©Copyright 2013
Jessica Chang

Energy-Aware Batch Scheduling

Jessica Chang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Reading Committee:

Samir Khuller, Chair

Anna R. Karlin, Chair

Arvind Krishnamurthy

Program Authorized to Offer Degree:
UW Computer Science and Engineering

University of Washington

Abstract

Energy-Aware Batch Scheduling

Jessica Chang

Co-Chairs of the Supervisory Committee:
Professor Samir Khuller
University of Maryland Computer Science

Professor Anna R. Karlin
Computer Science and Engineering

The subject of this thesis concerns algorithmic aspects of energy-related trends in computation. Within recent years, the computationally heaviest jobs are executed over massive corpora residing at data centers, whose energy profiles differ from those of standard desktops, and whose power consumption translates into an operational budget on the order of millions of dollars. Designing schedules to optimize notions of power consumption is sometimes at odds with standard techniques prevalent in the classical scheduling theory, which has traditionally focused on objectives capturing the interests of the scheduler (e.g., makespan) or of individual jobs (e.g., flow time).

Thus, we study a fairly simple model of energy usage: the *active time* scheduling model. Formally, we are interested in scheduling a set of n jobs, each with release time, deadline and processing requirement, on a system comprised of a single batch machine, with batch parameter B. (The machine can be working on up to B jobs at the same time.) The cost of a schedule is simply the amount of time that the system, i.e., the machine, is working. This is akin to the magnitude of the schedule's projection onto the time axis. The underlying assumption is that the cost for the machine to be active is roughly the same, regardless of whether it is working on only

one job or operating at full capacity. We give a fast algorithm for the basic case in which jobs are unit length and time is slotted. When time is not slotted, the problem still admits a polynomial-time solution, albeit of higher time complexity, via dynamic programming techniques. On the other hand, when time is slotted, but jobs may have multiple feasible intervals, the problem is NP-hard for $B > 2$, and otherwise can be solved via b-matching techniques. When jobs are arbitrary in length and can be preempted, but have a single feasible interval, we also show that a large class of algorithms is 5-approximate.

We also empirically compare algorithms within this class with the incumbent greedy algorithm due to Wolsey; the latter algorithm is widely implemented in practice. This comparison is cast within a framework general enough to capture other canonical covering problems, most notably Capacitated Set Cover. In particular, we design a heuristic LPO that essentially complements Wolsey's algorithm and propose optimizations to both approaches. At a high level, our findings solidly establish LPO as a competitive, if not superior, alternative to Wolsey's algorithm, with respect to both solution quality and number of subroutine calls made.

Finally, this thesis makes theoretical headway on the well-studied *busy time* problem. The key assumption that sets this apart from the aforementioned active time problem is that under the busy time model, the system has access to an *unlimited* number of identical batch machines. For non-preemptive jobs of arbitrary length, we give a 3-approximation that leverages important insights for the special case where the instance consists of interval jobs. When preemption is permitted, we give an exact algorithm for unbounded B; this result yields a simple 2-approximation for bounded B.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

ACTIVE TIME: amount of time that a machine is not idle. Also known as a machine's *busy time*.

BUSY TIME: , of a machine: see *active time*.

BUSY TIME: , of a schedule: the cumulative busy time over all machines used.

DEMAND: , of an interval job instance at time t: the number of jobs for which t is contained in their feasible regions.

INTERVAL JOB: : one whose length p_j is equal to the size its feasible region, i.e., $p_j = d_j - r_j$.

MAKESPAN: finish time of the last job to complete.

THROUGHPUT: number of jobs completed.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Samir Khuller, who I first met in undergraduate Algorithms in 2005. Since then, he has played a pivotal role in my development as a researcher. His confidence in me has been unwavering, from the summer of the REU to the last few years of my graduate studies.

I am also grateful for Anna Karlin, especially for her guidance during my earlier years, and also for pushing me to pursue research problems that I found exciting. A special thanks to the Department of Computer Science and Engineering at the University of Washington for graciously allowing me to conduct the bulk of my thesis-related research remotely, and to the University of Maryland Institute for Advanced Computer Studies for generously hosting me. The National Science Foundation supported a substantial portion of my graduate career[1]. This effort would not have completed without the flexibility and support of these people and institutions.

Also, thanks to Professor Hal Gabow of the University of Colorado for his insightful collaborations, as well as to Thach Nguyen and Koyel Mukherjee, with whom countless hours of brainstorming and discussion were spent.

I would like to thank Elisa Celis and Widad Machmouchi; what started as collegiate relationships at WIT 2008 has grown into valued personal friendships. Finally, thanks to my parents, Betsy and Al Lin, Xiaoxue Huang, Elaine Chan, Jessica Chao, Tiffany Fu, Jennifer Hsu, and Annie Lam, for their patience and continued support.

[1]NSF Graduate Research Fellowship, NSF CCF-1016509, NSF CCF-1217890, NSF CCF-0937865, and NSF ADVANCE grant.

DEDICATION

to my parents, without whom this endeavor would
have been impossible

Chapter 1

INTRODUCTION

The subject of this thesis concerns algorithmic aspects of energy-related issues in computation. Of particular interest are scheduling problems motivated by the amount of time that a system is "active". Though inherently simple, the relevant objective functions are quite different and sometimes at odds with standard objectives considered in classical scheduling theory, as highlighted by the following operational problem.

> Suppose that a ship can carry up to B cargo containers from port X across the ocean to port Y. Containers needing to be transported from X to Y have delivery requirements in the form of release times and deadlines. What is the minimum number of round trips required to deliver all the containers on time?

The obvious and key assumption here is that the cost to send the ship is roughly constant, regardless of the ship's load, and the goal is to minimize the number of trips made while meeting delivery requirements.

This dissertation examines several problems of this flavor. In general, we will be interested in scheduling a set of n jobs, each with a release time, deadline, and processing requirement, on a given batch system, with batch parameter $B > 0$. The quality of a schedule is measured by the total amount of time spent doing work; a machine is considered "active" or "busy" regardless of whether it is working on a single job or operating at full capacity. Intuitively speaking, the corresponding visualization of this measure would be the magnitude of a schedule's "projection" onto the time

axis. We primarily consider two models in this work, the *active time* model and the *busy time* model, the distinction between which lies in the resource assumptions made. Both preemptive and non-preemptive problems are studied. Most of the results discussed in this thesis were taken from the following set of papers [16, 17, 18].

1.1 Energy Consumption at Data Centers

Within the last twenty years, it has become standard for the most financially expensive computations to be executed over massive corpora residing at data centers. The role of data centers has evolved into the backbone of daily function in nearly every sector of the economy, from finance to academia, healthcare to infrastructural operations [28]. Between facility maintenance, cooling requirements and powering hardware, the operation of a typical data center is complex, demands an annual power budget on the order of millions of dollars, and leaves more than a negligible carbon footprint. As server technology has made great strides and as smaller data centers have slowly consolidated into larger ones, data center power densities have increased significantly. As a result, it is not uncommon for the cost of power and cooling to comprise the majority of a data center's operating budget. In 2006, the electricity consumed by U.S. data centers alone accounted for 1.5 percent of national energy usage, and totaled roughly $450 million [28]. Consequently, the energy consumption of data centers has become a primary topic of discussion across many communities [28, 26, 2, 47, 51].

Although the power consumed by server processors accounts for less than two percent of data center power costs [2], decisions made by scheduling policies at the processor level drastically affect data center efficiency, the increase of which can counter soaring costs in power and cooling [47]. As highlighted in the following quote, this calls for scheduling algorithms expressly tailored toward energy efficiency, in contrast to the classical objectives studied in the past.

> "Potential benefits include increased data center capacity and reduced

> capital expenditures as well as reduced power and cooling costs with power-aware job scheduling ... However, current batch job scheduling algorithms and configurations are tuned only to optimize performance; energy efficiency has been ignored" [47].

Manufacturers are also designing multi-core chips that can transition processors between sleep and active states without excessive overhead. Furthermore, Gandhi et al. [39] recently observe that despite setup times and the energy cost currently associated with transitions to the fully active state, a policy allowing for various sleep states is advantageous to the typical *AlwaysOn* configuration. In fact, they conclude that as the size of a data center increases, the impact of setup time on performance diminishes.

1.2 Energy Usage of Mobile Devices

Perhaps a more ubiquitous class of computation are those executed on mobile devices. As cellular devices have gradually become the global standard, the technology supporting them has also made undeniable and impressive strides. However, one of the most limiting constraints on cellular devices is the battery. Many popular applications for mobile devices heavily tax the battery (and network) due to high-speed and/or high-quantity data transfer, e.g., streaming content, regular updates via news or reader applications. As hardware breakthroughs continue to advance, it is increasingly clear that there is an express need for intelligent energy usage in mobile devices, from accurate energy profiling to identification of power drains to various dynamic power management strategies [74].

Opening (and maintaining) network connections are one source of significant drain on the cellular battery [78, 74]. Understanding the extent to which this cost can be mitigated is imperative to prolonging battery life. Standard software designed for the desktop machine does not fare well on the mobile device, which does not benefit

from wired network connections nor external AC sources. In particular, sending or retrieving data regularly in small bursts on a mobile device heavily drains power and also strains the network. In some cases, the solution is to prefetch the data in larger batches to reduce the number of network connections needed [44]. Indeed, a sizable effort by application developers and experts in networks has been mounted in response to this problem.

Classical scheduling theory for batch processors has traditionally focused on the optimization of objectives that capture the interests of the scheduler (e.g., minimizing makespan) or those of individual jobs (e.g., minimizing maximum tardiness, minimizing flow time). By and large, these results do not take energy constraints into consideration; in particular, they are ill-equipped to answer algorithmic questions arising from energy-driven contexts that have surfaced in the past couple of decades.

This thesis explores several scheduling problems through an energy-tinted lens. It extends the existent energy-motivated scheduling knowledge base, via the proposal of the simple and intuitive *active time* model and the study of the related *busy time* model. Both models cleanly capture many central issues in this space. This work also includes an empirical investigation of heuristics commonly applied to a wide class of computational problems.

1.3 Active Time Model

In the *active time* model, one is given a set $\mathcal{J}$ of n jobs, where each job J_i has an integer length p_i as well as a set T_i of time intervals in which it can be feasibly scheduled. $T_i = \{I_k^i = [r_k^i, d_k^i)\}_{k=1}^{m_i}$ is a non-empty set of m_i disjoint intervals, with boundaries at integer time points. The value m_i being one is equivalent to a job having a single release time and deadline, and in such cases, we will denote the release time of job J_i (deadline, respectively) by r_i (d_i, respectively). We assume that each job can "fit" in

its feasible region, i.e., that for all jobs J_i,

$$\sum_{k=1}^{m_i} |I_k^i| \geq \ell_i$$

For ease of notation, job J_i may sometimes be referred to simply as job i. Time is divided into unit length timeslots. We will denote by $\mathcal{T}$ the set of relevant timeslots from 1 to $d_{\max}$, where $d_{\max} = \max_i d_i$ is an upperbound on the makespan of any active time schedule.

In addition, one is given a batch machine that can work simultaneously on up to B different jobs at a time, as well as a memory storage unit holding the data that each job must access. The parameter B, a positive integer, is called the *batch capacity* of the system. If the machine schedules any jobs at timeslot t, the storage unit must be "on" at t and the system is considered *active at* t, regardless of whether it is processing one job or operating at full capacity. The notion of active time hinges on the assumption that whenever the storage unit is on, a constant power cost is incurred, whether the processor is working on B jobs or a single job, i.e. that the power consumed by the processor is negligible to that of the storage unit, as is commonly observed in data centers [2, 47].

The goal of the active time problem is to satisfy all n jobs, i.e., schedule them within their feasible regions, in a way that minimizes the number of slots during which the machine is active. In other words, subject to each job i being scheduled in p_i slots of its feasible region T_i, and subject to at most B jobs being scheduled at any slot, we would like to minimize the total active time spent scheduling the jobs.

For example, an active time instance is depicted in Figure 1.1. The batch capacity B is two. Jobs X and Y are unit length, but job Z has length two. The optimal solution schedules jobs X and Z in slot 2 and jobs Y and Z in slot 3, for a minimum active time of 2. It is easy to see that if any jobs are scheduled in the first slot, then the schedule cannot capitalize on parallelism with job Z, and the active time would be higher than necessary.

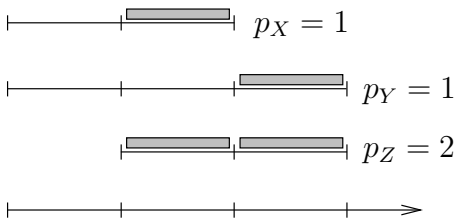


Figure 1.1: The optimal schedule of an active time instance in which B is two.

We consider both preemptive and non-preemptive variants of the active time problems. Within the scope of this thesis, preemption is permitted only at integer time points. In Graham's notational convention, the active time problem is $P|\ r_j, d_j, p_j, pmtn^+|\sum_t a_t$, where a_t is 1 whenever the schedule is active at timeslot t, and 0 otherwise.

Feasibility. In general, there may be instances for which there is no schedule that feasibly satisfies all jobs. However, this case is easy to detect. When jobs are unit in length, one can determine feasibility via an Earliest-Deadline-First (EDF) computation: order jobs in non-decreasing order of deadline. For each slot t from 1 to $d_{\max}$, greedily schedule up to B available jobs at t, favoring those of earlier deadline (breaking ties arbitrarily). Any feasible schedule can be modified via a sequence of swapping operations into one in which for any pair of jobs i and j with $d_i > d_j$, either i is scheduled no earlier than j, or i is scheduled strictly before r_j. Subject to tie-breaking, the EDF approach will find the (left-shifted) feasible schedule having exactly this property.

In fact, for any instance of the general active time problem, one can still determine feasibility via a simple maximum flow computation. Define $G_{\mathbf{feas}}$ to be the flow network whose vertex set is a source s, a sink t and a bipartite subgraph $(\mathcal{J}, \mathcal{T})$ where $\mathcal{J}$ contains a node x_i for every job i and $\mathcal{T}$ contains a node for every slot of time, from 1 to T. For each node $x_i \in \mathcal{J}$, there is an edge from s to x_i with capacity p_i. For any job i that is feasible in slot t, there is an edge of capacity one between x_i

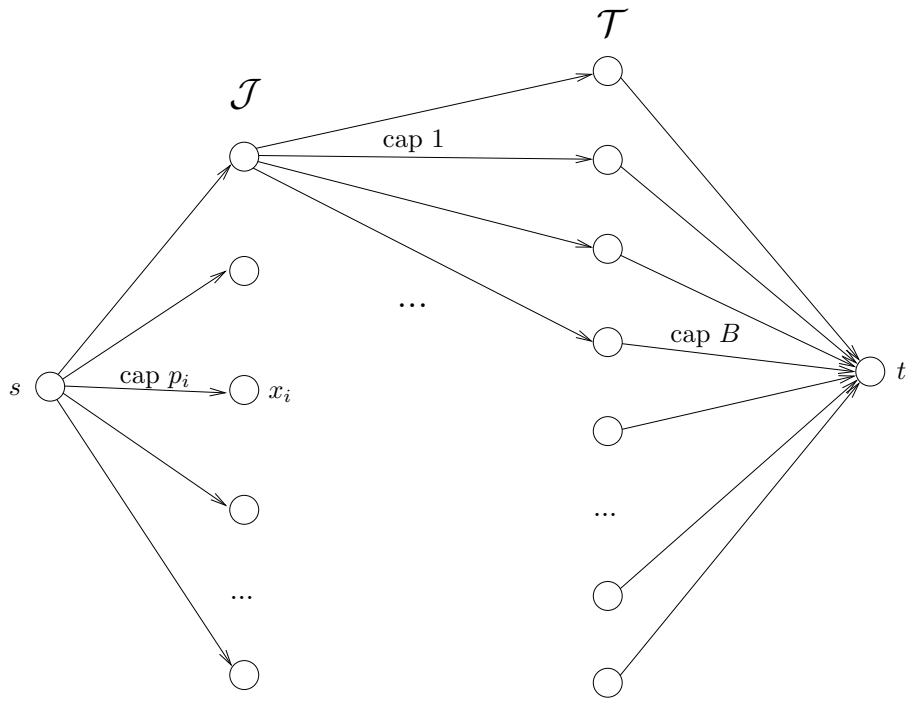


Figure 1.2: Flow network $G_{\mathbf{feas}}$. An integral flow of value $\sum_i p_i$ corresponds to a feasible schedule.

and t's node in $\mathcal{T}$. Finally, there exists an edge of capacity B between each slot node of $\mathcal{T}$ and the sink (see Figure 1.2). An active time instance has a feasible schedule if and only if the maximum flow that can be sent on the corresponding graph $G_{\mathbf{feas}}$ has value $L = \sum_{i=1}^{n} p_i$.

In general, classic scheduling algorithms do not immediately minimize active time. For example, Earliest-Deadline-First (EDF) policies schedule jobs as soon as they become available, making no effort to consolidate jobs into bundles. Futhermore, the standard relationship between a feasible schedule and matchings is complicated by the active time objective; the equivalent goal is to find a matching that matches all job nodes, but minimizes the number of slot nodes in the matching.

Contributions. Our first glimpse into the active time scheduling problem pertains to the special case in which jobs are unit length and have a single release time and deadline. On these types of instances, we show that minimum active time can be achieved via LazyActivation, a fast 2-pass algorithm. LazyActivation is greedy in nature, favoring jobs of earlier deadlines. However, unlike many Earliest-Deadline-First protocols, LazyActivation schedules jobs as late as possible, in the hopes of batching them with other jobs that are not released until later. Specifically, LazyActivation considers jobs in non-decreasing order of deadline, attempting to schedule jobs as late as possible, i.e., at their deadlines. To handle situations in which strictly more than B jobs have the same deadline, the algorithm a priori modifies job deadlines in a way that does not change the minimum active time and so that there are at most B jobs that share a deadline. LazyActivation has the additional property of maximizing throughput on instances where it is impossible to satisfy all jobs. In particular, if $n' \leq n$ is the maximum number of jobs that can be satisfied in a schedule, then LazyActivation returns a schedule that satisfies n' jobs and among all such schedules, minimizes active time.

When the release times and deadlines are permitted to take on real values, non-preemptively scheduling unit length jobs to minimize the number of batches can be solved optimally in polynomial time via dynamic programming. This objective differs slightly from active time: a batch must start all its jobs at the same time and the system may work on at most one batch at a time. Even so, this problem is a strict generalization of the previous one. We extend the result to address cases in which a budget on the number of active slots is imposed. These two dynamic programming results are our only active time results pertaining specifically to non-integral release times and deadlines; but for this small technicality, one can think of them as solving active time for equal-length jobs under the time-slotted model. The dynamic programs are in the same vein as those of Baptiste [9] and rest on the observation that there are without loss a quadratic number of time points at which a batch can start.

We consider two generalizations to the problem of scheduling unit jobs with single feasible intervals. For unit length jobs having multiple feasible regions and for batch capacity $B = 2$, there are exact algorithms based on identifying a certain kind of maximum cardinality b-matching on a graph constructed from the active time instance. In practice, it is often the case that jobs are periodic in nature; there is a sizeable research effort in periodic scheduling. Moreover, a job's feasible region may be constrained by irregular externalities, e.g. the availability of another resource or a user's schedule. The active time model is general enough to capture jobs of this nature. For large B, minimizing active time becomes NP-complete via a reduction from Vertex Cover: each vertex corresponds to a timeslot and each edge corresponds to a job that is feasible exactly in slots t_u and t_v, where $e = (u, v)$. Setting B to the maximum degree of any vertex in the graph, there exists a vertex cover of size k if and only if there is a feasible schedule with active time k.

In an orthogonal generalization, minimizing active time for non-preemptive arbitrary length jobs with a single feasible interval (i.e., a release time and a deadline) is NP-complete, while the problem's complexity when preemption is permitted remains unclear. Nevertheless, in this case, we show that minimal feasible solutions have bounded active time. A *minimal* feasible solution is a feasible set $\mathcal{S}$ of active slots such that no strict subset $\mathcal{S}' \subset \mathcal{S}$ of slots can feasibly support the entire job set. In other words, shutting down any active slot in $\mathcal{S}$ would render the modified solution infeasible[1]. We show via a charging argument that any algorithm returning minimal feasible solutions is 5-approximate. An empirical investigation of this particular algorithm is also given.

[1]In this context, we sometimes refer to a set of slots as a "solution" since given a set of active slots, determining the schedule itself, i.e., the assignment of jobs to slots, can be computed via a network flow computation, similar to the one in Figure 1.2.

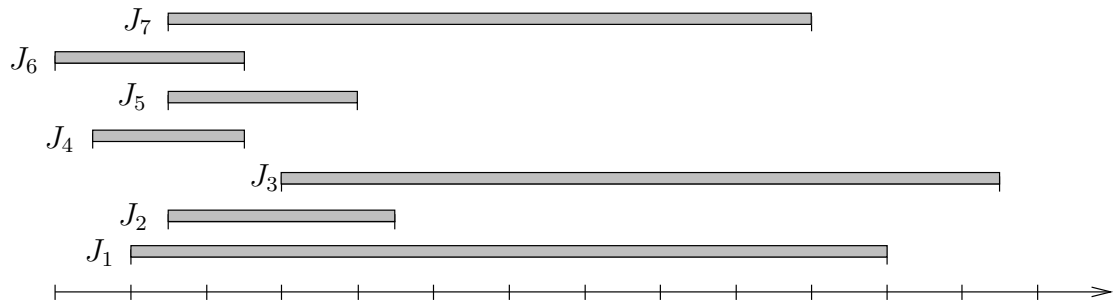
1.4 Busy Time Model

One existing notion most closely related to the active time model is that of *busy time*, first introduced by Winkler and Zhang [83]. Under this model, the system consists of *multiple* identical batch machines. Again, n jobs, each with release time, deadline and processing length, must be scheduled on these machines so that no machine is working on more than B jobs at any given time. The goal is to assign jobs to machines, i.e., partition the job set $\mathcal{J}$ into bundles, specifying their start times, so as to minimize the cumulative busy time over all machines. The key assumption that distinctively sets this model apart from that of active time is that the scheduler has access to an *unlimited* number of batch machines. Thus, every instance is feasible, since in the worst case, each job can always be scheduled separately on its own machine.

