefining the “valid” configuration. Previously the validity of configuration was only about the consistency of the execution segments; now we also consider a configuration with load at some timeslot bigger than L as invalid. The other parts of the algorithm $\mathcal{E}$ remain the same.

It is clear that we list all possible configurations. A configuration is deleted only when it is invalid, it has some timeslots with load more than L or it is identical to another configuration with lower cost Hence in the end we get an optimal schedule and in the schedule each timeslot has load at most L.

Corollary 6.16. *The* $\mathsf{GRID}_{\mathsf{limited}}$ *problem is fixed parameter tractable with respect to the maximum width of jobs, the maximum number of overlapped feasible intervals, and the maximum length of windows.*

Corollary 6.17. *The* $\mathsf{GRID}_{\mathsf{limited}}$ *problem is fixed parameter tractable with respect to the maximum width of jobs, and the maximum number of overlapped feasible intervals.*

6.3 Summary

In this chapter, we adapted our techniques to solve other problems. We proved that the online algorithm we proposed in Section 5.1 is asymptotically optimal for the smart grid problem with objective minimizing maximum power request. Since the Machine Minimization problem is a special case of the $\mathsf{GRID}_{\mathsf{peak}}$ problem, the algorithm is also asymptotically optimal for the Machine Minimization problem.

We also investigated the potential of the exact algorithm framework proposed in Section 4.3. We showed that the framework can be adapted to solve other demand response management problems by simply changing the check conditions in the algorithm. It also showed that these problems are all fixed-parameter tractable with respect to the same set of parameters.

Chapter 7

Conclusion

In this thesis, we study algorithms for problems within the general area of the smart grid scheduling.

In Chapter 4, we proved that the GRID problem is NP-hard, even when preemption is allowed, or the jobs have common feasible intervals and unit width/heights. We then show that for a special case where jobs have unit width and unit height, the GRID problem is polynomial time solvable. The polynomial time algorithm is based on a graph structure which captures all feasible assignments. We show that maintaining the graph and querying the graph can be done in polynomial time. By a simple checking on the graph, we can know if the current assignment is optimal. If the current assignment is not optimal, by polynomial time shifting of the jobs, the optimal schedule can be achieved easily.

By generalizing the optimal algorithm for the unit case, we proposed an approximation algorithm for jobs with arbitrary widths and arbitrary heights. The approximation algorithm is based on classifying jobs by their widths and heights. For each class, jobs are treated as unit-size jobs by necessary modifications and the optimal algorithm for unit-size jobs is performed.

The approximation only gives a very initial result. A potential direction is to investigate other approaches and see if there are more accurate approximation algorithms. Further, for complexity interests, it would be interesting to find an approximation lower bound.

In Chapter 5, we proposed a $(36(1 + \lceil \log \frac{w_{\max}}{w_{\min}} \rceil))^{\alpha} \cdot (\min\{(8(e + e^2)^{\alpha}, \frac{(2\alpha)^{\alpha}}{2}\} + 1)$-competitive online algorithm for the jobs with arbitrary widths, arbitrary heights, and arbitrary feasible intervals. We also prove that for any deterministic online algorithm, the competitive ratio is at least $(\frac{1}{3} \log \frac{w_{\max}}{w_{\min}})^{\alpha}$ for arbitrary $\alpha > 1$. We also proved that for more restricted input, the simple strategies as First-Fit, Next-Fit, or Best-Fit can achieve good competitive ratio.

Some of our online algorithms are based on the analogy to the dynamic voltage/speed scaling algorithm. It gives us fascinating views on the relations of these energy-aware problems. However, it is also necessary to look at different approaches. Furthermore, the parameters we considered so far are $w_{\max}$ and $w_{\min}$. Trying other parameters might give us new point of views.