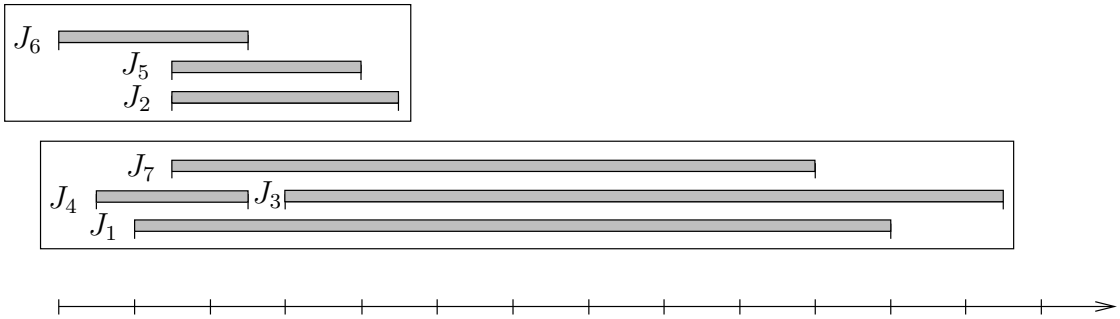
A notational comment: in the busy time literature, the total time that a machine is active (i.e., not idle) is called its *busy time*. Even though this is conceptually the same as a machine's active time as defined in the previous section, we will call it "busy time" in the context of the busy time model and "active time" in the context of the active time model.

The busy time problem was initially motivated by the following application that, though not driven by energy issues per se, exhibits characteristics similar to those of previously discussed applications.

Communication across an optical network is established in the form of *lightpaths*, each one spanning multiple links of the network to travel from one point to another. At each node of the network sits a switching unit, e.g., an Optical Add-Drop Multiplexer (OADM). Lightpaths must be assigned to various wavelengths so that on any link of the network, the number of lightpaths sharing the same wavelength does not exceed a certain limit, known as the *grooming* factor [46]. One central goal to designing good networks and supporting signal demands on them involves minimizing the cost



(a) A collection of interval jobs, ordered arbitrarily.



(b) Optimal packing of the jobs on two machines with $B = 3$. Jobs J_6, J_5 and J_2 form one bundle whose busy time is four, and the remaining jobs form the second bundle with busy time twelve, for a total cost of sixteen.

Figure 1.3: An instance of the busy time problem on interval jobs.

of hardware, in particular, the switching units of the network; the problem in its full generality involves routing challenges and is quite complex.

Consider the special case in which the network topology is a line, with OADMs at each intermediate node. Each lightpath must pass through a portion of the network corresponding to a subinterval on the line, so that the number of lightpaths sharing a single OADM does not exceed the grooming factor. Minimizing the number of switching units needed on a line topology is a special case of the busy time problem: time corresponds to the line network; batch machines to wavelengths; batch capacity to grooming factor; jobs to lightpaths; and a machine's busy time to the number of OADMs needed a group of lightpaths sharing the same wavelength [59].

In fact, this bring us to a special case of the busy time problem deserving particular attention: the scheduling of interval jobs.

Definition 1. *An **interval job** has the property that its processing length is equal to the size of its feasible region, i.e. the difference between its release time and deadline.*

Since in this case there is no question about when a job starts, the problem amounts to finding a feasible partition of the job set so as to minimize the sum of the busy time of each partition (see Figure 1.3). Each partition is assigned to a different machine, whose busy time is completely determined by the set of jobs assigned to it. In this sense, the interval jobs case appears to be quite restrictive. Even so, the challenge of optimally packing them onto machines is NP-complete [83]. Flammini et al. [33] gave a 4-approximation for the interval jobs case. Their FirstFit algorithm iterates over jobs in non-increasing order of length, assigning them to the machine of lowest index that can feasibly accept it. Subsequently, Khandekar et al. proved that the analysis of FirstFit extends to a 4-approximation for the general problem in which jobs may not be interval[2] [59]. They first solve via dynamic programming the instance sans the batch constraint, i.e., with B unbounded. This step fixes the starting time of each job, transforming the instance into one of interval jobs, after which FirstFit can be applied. Their techniques are clean, but do not suggest that every α-approximation for the interval jobs case immediately gives rise to an α-approximation for the general case.

Unknown to Flammini et al., existing results already subsumed theirs. Before describing them, the following concept will be helpful.

Definition 2. *The demand profile of an instance I is given by*

$$DP_I(t) = \left\lceil \frac{dem(t)}{B} \right\rceil$$

[2]Their result is a 5-approximation for a generalization of busy time which allows jobs to impose arbitrary demand or load on the machine. However, under the standard busy time problem (i.e., when job demands are unit), the approximation bound decreases to four.

for every time t, where $dem(t)$ is the number of jobs J_i such that $t \in [r_i, d_i)$. Let DP_I be $\sum_t DP_I(t)$.

Since $DP_I(t)$ is a lowerbound on the number of bundles spanning t in any feasible solution, DP_I is less than or equal to the $OPT(I)$, the busy time of the optimal solution.

Alicherry and Bhatia gave a 2-approximation [6] for a line coloring problem that captures exactly busy time on interval jobs. Their algorithm proceeds in phases, each of which identifies two sets of disjoint jobs. Each set is colored a different color and they show that the number of colors used at each time point t is no more than twice $DP_I(t)$. Kumar and Rudra later gave another 2-approximation [64]. Informally, the first phase of Kumar and Rudra's algorithm packs jobs infeasibly into the demand profile. However, their analysis demonstrates that the infeasibility is limited: no more than $2B$ jobs occupy the same space of the demand profile via this assignment. This is the key that allows them to then divide the packing into a feasible solution that is at most twice DP_I. Though these results give the best guarantee known for the interval jobs case, their extensions to the general busy time problem remain unclear. Extension of either algorithm via the approach by Khandekar et al. [59] trivially admits a 4-approximation. We discuss it in the context of Kumar and Rudra's result.

Observation 1. *There exists an extension of Kumar and Rudra's algorithm that is 4-approximate for the problem of minimizing busy time.*

Let $\mathcal{J}'$ denote the interval job instance acquired from solving the non-interval job instance $\mathcal{J}$ with unbounded B. Then, $DP_{\mathcal{J}'}$ is at most twice $OPT(\mathcal{J})$: the contribution to $DP_{\mathcal{J}'}$ of the top-most "layer" of the demand profile at every point of time is equal to the *span* of $\mathcal{J}'$, i.e., the size of the union of the jobs' intervals, taken over jobs in $\mathcal{J}'$. The span of $\mathcal{J}'$ lowerbounds $OPT(\mathcal{J})$ by definition. The rest of $DP_{\mathcal{J}'}$ can be charged to $\frac{\sum_j p_j}{B}$, which also lowerbounds $OPT(\mathcal{J})$. In summary, we have that

$$RK(\mathcal{J}') \leq 2DP_{\mathcal{J}'} \tag{1.1}$$

$$\leq 2(2OPT(\mathcal{J})) \tag{1.2}$$

where $RK(\mathcal{J}')$ is the busy time of Kumar and Rudra's algorithm on $\mathcal{J}'$.

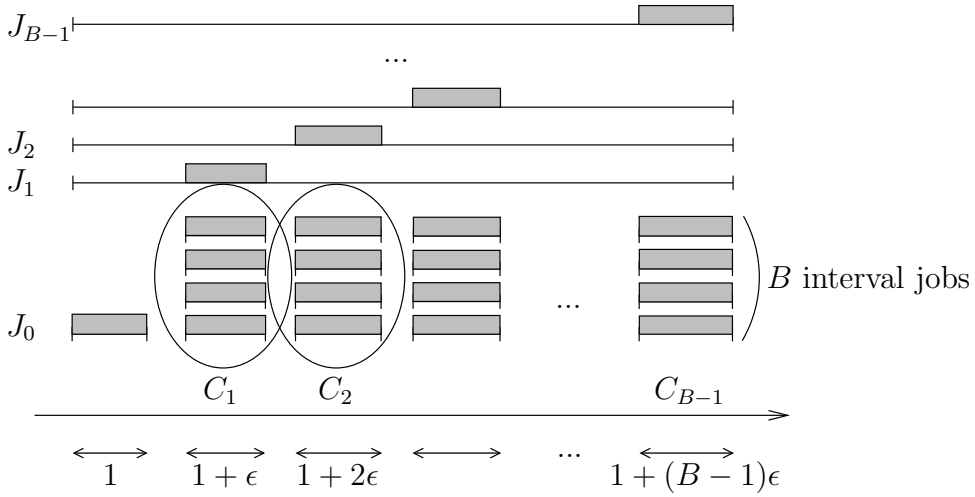
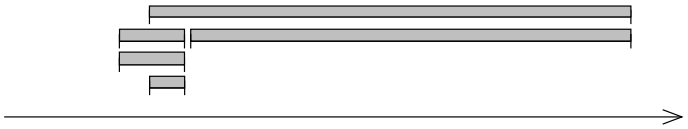
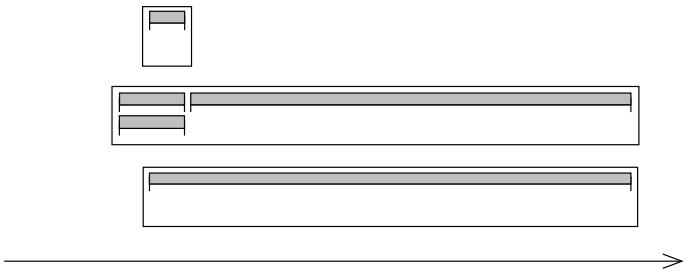


Figure 1.4: Fixing jobs by solving the problem for unbounded B can incur a cost approaching two as B increases without bound.

It is not obvious whether this analysis can be tightened. For example, consider the instance in Figure 1.4. Inequality 1.2 is tight: minimizing busy time with unbounded B is equivalent to minimizing the span, and thus the dynamic program will align the non-interval job J_i with the group C_i of interval jobs instead of optimally aligning them with the left-most job J_0. On the other hand, Figure 1.5(a) contains an instance of interval jobs in which the busy time of Kumar and Rudra's algorithm may be arbitrarily close to twice that of the optimal solution. Such a possible packing is shown in Figure 1.5(b). On this same instance, Alicherry and Bhatia's algorithm can also have cost close to twice that of the optimal solution. Whether their algorithms can be extended to handle non-interval jobs by a factor better than 4 is unclear, but



(a) An interval jobs instance on which Kumar and Rudra's algorithm can do poorly ($B = 2$).



(b) The busy time of Kumar and Rudra's algorithm for the instance in (b) can be arbitrarily close to twice that of the optimal solution ($B = 2$).

Figure 1.5: Busy time instances on which known algorithms do poorly.

the evidence mildly suggests that such an argument would be non-trivial.

Our contribution to this body of work is a 3-approximation for the non-interval job setting. The bound is acquired by a non-trivial 3-approximation for the interval jobs case, which is then extended to the general case via techniques akin to those of Khandekar et al. [59]. The 3-approximation for interval jobs is obtained via the algorithm GreedyTracking that greedily identifies whole *tracks* of machines at a time. A track is a subset of jobs whose feasible regions are disjoint. Specifically, GreedyTracking iteratively identifies among the unassigned jobs a subset $\mathcal{T}$ of disjoint jobs that maximizes the sum of job lengths in $\mathcal{T}$, denoted the *span* of $\mathcal{T}$. This approach,

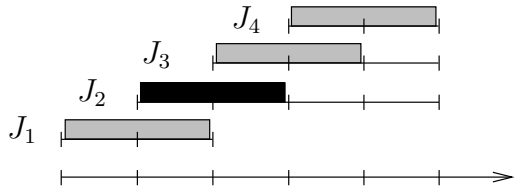
though greedy in nature, is less myopic than algorithms that assign individual jobs to machines one-by-one. The analysis of GreedyTracking depends on the fact that each identified track has span at least half that of the remaining set of unscheduled jobs. This property is crucial to the establishment of the algorithm's bound; the rest of the proof follows from lowerbounds on an optimal solution.

We also show that in the case of preemptive busy time scheduling, there is an exact algorithm for unbounded B. (When B is unbounded, all busy time solutions use a single machine. In this sense, the result for unbounded B falls under both the active time and busy time models.) The underlying idea of this algorithm is reminiscent of LazyActivation's analysis and is perhaps most easily illustrated by the following observation: because there is no parallelism constraint, every job of earliest deadline d_1 without loss is scheduled as late as possible in an optimal solution, i.e., so that it ends precisely at its deadline. Furthermore, later jobs can also be scheduled here (at least partially) at no additional cost. This result in turn can be extended to a 2-approximation for minimizing busy time with bounded B.

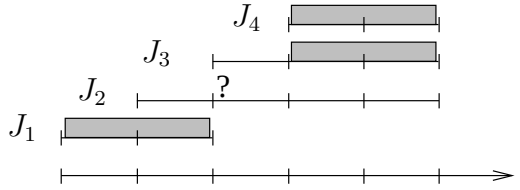
1.5 Truck versus Oven Batch Models

It is worth explicitly noting a point that until now has subsisted as an unstated subtlety. Under some batch models, one requires jobs belonging to the same batch to start at the same time. Here, the notion of a "batch" is clear; one batch is distinct from another. The motivating shipping problem at the beginning of this chapter falls under this model, which we call the *truck* batch model: once the proverbial truck has departed, no other jobs can be added to it.

In other contexts, the batch model may permit the system to process jobs simultaneously without enforcing that they start at the same time. This is akin to a pizza restaurant with an oven that can bake B pizzas at once: it is acceptable to open the oven and start baking a new pizza even if the baking of another pizza has already started. This is the *oven* batch model; here, the notion of a "batch" is less clear and



(a) Under the oven model, the minimum active time is five.



(b) Under the truck model, the instance is infeasible.

Figure 1.6: Truck versus oven batch models (B is two).

batch capacity B serves more as a bound on the system's parallelism.

Thus, the truck model is more restrictive than the oven model. An example highlighting this distinction is depicted in Figure 1.6. Each of the four non-preemptive jobs has length two. Under the oven model, job J_2 is permitted to start at slot 2 while job J_1 is in the midst of finishing. However, under the truck model, the instance is infeasible: job J_2 cannot start until after slot 2, but must finish before slot 4.

The distinction between truck and oven batch models is vacuous when time is slotted and jobs are unit length: all feasible schedules implicitly start jobs within their batches at the same time. The active time result for unit length jobs without integral release times and deadlines operates under the truck model. On the other hand, the preemptive results for arbitrary-length jobs fall strictly under the oven model and not the truck model, as do the busy time results.

1.6 Min-Edge Cost Flow Framework for Covering Problems

Flow networks have played a pivotal role in the design of algorithms, with an impact spanning several areas of computer science and operations research, from network design and computer vision to scheduling and routing [1]. In particular, many important covering problems are special cases of the fundamental minimum-edge cost flow problem (MECF), including active time scheduling. Several of them have direct connections to applications outside of computer science; from understanding glycoprotein formation [49] to identifying galaxies and quasars via spectroscopic data [72], the effects of good algorithms for covering problems are far-reaching. We investigate the relationship between MECF and active time scheduling as well as other well-studied covering problems; in this process, we develop powerful heuristics and optimizations and empirically demonstrate their merits in scheduling contexts and other covering settings.

Recall the minimum-edge cost flow problem: G is a directed graph given by $G = (V, A)$ with a specified source s and sink t. Each arc a has a corresponding integral capacity $c(a)$ and price $\kappa(a)$. Given a target flow value f^*, the goal is to find a flow function on G realizing at least f^* units of flow from s to t with minimum cost. The difficulty of MECF is that it adheres to the fixed cost model, under which any positive flow sent across arc a incurs the entire cost $\kappa(a)$ of the arc. In other words, if A' is the subset of arcs along which positive flow is sent, the total cost of the flow is $\sum_{a \in A'} \kappa(a)$.

Our study pertains to the following framework of MECF, henceforth denoted *Cov-MECF* . The graph G is such that $G - \{s, t\}$ is a bipartite graph (X, Y, E), s has arcs only to X and t has arcs only from Y. (All edges in E are oriented from X to Y.) In addition, the capacities $c(a)$ are unit for $a \in E$. The arcs (y, t) also have costs κ_y for all $y \in Y$; the prices of all other arcs are zero. See Figure 1.7(a). For ease of notation, we will denote the capacities for edges from s to $x \in X$ as c_x and capacities

from $y \in Y$ to t as c_y.

Several covering problems fit into this framework, including active time scheduling as well as the following canonical problem. In Set Cover, there is a ground set $\mathcal{U} = \{u_1, \ldots, u_n\}$ of elements and a collection $\mathcal{S} \subseteq 2^{\mathcal{U}}$ of subsets $S_1, \ldots, S_m$. The goal is to find a minimum cardinality subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\cup_{i \in \mathcal{S}'} S_i = \mathcal{U}$. An $O(\log n)$-approximation exists for Set Cover [56, 71, 20], that greedily selects the set that minimizes the cost to marginal benefit ratio until all ground elements are covered. Feige [32] provides a matching lower bound. Wolsey extended the $O(\log n)$-approximation to a larger class of covering problems to which all problems considered here belong [84]. The connection between *Cov-MECF* and a natural generalization of Set Cover called Capacitated Set Cover follows.

Capacitated Set Cover. In Capacitated Set Cover (CapSC), each set S_j has a capacity $k(j)$ and the goal is to find the smallest collection of sets, with the additional requirement that set S_j covers at most $k(j)$ elements. This problem can be cast into the *Cov-MECF* framework by creating the following graph G' (depicted in Figure 1.7(b)). There is an "element node" x_i for every ground element $u_i \in \mathcal{U}$, as well as a "set node" y_j for every set $S_j \in \mathcal{S}$. In addition, there is a source s and an edge (s, x_i) for every x_i, with unit capacity and zero cost. Create a sink t, and insert edge (y_j, t) for every y_j, with capacity $k(j)$ and unit cost. Finally, there is an edge (x_i, y_j) if and only if element u_i can be covered by set S_j in the original instance. Identifying a capacity-respecting set cover of minimum size amounts to finding in G' a minimum edge-cost flow of value equal to n.

As this problem generalizes Set Cover, the lowerbounds on the approximation guarantee apply. Wolsey's greedy algorithm is an $O(\log n)$-approximation [84] [3]. We highlight that unlike its uncapacitated counterpart, computing the marginal benefit

[3] Wolsey proved, in fact, it is an $O(\log(\max_j k(j))$-approximation.

of adding a set to a partial cover requires a flow computation.

Despite the dire state of affairs at the theoretical level, in the real world, there remains an undeniable need to solve problems of a set cover flavor. One commonly implemented algorithm for doing so is in fact Wolsey's greedy algorithm. Despite its tight $O(\log n)$ guarantee, studies indicate that in practice, it fares much better. For instance, in their widely-cited result, Kempe et al. [58] examine the influence function in social networks, demonstrating that not only does it exhibit submodularity, Wolsey's greedy algorithm in fact returns solutions that are much closer to the optimal solution than its worst-case bound would imply. The approach has since been applied to a plethora of problems within the social networks literature, ranging from outbreak detection in networks [67] to the spread of information via the blogosphere [55].

The primary disadvantage of Wolsey's algorithm is that each iteration may involve computing the marginal benefit for many candidate sets, requiring a high number of subroutine flow calls. For large instances, this can render Wolsey's algorithm unacceptably slow, limiting the size of the input on which it can be run. This issue has been the subject of considerable investigation within the last decade. Leskovec et al. [67] show that if sets are considered in a particular order, irrelevant computations can be avoided, thereby reducing the total number of flow calls. Their algorithm CELF maintains the spirit of Wolsey's greedy algorithm and was recently improved to CELF++ by Goyel et al. [48]. The concern is also a focus of this study.

Despite the fact that it is a special case of MECF, *Cov-MECF* still subsumes many central problems. One benefit of this formation is that it captures multi-set multi-cover problems. In other words, one can demand that elements be covered multiple times. This is an imperative aspect of many scheduling problems, in particular, when jobs of non-unit length must be scheduled.

Contributions. First, we construct a naïve implementation MinFeas that computes

the minimum feasible solution in a single iteration over the sets, shutting down a set if the resulting instance remains feasible; recall that this is provably a 5-approximation for the active time scheduling problem with arbitrary length jobs. We find that while MinFeas's solutions are not as close to optimal as those of Wolsey's algorithm, they tend to be better than the worst-case guarantee suggests. However, the primary advantage of MinFeas is its speed; the number of flow computations is an order of magnitude less than that of Wolsey's algorithm. We also introduce the more general *LPO* algorithm for problems within the *Cov-MECF* framework. *LPO* is an intelligent variation of MinFeas that inherently complements Wolsey's algorithm. While Wolsey's approach builds a cover from the empty set, *LPO* iteratively trims off sets from the initial feasible cover $\mathcal{C} = \mathcal{S}$. The essence of *LPO* harnesses the fractional optimal solution to guide the selection of the next set to remove. We apply the analogs of Leskovec et al.'s [67] optimizations to *LPO* and empirically compare the performance to that of Wolsey' greedy algorithm. We find that despite its lack of theoretical bounds, on an overwhelming majority of instances, *LPO* returns solutions that are closer to optimal than Wolsey's algorithm. We also discover that the performance verdict is largely dictated by the specific covering problem. In particular, for instances of Capacitated Vertex Cover and Active Time Scheduling (discussed below), *LPO* is faster than Wolsey's algorithm. The reverse is true for general Capacitated Set Cover; however we find that on these instances, the optimization of Leskovec et al. levels their performances rather evenly.

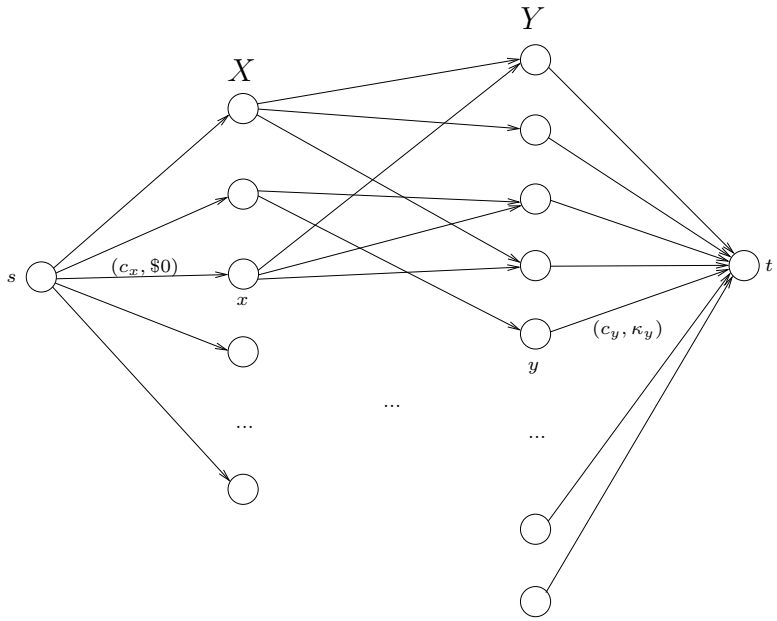
We further propose an additional "freezing" step. Before *LPO* (Wolsey's algorithm, respectively) is run, the LP relaxation of the covering problem is solved. Any set variables that take on integral values in the fractional solution are *frozen* to those values (0 or 1, typically) prior to the invoking of the algorithm proper. Our experiments indicate that more often than not, freezing improves rather than hurts the quality of the solution. More importantly, freezing drastically reduces the number of flow calls, since the majority of the variables attain integral values in the fractional

solution.

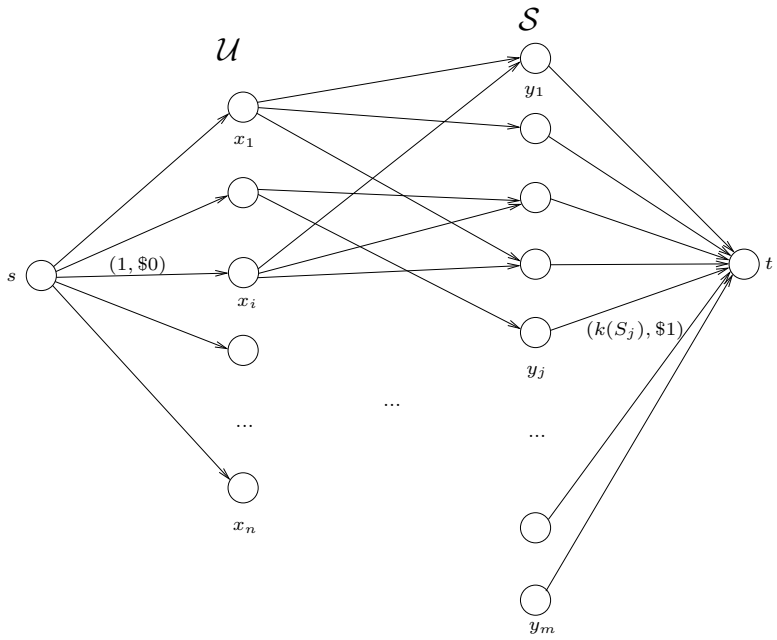
We note that on smaller-sized input, directly solving for the optimal integral solution may be faster than executing the heuristics mentioned above, given their iterative nature. Thanks to the current technology of mixed integer program solvers, it is not until the instance size is rather large that we truly see the cost of computing the optimal solution via exponential heuristics. Our discussion on performance pertains to approaches that scale and thus focuses on the comparison of the polynomial-time heuristics and their optimizations.

1.7 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 contains a discourse of literature that is relevant to the active time model, the busy time model, and the *Cov-MECF* framework. In particular, a discussion ensues of the current state-of-affairs in batch scheduling and also other energy-related scheduling problems. Following that, Chapter 3 explores the simple LazyActivation algorithm for the most restrictive of active time scheduling cases: unit length jobs with slotted time. In Chapter 4, we demonstrate how to minimize the number of batches for unit length jobs when release times and deadlines are taken over the reals and time is not slotted, i.e., batches may begin at any point in time. Chapters 5 and 6 examine two orthogonal generalizations of the unit-length, single release time and deadline case: arbitrary T_i and arbitrary length jobs, respectively. An empirical examination of heuristics for the latter problem is contained in Chapter 7; this investigation pertains to a large class of capacitated covering problems that includes active time scheduling. Finally, results within the busy time model are examined in Chapter 8, and the thesis closes with conclusions in Chapter 9.



(a) The general *Cov-MECF* framework.



(b) Capacitated Set Cover.

Figure 1.7: The *Cov-MECF* framework.

Chapter 2

RELATED WORK

The problems considered in this thesis are in essence scheduling problems. With roots arguably older than the field of computer science itself, the theory of scheduling is extensive. As such, we begin this chapter with a brief survey of classic results for batch scheduling of unit length jobs. Then we peruse the more recent and energy-motivated scheduling literature, specifically the relevant busy time results. Following that is a discussion of the covering problems captured by the *Cov-MECF* framework and their known solutions, both in theory and implementation.

2.1 Parallel and Batch Scheduling Theory

The following problems fall under the oven batch scheduling model, in which a job A classical problem related to our work is the well known "Scheduling unit jobs on B processors with precedence constraints", in which n unit jobs are given with precedence constraints and the goal is to schedule the jobs on B processors to minimize the maximum completion time. Again this can be viewed as minimizing the active time. For arbitrary B, the problem is NP-complete [42]. For fixed B, the problem is known to be W[2]-hard [13]. For the case where $B = 2$, this problem can be solved optimally in polynomial time [34, 35]. Garey and Johnson [43, 41] consider the problem of scheduling unit jobs with integer release times and deadlines with precedence constraints, providing polynomial time algorithms for $B = 2$. In addition, their algorithm finds a schedule with minimum lateness. This was extended to the case of real release times and deadlines by Wu and Jaffar [85], who gave an $O(n^4)$ algorithm. Their primary technique involves computing successor-

tree-consistent deadlines, which effectively upper bounds the latest completion time for each job. Successor-tree-consistency allows the optimal schedule to be computed via a slight variation to Simons forward scheduling algorithm for independent unit length jobs. For scheduling unit length jobs with arbitrary release times and deadlines on B processors to minimize the sum of completion times, Simons and Warmuth [81] extended the work by Simons [80] giving an algorithm with running time $O(n^2B)$ to find a feasible solution. For constant B, the running time is improved in [70]. With regards to the relationship between preemption and non-preemption, Coffman and Garey show that for the classical two-processor scheduling problem, the ratio between the optimal preemptive solution and the optimal non-preemptive one is at most 4/3 [21]. In that problem, there are no release times or deadlines, but there are precedence constraints and the goal is to minimize the makespan.

The type of batch processing considered in this thesis is similar to the work initiated by Ikura and Gimple [52]. Their algorithm is designed to minimize completion time for batch processing on a single machine for the special case of *agreeable*[1] release times and deadlines. Baptiste [9] extended the Ikura and Gimple results to general release times and deadlines and an efficient algorithm was recently given by Condotta et al. [22]. All of these works focus merely on trying to find a feasible schedule (which then can be used as a subroutine to minimize maximum lateness). However in our problem, in addition we wish to minimize the number of batches. Some of these results were improved by Koehler and Khuller [62].

2.2 Scheduling with Chain Precedence Constraints

A notion similar to preemption at integer time points is that of scheduling jobs with *chain* precedence constraints. Precedence constraints are typically modeled as a graph in which there is a vertex for each job, and arc (i, j) exists if and only if job J_i must be

[1]When the ordering of jobs by release times is the same as the ordering of jobs by deadlines.

completed before job J_j begins. Imposing precedence constraints on a set of jobs in general makes scheduling NP-hard. However, for structured cases of constraints, this is not always the case. An instance has chain precedence constraints if the directed precedence graph can be partitioned into a set of disjoint paths ("chains").

One can consider integral-length jobs that can be preempted at integer time points as a special case of chains precedence constraint scheduling. Job j of integral length p_j can be viewed as a chain of p_j unit-length jobs ("links") $k_1, \ldots, k_{p_j}$ in which the release times and deadlines are all identically r_j and d_j, respectively. In this sense, chains scheduling is a more general setting.

Lenstra and Kan [65] proved NP-hardness for scheduling unit-length jobs with chain-like precedence constraints on a sequential machine to minimize the number of late jobs, as well as cumulative weighted tardiness. Du et al. [27] proved that on a parallel machine, it is NP-hard to schedule arbitrary-length jobs with chain-like precedence constraints to minimize makespan. The reduction is from 3-PARTITION and requires jobs within the same chain to have different release times and deadlines. On the other hand, if the jobs are all unit in length, minimizing the makespan under tree-like precedence constraints can be done in polynomial time via Hu's classic algorithm [50]. Several problems for chain-like precedence constraints have been studied over the years, falling on both sides of the P/NP-complete dichotomy. For a more complete picture, the reader is encouraged to consult the Handbook of Scheduling [68].

2.3 Power-aware Scheduling

Power management strategies have been widely studied in the scheduling literature [3, 4, 53, 86, 54, 10]. What follows is a discussion of a few of the most relevant problems.

2.3.1 Scheduling to Minimize Gaps

Baptiste [10] examines a related problem of "min gap" scheduling of unit-length jobs on a single processor to minimize the number of idle intervals; in this model, the algorithm must determine when the processor sleeps. Baptiste gives an optimal dynamic programming algorithm which builds from a dominance property of the optimal offline schedule. Baptiste, Chrobak and Dürr in [11] improve the running time; their algorithm in fact applies to the generalized setting in which jobs have arbitrary processing times. This work was subsequently extended to handle multiple processors by Demaine et al. [24], who also provide an approximation algorithm for the case where each job has multiple intervals in which it can be scheduled. They also give $\Omega(\log n)$ lower bounds on the approximation ratio. The cost function of this lower bound does not apply to the problems studied in this paper.

Demaine and Zadimoghaddam [25] recently considered a different scheduling problem of minimizing cumulative energy consumption for unit-length jobs over multiple processors, Their model is quite general in that the feasible time slots in which each job may be scheduled may not comprise a single time interval. Also, each processor has an arbitrary (unrelated) energy consumption function and can go to a sleep state. demonstrating that the coverage function is submodular, i.e. that the greedy algorithm yields an $O(\log n)$-approximation.

2.3.2 Speed Scaling

Also related is the *dynamic speed scaling* problem, in which the scheduler determines the non-negative speed at which the processor runs. For a single processor, Yao et al. [86] give an exact offline solution that minimizes the total power consumption when the power is a convex function of the speed. Irani et al. [54] study an extended problem in which the machine can also be put into the "sleep" state, during which period no cost is incurred other than the constant wake-up cost. They present a 2-

approximation and an $O(1)$-competitive algorithm in the online setting. The problem was recently shown to be NP-complete and the approximation improved to 4/3 by Albers and Antoniadis [5]. Despite the significant results in [54], its authors acknowledge that a continuous power function is unrealistic; in practice, systems run at a finite number of potential speeds. Our work is the special case in which power is represented by a step function.

Li and Yao [69] consider a discretized version of the problem, in which the system may operate at one of a finite number of speeds. Their algorithm is exact and runs in time $O(dn \log n)$, where d is the number of possible speeds. The main idea is to first partition the jobs, and then to determine the speeds of these jobs, partition by partition. However, their model is not quite the same as ours; despite the discretization of the speeds, they still assume that the underlying power function is convex and therefore cannot capture the step from zero to positive speed.

2.3.3 Unrelated Machine Activation Scheduling

One general scheduling model is considered by Khuller et al. [60], in which jobs must be assigned to machines, each machine has an associated cost and capacity, and subject to staying within a budget, the goal is to purchase a set of machines and assign jobs to them to minimize the maximum load on any machine. This problem is clearly Set Cover-hard. Khuller et al. give a greedy algorithm achieving twice the optimal makespan, when it is permitted to violate the budget by a factor of $O(\ln n)$.

2.3.4 Busy Time Scheduling

The busy time problem was initiated by Winkler and Zhang [83] in the context of *fiber minimization* in optical network design. They proved that the problem was NP-complete when jobs are non-preemptive, via a simple reduction from Circular

Arc Graph Coloring[2]. Approximations for several special cases of the busy time problem are known. Alicherry and Bhatia [6] gave the first and currently best 2-approximation for the special case pertaining to interval jobs, i.e. jobs whose lengths are equal to the difference between their deadlines and release times. Their result is cast as a line coloring problem, in which intervals (demands) span parts of a line, and the goal is to find a valid coloring of the demand set that minimizes a cost function that captures objectives like busy time. Rudra and Kumar [64] later provided another 2-approximation, drawing from techniques by Gergov [45] for the Dynamic Storage Allocation problem. However, neither of their results immediately extends to busy time scheduling of non-interval jobs, as discussed in the Introduction.

Flammini et al. [33] give a 4-approximation for interval jobs, via the simple algorithm FirstFit, that in non-increasing order of job length, greedily assigns jobs to machines of lowest index. They also provide stronger guarantees for some special cases, e.g., proper instances where no job's window is contained within that of another. Khandekar et al. [59] consider the busy time generalization in which jobs are not necessarily interval jobs, and further more, each job has a demand on the machine that processes it. The corresponding batch constraint is that at no time may a machine be working jobs whose cumulative demand exceeds B. In the standard busy time problem, jobs have unit demand. The authors develop a 5-approximation for this case; when the demands are unit, the approximation guarantee of their algorithm becomes four. The main idea is to first cluster the jobs under the assumption that each batch has infinite capacity and then to fix this as the position of the job by modifying the release time and deadline, thus converting it to an interval job. The algorithm then partitions the jobs by demand into two categories and uses a greedy method to schedule the jobs. One significant contribution of their work is an extension of FirstFit's analysis to the busy time problem on non-interval jobs, yielding the

[2]The batch capacity B is sometimes denoted μ or g in the fiber optics literature.

first approximation for the general case. The busy time results in this thesis exhibit a similar property (see Chapter 8). A number of applications are mentioned in [59, 33].

We refer the reader to surveys [53, 3, 4] for a more comprehensive overview of the latest scheduling results for power management problems.

2.4 Capacitated Covering Problems

We close this chapter with a discourse on the covering problems other than active time scheduling that fit within the *Cov-MECF* framework.

2.4.1 Capacitated Set Cover

From a theoretical perspective, there are few unanswered questions about Capacitated Set Cover. Wolsey gave an $O(\log n)$-approximation [84] and it is asymptotically tight [32].

Nonetheless, variations of this problem arise in practice. Interest in variations of the Set Cover problem extends to the Sloan Digital Sky Survey project, an effort toward surveying the history of astronomy. Lupton et al. [72] studied a special geometric case of Capacitated Set Cover in the context of collecting telescopic snapshots of the universe. The telescope may be pointed to a "disc"-shaped area in the sky to capture a high-resolution image for processing; the restriction is that the number of galaxies per image that can be processed cannot exceed a given constant. The goal is to process each object of interest via the smallest number of snapshots. Finding good solutions to this problem has implications on the order of millions of dollars. For this NP-hard problem, Lupton et al. propose a fast, greedy heuristic whose solutions are in general about 20 percent better than their near-uniform counterparts. Their method is faster than standard IP approaches in the operations research literature

and in practice, yields solutions of impressive quality.[3]

2.4.2 Capacitated Vertex Cover

In the standard Vertex Cover problem, we are given a graph $G = (V, E)$ with weights $w(v)$ on the vertices. The goal is to find a subset $S \subseteq V$ of minimum weight such that every edge is incident to a vertex in S. This is a special case of Set Cover in the sense that each element can be covered by only two sets.

In the capacitated version, vertex v has capacity $k(v)$, i.e. an upperbound on the number of edges that can be assigned to it. The goal is to find a minimize size cover that respects the capacities. When the capacities are soft, then multiple copies of a vertex v may be selected; each copy is permitted to cover up to $k(v)$ edges, and the cost $w(v)$ is incurred for each copy of v in the vertex cover. This problem is NP-hard. Guha et al. [49] give a primal-dual 2-approximation for the soft capacities case. For the unweighted hard capacities version, a 2-approximation was given by Gandhi et al. [40], improving upon the initial bound of three given by Chuzhoy and Naor [19]; the weighted case is Set Cover-hard. Extensions to hypergraphs were made by Khuller and Saha [79]. As far as we know, there is no experimental research suggesting heuristics with performance better than twice the optimal solution.

For uncapacitated vertex cover, Nemhauser and Trotter proved that there exists a optimal solution to the relaxation of vertex cover's LP in which each variable takes on values $0, \frac{1}{2}$, or 1 [76]. While this result does not extend to the capacitated setting, we empirically observe that often times, several variables attain integral values in the fractional relaxation, yielding a significant performance boost via the freezing optimization.

---

[3]We note that significant theoretical attention has been given to a special case of geometric set cover, in which ground elements are points and sets are objects in Euclidean space [75, 82, 15, 30, 7, 8]. Many of the results generalize to multicovering variants, but under a different notion of capacitated sets.

All of these problems are captured in the *Cov-MECF* framework. Despite its wide array of applications, MECF is in general less understood compared to its well-studied min-cost flow counterpart, even when restricted to the special case considered here. MECF is a classic NP-hard problem, listed as problem [ND32] in Garey and Johnson's *Computers and Intractablity: A Guide to the Theory of NP-Completeness* [42]. Krumke et al. [63] showed that unless $NP \subseteq DTIME(n^{O(\log \log n)})$, there exists no $(1-\epsilon)\ln f^*$-approximation for any constant $\epsilon > 0$. Even et al. [29] established that unless $NP \subseteq DTIME(n^{polylog(n)})$, there does not exist a $2^{\log^{1-\epsilon} n}$-approximation, for any constant $\epsilon > 0$.

Chapter 3

ACTIVE TIME: A BASIC PROBLEM

In this chapter, we examine a fairly restrictive case of active time scheduling, in which every job has length one, a release time, a deadline, and in which time is slotted, i.e. jobs can start and finish only at integral time points. In Graham's standard notation, this problem is $P|p_j = 1, r_j, d_j| \sum_t a_t$. We introduce LazyActivation, a polynomial-time greedy algorithm that minimizes active time exactly.

3.1 Preliminaries

Before getting to the result itself, let us first establish preliminary assumptions that will be used for the remainder of this chapter.

Lemma 1. *For all active time instances I having unit length jobs with release times and deadlines, one can convert I into an instance I' with equivalent minimum active time, such that in I', $d_{\max}$ is $O(n)$.*

Proof. Order the jobs by release time, i.e., such that $0 \leq r_1 \leq \ldots \leq r_n$. Without loss of generality, no deadlines are after $r_n + n$, and thus $d_{\max} \leq r_n + n$. Thus, it suffices to give an I' in which $r_n = O(n)$.

Create I' iteratively from I to have the property that $r_j < j$ via the following procedure. Consider jobs j in non-decreasing order of release time. Intuitively, if job j is released at or after slot j, then only $j-1$ jobs have been released by slot j and no schedule needs all j slots for them. In particular, if for all jobs $j' < j$, the invariant $r_{j'} < j'$ holds, then slot j is without loss not active in a feasible schedule (for example, left-shifting any feasible schedule will yield this property). More generally, suppose that $r_j = j + k \geq j$ for some $k \geq 0$. Then in interval $[0, j + k)$, only $j - 1$ jobs

have been released and slots $j, j+1, \ldots, j+k$ are inactive in any left-shifted feasible schedule. Thus, we can remove or "contract" them: decrease by $k+1$ all release times and deadlines that occur at or after slot $j+k$.

One can implement this procedure trivially in $O(n^2)$ time by decrementing release times or deadlines one-by-one, possibly $O(n)$ times. To improve the running time to $O(n)$ plus the time to sort, instead maintain the amount by which each subsequent release time or deadline should be decreased. This amount is equivalent to the number of slots that have been "contracted" and can only increase as more release times are processed. The precise definition of this process is in Algorithm 1. In Line 3, $r_j - \textbf{numContracted}$ is what the modified release time would be if no more slots were contracted; if this is greater than or equal to j, then the invariant does not hold and more slots should be contracted (the $k+1$ slots immediately preceding r_j, to be precise). Lines 10 through 12 keep deadlines consistent with slots that are contracted because of a future release time.

□

Henceforth, we will assume that $d_{\max} = O(n)$. For the moment, we will also assume the instance is feasible, since this is easy to check by an EDF or flow computation. Infeasibility will be addressed at the end of this chapter.

3.2 Algorithm LazyActivation

We now provide a high level description of the algorithm, followed by pseudo-code and the proof of optimality. First, we say job j is *available* at t if it is unscheduled and $t \in [r_j, d_j)$. Denote the distinct deadlines by $d_{i_1} < d_{i_2} < \ldots < d_{i_k} = d_{\max}$, and let S_p be the set of jobs with deadline d_{i_p}. Then $S_1 \cup S_2 \ldots \cup S_k$ is the entire job set.