In Chapter 6 we investigate the interval graphs-based framework for exact solutions, which was proposed in Chapter 4, can be adapted to solve different demand management problems. By the adaption, we also proved that these problems are all fixed-parameterized tractable with respect to maximum width of jobs, and the maximum number of overlapped feasible intervals. It would be interesting to see how to adapt this framework to more other problems. On the other hand, the competitive ratio of our online algorithm also dependents on the ratio $\frac{w_{\max}}{w_{\min}}$. It raises a curious question: does the parameterized complexity direct us to possible competitive algorithms?

The online algorithm proposed in Chapter 5 is proved in Chapter 6 to be asymptotically for the Machine Minimization problem. It would be interesting to look into the relations between these optimization problems. We compared and contrasted many classical optimization problems and the $\mathsf{GRID}/\mathsf{GRID}_{\mathsf{peak}}$ problems in Chapter 3. It would be exciting to dive into the problems and investigate how the constraints of the problems, preemptive or non-preemptive, discrete or continuous, allowing reshaping or not, affect the strategy design and analysis. Moreover, how these properties affect on the complexity of the problems.

For the GRID problem, there are a number of potential directions. It would be interesting to consider the jobs with precedence constraints. Also, users might be willing to pay more for finishing certain jobs earlier. It is another interesting problem that how to minimize a linear combination of the cost and the (weighted) completion time, which might be a balance between the cost and the happiness of the users.

In real world, the power plants have limited power, which means there is a power request upper bound. If the power request is over this bound, it might causes the shot down of the power plant. Hence it is essential to consider minimizing the total cost under the power limit. We proposed an exact algorithm for this problem in Chapter 6, and further research in the online setting is needed.

In UK there is an "Economy 7" tariff which costs different price per unit of power at day and night. It is also interesting to fit the GRID problem in such kind of tariff. That is, at different time, the cost functions are different.

Some of the users might be able to generate energy. In this case, the scheduling strategy has to take it into consideration. Depending on the capability of storage and trading of energy, the scheduling problem becomes even more complicated.

Moreover, consider different power generators as different machines, the GRID problem can be extended to multiple machines model. Different power generators might hive different cost function, and different job request might cost differently on different power generators. It brings more interesting and unsolved problems.

Bibliography

[1] Soroush Alamdari, Therese Biedl, Timothy M Chan, Elyot Grant, Krishnam Raju Jampani, Srinivasan Keshav, Anna Lubiw, and Vinayak Pathak, *Smart-grid electricity allocation via strip packing with slicing*, WADS, Springer, 2013, pp. 25–36.

[2] Susanne Albers, *Energy-efficient algorithms*, Communication ACM **53** (2010), no. 5, 86–96.

[3] Antonios Antoniadis and Chien-Chung Huang, *Non-preemptive speed scaling*, J. Scheduling **16** (2013), no. 4, 385–394.

[4] Yossi Azar, *On-line load balancing*, Online Algorithms, Springer, 1998, pp. 178–195.

[5] Evripidis Bampis, Alexander V. Kononov, Dimitrios Letsios, Giorgio Lucarelli, and Ioannis Nemparis, *From preemptive to non-preemptive speed-scaling scheduling*, Discrete Applied Mathematics **181** (2015), 11–20.

[6] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs, *Speed scaling to manage energy and temperature*, J. ACM **54** (2007), no. 1, 3:1–3:39.

[7] Paul C. Bell and Prudence W. H. Wong, *Multiprocessor speed scaling for jobs with arbitrary sizes and deadlines*, J. Comb. Optim. **29** (2015), no. 4, 739–749.

[8] Allan Borodin and Ran El-Yaniv, *Online computation and competitive analysis*, Cambridge University Press, New York, NY, USA, 1998.

[9] Mihai Burcea, Wing-Kai Hon, Hsiang-Hsuan Liu, Prudence W. H. Wong, and David K. Y. Yau, *Scheduling for electricity cost in smart grid*, Combinatorial Optimization and Applications - 7th International Conference, COCOA 2013, Chengdu, China, December 12-14, 2013, Proceedings (Peter Widmayer, Yinfeng Xu, and Binhai Zhu, eds.), Lecture Notes in Computer Science, vol. 8287, Springer, 2013, pp. 306–317.