Jobs are processed in two phases. In Phase I we scan the jobs in order of *decreasing* deadline. We do not schedule any jobs, but only modify the deadlines of jobs to create

```
1  numContracted ← 0;
2  for j ← 1 to n do
3      if r_j − numContracted = j + k for some k ≥ 0 then
4          numContracted ← numContracted + k + 1;
5      end
6      r_j ← r_j − numContracted;
7      forall the deadlines d ∈ [r_j, r_{j+1}) do
8          d ← d − numContracted;
9      end
10     forall the deadlines d ∈ [r_{j−1}, r_j) do
11         d ← max(d, r_j);
12     end
13 end
```

Algorithm 1: Converting an instance so that $d_{\max} = O(n)$.

a new instance, whose optimal solution is equivalent to that of the original instance. The desired property of the new instance is that at most B jobs will have the same deadline. Process the time slots from right to left. At slot D, let S be the set of jobs that currently have deadline D. From S, select $\max(0, |S| - B)$ jobs with earliest release times and decrement their deadlines by one. If $|S| \leq B$ then we do not modify the deadlines of jobs in S. (Note that a job may have its deadline reduced multiple times since it may be processed repeatedly.)

After the first phase, let $d'_{\ell_1} < \ldots < d'_{\ell_{k'}}$ be the set of distinct modified deadlines, and let S'_p refer to the jobs of modified deadline d'_{ℓ_p}. We now describe Phase II in which jobs are actually scheduled. Initially all jobs are *unscheduled*. As the algorithm assigns jobs to active time slots, we change the status of jobs to *scheduled*. Once a job is scheduled, it remains scheduled. Once a slot is declared active, it will remain active for the entire duration of the algorithm.

Our algorithm, in general, schedules jobs in increasing order by deadline. As we will see shortly, this is not quite true since the algorithm may schedule some jobs of later deadline if there is available space earlier. The algorithm considers S'_p in increasing order of p. At the time at which S'_p is processed, if there are still unscheduled jobs of S'_p, the algorithm opens slot d'_{ℓ_p}. If fewer than B jobs get scheduled there, the algorithm attempts to schedule additional available jobs as *fillers*. It selects from the available jobs via EDF, until slot t is full or no more jobs are available. The formal description of the algorithm does exactly this, but via a preprocessing step that allows for faster running time and cleaner analysis.

3.2.1 Formal Algorithm Description

Let $B > 0$ be the number of jobs that the system can satisfy in a single time slot. For every job j, denote j's release time and deadline by r_j and d_j, respectively. We index jobs in order of non-decreasing deadline, i.e., such that $d_1 \leq d_2 \leq \ldots \leq d_n = d_{\max}$. Normalize to 0 the earliest release time of any job. Then without loss, all feasible

schedules are active only within the interval $[0, d_{\max}]$.

In the first phase the algorithm scans the jobs from right to left in decreasing deadline order. At each step it considers the set of jobs with a common deadline and leave up to B jobs with the latest release times untouched. It modifies the deadlines of the rest of the jobs in this set, decrementing them each by one, and then continue processing the jobs.

The algorithm for the second phase simultaneously maintains a set W of active time slots and a set J of satisfied jobs, both of which are initially empty. In each iteration, it looks at the unsatisfied job j^* of earliest deadline, i.e., $j^* = \arg\min_{j \notin J} d_j$. Let d^* be j^*'s deadline, and let J^* be the set of all unsatisfied jobs with the same deadline d^*. The algorithm activates the latest possible time slot that can satisfy J^* and adds it to W. Only one slot is needed to satisfy J^* since $|J^*| \leq B$.

The algorithm first assigns all jobs in J^* to the newly activated time slot t; i.e., jobs in J^* are added to J. If J^* consists of strictly fewer than B jobs, then the algorithm greedily assigns to the remaining space "filler" jobs that are available and unscheduled, again making selections based on EDF. These filler jobs will also be added to J.

The following pseudocode formalizes the above description of the second phase. *SelectFillers* takes as input the set of active slots W and the set of scheduled jobs J, returning the set of filler jobs J'. These jobs are then added to the set of scheduled jobs.

3.2.2 Analysis of the Algorithm

It is easy to implement this algorithm in time $O(n \log n)$ using standard data structures. What is not completely obvious is why it computes an optimal solution. Suppose the initial instance I is transformed by Phase I to a modified instance I'. We prove the following properties about an optimal solution for I'.

```
1 J ← ∅; W ← ∅ ;
2 while ∃ j ∉ J do
3     d* ← arg min_{j∉J} d_j ;
4     J* ← {j ∉ J : d_j = d*} ;
5     W ← W ∪ d* ;
6     J ← J ∪ J* ;
7     J' ← SelectFillers(W, J) ;
8     J ← J ∪ J' ;
9 end
```

Algorithm 2: Lazy Activation Algorithm

```
1 Choose available fillers based on EDF to fill the B − |J*| empty spots.
```

Algorithm 3: SelectFillers(W, J)

Proposition 1. *An optimal solution for I' has the same number of active slots as an optimal solution for the original instance I.*

Proof. It is easy to see that any feasible solution for I' is feasible for I since preprocessing only created a more constrained instance in the transformation; each job's window in I' is a subset of its window in the original instance I. We now argue that a solution for I can be transformed to a solution for I' using the same number of slots. Suppose a feasible schedule σ for I is infeasible for I' due to a job x. In other words, σ schedules job x after its modified deadline in I'. We can argue this step by step, by showing that decrementing the deadline of a single job does not change the optimal solution; since the modification is done by a sequence of such operations, the optimal solution is preserved. Assume that the deadline of x was reduced by one. In the instance I', we have B jobs with deadline d_x, out of which at most $B - 1$ jobs can be scheduled with x. Hence there is at least one job scheduled earlier whose

deadline is still d_x. Since its release time cannot be before the release time of x, we can exchange these two jobs. This makes the schedule feasible for I', and this establishes the proposition. □

Proposition 2. *Without loss of generality, an optimal solution for I' only opens slots that are deadlines.*

Proof. Consider the optimal solution activating the least number of non-deadline slots. Among the active slots that are not deadlines, let t be the right-most (i.e., latest) one. Suppose X is the set of jobs that are assigned to t, and suppose we shifted X as late as possible, keeping the jobs of X together while maintaining the feasibility of every job in it. X cannot end up at a deadline slot; otherwise, we would have reduced the number of non-deadline slots by one. Therefore, X must end up "stuck" at a non-deadline slot, unable to shift right further due to the presence of other jobs. More specifically, denote by t' the earliest deadline after the non-deadline slot at which X got stuck. Then, there must be other jobs assigned to each slot between the slot where X got stuck and t'. These jobs are in some sense "blocking" X. Let Y denote the set of blocking jobs that are scheduled at deadline slot t'. From the set $X \cup Y$, schedule at t' the B jobs with the earliest deadlines. Since at most B jobs have deadline t', all the remaining jobs of $X \cup Y$ have later deadlines and the argument can be repeated. □

Theorem 1. *On instance I', there exists an optimal solution in which the slot d_1 is filled with available jobs of earliest deadline.*

Proof. Due to the above propositions, the jobs with deadline d_1 are all scheduled in time slot d_1 in the optimal solution, without loss of generality. In addition, to fill the remaining slots, the optimal solution again without loss selects available jobs of earlier deadline. Otherwise one can exchange the jobs to achieve this property. □

The correctness of Lazy Activation immediately follows by inductively applying Theorem 1.

3.2.3 Infeasible Instances

In this section, we consider the behavior of the Lazy Activation algorithm on instances for which it is impossible to schedule all jobs within their individual windows of feasibility. We show that Lazy Activation maximizes the number of jobs satisfied. In fact, we see that it does so in the fewest number of active timeslots. Denote by $\mathcal{S}_{LA}$ the schedule returned by Lazy Activation. In Phase I, it is possible for a job's deadline to be decremented all the way to its release time; in this case, we say that the job's window has *collapsed*.

Proposition 3. *On infeasible instances, Lazy Activation maximizes the number of jobs satisfied.*

Proof. One can use an argument similar to Proposition 1 to show that even if Phase I collapses the windows of some jobs, it does not change the maximum number of jobs that can be scheduled. For completeness' sake, we detail it here. As before, we argue step-by-step that each decrement changing the instance from I to I' does not affect the maximum throughput. Suppose a deadline d_x of job x is reduced by one. Let σ be a feasible schedule on I achieving maximum throughput on I. We can transform σ into σ' that is feasible on instance I' and that satisfies the same number of jobs. If σ does not schedule job x at d_x, then σ is already feasible on I'. Suppose σ schedules x at d_x. Then σ can do at most $B-1$ other jobs at d_x. In I', there are B jobs with deadline d_x and release time at least r_x. Thus, there exists a job j which is not scheduled by σ at d_x and which has release time at least r_x. Modify σ by swapping jobs j and x. (If j was not scheduled in σ, then schedule j and not x.) The new schedule satisfies the same number of jobs and is also feasible for I'.

Thus, the modification of deadlines in Phase I does not change the maximum number of jobs which can be scheduled. In particular, jobs whose windows collapse in Phase I without loss of generality are also dropped in some throughput-maximizing schedule. In Phase II, Lazy Activation schedules every job whose window has not

collapsed, either at its deadline or earlier, i.e., as a filler. Therefore, Lazy Activation maximizes the number of jobs satisfied. □

Proposition 4. *If Lazy Activation's schedule $\mathcal{S}_{LA}$ satisfies $n' < n$ jobs in k active slots, then any schedule $\mathcal{S}$ satisfying n' jobs does so in at least k active time slots.*

Proof. Suppose that Lazy Activation collapses $\kappa = n - n'$ jobs, denoted $j_{a_1} \dots j_{a_\kappa}$. Let α_t be the number of jobs that had deadline t at the start of the iteration in which Phase I processed t as the deadline. Deadlines are decremented precisely when $\alpha_t > B$. For each collapsed job j_{a_i}, we will identify an interval I_{a_i} of excess demand by intuitively "unrolling" the iterations of Phase I starting from the point of collapse. More formally, set t' to the latest slot such that every slot $t \in [r_j, t']$ is such that $\alpha_t > B$. Define I_{a_i} to be $[r_j, t')$. Notice that t' is the original deadline of some job. Let $J_{a_i} = \{j : [r_j, d_j) \subseteq I_{a_i}\}$. Then the collapsed job $j_{a_i} \in J_{a_i}$ and also $|J_{a_i}| > B \cdot |I_{a_i}|$. In fact, J_{a_i} consists exactly of two types of jobs: jobs which have collapsed and $B \cdot |I_{a_i}|$ jobs which have not collapsed. Lazy Activation schedules the latter job set in I_{a_i} at B jobs per slot.

Now partition the original instance (J, T) into two subinstances (J_1, T_1) and (J_2, T_2), where $J_1 = \bigcup_{i=1}^{k} J_{a_i}$ and $T_1 = \bigcup_{i=1}^{k} I_{a_i}$. Obviously jobs of J_1 cannot be scheduled in slots of T_2 by definition. Notice that since $\mathcal{S}$ maximizes throughput, it necessarily schedules J_2 in T_2: suppose there exists a job $j \in J_2$ that is scheduled by $\mathcal{S}$ in some interval I_{a_i}. Then since $|J_{a_i}| > B \cdot |I_{a_i}|$, there is some job of J_{a_i} that is missed by $\mathcal{S}$. Since it is possible to schedule all jobs of J_2 only in slots of T_2 ($\mathcal{S}_{LA}$ is such an example), missing that many jobs of J_{a_i} was unnecessary. This contradicts the fact that $\mathcal{S}$ maximizes throughput.

Then the active time $A(\mathcal{S})$ of $\mathcal{S}$ ($\mathcal{S}_{LA}$, respectively) can be decomposed into two components: the active time $A_1(\mathcal{S})$ spent satisfying J_1 and the active time $A_2(\mathcal{S})$

spent satisfying J_2. Then,

$$\begin{aligned} A(\mathcal{S}) &= A_1(\mathcal{S}) + A_2(\mathcal{S}) \\ &\geq A_1(\mathcal{S}_{LA}) + A_2(\mathcal{S}_{LA}) \\ &= A(\mathcal{S}_{LA}) \\ &= k \end{aligned}$$

where the inequality follows from the facts that (1) Lazy Activation minimizes active time on feasible instances and (2) there are exactly $B \cdot |I_{a_i}|$ jobs contained in each I_{a_i} that have not collapsed and Lazy Activation schedules all of them. Thus whenever Lazy Activation is active in T_1, it schedules B jobs per slot. □

Chapter 4

ACTIVE TIME FOR NON-SLOTTED TIME

In this chapter, we consider a generalization of the previously discussed problem. Suppose that the unit length jobs have release times and deadlines over the reals and that we want to minimize the number of *batches* in the schedule, where a batch is a set of at most B jobs which all start and finish at the same time and where the system can work on at most one batch at a time. This objective is slightly more restrictive than active time; even so, scheduling unit length jobs with integer release times and deadlines to minimize active time is clearly a special case of this, since in the former problem, the batch property holds without loss.

We describe a simple dynamic program which determines in $O(n^8)$ time the minimum number of batches needed to non-preemptively satisfy all jobs, i.e., $1|p\text{-batch}, B < n, r_i, d_i, p_i = 1|K$, where K is the number of batches in the schedule. The dynamic program (and therefore, the notation) is similar to that found in [9], on which several dynamic programming results in this area of scheduling are based. Suppose throughout that jobs are listed in non-decreasing order by deadline, i.e., $d_1 \leq d_2 \leq \ldots \leq d_n$.

Definition 3. *For job k and for interval $[t_\ell, t_r]$ where $r_k \in [t_\ell, t_r]$ and $t_r + 1 \leq d_k$, let the set of jobs $U_k(t_\ell, t_r) = \{j \leq k : r_j \in [t_\ell, t_r]\}$. Also let $U_0(t_\ell, t_r) = \emptyset$ for all intervals $[t_\ell, t_r]$.*

Figure 4.1 shows $U_k(t_\ell, t_r)$. Observe that the jobs in $U_k(t_\ell, t_r)$ must be scheduled entirely in $[t_\ell, d_k)$ in any feasible schedule. We restrict ourselves without loss to the space of schedules obeying the EDF Principle: for any pair of jobs i and j where $i < j$, either job i is scheduled before or with job j, or r_i is after the point at which job j is scheduled. Indeed, if a schedule contains a pair of jobs i and j for which this

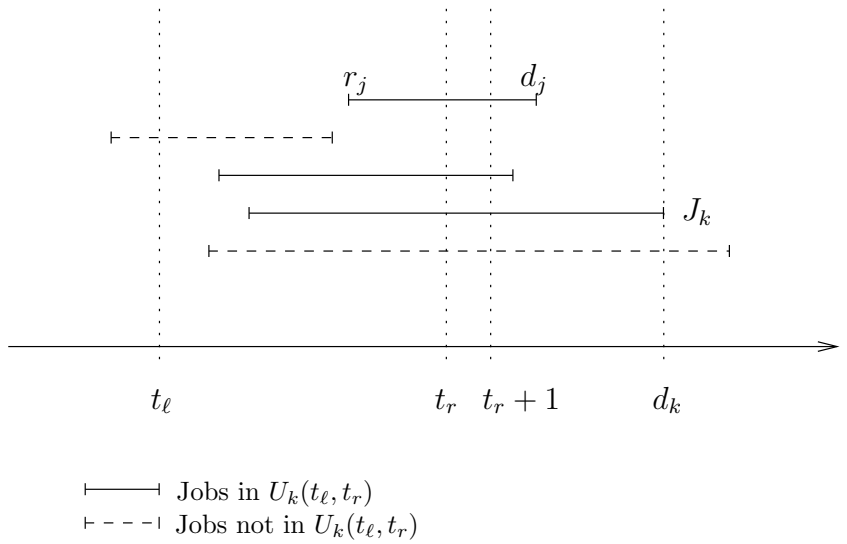


Figure 4.1: Unit jobs of $U_k(t_\ell, t_r)$.

does not hold, then one can swap them in the schedule without affecting feasibility or the number of batches.

Definition 4. *Let $P(k, t_\ell, t_r, \mu_r)$ be the minimum number of batches required to schedule $U_k(t_\ell, t_r)$ such that (1) all batches start within $[t_\ell + 1, t_r]$, (2) at most $B - \mu_r$ of these jobs are scheduled in the batch starting at t_r, and (3) if $\mu_r > 0$, then the batch starting at t_r does not count toward the number of batches.*

If such a schedule does not exist, e.g., if $k > 0$ and $t_\ell + 1 > t_r$, then let $P(k, t_\ell, t_r, \mu_r) = \infty$. Let $P_t(k, t_\ell, t_r, \mu_r)$ be the value of $P(k, t_\ell, t_r, \mu_r)$ subject to job k being scheduled in a batch that starts at time t. Then

$$P(k, t_\ell, t_r, \mu_r) = \min_t P_t(k, t_\ell, t_r, \mu_r)$$

Per Baptiste's observation in [10], it is enough to iterate over a set of $O(n^2)$ possible start times for batches. For a fixed t, one can compute $P_t(k, t_\ell, t_r, \mu_r)$ as follows. Let $L = \{j \leq k : r_j \in [t_\ell, t]\}$ be the jobs of $U_k(t_\ell, t_r)$ which are released before or at t. Similarly, let $R = \{j \leq k : r_j \in (t, t_r]\}$ be the jobs of $U_k(t_\ell, t_r)$ released after t. The

observation above implies that if $i \in L$, then i is scheduled before or at t. On the other hand, if $i \in R$, then job i must be scheduled after t. Let $k_L \in \arg\max_{i \in L \setminus \{k\}} d_i$, or 0 if $L \setminus \{k\} = \emptyset$. Similarly, let $k_R \in \arg\max_{i \in R} d_i$, or 0 if $R = \emptyset$.

If $t = t_r$, then $L = U_k(t_\ell, t_r)$, R is empty and k_L (if positive) is the second-to-latest deadline in $U_k(t_\ell, t_r)$. Then,

$$P_t(k, t_\ell, t_r, \mu_r) = \begin{cases} P(k_L, t_\ell, t_r, \mu_r + 1) & \text{if } 0 < \mu_r < B \\ 1 + P(k_L, t_\ell, t_r, 1) & \text{if } \mu_r = 0 \\ \infty & \text{otherwise} \end{cases} \tag{4.1}$$

Consider $t < t_r$. If $\mu_r > 0$, then for $t \in (t_r - 1, t_r)$, $P_t(k, t_\ell, t_r, \mu_r) = \infty$, since starting batches at t and at t_r will violate the constraint that at any given time, there is at most one batch running. Otherwise, i.e., if $t < t_r - 1$ or $\mu_r = 0$,

$$P_t(k, t_\ell, t_r, \mu_r) = 1 + P(k_L, t_\ell, t, 1) + P(k_R, t, t_r, \mu_r)$$

Finding the optimal number of batches is equivalent to computing $P(n, \min_i r_i - p, d_n - 1, 0)$. Since there are $O(n)$ jobs and $O(n^2)$ interesting times to consider, the total time is $O(n^7 B) = O(n^8)$. This was most recently improved to $O(n^3)$ by Koehler and Khuller via a non-DP approach [62].

4.1 Maximizing Throughput in $\mathcal{K}$ Batches

On instances where it is impossible to schedule every job, one can apply a technique of the same vein to maximize throughput. In particular, one can maximize throughput subject to an upper bound $\mathcal{K}$ on the number of batches. Define $P(k, t_\ell, t_r, \mu_r, \kappa)$ to be the maximum number of jobs of $U_k(t_\ell, t_r)$ which can be scheduled in at most κ batches starting in $[t_\ell + 1, t_r]$, where as before, if $\mu_r > 0$, one can schedule up to $B - \mu_r$ jobs in the batch starting at t_r without incurring cost toward the batch budget κ.

To understand the computation of $P(k, t_\ell, t_r, \mu_r, \kappa)$, let t be the starting time of the batch which satisfies job k (if such a time exists). Consider where the rest of

$U_k(t_\ell, t_r)$ may be satisfied within $[t_\ell+1, t_r]$. Denote by α (β, respectively) the number of them which start in $[t_\ell + 1, t]$ ($(t, t_r]$, respectively). Observe that $\beta \leq \kappa - \alpha$.

There are four cases. Suppose job k is not satisfied in the optimal solution. Then $P(k, t_\ell, t_r, \mu_r, \kappa) = P(k', t_\ell, t_r, \mu_r, \kappa)$ where $k' = \arg\max_{i \in U_k(t_\ell, t_r) \setminus \{k\}} d_i$, i.e., the job of next latest deadline in $U_k(t_\ell, t_r)$. (If $U_k(t_\ell, t_r) = \emptyset$, then $k' = 0$.) If job k is satisfied in the schedule, then define L, R, k_L and k_R as in Section 3. Suppose that job k is satisfied in the batch starting at $t = t_r$. Then

$$P(k, t_\ell, t_r, \mu_r, \kappa) = \begin{cases} 1 + P(k_L, t_\ell, t_r, 1, \kappa - 1) & \text{if } \mu_r = 0 \\ 1 + P(k_L, t_\ell, t_r, \mu_r + 1, \kappa) & \text{if } 0 < \mu_r < B \\ -\infty & \text{otherwise} \end{cases} \tag{4.2}$$

If job k is satisfied in a batch which starts at $t \leq t_r - 1$, or if $\mu_r = 0$ and $t \leq t_r$, then let α be the number of batches used to satisfy jobs in L; this includes the batch satisfying job k, so $\alpha > 0$. Then $\beta = \kappa - \alpha$ is the budget on the number of batches used to satisfy R. Note that for the case where $\mu_r = 0$, if R is not empty, then any α corresponding to a feasible schedule will be strictly less than κ, i.e., $\beta > 0$.

$$P(k, t_\ell, t_r, \mu_r, \kappa) = \begin{cases} \max_{0<\alpha\leq\kappa}(1 + P(k_L, t_\ell, t, 1, \alpha - 1) & \text{if } 0 \leq \mu_r < B \\ \qquad + P(k_R, t, t_r, \mu_r, \kappa - \alpha)) & \\ P(k, t_\ell, t_r - 1, 0, \kappa) & \text{otherwise} \end{cases} \tag{4.3}$$

Finally, if $\mu_r > 0$ and $t \in (t_r - 1, t_r)$, then $P(k, t_\ell, t_r, \mu_r, \kappa) = -\infty$, since the machine cannot work on multiple batches at the same time. $P(k, t_\ell, t_r, \mu_r, \kappa)$ is the maximum over these values. Since there are $O(n)$ possible values for k, $O(n^2)$ possible values for both t_r and t_ℓ, and $O(n)$ possible values for μ_r, and since $\kappa = O(n)$, there are $O(n^7)$ possible values of $P(k, t_\ell, t_r, \mu_r, \kappa)$ that need to be computed. Computing each one costs $O(n)$, yielding a total running time of $O(n^7\mathcal{K})$. We note that if the time model is slotted, then there are $O(n)$ possible values for both t_r and t_ℓ, and the total running time is instead $O(n^5\mathcal{K})$. This algorithm can be extended to the

case in which each job J_k has an associated profit v_k and the goal is to schedule a profit-maximizing set of jobs in $\mathcal{K}$ slots.

Chapter 5

ACTIVE TIME FOR MULTIPLE FEASIBLE WINDOWS

We now shift our attention to the more general problem in which a job may have multiple feasible regions, i.e., T_i need not be comprised of a single interval. For any $B \geq 3$, we show that the problem is NP-hard and discuss its relationship to other classic covering problems. We also develop an efficient algorithm for the two processor case ($B = 2$).

5.1 Connections to Other Problems

Capacitated Vertex Cover. There is an interesting relationship between the activation problem of unit-length jobs with multiple feasible windows and the problem of vertex cover with capacity constraints. In the latter problem, we are given a graph $G = (V, E)$ and a capacity function $k(v)$ for each vertex. The goal is to pick a subset S of vertices and to determine an assignment of edges to vertices in S, so that each edge is assigned to an incident vertex in S, and so that each vertex v has at most $k(v)$ adjacent edges assigned to it. When multiple copies of a vertex may be chosen to be part of the cover, a primal-dual 2-approximation is given in [49]. In the hard capacities version of the problem (VCHC), a bounded number of copies of a vertex may be selected. There is a 2-approximation using LP rounding [40], improving the previous bound of 3 given in [19].

In the special case of the activation problem in which each job has exactly two feasible time slots, there is an equivalence with VCHC with uniform capacities $k(v) = B$. Since jobs need not have adjacent feasible time slots, one can view time as a set of slots rather than an ordering of slots. Then, the equivalence follows: the slots are

the vertices and the jobs are the edges. For any valid capacity-respecting vertex cover of size C, there exists a corresponding activation schedule of cost C (and vice-versa). One implication of our result is that we can solve VCHC optimally when $k(v) = 2$. Furthermore, the 2-approximation for VCHC in [40] applies to the activation problem when each job has two feasible slots.

There is a similar relationship between VCHC on hypergraphs and the activation problem where jobs may have more than two feasible slots, and for size g hyper edges, an $O(g)$-approximation has recently been developed [79].

Capacitated K-center. Another previously studied problem is the K-center problem with load capacity constraints [12, 61]. Given an edge weighted graph satisfying the metric property, the goal is to pick K nodes (called centers) and assign each vertex to a chosen center that is "close" to it. We should not assign more than B nodes to any chosen center and want to minimize the value $d_{\max}$ such that each node is assigned to a center within distance $d_{\max}$ of it. Previous work culminated in a 5-approximation for the problem where multiple copies of the same node may be chosen as a center. If only one copy can be chosen then the bound goes up to 6.

There is the following correspondence between this and the following instance of the activation problem. First, guess the maximum value $d_{\max}$ and create the (unweighted) graph $G^{d_{max}}$ induced by edges having weight at most d_{max}. Then create a job J_i and a timeslot t_i for each node i in the original graph G, and let job J_i be feasible in slot t_j if and only if node i is within distance d_{max} of node (or center) j in G, i.e., if edge (i, j) exists in $G^{d_{max}}$. (So t_i should be a feasible slot for J_i.) Then, finding K centers to which all n vertices can be assigned in $G^{d_{max}}$ is equivalent to opening K timeslots and feasibly scheduling all n jobs in them so that no slot is assigned more than B jobs. Since we can solve the activation problem when $B = 2$, we can optimally solve the K-center problem when at most two nodes can be assigned to a given center.

5.2 Proof of NP-hardness for B=3

We prove that the activation problem for jobs of arbitrary T_i is NP-hard when $B = 3$ via a reduction from 3-Exact-Cover, which is known to be NP-hard [42]. Given a collection X of n elements and a collection of subsets $S_1, \ldots, S_m$, each containing exactly three elements from X. Is there a sub-collection of exactly $\frac{n}{3}$ subsets that exactly cover all of X? We can view this problem as a bipartite graph where one side has the elements X and the other side has a vertex for each subset S_i, and edges denote membership. This problem maps to the question of finding a dominating set of size exactly $\frac{n}{3}$ where we are only allowed to select from the side containing subsets. The relationship with the scheduling problem is now obvious: selecting a subset is akin to activating a certain time slot where the set X corresponds to a collection of jobs. The edges specify in which time slots a job can be scheduled. Since the subsets each have size exactly three, there is an exact cover of size $\frac{n}{3}$ if and only if there is a schedule with exactly $\frac{n}{3}$ active slots.

We note that there is an $O(\log n)$ approximation for this problem (for any B) by an easy reduction to a submodular cover problem. This result follows from the classical covering results due to Wolsey [84].

5.3 Algorithms for $B = 2$

The state of affairs is less pessimistic when B is two. Optimal approaches arise out of a proper understanding of the connection between minimum active time and b-matchings, a natural generalization of matchings, on a special graph. We present a simple algorithm and in the process highlight the key intuitions that underlie the faster algorithm.

5.3.1 A Sidebar on b-matchings

First, we review b-matchings.

Definition 5. *Let $G = (V, E)$ be a graph with degree constraints $b : V \to \mathbb{Z}^+$. A* ***b-matching*** *of G is a subset of edges such that the number of matched edges incident on a node v is at most $b(v)$.*

Standard matchings are b-matchings with unit degree constraints. Just as with standard matchings, the *cardinality* of a b-matching is the number of edges in it, and a *perfect* b-matching is one in which each node v is incident to *exactly* $b(v)$ matched edges. Algorithms for finding b-matchings of maximum cardinality are well-known and, as with finding maximum matchings, depend heavily on the identification of augmenting paths. As with standard matchings, if a perfect matching on a graph exists, algorithms for maximum cardinality matchings will find it. In his classic result, Gabow [36] gave a $O(nm \log n)$ time algorithm for finding a b-matching, where n is the number of vertices and m the number of edges in the graph.

Often in the literature, the terms *b-matching* and *degree-constrained subgraph* are used interchangeably. Due to the strong connection with matchings, we will stick to the former as our term of choice.

5.3.2 A First Algorithm

Since B is two, in any schedule, each slot is assigned either no jobs (i.e., is idle), one job or two jobs. Since every job must be scheduled, minimizing the active time is equivalent to minimizing the number of slots that are active yet working on only one job, henceforth denoted as *half-full* slots. If k denotes the minimum number of such slots, then the active time is necessarily

$$\frac{n-k}{2} + k = \frac{n+k}{2}$$

Thus, identifying an optimal schedule can be reduced to finding the minimum k such that there exists a feasible schedule with exactly k half-full slots. For a given instance I and a guess k on the number of half-full slots, construct a graph $G_k(I)$ as follows

(see Figure 5.1). $G_k(I)$ has a *job node* x_i for every job J_i in I; in Figure 5.1, they are the set of nodes in the top left, denoted $\mathcal{J}$. $G_k(I)$ also has a *slot node* y_t for every time slot t from 1 to $d_{\max}$. Job nodes have edges only to slot nodes. In particular, job node x_i has an edge to slot node y_t if and only if job J_i is feasible in slot t. Each job node x_i also has degree constraint $b(x_i) = 1$; a matched edge will correspond to that job being assigned to that slot.

In addition, for every slot node y_t, there are two *peripheral nodes* $z_{t,1}$ and $z_{t,2}$. The graph has edges $(y_t, z_{t,1}), (y_t, z_{t,2})$, and $(z_{t,1}, z_{t,2})$ (see Figure 5.1). For every t and $i = 1, 2$, node $z_{t,i}$ has degree constraint one while node y_t has degree constraint two.

Finally, the graph has a special node H with an edge to each slot node y_t and degree constraint k.

Lemma 2. *There exists a perfect b-matching on $G_k(I)$ if and only if there exists a feasible schedule with exactly k half-full slots.*

Proof. Take any active time schedule $\mathcal{S}$ with exactly k half-full slots and construct the b-matching by matching each job node x_i to the slot node corresponding to the slot in which $\mathcal{S}$ scheduled job J_i. Since $\mathcal{S}$ schedules all jobs, the degree constraints of the job nodes are met with equality. For each half-full slot t in $\mathcal{S}$, also include the edge (y_t, H) in the b-matching. Since $\mathcal{S}$ has exactly k half-full slots, the degree constraint of H is also met with equality. For each slot t, if t is active in $\mathcal{S}$, its degree constraint is already met, so match $z_{t,1}$ to $z_{t,2}$. Finally, for each idle slot t, match y_t to both $z_{t,1}$ and $z_{t,2}$. This concludes the description of a valid perfect b-matching.

Now take any perfect b-matching on $G_k(I)$ and consider the schedule $\mathcal{S}$ in which a job J_i is satisfied in slot t if and only if (x_i, y_t) is in the b-matching. This is a feasible schedule since job nodes have degree constraint one and no slot node can be matched to more than two nodes.

Since the b-matching is perfect, each slot node y_t must be incident to exactly two matched edges. By construction, this means that if y_t is matched to one of its

peripheral nodes, then it must be matched to both peripheral nodes, and thus to no job nodes; these slots are idle in $\mathcal{S}$. On the other hand, if y_t is matched to neither peripheral node, then the peripheral nodes are matched to each other and the slot node is matched either to two job nodes or to a job node and H; regardless, these slots are active in $\mathcal{S}$, exactly k of which are satisfying only one job, by H's degree constraint k. Thus, $\mathcal{S}$ is a feasible schedule with k half-full slots and active time $\frac{n+k}{2}$. □

We seek the minimum value of k for which there exists a feasible schedule with exactly k half-full slots. This may require finding maximum b-matchings on as many as $\min\{n, d_{\max}\} = O(n)$ graphs to find the desired k and the corresponding optimal schedule. By the classic result due to Gabow [36], perfect b-matchings can be found in time $O(n'm' \log n')$ where n' is the number of vertices and m' the number of edges. In this case, each $G_k(I)$ has $O(n + d_{\max})$ nodes and $O(n \cdot d_{\max})$ edges, where n is the number of jobs and $d_{\max}$ is the number of slot nodes. Thus, the total running time to find the minimum active time schedule is $O(n^2 \cdot d_{\max}(n + d_{\max}) \log(n + d_{\max}))$.

5.3.3 A Faster Algorithm

One can circumvent searching for the correct k by looking for a maximum cardinality b-matching on a slightly modified graph. Construct the following graph $G'(I)$. The vertices are comprised only of job nodes $\mathcal{J}$ and slot nodes $\mathcal{T}$. The edge set is

$$\{(x_i, y_t) : \text{job } J_i \text{ can be scheduled in time slot } t\} \cup \{(y_t, y_t) : y_t \in \mathcal{T}\}.$$

The degree constraint $b(x_i)$ for every job node $x_i \in \mathcal{J}$ is one, and $b(y_t)$ is two for every slot node $y_t \in \mathcal{T}$. By definition, a loop (y_t, y_t) contributes two to the degree of node y_t. Let n be the number of jobs and m the number of given pairs (i, t) where job J_i can be scheduled at time t. Observe that G' has $O(m)$ vertices and $O(m)$ edges, since we can assume every slot node y_t is incident to some edge of the form (x_i, y_t).

Any b-matching maps to a schedule: a matched edge between job node x_i and slot node y_t corresponds to job J_i being scheduled in slot t. Conversely, every schedule maps to a b-matching whose vertex set is the union of nodes corresponding to scheduled jobs and time nodes. In particular, the edges of the b-matching are either of the form (x_i, y_t) where job J_i was scheduled in slot t, or they are loops (y_t, y_t) for each inactive time slot node $y_t \in \mathcal{T}$. Also any b-matching M that covers ι jobs and contains λ loops has cardinality

$$|M| = \iota + \lambda. \tag{5.1}$$

Lemma 3. *A maximum cardinality b-matching M of G' minimizes the number of active slots used by any schedule of $|V(M) \cap \mathcal{J}|$ jobs, where $V(M)$ is the set of nodes incident to matched edges.*

Proof. Let M be a maximum cardinality b-matching. M covers $\iota = |V(M) \cap \mathcal{J}|$ jobs. (5.1) shows no schedule of ι jobs contains more loops than M. M contains the loop (y_t, y_t) precisely when t is not active. Thus M minimizes the number of active slots for schedules of ι jobs. □

In G', the number of loops in any simple path P can be at most $\lceil |P|/2 \rceil$. So, in the algorithms for finding maximum cardinality b-matchings and maximum matchings, an augmenting path still has length $O(n)$. A maximum cardinality b-matching on G' can be found in time $O(\sqrt{n}m)$, via non-bipartite matching approaches. We describe two ways to handle the loops.

The first approach reduces the problem to maximum cardinality matching in a graph called the MG graph. To do this, modify G' by replacing each time slot vertex y_t with two vertices y_{t_1}, y_{t_2}. Replace each edge (x_i, y_t) by two edges from x_i to the two replacement vertices for y_t. Finally replace each loop (y_t, y_t) by an edge joining the two corresponding replacement vertices.

A b-matching M corresponds to a matching M' in a natural way: if a time slot node y_t is "active" in M then M' matches corresponding edges of the form (x_i, y_{t_i}),

for $i = 1, 2$. If the loop (y_t, y_t) is in M then M' matches the replacement edge (y_{t_1}, y_{t_2}). Thus it is easy to see that a maximum cardinality b-matching corresponds to a maximum cardinality matching of the same size. The cardinality matching algorithm of Micali and Vazirani [73, 37] gives the desired time bound. (This approach works for the versions of the problem that we will consider on unit jobs; it does not work when the jobs have longer length since a pair of edges (x_i, y_{t_1}), (x_i, y_{t_2}) might get matched.)

The second approach involves an algorithm for maximum cardinality b-matching on general graphs. If such an algorithm does not handle loops directly, modify G' to BG by replacing each loop (y_t, y_t) by a triangle $(y_t, y_{t_1}, y_{t_2}, y_t)$, where each y_{t_i} is a new vertex with degree constraint $b(y_{t_i}) = 1$ for $i = 1, 2$. A b-matching in G' corresponds to a b-matching in the new graph that contains exactly $|\mathcal{T}|$ more edges, one from each triangle. The cardinality algorithm of Gabow and Tarjan [38] gives the desired time bound.

In any case, either approach yields ways to find maximum cardinality b-matchings on graphs with loops. What is less obvious is how to find a maximum cardinality b-matching in which every job node is matched. Toward that end, denote by ι^* the greatest number of jobs that can be scheduled in I. For feasible instances in which all jobs can be scheduled, $\iota^* = n$. Clearly ι^* can be computed in time $O(\sqrt{n}m)$ by finding a maximum cardinality b-matching on the bipartite graph $G^\star$ formed by removing all loops from G' [31].

Theorem 2. *Among all schedules satisfying the maximum number of jobs (ι^*), one that minimizes the number of active slots can be found in time $O(\sqrt{n}m)$.*

Proof. The algorithm first finds a b-matching M_0 matching ι^* jobs, e.g., by computing one in $G^\star$. With respect to G', M_0 is a valid b-matching, but not necessarily one of maximum cardinality. The algorithm then converts M_0 to the desired b-matching via an algorithm for finding a maximum cardinality b-matching on G', but using M_0 as

the initial b-matching.

The correctness of this approach follows from the fact that the b-matching algorithm works by augmenting paths. This implies that as the initial b-matching M_0 is enlarged to the final b-matching, no vertex's degree decreases. In particular, once a job node is matched, i.e., in M_0, it remains matched even as M_0 is augmented. Thus the final solution schedules the same jobs as M_0, and by Lemma 3, corresponds to a schedule minimizing the number of active slots among all schedules satisfying ι^* jobs.

Using M_0 as the initial b-matching does not affect the time bound, since the algorithm begins by finding augmenting paths of length 1, i.e., an arbitrary set of edges satisfying the degree constraints [38, 73]. □

The proof indicates that the choice of which ι^* jobs to schedule is irrelevant – the minimum number of active slots for a schedule of ι^* jobs can be achieved using *any* set of ι^* jobs that can be scheduled together.

Our algorithm can be extended to different variations of the active time problem. For example the following corollary models a situation where power is limited.

Corollary 1. *For any given integer budget α on the number of active slots, a schedule for the maximum number of jobs (ι^*) using at most α active slots can be found in time $O(\sqrt{n}m)$.*

Proof. Start by constructing the b-matching M^* of Theorem 2 that schedules ι^* jobs and contains, say, λ^* loops. Let τ_1 and τ_2 be such that M^* schedules τ_i time slots with i jobs. Then $\iota^* = \tau_1 + 2\tau_2$. Let $\Delta = \alpha - (\tau_1 + \tau_2)$ be the number of time slots by which M^* has exceeded the active time budget, and consider the following cases for Δ.

Case $\Delta \leq 0$: M^*'s active time is already less than or equal to α.

Case $0 < \Delta \leq \tau_1$: Choose any Δ time slots scheduling just one job. Unschedule those jobs in the schedule. This gives a new schedule with exactly α active slots. In

the corresponding b-matching, remove those matched edges between job nodes and slot nodes, and add the corresponding self-loops. The resulting b-matching M has $iota^* - \Delta$ jobs and $\lambda^* + \Delta$ loops. (5.1) shows $|M| = |M^*|$. Since fixing the number of active slots, e.g., to α, fixes λ, (5.1) also shows M schedules the greatest possible number of jobs.

Case $\tau_1 < \Delta$: In M^*, unschedule the jobs that are scheduled in the τ_1 time slots with one job, or scheduled in $\Delta - \tau_1$ other time slots (chosen arbitrarily from the τ_2 slots with two jobs). The result is a schedule with α active slots, each processing two jobs. This schedule obviously satisfies the greatest possible number of jobs. □

The proof shows that the b-matching M^* is easily turned into a table (of $\tau_1 + \tau_2$ entries) that gives the desired schedule for every budget value of α.

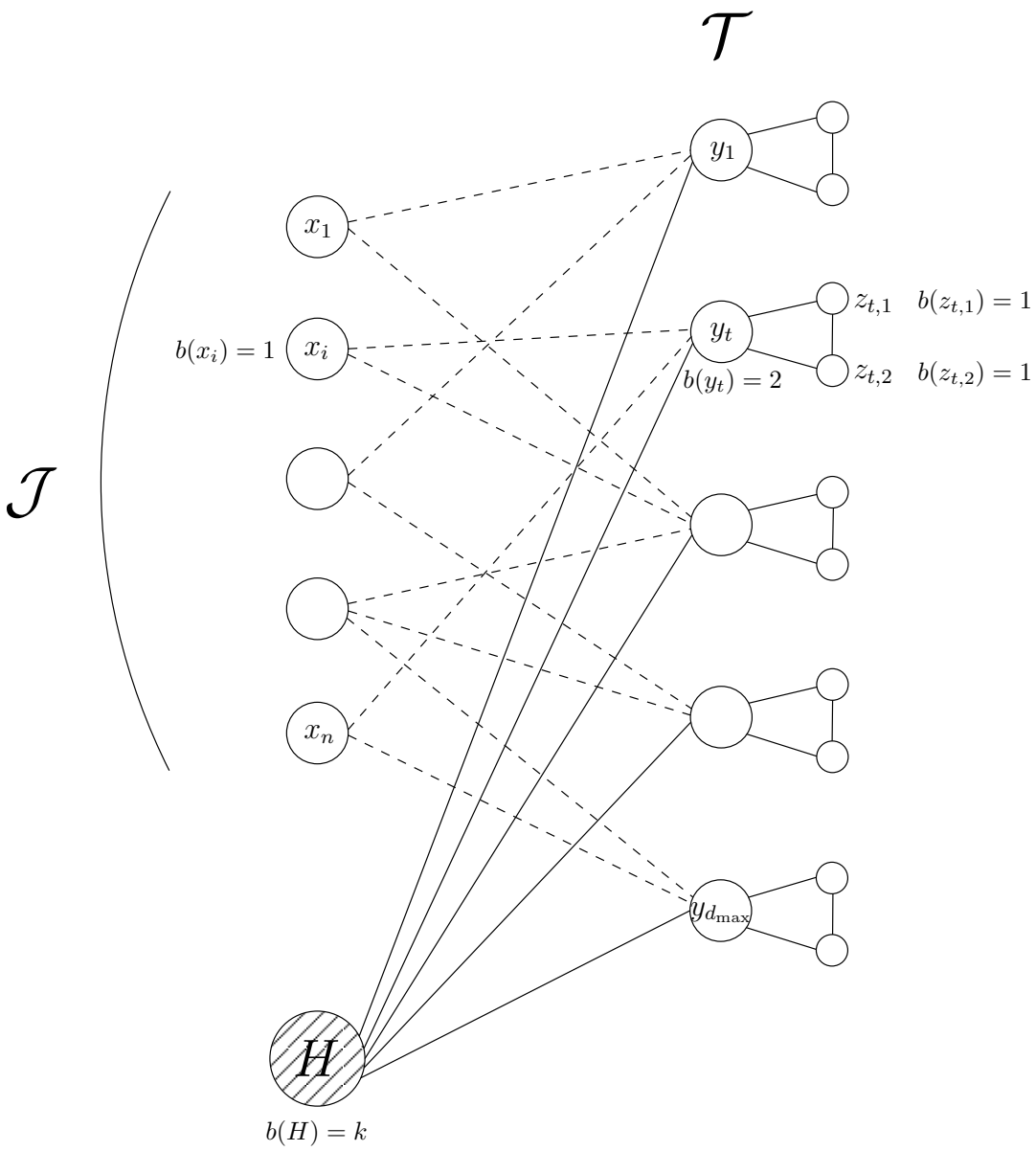


Figure 5.1: Graph $G_k(I)$ and the degree constraint b for each node.