[10] ———, *Scheduling for electricity cost in a smart grid*, J. Scheduling **19** (2016), no. 6, 687–699.

[11] Stéphane Caron and George Kesidis, *Incentive-based energy consumption scheduling algorithms for the smart grid*, IEEE Smart Grid Comm., 2010, pp. 391–396.

[12] Chen Chen, K. G. Nagananda, Gang Xiong, Shalinee Kishore, and Lawrence V. Snyder, *A communication-based appliance scheduling scheme for consumer-premise energy management systems*, IEEE Trans. Smart Grid **4** (2013), no. 1, 56–65.

[13] Lin Chen, Nicole Megow, and Kevin Schewior, *An o(log* m*)-competitive algorithm for online machine minimization*, Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016 (Robert Krauthgamer, ed.), SIAM, 2016, pp. 155–163.

[14] Julia Chuzhoy and Paolo Codenotti, *Erratum: Resource minimization job scheduling*, APPROX-RANDOM, 2009.

[15] Julia Chuzhoy, Sudipto Guha, Sanjeev Khanna, and Joseph Naor, *Machine minimization for scheduling jobs with interval constraints*, 45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings, IEEE Computer Society, 2004, pp. 81–90.

[16] Julia Chuzhoy, Sudipto Guha, Sanjeev Khanna, and Joseph Seffi Naor, *Machine minimization for scheduling jobs with interval constraints*, FOCS, IEEE, 2004, pp. 81–90.

[17] Mark Cieliebak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer, *Scheduling with release times and deadlines on a minimum number of machines*, IFIP TCS, Springer, 2004, pp. 209–222.

[18] Mark Cieliebak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer, *Scheduling with release times and deadlines on a minimum number of machines*, Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France (Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, eds.), IFIP, vol. 155, Kluwer/Springer, 2004, pp. 209–222.

[19] Nikhil R. Devanur, Konstantin Makarychev, Debmalya Panigrahi, and Grigory Yaroslavtsev, *Online algorithms for machine minimization*, CoRR **abs/1403.0486** (2014).

[20] Irit Dinur, Klaus Jansen, Joseph Naor, and José D. P. Rolim (eds.), *Approximation, randomization, and combinatorial optimization. algorithms and techniques, 12th international workshop, APPROX 2009, and 13th international workshop, RANDOM 2009, berkeley, ca, usa, august 21-23, 2009. proceedings*, Lecture Notes in Computer Science, vol. 5687, Springer, 2009.

[21] Marijana Živić Djurović, Aleksandar Milačić, and Marko Kršulja, *A simplified model of quadratic cost function for thermal generators*, DAAAM, 2012, pp. 25–28.

[22] Jack Edmonds and Richard M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19** (1972), no. 2, 248–264.

[23] European Commission, *Europen smartgrids technology platform*, `ftp://ftp.cordis.europa.eu/pub/fp7/energy/docs/smartgrids_en.pdf`, 2006.

[24] Kan Fang, Nelson A. Uhan, Fu Zhao, and John W. Sutherland, *Scheduling on a single machine under time-of-use electricity tariffs*, Annals OR **238** (2016), no. 1-2, 199–227.

[25] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang, *Smart grid – the new and improved power grid: A survey*, IEEE Communications Surveys Tutorials **14** (2012), no. 4, 944–980.

[26] Hassan Farhangi, *The path of the smart grid*, IEEE Power and Energy Mag. **8** (2010), no. 1, 18–28.

[27] Xin Feng, Yinfeng Xu, and Feifeng Zheng, *Online scheduling for electricity cost in smart grid*, COCOA, Springer, 2015, pp. 783–793.

[28] Delbert Fulkerson and Oliver Gross, *Incidence matrices and interval graphs*, Pacific Journal of Mathematics **15** (1965), no. 3, 835–855.

[29] M. R. Garey, Ronald L. Graham, David S. Johnson, and Andrew Chi-Chih Yao, *Resource constrained scheduling as generalized bin packing*, J. Comb. Theory, Ser. A **21** (1976), no. 3, 257–298.

[30] Michael R. Garey and David S. Johnson, *Computers and intractability; a guide to the theory of np-completeness*, W. H. Freeman & Co., New York, NY, USA, 1990.

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

[32] Rudolf Halin, *Some remarks on interval graphs*, Combinatorica **2** (1982), no. 3, 297–304.

[33] K. Hamilton and N. Gulhar, *Taking demand response to the next level*, IEEE Power and Energy Mag. **8** (2010), no. 3, 60–65.

[34] Dorit S. Hochbaum and J. George Shanthikumar, *Convex separable optimization is not much harder than linear optimization*, J. ACM **37** (1990), no. 4, 843–862.

[35] Wing-Kai Hon, Hsiang-Hsuan Liu, and Prudence W.H. Wong, *Online nonpreemptive scheduling for electricity cost in smart grid*, MAPSP, 2015, pp. 193–195.

[36] WA Horn, *Some simple scheduling algorithms*, Naval Research Logistics Quarterly **21** (1974), no. 1, 177–185.

[37] Chien-Chung Huang and Sebastian Ott, *New results for non-preemptive speed scaling*, Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II (Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, eds.), Lecture Notes in Computer Science, vol. 8635, Springer, 2014, pp. 360–371.

[38] Ali Ipakchi and Farrokh Albuyeh, *Grid of the future*, IEEE Power & Energy Mag. **7** (2009), no. 2, 52–62.

[39] Tohru Ishihara and Hiroto Yasuura, *Voltage scheduling problem for dynamically variable voltage processors*, Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998, Monterey, California, USA, August 10-12, 1998 (Anantha Chandrakasan and Sayfe Kiaei, eds.), ACM, 1998, pp. 197–202.

[40] Landis D. Kannberg, David P. Chassin, JohN G. DeSteese, Steve G. Hauser, Michael C. Kintner-Meyer, Robert G. Pratt, Lawrence A. Schienbein, and W. Michael Warwick, *GridWiseTM: The benefits of a transformed energy system*, arXiv preprint nlin/0409035 (2004).

[41] Mohammad M Karbasioun, Gennady Shaikhet, Evangelos Kranakis, and Ioannis Lambadaris, *Power strip packing of malleable demands in smart grid*, ICC, IEEE, 2013, pp. 4261–4265.

[42] Alexander V. Karzanov and S. Thomas McCormick, *Polynomial methods for separable convex optimization in unimodular linear spaces with applications*, SIAM J. Comput. **26** (1997), no. 4, 1245–1275.

[43] Ton Kloks and et al., *Advances in graph algorithms*, 2013.

[44] Iordanis Koutsopoulos and Leandros Tassiulas, *Control and optimization meet the smart power grid: Scheduling of power demands for optimal energy management*, e-Energy, ACM, 2011, pp. 41–50.

[45] ______, *Control and optimization meet the smart power grid: scheduling of power demands for optimal energy management*, 2nd International Conference on Energy-Efficient Computing and Networking 2011, e-Energy '11, New York, NY, USA - May 31 - June 01, 2011 (Hermann de Meer and David Hutchison, eds.), ACM, 2011, pp. 41–50.

[46] Rajamani Krishnan, *Meters of tomorrow [in my view]*, IEEE Power and Energy Magazine **6** (2008), no. 2, 96–94.

[47] Woo-Cheol Kwon and Taewhan Kim, *Optimal voltage allocation techniques for dynamically variable voltage processors*, ACM Trans. Embedded Comput. Syst. **4** (2005), no. 1, 211–230.

[48] Joseph Leung, Laurie Kelly, and James H. Anderson, *Handbook of scheduling: Algorithms, models, and performance analysis*, CRC Press, Inc., Boca Raton, FL, USA, 2004.