Chapter 6

ACTIVE TIME FOR ARBITRARY LENGTH JOBS

6.1 Non-preemptive Scheduling

Consider an active time instance in which jobs may have varying length p_i. For convenience, we will assume the lengths are integral. If the jobs cannot be preempted, then even the feasibility problem is easily NP-hard from 3-PARTITION. For the sake of completeness, we describe it here. In 3-PARTITION, one is given a sequence of integers $x_1, \ldots, x_{3m}$ so that each integer x_i is between $\frac{X}{2}$ and $\frac{X}{4}$ where $mX = \sum_{i=1}^{3m} x_i$. The goal is to partition the integers into exactly m triples so that each triple sums exactly to X. For each integer x_i, create a job of length x_i, with release time 0 and deadline X. Then for $B = m$, there exists a partition of the elements if and only if there is feasible schedule of all the jobs.

6.2 Preemptive Scheduling

For the case in which jobs can be preempted at integer boundaries, we give a simple 5-approximation.

A job j is said to be *live at* t if $t \in [r_j, d_j)$. A slot is *active* if at least one job is scheduled in it and *inactive* otherwise. We say that an active slot is *almost-full* if there are at least $B/2$ jobs assigned to it and *non-full* otherwise.

A feasible solution $\mathcal{S}$ is a set of activated time slots and an assignment of jobs to slots such that every job j is assigned to p_j active slots and no active slot is assigned more than B jobs. (In fact, once the set of slots has been determined, the assignment of jobs to slots can be found via a max flow computation.) For a feasible solution to

be minimal means that there is no way to close down any active slot and still satisfy the entire job set.

Definition 6. *A minimal feasible solution $\mathcal{S}$ is **left-shifted** means that for every job j, if j is scheduled at slot t, then j is also scheduled in every possible earlier non-full slot t', i.e., t' such that $r_j \leq t' < t$.*

For every feasible solution $\mathcal{S}$, it is easy to find a minimal feasible solution $\mathcal{S}'$ that activates the same set of slots as $\mathcal{S}$.

Definition 7. *Job j is **non-full-rigid** with respect to a solution $\mathcal{S}$ means that j is scheduled in every non-full slot t' where it is live (of which there is at least one).*

Observation: For every non-full slot t in a minimal feasible solution $\mathcal{S}$, there must exist a non-full-rigid job in $\mathcal{J}_t$. If there had existed a t such that every job j in $\mathcal{J}_t$ had another non-full slot t_j in its window in which it is not scheduled, then $\mathcal{S}$ would not be minimal, since we could shut down t, scheduling each job j of $\mathcal{J}_t$ in t_j instead.

Theorem 3. *Let $\mathcal{S}$ be any left-shifted minimal feasible solution. Then the number of active slots in $\mathcal{S}$ is at most 5 times OPT, where OPT denotes the cost of an optimal solution.*

Proof. Active almost-full slots of $\mathcal{S}$ are charged directly to the quantity $\frac{2\sum_j p_j}{B}$, which is at most $2 \cdot OPT$. We now identify a subset of jobs to which we can charge the cost of $\mathcal{S}$'s active non-full slots; this subset should have the property that at most three of its jobs are live at any slot.

For each non-full slot t from 0 to $d_{\max}$, identify a non-full-rigid job $j \in \mathcal{J}_t$; if there are multiple non-full-rigid jobs, favor the one of later deadline (breaking ties arbitrarily). Charge to j the slot t as well as all subsequent non-full slots $t' \leq d_j$. Then, move to the earliest uncharged non-full slot and repeat. Denote by $\mathcal{J}^\star$ the set of jobs charged during this process. The key property is that at most three jobs of $\mathcal{J}^\star$

are live at any point in time; this is proven below in Lemma 5. In other words, $\mathcal{J}^\star$ can be partitioned into three subsets $(\mathcal{J}_A^\star, \mathcal{J}_B^\star, \mathcal{J}_C^\star)$ so that jobs in the same partition have disjoint feasible intervals. This implies that for each partition $\mathcal{J}_i^\star$,

$$\sum_{j \in \mathcal{J}_i^\star} p_j \leq OPT$$

Then, letting $NF(\mathcal{S})$ denote the number of non-full slots in $\mathcal{S}$ and $AF(\mathcal{S})$ the number of almost-full slots, the total cost of $\mathcal{S}$ is bounded by

$$\begin{aligned}
|\mathcal{S}| &= AF(\mathcal{S}) + NF(\mathcal{S}) \\
&\leq 2\ OPT + \sum_{j \in \mathcal{J}^\star} p_j \\
&= 2\ OPT + \sum_{j \in \mathcal{J}_A^\star} p_j + \sum_{j \in \mathcal{J}_B^\star} p_j + \sum_{j \in \mathcal{J}_C^\star} p_j \\
&\leq 2\ OPT + 3\ OPT \\
&= 5\ OPT
\end{aligned}$$

□

We now prove that $\mathcal{J}^\star$ has at most three jobs live at any point in time.

Lemma 4. *No more than two jobs of $\mathcal{J}^\star$ are live at any non-full slot t.*

Proof. Suppose by way of contradiction that there existed a non-full slot t and three non-full-rigid jobs of $\mathcal{J}^\star$ that were all live at t. Denote them by $j_{\alpha_1}, j_{\alpha_2}$ and j_{α_3}, where $d_{\alpha_1} \leq d_{\alpha_2} \leq d_{\alpha_3}$ (breaking ties in a way that is consistent with the charging procedure). See Figure 6.1. The gray-colored slots are either idle or full; either way, they are not non-full. The latest non-full slot t_1 in j_{α_1}'s window must have been charged to j_{α_1}. Then, the charging procedure moves to the next non-full slot $t_2 > d_{\alpha_1}$. By assumption, j_{α_2} and j_{α_3} have been released before t_2, so both jobs are candidates for being chosen as the next job to charge. The charging procedure selects

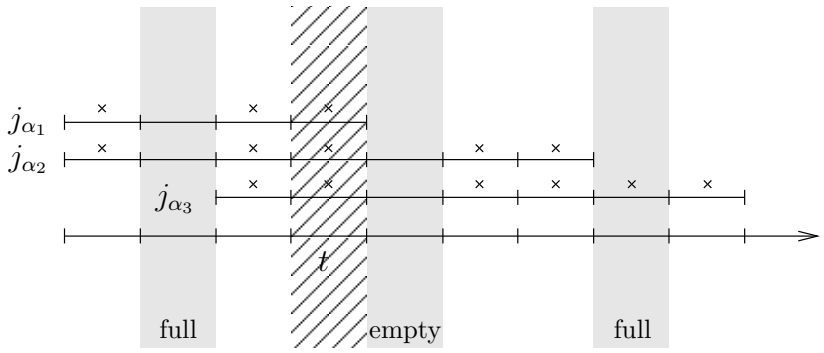


Figure 6.1: Hypothetical three non-full-rigid jobs live at the same time t. "x"'s denote one unit of a job being scheduled in that time slot.

the non-full-rigid job of the latest deadline. In particular, this implies that job j_{α_2} should not be charged by any non-full slots, a contradiction.

□

Lemma 5. *No more than three jobs of $\mathcal{J}^\star$ are live at any slot t.*

Proof. If t is a non-full slot, the claim follows trivially by Lemma 4. Suppose t is not a non-full slot, i.e., t is either inactive, or it is active and almost-full. Consider the jobs of $\mathcal{J}^\star$ that could be live at t: they must also be live either at the closest non-full slot t' to t's left, or the closest non-full slot t'' to t's right.

At most two jobs of $\mathcal{J}^\star$ can be live at t', so it is enough to argue that at most one job of $\mathcal{J}^\star$ can be live at t but not at t'. Consider all non-full-rigid jobs that are live at t but not at t' (see Figure 6.2). Each such job has release time in $[t'+1, t)$, and one of them, denoted $j^\star$, must have maximum deadline. No other non-full-rigid job that is live at t but not at t' can be in $\mathcal{J}^\star$: $j^\star$ will be picked over it. Thus, at most one job of $\mathcal{J}^\star$ can be live at t but not at t'.

□

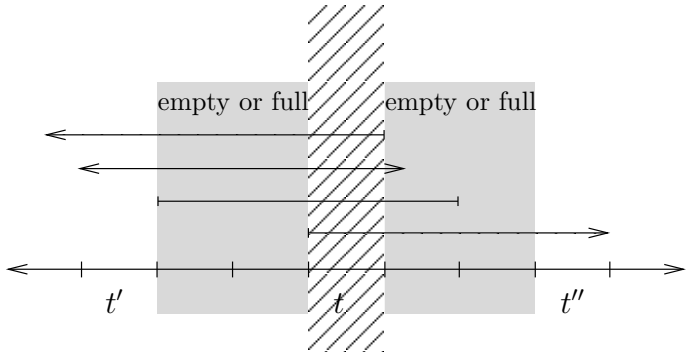


Figure 6.2: At most three jobs of $\mathcal{J}^*$ can be live in any slot.

6.2.1 Problem Complexity

The complexity of this problem remains unresolved. Natural extensions of the polynomial time LazyActivation algorithm immediately run into a couple of difficulties when jobs may vary in length. For example, it is no longer clear how to do the preprocessing step (Phase I) of LazyActivation: when more than B jobs share a deadline, whose deadlines should be decremented? Part of this difficulty is highlighted in the instance in Figure 6.3. Since jobs J_5, J_6, and J_7 tie up slots 8 and 9, it would be a severe mistake to decrement the deadline of job J_2. On the other hand, if job J_5 did not exist, then it would be a mistake to decrement the deadline of job J_1. In general, it may be difficult to decide which subset of jobs should have their deadlines decremented, among a set of jobs sharing the same deadline; one cannot afford to consider every possible subset. Furthermore, it is not clear how to choose the filler jobs in Phase II, since for similar reasons, identifying the "most constrained" jobs can be a challenge.

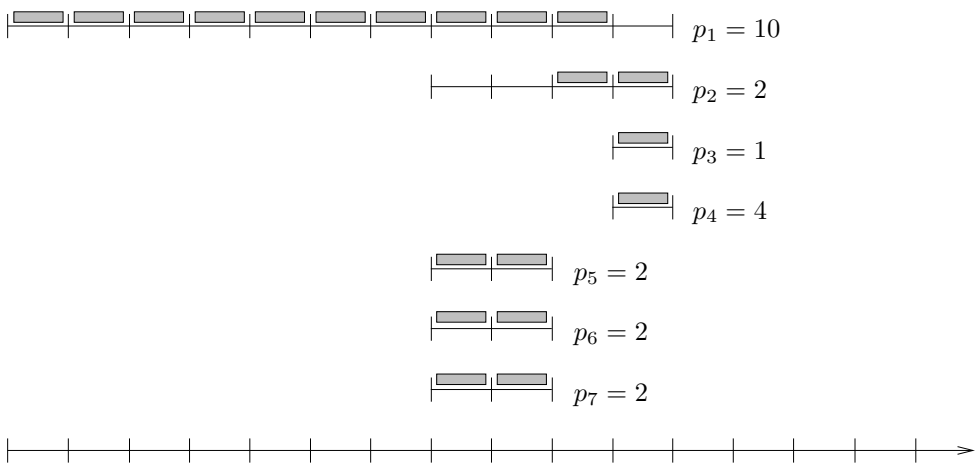


Figure 6.3: The difficulty of pre-processing for arbitrary-length jobs ($B = 3$). Recall that p_i denotes the length of job J_i.

Chapter 7

EMPIRICAL STUDIES OF CAPACITATED COVERING PROBLEMS

This chapter discusses the empirical value of various covering problems captured by the *Cov-MECF* framework defined in the Introduction (see Section 1.6). We remind the reader of the *Cov-MECF* framework: given a graph G and two nodes $s, t \in G$, $G - \{s, t\}$ forms a bipartite graph (X, Y, E). The arcs of G are from s to nodes of X; nodes of Y to t; or from X to Y as specified by E (see Figure 1.7). In addition, the capacities $c(a)$ are unit for $a \in E$ and otherwise depend on the particular covering problem at hand. Further, for every $y \in Y$, the arc (y, t) has cost κ_y; all others have cost zero.

The *Cov-MECF* framework captures not just the active time problem, but also Capacitated Set Cover and the following well-studied special case.

Capacitated Vertex Cover. In the standard Vertex Cover problem, we are given an undirected graph with weights $w(v)$ on the vertices. The goal is to find a subset S of vertices of minimum weight such that every edge is incident to a vertex in S. This is a special case of Set Cover in the sense that each element can be covered by only two sets.

In the capacitated version, each vertex v has capacity $k(v)$, i.e. an upperbound on the number of edges that can be assigned to it. The goal is to find a minimize size cover that respects the capacities. When the capacities are soft, then multiple copies of a vertex v may be selected; each copy is permitted to cover up to $k(v)$ edges, and the cost $w(v)$ is incurred for each copy of v in the vertex cover. This

problem is NP-hard. Guha et al. [49] give a primal-dual 2-approximation for the soft capacities case. For the unweighted hard capacities version, a 2-approximation is due to Gandhi et al. [40], improving upon the initial bound of three given by Chuzhoy and Naor [19]; the weighted case is Set Cover-hard. Extensions to hypergraphs were made by Khuller and Saha [79]. As far as we know, there is no experimental research suggesting heuristics with performance better than twice the optimal solution.

7.1 Algorithms for Covering

We continue this chapter with a review of Wolsey's greedy algorithm. Then we introduce the *LPO* heuristic. We compare Wolsey's algorithm and *LPO* with the following simple algorithm, denoted MinFeas. Consider each slot t in increasing order, shutting it down as long as doing so does not violate the feasibility of the instance. In Chapter 6, it was proven that for active time scheduling problems, MinFeas is one of many algorithms that are 5-approximate. As we will see, *LPO* is also in this class of 5-approximate solutions, but it is a bit more sophisticated and thus computationally expensive.

Finally, we describe the optimizations applied to these algorithms. Even though for simplicity's sake, the discussion is given in the context of Capacitated Set Cover, the algorithms and optimizations can be applied to other covering problems within this framework.

7.1.1 Wolsey's Algorithm

Wolsey's greedy algorithm begins with the empty cover $\mathcal{C}$ and iteratively adds sets to it until all ground elements can be covered. In each iteration, it selects the set that minimizes the cost to marginal benefit ratio. More formally, it adds to $\mathcal{C}$ the set S that minimizes $w(\mathcal{C}, S)$, with

$$w(\mathcal{C}, S) = \frac{c(S)}{f(\mathcal{C} \cup \{S\}) - f(\mathcal{C})}$$

for $f(\mathcal{C} \cup \{S\}) - f(\mathcal{C}) > 0$ and unbounded otherwise, and where $c(S)$ is the cost of S and $f(\mathcal{C}')$ is the maximum number of elements that can be covered by $\mathcal{C}'$.

7.1.2 LPO

While Wolsey's algorithm starts with the empty solution and greedily adds sets to it until it becomes feasible, the LP oracle algorithm (LPO) begins with the feasible solution $\mathcal{C}' = \mathcal{S}$ and greedily removes sets until $\mathcal{C}'$ is a minimal cover, i.e. no more sets can be removed without violating feasibility.

At the heart of LPO is an LP solver that provides "hints" as to which sets should be closed. Over the course of the algorithm, LPO maintains a cover $\mathcal{C}$ of sets that have not yet been closed, i.e. removed. In iteration i, it determines the set $\mathcal{S}_i$ of sets where $\mathcal{S}_i = \{S \in \mathcal{C} : \mathcal{C} \backslash \{S\} \text{ is feasible } \}$. In other words, $\mathcal{S}_i$ is the sets that are candidates for being removed from $\mathcal{C}$. LPO closes the set S' such that

$$S' \in \arg\min_{S \in \mathcal{S}_i} \ell(\mathcal{C} \backslash \{S\})$$

where $\ell(\mathcal{C}')$ is the fractional cost of the instance in which only sets in $\mathcal{C}'$ can be open (even partially open). The value $\ell(\mathcal{C} \backslash \{S\})$ can be interpreted as the fractional cost of removing set S from $\mathcal{C}$. Intuitively, LPO performs well when $\ell(\mathcal{C} \backslash \{S\})$ corresponds to the effect of closing S on the integral solution. LPO repeats these steps until there are no more open sets that can be closed. The intuition behind this is that the LP oracle gives warning when closing a set might impose a heavy cost in later iterations. In some sense, it gives an idea of which sets are more important than others.

As an example, consider the following instance of active time scheduling of unit jobs with batch parameter $B = 5$ (Figure 7.1). There is a block of four rigid jobs, each of length four, which must be scheduled in slots six through nine. These slots each have room for one more job unit. What does the rest of the optimal solution look like? If the last slot is open, then one can achieve an active time of 6 (or $B + 1$ in general). However, if the last slot is closed, then the unit jobs are forced to occupy

the remainder of the otherwise rigid block, which will then force the long chain of 5 (or B) to be scheduled earlier, for a total active time of 9 (or $2B - 1$). This is suboptimal, and LPO can detect this suboptimality. It cannot feasibly close slots six through nine. If it closes the last slot, the cost of the scheduling will necessarily go up; the fractional solution subject to slot 10 being closed reflects this.

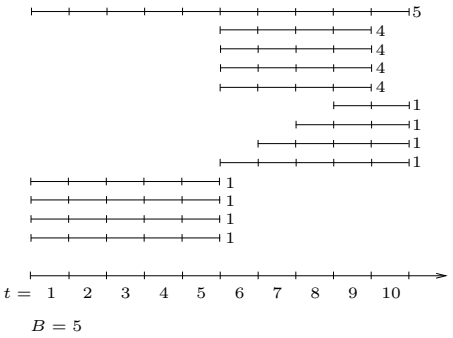


Figure 7.1: An example of Active Time Scheduling, highlighting the strength of LPO.

7.1.3 Optimizations

The primary disadvantage of Wolsey's algorithm is that it can require $\Theta(m^2)$ computations of $w(\mathcal{C}, S)$, each being quite costly. In particular, for problems involving hard capacities, a maximum flow computation is necessary to determine $w(\mathcal{C}, S)$; it may be that the elements in S can be covered by sets already in $\mathcal{C}$, but that due to the capacity constraints, the addition of S to $\mathcal{C}$ would still increase the total number of elements that can be covered. LPO is plagued by a similar affliction, in that it can necessitate several computations of $\ell(\mathcal{C}')$, especially on instances for which there are many feasible covers of small cardinality. The following optimizations were designed to reduce the number of such flow computations without compromising solution quality.

Optimization: Ordering sets [67]. The following observation can reduce Wolsey's number of computations, without affecting the cost of the solution, up to tiebreaking. Due to the submodularity of f, the value of $w(\mathcal{C}, S)$ for a given set S can only increase as $\mathcal{C}$ grows. Therefore one can delay recomputing $w(\mathcal{C}, S)$ for sets S that already had high values in previous iterations, since they are guaranteed not to be selected in the current iteration. More formally, let $\mathcal{C}_i$ be the partial cover at the beginning of the ith iteration. Suppose that $S_1, S_2, \ldots, S_m$ is the order by which the algorithm computes $w(\mathcal{C}_i, \cdot)$. Given a particular set S_j, let B_j denote the minimum $w(\mathcal{C}, \cdot)$ value among the first j sets, i.e.

$$B_j = \min_{S \in \{S_1, \ldots, S_j\}} w(\mathcal{C}_i, S)$$

If, for set S_{j+1} and some earlier iteration $i' < i$, $B_j < w(\mathcal{C}_{i'}, S_{j+1})$, then it follows that $B_j < w(\mathcal{C}_i, S_{j+1})$ as well, since $w(\mathcal{C}_{i'}, S_{j+1}) \leq w(\mathcal{C}_i, S_{j+1})$. Thus we know that S_{j+1} will not be the next set added to $\mathcal{C}_i$, and there is no need to compute $w(\mathcal{C}_i, S_{j+1})$ explicitly. This observation holds regardless of the order by which the algorithm iterates over sets. Thus, if it considers sets in increasing order of their most recently computed $w(\cdot, S_j)$ value, the number of computations can only further decrease while the solution cost remains the same (again, up to tiebreaking).

As observed in [67], applying this optimization to Wolsey's algorithm yields a significant improvement in performance; we demonstrate this independently in our results. Since the the spirit of the original algorithm is preserved, the $O(\log n)$ guarantee still holds, though we will see that in practice the solution quality is much better.