[49] Husheng Li and Robert Caiming Qiu, *Need-based communication for smart grid: When to inquire power price?*, Proceedings of the Global Communications Conference, 2010. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA, IEEE, 2010, pp. 1–5.

[50] Minming Li, *Approximation algorithms for variable voltage processors: Min energy, max throughput and online heuristics*, Theor. Comput. Sci. **412** (2011), no. 32, 4074–4080.

[51] Minming Li and F. Frances Yao, *An efficient algorithm for computing optimal discrete voltage schedules*, SIAM J. Comput. **35** (2005), no. 3, 658–671.

[52] Minming Li, Frances F. Yao, and Hao Yuan, *An \$o(n^2)\$ algorithm for computing optimal continuous voltage schedules*, CoRR **abs/1408.5995** (2014).

[53] Zhuo Li and Qilian Liang, *Performance analysis of multiuser selection scheme in dynamic home area networks for smart grid communications*, IEEE Trans. Smart Grid **4** (2013), no. 1, 13–20.

[54] Fu-Hong Liu, Hsiang-Hsuan Liu, and Prudence W. H. Wong, *Optimal nonpreemptive scheduling in a smart grid model*, 27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia (Seok-Hee Hong, ed.), LIPIcs, vol. 64, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 53:1–53:13.

[55] ———, *Optimal nonpreemptive scheduling in a smart grid model*, CoRR **abs/1602.06659** (2016).

[56] Alvaro Llaria, Jaime Jiménez, and Octavian Curea, *Study on communication technologies for the optimal operation of smart grids*, Trans. Emerging Telecommunications Technologies **25** (2014), no. 10, 1009–1019.

[57] Lockheed Martin, *SEELoad^{TM} Solution*, `http://www.lockheedmartin.co.uk/us/products/energy-solutions/seesuite/seeload.html`.

[58] Thillainathan Logenthiran, Dipti Srinivasan, and Tan Zong Shun, *Demand side management in smart grid using heuristic optimization*, IEEE Trans. Smart Grid **3** (2012), no. 3, 1244–1252.

[59] T. Joseph. Lui., Warwick Stirling, and Henry O. Marcy, *Get smart*, IEEE Power & Energy Mag. **8** (2010), no. 3, 66–78.

[60] Chris Y. T. Ma, David K. Y. Yau, and Nageswara S. V. Rao, *Scalable solutions of markov games for smart-grid infrastructure protection*, IEEE Trans. Smart Grid **4** (2013), no. 1, 47–55.

[61] Sabita Maharjan, Quanyan Zhu, Yan Zhang, Stein Gjessing, and Tamer Basar, *Dependable demand response management in the smart grid: A stackelberg game approach*, IEEE Trans. Smart Grid **4** (2013), no. 1, 120–132.

[62] Gilbert M Masters, *Renewable and efficient electric power systems*, John Wiley & Sons, 2013.

[63] Amir-Hamed Mohsenian-Rad, Vincent W.S. Wong, Juri Jatskevich, and Robert Schober, *Optimal and autonomous incentive-based energy consumption scheduling algorithm for smart grid*, ISGT, IEEE, 2010, pp. 1–6.

[64] Amir-Hamed Mohsenian-Rad, Vincent WS Wong, Juri Jatskevich, Robert Schober, and Alberto Leon-Garcia, *Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid*, IEEE Trans. Smart Grid **1** (2010), no. 3, 320–331.

[65] Balakrishnan Narayanaswamy, Vikas K. Garg, and T. S. Jayram, *Online optimization for the smart (micro) grid*, Proceedings of the 3rd International Conference on Energy-Efficient Computing and Networking, e-Energy'12, Madrid, Spain, May 9-11, 2012 (Marco Ajmone Marsan, Suresh Goyal, Shugong Xu, Antonio Fernández, Milan Prodanovic, and Ken Christensen, eds.), ACM, 2012, p. 19.

[66] Rolf Niedermeier, *Invitation to fixed-parameter algorithms*, Oxford University Press, Oxford, 2006.