One can apply the analogous optimization to LPO, since for any set S, $\ell(\mathcal{C} \backslash \{S\})$ can only increase as $\mathcal{C}$ gets smaller with each passing iteration. We omit the details since they are a straightforward modification of the optimization as applied to Wolsey's algorithm.

Optimization: Freezing set variables. We say a set is *closed* if it is not in

the cover and *open* otherwise. Simply stated, the following preprocessing step is done prior to the execution of the (original or otherwise optimized) algorithm: first solve the LP relaxation (see Appendix 7.4). Any sets having values 0 or 1 in the fractional optimal solution are *frozen* to being closed or open, respectively. Then run Wolsey's algorithm (LPO, respectively) on the remaining *unfrozen* sets, until a feasible solution is attained (until no more unfrozen sets can be closed, respectively). Note that for this optimization applied to Wolsey's algorithm, the theoretical bounds no longer hold. In spite of this, we empirically demonstrate that Wolsey and LPO with both modifications return solutions that are still quite close to optimal. In particular, not only does the freezing optimization frequently correct suboptimal solutions, it also severely reduces the number of flow evaluations that would have been necessary otherwise.

7.2 Implementation

All implementations were developed in Java, invoking Gurobi (version 5.5) wherever a flow or LP call was needed. Experiments were performed on a shared machine with twenty-four 2.20gHz hexa-core processors and 64 gb RAM.

7.2.1 Data Sets

For ease of notation, we abbreviate active time scheduling to ATS, Capacitated Set Cover to CapSC, and Capacitated Vertex Cover to CapVC. Several structured random instances of ATS, CapSC and CapVC were generated. Out of convenience, we describe CapSC in the context of jobs and slots, rather than ground elements and sets. We remind the reader that the primary differences between CapSC and ATS are that (1) capacities need not be uniform in CapSC, (2) a job's feasibility windows need not comprise a single interval in CapSC, and (3) jobs may not be unit length in ATS. Given parameters (N_1, N_2, M_1, M_2, B), instances of n jobs for $N_1 \leq n \leq N_2$ random and T slots for $M_1 \leq T \leq M_2$ random were created. For a CapSC instance, capacities

bounded by B were independently and randomly chosen for each slot, and each job randomly selected the slots that could cover it. If the instance was that of ATS, a single capacity $B' < B$ was randomly selected, and each job randomly chose its release time, a deadline after its release time, and a job length no more than the length of its feasible window. Infeasible instances were dropped and replaced with feasible ones.

Fifteen CapSC instances and fifteen ATS instances were generated with parameters $(25, 50, 30, 60, 6)$. These data sets are denoted "small" sets. The "medium" data set consisted of five CapSC instances and five ATS instances, generated with parameters $(50, 100, 75, 180, 7)$. For the CapVC testbed and parameters (N_1, N_2, P_1, P_2), a random graph $G(n, p)$ was created where n was uniformly selected at random from $[N_1, N_2]$ and with p similarly selected from the range $[P_1, P_2]$. Then, capacity $k(v)$ was computed as $k(v) \sim \mathcal{U}(0, d(v))$. Infeasible instances were dropped and replaced with feasible ones. Twenty instances were generated from each of the following parameter vectors: $(15, 50, 0.2, 0.95)$ and $(80, 130, 0.2, 0.6)$.

Our heuristics were also tested on modified graph instances taken from the public testbed given by the DIMACS Implementation Challenge [57]. Some graphs were originally intended to test Clique algorithms and are quite dense. Others were intended to test heuristics for the k-coloring problem and are less so. Finally, we evaluate our heuristics on two instances that model large-scale social interaction and connectivity structure, respectively. Newman's NetSci graph is a collaboration graph between authors publishing in Network Science [77]. It contains 1589 vertices and 2,742 edges. We also consider a network of peer-to-peer file sharing connections; it contains 6,301 vertices, one per Gnutella host, and 20,777 edges denoting observed connection between two hosts [66]. Graph sizes are contained in Table 7.1.

	n	m	Number of Frozen Variables
johnson8-2-4	210	28	0
hamming6-4	704	64	0
hamming6-2	1824	64	0
mulsoli4	3946	185	11
school1	19095	385	23
NetSci	2742	1589	1300
Gnutella	20777	6301	6301

Table 7.1: Statistics describing DIMACS, NetSci, and Gnutella graphs.

We note that for instances $G = (V, E)$ of CapVC, the X side of the *Cov-MECF* framework has $|X| = |E|$, while the Y side has $|Y| = |V|$. The out-degree of each node $x \in X$ is two, since each edge can be covered only by its two incident nodes.

7.2.2 Inducing Capacities

The DIMACS graphs and the real world graphs are instances of standard Vertex Cover; we aggressively imposed capacities that were as low as possible, without rendering the instance infeasible in the following way. For a given instance, assign to vertex v a capacity that is selected independently and uniformly from the range $[\lceil \lambda d(v) \rceil, d(v)]$, where $d(v)$ is the degree of vertex v and $0 < \lambda < 1$ is a constant. The subsequent procedure was followed to ensure that feasibility was maintained: for each candidate λ, assign capacities until feasibility is achieved or the fifth attempt has been made. If the instance is infeasible, increase λ by 0.05 and repeat. This process is guaranteed to return feasible capacity assignments as long as the uncapacitated version is feasible. In our case, λ was initialized to 0.8.

7.3 Results and Discussion

In this section, we discuss the results of the various implementations of Wolsey's algorithm, *LPO* and MinFeas. (We limit the discussion of MinFeas to the scheduling problem for which it was designed, though statistics of its performance on other testbeds are also reported.) We evaluate them on solution quality and efficiency of computation perspectives. For the latter, the primary metric considered is the total number of flow (or LP) evaluations made. We draw no conclusions based on timing constructs since the experiments were executed on a shared machine. However, we note that Wolsey's algorithm involves max flow computations while *LPO* calls the relaxation of min-edge cost flow computations. On some of the larger graphs, the vanilla implementations of Wolsey and *LPO* did not complete, and only the results of their optimized versions are reported. For the sake of notational convenience, ALG/OO will refer to the algorithm ALG with optimization by order. ALG/OOF refers to ALG with both optimization by order and optimization via freezing.

7.3.1 Solution Quality

	Num. Inst.	*LPO*	*LPO*/OO	*LPO*/OOF
ATS	20	0	0	0
CapVC	40	1	2	2
CapSC	20	1	1	1

Table 7.2: Number of *suboptimal* instances for all implementations of *LPO* on random testbeds. The total number of instances in each testbed is also given in the first column.

On the scheduling instances (lines 1 of Tables 7.2 and 7.3), *LPO* outperformed Wolsey's algorithm, as did MinFeas. In fact, at least on scheduling instances, for

	Num. Inst.	Wol	Wol/OO	Wol/OOF	MFS	MFS/F
ATS	20	3	5	0	0	0
CapVC	40	17	15	5	27	8
CapSC	20	5	6	5	35	16

Table 7.3: Number of *suboptimal* instances for all implementations of Wolsey's algorithm and MinFeas (MFS) on random testbeds.

which it was designed, MinFeas's empirical performance seems to be significantly more impressive than its theoretical bound of five might suggest. There exist instances on which MinFeas is suboptimal, but none of them were present in the random testbed. Wolsey's algorithm was suboptimal on several scheduling instances, even with intelligent ordering of the sets. In some cases, intelligent ordering actually hurt Wolsey's solution due to the change in tie-breaking. All of Wolsey's suboptimalities were corrected by freezing. On CapVC instances, *LPO* underperformed on two instances while Wolsey's algorithm was suboptimal on as many as seventeen (twelve of which were corrected by optimizations). *LPO* and Wolsey's algorithms maintained similar performance on the CapSC testbed. Though the quality of Wolsey's solutions are in general poorer than those of *LPO*, many suboptimalities were corrected by optimizations, the bulk attributed to freezing.

For the DIMACS and real world graphs (Tables 7.4 and 7.5), *LPO* achieved optimality on all but two of them while Wolsey's algorithm underachieved on at least four graphs, failing to complete on two others. Nevertheless, the performance is reasonable, relative to the performance of MinFeas.

	OPT	*LPO*	*LPO*/OO	*LPO*/OOF
johnson8-2-4	21	21	21	21
hamming6-4	52	52	52	52
hamming6-2	62	62	62	62
mulsoli4	99	99	99	99
school1	344	348	347	347
NetSci	944	944	944	945
Gnutella	4434[¶]	–	–	4434

Table 7.4: Optimal and LPO's solutions for DIMACS, NetSci and Gnutella graphs.

	Wol	Wol/OO	Wol/OOF	MFS	MFS/F
johnson8-2-4	23	24	24	25	25
hamming6-4	56	56	56	56	56
hamming6-2	62	62	62	62	62
mulsoli4	99	99	99	173	173
school1	349	348	348	357	357
NetSci	947	947	944	1034	1034
Gnutella	–	–	4434	4836	4836

Table 7.5: Wolsey algorithms' and MinFeas' solutions for DIMACS, NetSci and Gnutella Graphs.

It is clear that while known worst-case guarantees for *LPO* cannot be as broadly applied as Wolsey's algorithm, *LPO* returns optimal solutions significantly more often

[2]Computing the optimal solution actually timed out. Freezing reveiled that the fractional solution was integral.

than Wolsey's implementations. On all of Wolsey's implementations, whenever any of the implementations were suboptimal, they were never so by much (see Table 7.6); this only confirms what has already been long accepted in practice: Wolsey's algorithm on the whole performs much better than its theoretical bound would suggest.

	LPO	*LPO*/OO	*LPO*/OOF	Wol	Wol/OO	Wol/OOF	MFS	MFS/F
ATS	1.000	1.000	1.000	1.002	1.003	1.000	1.003	1.000
CapVC	1.000	1.001	1.001	1.007	1.006	1.002	1.033	1.006
CapSC	1.011	1.011	1.011	1.091	1.091	1.043	2.831	1.223

Table 7.6: Average ALG:OPT ratio for all implementations of LPO, Wolsey's and MinFeas algorithms on the random testbeds.

We point out that the freezing optimization did not hurt LPO's solution quality in any of the random instances. This does not come as a complete surprise since LPO's subroutine involves solving the LP relaxation of the problem; this is the exact LP relaxation that the freezing step solves, and the variables that are frozen to zero are in fact the sets that LPO would have discarded first anyway. For Wolsey's algorithms and MinFeas, freezing also helped much more often than it hurt.

7.3.2 Computational Efficiency

On the ATS, CapVC and DIMACS instances, the vanilla implementation of LPO required significantly fewer subroutine calls than its Wolsey counterpart; the reverse is true for the CapSC instances. This is to be expected, since by design, LPO and Wolsey's algorithm are complementary in their approaches. On instances for which LPO's final solution consists of a small number of sets, LPO will make many calls. Similarly, on instances where feasible covers tend to be large, Wolsey's algorithm will involve many more iterations. The multicover property of the ATS instances results

in higher cardinality solutions. The sparsity of the CapVC instances, i.e. that each edge can be covered by at most two vertices, leads to a similar effect. The number of subroutine calls required by MinFeas was orders of magnitude less than *LPO* and Wolsey's algorithm. In the worst case, both *LPO* and Wolsey's algorithm may require $\Theta(m^2)$ calls, while MinFeas needs $O(m)$.

		LPO	*LPO*/OO	*LPO*/OOF
ATS	small	440.07	85.40	2.47
	medium	3700.80	271.00	3.40
CapVC	small	144.55	51.65	8.10
	medium	525.05	164.70	15.45
CapSC	small	1141.53	104.27	23.33
	medium	7880.20	295.60	84.00

Table 7.7: Average number of calls for random testbeds for *LPO*.

		Wol	Wol/OO	Wol/OOF	MFS	MFS/F
ATS	small	937.07	243.00	2.67	43.20	1.27
	medium	9591.00	1109.80	4.20	136.00	1.80
CapVC	small	546.40	126.30	13.60	30.20	4.20
	medium	5859.25	472.25	28.45	105.75	7.25
CapSC	small	179.87	118.80	35.60	46.60	9.87
	medium	471.60	330.60	82.60	121.40	21.20

Table 7.8: Average number of calls for random testbeds for Wolsey's algorithm and for MinFeas.

Optimization: Ordering sets

This particular optimization yields significant improvement in the number of calls, though the extent of its effect varies from testbed to testbed. On instances where Wolsey's vanilla algorithm performed better than *LPO*'s, intelligently ordering sets leveled the playing field. The reverse is not true; on instances where vanilla *LPO* already involved fewer calls than vanilla Wolsey, *LPO*/OO continued to excel.

Table 7.9 contains the average percentage reduction per implemention, per generated testbed. On ATS instances, *LPO* enjoyed greater improvement than Wolsey. For example, on the medium set, the optimization reduced *LPO*'s average number of calls from 3701 to 271, while it only reduced Wolsey's average number of calls from 9591 to 1110.

On CapSC instances, this optimization in some sense leveled the playing field for *LPO*. *LPO* observed an average 92 percent reduction in the number of calls while Wolsey witnessed only a 33 percent reduction. However, this reduction is misleading; it is only because vanilla *LPO* involved so many more calls than vanilla Wolsey. For example, on the medium data set, the optimization reduced *LPO*'s average number of calls from 7880 calls to 296 calls and Wolsey's from 472 calls to 331.

On most of the DIMACS instances, the number of calls invoked by *LPO* was at least fifty percent less than the number of calls made by Wolsey. Intelligently ordering the sets yielded an average 72 percent reduction in *LPO*'s number of calls and a 77 percent reduction in that of Wolsey's. Because each edge can be covered by at most two vertices, the final solution for both *LPO* and Wolsey tended to be of higher cardinality, relative to the total number of vertices. This explains why *LPO* on average made fewer calls than Wolsey, even with optimizations applied. On the other hand, the number of calls made by Wolsey is significantly less on the NetSci graph than that of *LPO*, at least on their optimized implementations.

Optimization: Freezing set variables

By and large, the freezing optimization further improved the performance of *LPO* and Wolsey, generally giving *LPO* the edge over Wolsey. In fact, for most of the random testbeds, *LPO*/OOF involved on average fewer subroutine calls than Wolsey/OOF (see Tables 7.7 and 7.8). The discussion that follows pertains to the percentage reduction in number of calls.

	LPO/OO	*LPO*/OOF	Wol/OO	Wol/OOF	MFS/F
ATS	0.812	0.995	0.770	0.997	0.973
CapVC	0.551	0.956	0.817	0.980	0.890
CapSC	0.916	0.979	0.329	0.805	0.792

Table 7.9: Average percentage reduction in the number of subroutine calls, compared to their vanilla counterparts.

For almost all of the scheduling instances, freezing yielded a feasible integral solution, resulting in similar performances by *LPO*/OOF, Wolsey/OOF and MinFeas/OOF. On the CapVC testbed, freezing witnessed an additional 55 percent reduction in *LPO*'s performance, for a combined 96 percent reduction in tandem with the ordering optimization. Wolsey's algorithm saw a total 98 percent reduction in the number of calls, 16 percent attributed to the freezing optimization. For the CapSC instances, the freezing step further reduced *LPO*'s number of calls by an additional six percent for a total 98 percent reduction. While intelligently ordering the sets reduced Wolsey's number of subroutine calls by 33 percent, the freezing step additionally reduced it by another 48, for a combined 81 percent reduction in the number of calls. This suggests that the impact of freezing on Wolsey's algorithm, lack of theoretical guarantee notwithstanding, is comparable to that of imposing an intelligent orders on sets, at least for Capacitated Set Cover.

	LPO	*LPO*/OO	*LPO*/OOF
johnson8-2-4	196	56	58
hamming6-4	754	200	202
hamming6-2	189	132	134
mulsoli4	12354	439	419
school1	13927	1649	1603
NetSci	818159	12857	8652
Gnutella	–	–	2

Table 7.10: Number of flow computations for DIMACS, NetSci and Gnutella graphs.

Freezing had negligible effect on most of the DIMACS instances, since the fractional solutions had few integral values. However, the opposite was true for both the NetSci graph and the Gnutella peer-to-peer network, the fraction solution of the latter achieving integrality. Thus, frozen implementations of both *LPO* and Wolsey's algorithm were impressive, in contrast to their less-optimized counterparts, many of which timed out after several hours. See Tables 7.10 and 7.11.

7.4 Cov-MECF Integer Program

The integer program capturing *Cov-MECF* is straight-forward and is given below. There are only two types of variables. Variables of the form $f_{*,\star}$ represent the flow sent along edge $(*, \star)$, and the variables u_y serve as indicators for whether flow is sent along edge (y, t).

	Wol	Wol/OO	Wol/OOF	MFS	MFS/F
johnson8-2-4	415	179	181	28	28
hamming6-4	2101	399	401	64	64
hamming6-2	2140	902	904	64	64
mulsoli4	13564	754	742	185	185
school1	73989	3963	3877	385	385
NetSci	1057800	5508	1278	1589	1589
Gnutella	–	–	2	6301	6301

Table 7.11: Number of flow computations made by Wolsey's algorithms for DIMACS, NetSci and Gnutella graphs.

$$
\begin{array}{lrcll}
\text{min.} & \sum_{y\in Y} \kappa_y u_y & & & \\
\text{s.t.} & \sum_{y\in\delta(x)} f_{x,y} & = & f_{s,x} & \forall\ x \in X \\
& \sum_{x\in\delta(y)} f_{y,t} & = & f_{y,t} & \forall\ y \in Y \\
& f_{s,x} & \leq & c_x & \forall\ x \in X \\
& f_{y,t} & \leq & c_y u_y & \forall\ y \in Y \\
& f_{x,y} & \leq & 1 & \forall\ (x,y) \in E \\
& \sum_x f_{s,x} & \geq & f^* & \\
& f_{x,y} & \leq & u_y & \forall\ (x,y) \in E \\
& u_y & \in & \{0,1\} & \forall\ y \in Y \\
& u_y, f_{s,x}, f_{x,y}, f_{y,t} & \geq & 0 & \forall\ x \in X, y \in Y
\end{array}
\tag{7.1}
$$

Chapter 8

BUSY TIME SCHEDULING

In this chapter, we consider the busy time model, in which one is given access to an unlimited number of identical parallel machines, each of which can be working on up to B jobs at once. The goal is to minimize the cumulative busy time of machines used. For non-preemptive jobs, to describe our result requires an in-depth look at the special case of interval jobs. Although the result for the interval job setting is not the best known, it admits an improved approximation for the general busy time problem (see Section 8.2). Results for preemptive busy time scheduling follow this discussion.

8.1 Interval Jobs

Recall that an interval job is one whose processing length is equal to the difference between its deadline and release time. For a given instance $\mathcal{J}$ of jobs, the cost of the optimal solution of $\mathcal{J}$ by $OPT(\mathcal{J})$. We denote by $OPT_\infty(\mathcal{J})$ the cost of the optimal solution for the instance $\mathcal{J}$ when unbounded parallelism is allowed.

Definition 8. *The length of a time interval $I = [a, b)$ is denoted by $\ell(I) = b - a$. For a set of intervals $\mathcal{I}$, the length of $\mathcal{I} = \sum_{I \in \mathcal{I}} \ell(I)$. The span of $\mathcal{I}$ is $Sp(\mathcal{I}) = \cup_{I \in \mathcal{I}} I$.*

The length $\ell(J_j)$ (span $\mathrm{Sp}(J_j)$, respectively) of an interval job J_j is the length (span, respectively) of $[r_j, d_j)$. Similarly, the length $\ell(\mathcal{J})$ (span $\mathrm{Sp}(\mathcal{J})$, respectively) of a set $\mathcal{J}$ of jobs is the length (span, respectively) of the set of intervals $\{[r_j, d_j) : J_j \in \mathcal{J}\}$.

The following lower bounds were introduced in [33] and hold trivially on any optimal solution for a given instance $\mathcal{J}$.

Observation 2. *For a given instance $\mathcal{J}$ of interval jobs and $B \geq 1$, the following lower bounds hold.*

1. *Mass bound: $OPT(\mathcal{J}) \geq \frac{\ell(\mathcal{J})}{B}$. We sometimes refer to $\ell(\mathcal{J})$ as the "mass" of $\mathcal{J}$.*

2. *Span bound: for $\mathcal{J}$ a set of interval jobs, $OPT(\mathcal{J}) \geq |Sp(\mathcal{J})|$. In particular, the size of the span of any subset of interval jobs is a lowerbound the busy time of any feasible solution.*

Without loss of generality, the busy time of a machine is contiguous. If it is not, we can break it up into disjoint periods of contiguous busy time, assigning each of them to different machines, without increasing the total busy time of the solution.

We begin this section with a description of GreedyTracking for instances of interval jobs. Given a feasible solution, one can think of each bundle $\mathcal{B}$ as the union of B individual *tracks* of jobs, where each track is a feasible set of jobs, i.e. having disjoint spans. The main idea behind the algorithm is to identify such tracks iteratively, bundling the first B tracks into a single bundle, the second B tracks into the second bundle, etc. FirstFit suffers from the fact that it greedily considers jobs one-by-one; GreedyTracking is less myopic in that it identifies jobs whole tracks at a time, but its grouping of tracks into bundles is otherwise indiscriminant.

More specifically, in iteration i, the algorithm identifies a feasible subset $\mathcal{T}_i \subseteq \mathcal{J} \setminus \bigcup_{\iota=1}^{i-1} \mathcal{T}_\iota$ of maximum length $\ell(\mathcal{T}_i)$ and assigns it to the ith track. One can find such a subset efficiently via weighted interval scheduling algorithms [23]. The first B tracks of jobs will form the first bundle $\mathcal{B}_1$; the second B tracks of jobs will form $\mathcal{B}_2$; and so on. If the final solution has κ bundles, the algorithm's total cost is $\sum_{i=1}^{\kappa} |\mathrm{Sp}(\mathcal{B}_i)|$. Any bundles whose span is not a single interval can then be broken into a set of disjoint bundles without increasing the busy time. The pseudocode for GreedyTracking is in Figure 4.