[67] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein, *Optimal time-critical scheduling via resource augmentation*, Algorithmica **32** (2002), no. 2, 163–200.

[68] Anshu Ranjan, Pramod P. Khargonekar, and Sartaj Sahni, *Offline preemptive scheduling of power demands to minimize peak power in smart grids*, IEEE Symposium on Computers and Communications, ISCC 2014, Funchal, Madeira, Portugal, June 23-26, 2014, IEEE Computer Society, 2014, pp. 1–6.

[69] ———, *Offline first fit scheduling in smart grids*, 2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015, IEEE Computer Society, 2015, pp. 758–763.

[70] REGEN Energy Inc, *ENVIROGRID^{TM} SMART GRID BUNDLE*, `http://www.regenenergy.com/press/announcing-the-envirogrid-smart-grid-bundle/`.

[71] Barna Saha, *Renting a cloud*, FSTTCS, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 437–448.

[72] Sergio Salinas, Ming Li, and Pan Li, *Multi-objective optimal energy consumption scheduling in smart grids*, IEEE Trans. Smart Grid **4** (2013), no. 1, 341–348.

[73] Pedram Samadi, Amir-Hamed Mohsenian-Rad, Robert Schober, Vincent WS Wong, and Juri Jatskevich, *Optimal real-time pricing algorithm based on utility maximization for smart grid*, SmartGridComm, IEEE, 2010, pp. 415–420.

[74] P. T. Sokkalingam, Ravindra K. Ahuja, and James B. Orlin, *New polynomial-time cycle-canceling algorithms for minimum-cost flows*, Networks **36** (2000), no. 1, 53–63.

[75] Arie Tamir, *A strongly polynomial algorithm for minimum convex separable quadratic cost flow problems on two-terminal series-parallel networks*, Math. Program. **59** (1993), 117–132.

[76] ShaoJie Tang, Qiuyuan Huang, Xiang-Yang Li, and Dapeng Wu, *Smoothing the energy consumption: Peak demand reduction in smart grid*, INFOCOM, IEEE, 2013, pp. 1133–1141.

[77] Toronto Hydro Corporation, *Peaksaver Program*, `http://www.peaksaver.com/peaksaver_THESL.html`.

[78] UK Department of Energy & Climate Change, *Smart grid: A more energy-efficient electricity supply for the UK*, `https://www.gov.uk/smart-grid-a-more-energy-efficient-electricity-supply-for-the-uk`, 2013.

[79] US Department of Energy, *The Smart Grid: An Introduction*, `http://www.oe.energy.gov/SmartGridIntroduction.htm`, 2009.

[80] László A. Végh, *Strongly polynomial algorithm for a class of minimum-cost flow problems with separable convex objectives*, Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012 (Howard J. Karloff and Toniann Pitassi, eds.), ACM, 2012, pp. 27–40.

[81] F. Frances Yao, Alan J. Demers, and Scott Shenker, *A scheduling model for reduced CPU energy*, 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995, IEEE Computer Society, 1995, pp. 374–382.

[82] Sean Yaw and Brendan Mumey, *An exact algorithm for non-preemptive peak demand job scheduling*, Combinatorial Optimization and Applications - 8th International Conference, COCOA 2014, Wailea, Maui, HI, USA, December 19-21, 2014, Proceedings (Zhao Zhang, Lidong Wu, Wen Xu, and Ding-Zhu Du, eds.), Lecture Notes in Computer Science, vol. 8881, Springer, 2014, pp. 3–12.

[83] Sean Yaw, Brendan Mumey, Erin McDonald, and Jennifer Lemke, *Peak demand scheduling in the smart grid*, 2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014, IEEE, 2014, pp. 770–775.

[84] Zpryme Research & Consulting, *Power systems of the future: The case for energy storage, distributed generation, and microgrids*, `http://smartgrid.ieee.org/images/features/smart_grid_survey.pdf`, 2012.