Input: $\mathcal{J}, B$

$\mathcal{S} \leftarrow \mathcal{J}$

$i \leftarrow 1$; $j \leftarrow 1$

while $\mathcal{S} \neq \emptyset$ **do**

    // $\mathcal{T}_i^j$ denotes the track j of bundle i

    $\mathcal{T}_i^j \leftarrow MWFS(\mathcal{S})$

    $\mathcal{S} \leftarrow \mathcal{S} \backslash \mathcal{T}_i^j$

    if $j < B$, **then** $j \leftarrow j + 1$

    else $i \leftarrow i + 1$; $j \leftarrow 1$

end

$\kappa \leftarrow i$

for $k \leftarrow 1, \ldots, \kappa$ **do**

    $\mathcal{B}_k \leftarrow \bigcup_{j=1}^{B} \mathcal{T}_k^j$

end

return $\{\mathcal{B}_k\}_{k=1}^{\kappa}$

Algorithm 4: GreedyTracking. $MWFS(\mathcal{S})$ returns the solution to the weighted interval scheduling problem where the weight $w(J_j) = p_j$ for job J_j.

We next prove a key property of GreedyTracking: the span of any track is at least half that of the remaining unscheduled jobs. In particular, the span of any bundle is at most twice that of the first track to be assigned to it.

Lemma 6. *Let $\mathcal{T}_i$ be the ith track found by* GreedyTracking. *Let $\mathcal{J}_i' \subseteq \mathcal{J}$ denote the set of unscheduled jobs $\mathcal{J} \backslash \bigcup_{\iota=1}^{i} \mathcal{T}_\iota$. Then $|Sp(\mathcal{J}_i' \cup \mathcal{T}_i)| \leq 2 \cdot |Sp(\mathcal{T}_i)|$.*

Proof. Consider a specific track $\mathcal{T}_i$ and the set $\mathcal{J}_i'$ of remaining jobs. It is possible that $\mathrm{Sp}(\mathcal{J}_i' \cup \mathcal{T}_i)$ is comprised of multiple maximal intervals; in such a case, individually applying the argument to each maximal interval will yield the result. Henceforth we will assume without loss that $\mathrm{Sp}(\mathcal{J}_i' \cup \mathcal{T}_i)$ is a single interval. For the remainder of

the proof, we drop the index i.

Order the jobs of $\mathcal{T} = \{J_{\alpha_1}, \ldots, J_{\alpha_L}\}$ by release time (and thus, deadline). Then, one can view $\mathrm{Sp}(\mathcal{J}')\backslash\mathrm{Sp}(\mathcal{T})$ as a set $\mathcal{I}$ of maximal disjoint intervals $I_0, \ldots, I_L$. Intuitively, intervals in $\mathcal{I}$ are those whose cost must be accounted for by, i.e. charged to, the span of track $\mathcal{T}_i$. It is enough to show that

$$\sum_{j=0}^{L} |I_j| \leq |\mathrm{Sp}(\mathcal{T})|$$

First, for any $I_j = [a, b)$, it is obvious that the minimum number of jobs in $\mathcal{J}'$ needed to cover or account for I_j is at most two: consider the jobs of $\mathcal{J}'$ whose spans contain a. One of these jobs has latest deadline and thus maximally accounts for I_j; the rest cannot account for any more of I_j. Similarly, of all the jobs of $\mathcal{J}'$ whose spans contain b, one of them has earliest release time; this job and the former job must span I_j, since no jobs of $\mathcal{J}'$ are completely contained in I_j by maximality of $\mathcal{T}$. Therefore, I_j can be accounted for by the spans of either one or two jobs in $\mathcal{J}'$.

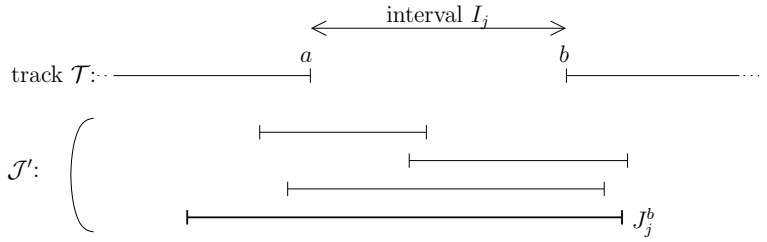


Figure 8.1: Interval I_j only needs one jobs of $\mathcal{J}'$ to span it. J_j^a is undefined.

We identify these *spanning jobs* algorithmically in the following way. Define $J_j^b \in \mathcal{J}'$ to be

$$J_j^b \in \mathrm{argmin}\{r_j : J_j \in \mathcal{J}' \text{ and } b \in [r_j, d_j)\}$$

If there are multiple such jobs with minimum release time, break ties in favor of the one with latest deadline. If the release time of J_j^b is at or before a (as in Figure 8.1),

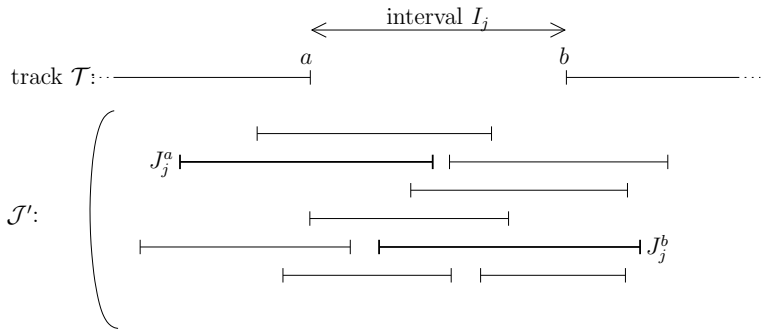


Figure 8.2: Interval I_j must be covered by two jobs of $\mathcal{J}'$.

then the $I_j \subseteq \mathrm{Sp}(J_j^b)$. If not, then there must exist another job in $\mathcal{J}'$ whose span contains $I_j \backslash \mathrm{Sp}(J_j^b)$ (as in Figure 8.2). Choose such a job J_j^a of earliest release time, again breaking ties toward jobs of later deadline. (If $I_j \subseteq \mathrm{Sp}(J_j^b)$, define J_j^a to be nil.) It is possible that a single job may cover multiple intervals and also that spanning jobs are not disjoint from one another. However, a consequence of selecting jobs this way is that if $\mathcal{J}'' = \bigcup_{j=0}^{L}(J_j^b \cup J_j^a)$ is the set of spanning jobs, then at most two jobs of $\mathcal{J}''$ overlap at any fixed point of time. Notice that $\mathrm{Sp}(\mathcal{T} \cup \mathcal{J}') = \mathrm{Sp}(\mathcal{T} \cup \mathcal{J}'')$.

Intuitively, our approach will use the maximality of $\mathcal{T}$ to bound $\sum_j |I_j|$ by the lengths of the spanning jobs, which in turn will be charged to the size of $\mathrm{Sp}(\mathcal{T})$. For each interval I_j that is accounted for by two jobs J_j^b and J_j^a, we call the deadline of the earlier job J_j^a a *breaking point*. Precisely, d_j^a is a breaking point only if $J_j^b \cap J_j^a \subseteq I_j$. Defining the left-most and right-most boundaries of $\mathrm{Sp}(\mathcal{T} \cup \mathcal{J}')$ to also be breaking points, it suffices to prove the bound between any two consecutive breaking points b_q and b_{q+1}. Let the jobs of $\mathcal{T}$ between b_q and b_{q+1} be $\mathcal{T}_q = \{J_{\alpha_c}, \ldots, J_{\alpha_k}\}$. Define $\mathcal{I}_q = \{I'_{c-1}, \ldots, I'_k\}$ where

$$I'_j = I_j \cap [b_q, b_{q+1})$$

and let the corresponding set of spanning jobs be $\mathcal{J}''_q$. If all of the jobs of $\mathcal{J}''_q$ were disjoint, then they would form a *replaceable set* for $\mathcal{T}_q$, i.e. $\mathcal{T} \backslash \mathcal{T}_q \cup \mathcal{J}''_q$ is a feasible track and so $|\mathrm{Sp}(\mathcal{J}''_q)| \leq |\mathrm{Sp}(\mathcal{T}_q)|$; then the lemma would follow easily. However, it

is possible that jobs of $\mathcal{J}''_q$ are not disjoint; despite this, with more careful attention we can still argue the bound in the following way. Since we are operating between two consecutive breaking points, any two jobs of $\mathcal{J}''_q$ must have some part of their intersection in the span of a job of $\mathcal{T}_q$. We will call such a job in $\mathcal{T}_q$ an *overloaded* job. The overloaded jobs partition $\mathcal{T}_q$ in such a way that for each partition, there is a natural replaceable set in $\mathcal{J}''_q$. Summing over the partition of $\mathcal{T}_q$, only the overloaded jobs will be double-counted; these correspond exactly to the jobs of $\mathcal{J}''_q$ whose entire lengths are needlessly contributing to $\sum_{j \in \mathcal{J}''_q} p_j$. More formally, let $\mathcal{H}_q \subseteq \mathcal{T}_q$ be the set of overloaded jobs, i.e.

$$\mathcal{H}_q = \{j : J_{\alpha_j} \in \mathcal{T}_q \text{ and the intersection of two jobs of } \mathcal{J}''_q \text{ is partially in } \mathrm{Sp}(J_{\alpha_j})\}$$

Denote the jobs of $\mathcal{H}_q$ by $\mathcal{H}_q = \{h_0, h_1, \ldots, h_d, h_{d+1}\}$, where h_0 and h_{d+1} act as "dummy overloaded jobs" of infinitesimal size at b_q and b_{q+1}. (Their purpose is just for notational ease in summing over $[b_q, b_{q+1})$.) See Figure 8.3, containing a maximal interval M of $\mathrm{Sp}(\mathcal{J}'' \cup \mathcal{T})$, partitioned by breaking points (dotted vertical lines). $J_{h_\ell}, J_{h_{\ell+1}}$ and $J_{h_{\ell+2}}$ are consecutive overloaded jobs. The sum of the lengths of intervals between consecutive overloaded jobs is bounded by the span of $\mathcal{J}''_{q,\ell}$. $\mathcal{J}''_{q,\ell}$ is a replaceable set for track jobs between J_{h_ℓ} and $J_{h_{\ell+1}}$ (inclusive), so the span of the former is upperbounded by that of the latter.

Consider two consecutive overloaded jobs $J_{\alpha_{h_\ell}}$ and $J_{\alpha_{h_{\ell+1}}}$. Let the relevant intervals $\mathcal{I}_{q,\ell}$ and spanning jobs $\mathcal{J}''_{q,\ell}$ be defined in the natural way:

- $\mathcal{I}_{q,\ell} = \{I'_j \in \mathcal{I}_q : h_\ell \leq j < h_{\ell+1}\}$ are the intervals between the overloaded jobs;
- $\mathcal{J}''_{q,\ell} = \{J^b_j \cup J^a_j : h_\ell \leq j < h_{\ell+1}\}$ are the jobs that account for these intervals

Notice that $\bigcup_{\ell=0}^{d} \mathcal{I}_{q,\ell} = \mathcal{I}_q$. Similarly, $\{\mathcal{J}''_{q,\ell}\}_{\ell=0}^{d}$ is a partition $\mathcal{J}''_q$. Furthermore, for each $\ell = 0, \ldots, d$, jobs in $\mathcal{J}''_{q,\ell}$ are disjoint and thus are a replaceable set for the jobs of $\mathcal{T}_q$ between $J_{\alpha_{h_\ell}}$ and $J_{\alpha_{h_{\ell+1}}}$, i.e.

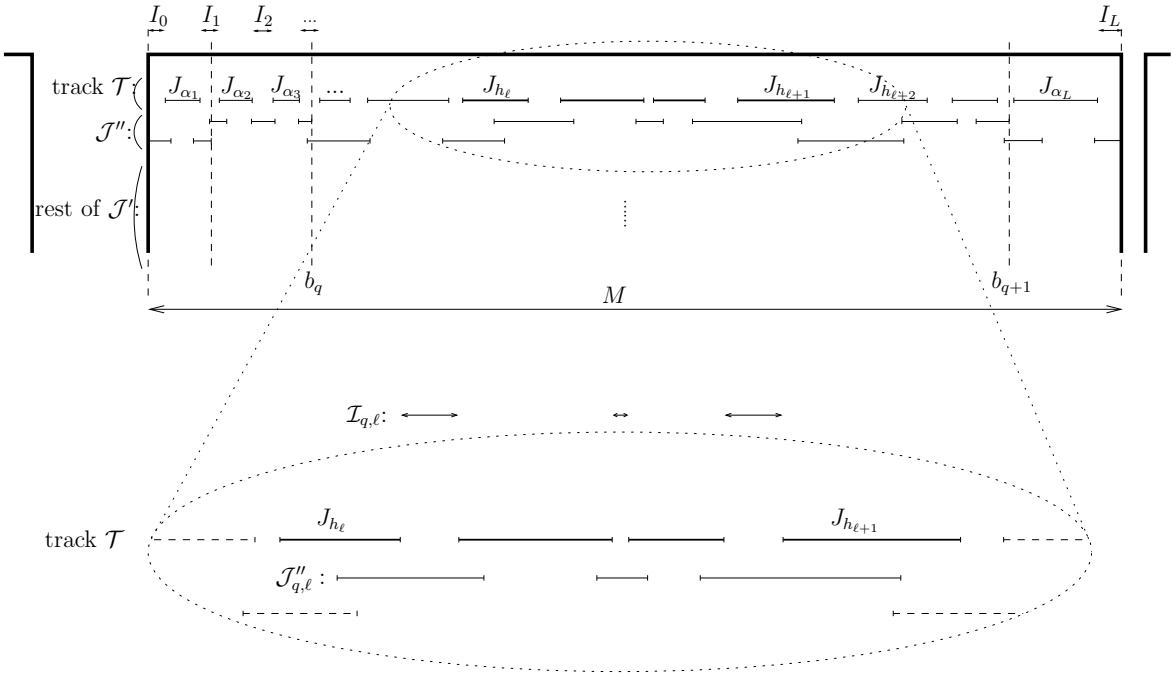


Figure 8.3: A maximal interval M of $\mathrm{Sp}(\mathcal{J}'' \cup \mathcal{T})$, partitioned by breaking points (dotted vertical lines).

$$|\mathrm{Sp}(\mathcal{J}''_{q,\ell})| = \sum_{j \in \mathcal{J}''_{q,\ell}} p_j \leq \sum_{j=h_\ell}^{h_{\ell+1}} p_{\alpha_j} \tag{8.1}$$

Summing over all overloaded jobs, we get that

$$\begin{aligned}\sum_{j=c-1}^{k} |I'_j| &\leq \sum_{j \in \mathcal{J}''_q} p_j - \sum_{j \in \mathcal{H}_q} p_{\alpha_j} \leq \left[\sum_{\ell=0}^{d} \sum_{j \in \mathcal{J}''_{q,\ell}} p_j \right] - \sum_{j \in \mathcal{H}_q} p_{\alpha_j} \\ &\leq \left[\sum_{\ell=0}^{d} \sum_{j=h_\ell}^{h_{\ell+1}} p_{\alpha_j} \right] - \sum_{j \in \mathcal{H}_q} p_{\alpha_j} = |\mathrm{Sp}(\mathcal{T}_q)| + \sum_{j \in \mathcal{H}_q} p_{\alpha_j} - \sum_{j \in \mathcal{H}_q} p_{\alpha_j} = |\mathrm{Sp}(\mathcal{T}_q)|\end{aligned} \tag{8.2}$$

Note that (8.2) follows from the fact that it is precisely the overloaded jobs that are double-counted; every other track job is counted once. Since the lengths of I_j's

between consecutive breaking points is bounded by the span of $\mathcal{T}$ between the breaking points, the entire proof follows. □

Lemma 7. *For every $i > 1$, the size of the span of bundle $\mathcal{B}_i$ can be bounded by the mass of the bundle $\mathcal{B}_{i-1}$ as follows: $|Sp(\mathcal{B}_i)| \leq 2\frac{\ell(\mathcal{B}_{i-1})}{B}$.*

Proof. Let $\mathcal{T}_i^1$ denote the first track of the bundle $\mathcal{B}_i$ and $\mathrm{Sp}(\mathcal{T}_i^1)$ the busy time of $\mathcal{T}_i^1$. We know by Lemma 6 that $|\mathrm{Sp}(\mathcal{B}_i)| \leq 2|\mathrm{Sp}(\mathcal{T}_i^1)|$.

Since $\mathcal{T}_i^1$ started the ith bundle, bundle $\mathcal{B}_{i-1}$ must already have had B tracks in it. Furthermore, the lengths (or busy time) of these tracks are longer than that of $\mathcal{T}_i^1$ since GreedyTracking chooses tracks in non-increasing order of length. We also know that the mass of the bundle $\mathcal{B}_{i-1}$ is the sum of the lengths of the B tracks constituting it and by definition of a track, all the jobs comprising $\mathcal{T}_i^1$ are disjoint. Hence, $\ell(\mathcal{B}_{i-1}) \geq B \cdot \ell(\mathcal{T}_i^1) = B \cdot \ell(\mathcal{T}_i^1)$. Therefore it follows that $|\mathrm{Sp}(\mathcal{B}_i)| \leq 2\frac{\ell(\mathcal{B}_{i-1})}{B}$. This completes the proof. □

Lemma 8. *The total busy time of all the bundles except the first one is at most twice that of an optimal solution for the entire instance. Specifically, $\sum_{i>1} |Sp(\mathcal{B}_i)| \leq 2OPT(\mathcal{J})$.*

Proof. This proof follows easily from Lemma 7. For any $i > 1$, $|\mathrm{Sp}(\mathcal{B}_i)| \leq 2\frac{\ell(\mathcal{B}_{i-1})}{B}$. Summing over all $i > 1$, we get the following

$$\sum_{i>1} |\mathrm{Sp}(\mathcal{B}_i)| \leq 2\frac{\sum_{i>1} \ell(\mathcal{B}_{i-1})}{B} = 2\frac{\sum_{i>1} \sum_{j \in \mathcal{B}_{i-1}} p_j}{B} \tag{8.3}$$

Therefore, we get the following:

$$\sum_{i>1} |\mathrm{Sp}(\mathcal{B}_i)| \leq 2\frac{\sum_{j \in \mathcal{J}} p_j}{B}. \tag{8.4}$$

By Observation 2, the cost of any optimal solution is at least the sum of the lengths of the jobs in the instance $\mathcal{J}$ divided by B.

$$OPT(\mathcal{J}) \geq \frac{\sum_{j \in \mathcal{J}} p_j}{B} \tag{8.5}$$

Hence, from Equations (8.4) and (8.5), we have the following.

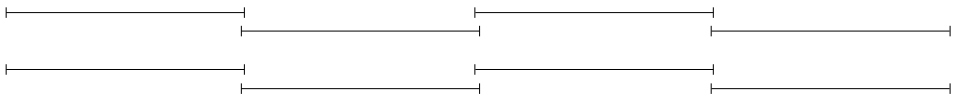
$$\sum_{i>1} |\mathrm{Sp}(\mathcal{B}_i)| \leq 2OPT(\mathcal{J}) \tag{8.6}$$

This completes the proof. □

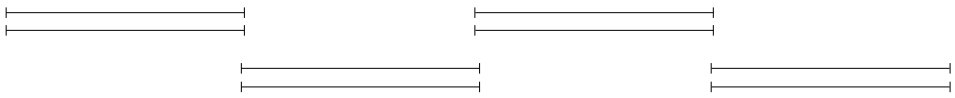
Theorem 4. *The cost of the algorithm is at most* 3 *times the cost of an optimal solution. Specifically,* $\sum_i |Sp(\mathcal{B}_i)| \leq 3OPT(\mathcal{J})$.

Proof. By Lemma 8, $\sum_{i>1} |\mathrm{Sp}(\mathcal{B}_i)| \leq 2OPT(\mathcal{J})$. Furthermore, $|\mathrm{Sp}(\mathcal{B}_1)| \leq OPT(\mathcal{J})$ by Observation 2. Therefore, $\sum_i |\mathrm{Sp}(\mathcal{B}_i)| \leq 3OPT(\mathcal{J})$. Hence, the algorithm approximates the optimal solution by a factor of at most three. □

Figure 8.4 shows that the approximation factor achieved by GreedyTracking is at least $2 - \epsilon$, for any $\epsilon > 0$. In this example, GreedyTracking identifies the tracks in an optimal way, but groups tracks into bundles in a suboptimal, misaligned manner.



(a) A possible packing by GreedyTracking



(b) Optimal packing.

Figure 8.4: A bad example for GreedyTracking. $B = 2$.

8.2 Implications for General Busy Time Scheduling

The general (non-interval job) variant of busy time was first considered by Khandekar et al. [59][1]. We are given a set $\mathcal{J}$ of jobs where each job $J_j \in \mathcal{J}$ needs to be processed within the time window $[r_j, d_j)$. They considered the most general version, where jobs not only have arbitrary processing times p_j but also arbitrary demand w_j. They provide a 5-approximation algorithm for this general problem.

The busy time problem is a restricted version of this one, where jobs can have arbitrary processing times p_j but demands are unit. We show that our 3-approximation algorithm for busy time scheduling of interval jobs implies a 3-approximation for busy time scheduling.

As a first step towards proving the 5-approximation for the general real-time scheduling problem with variable demands, Khandekar et al. [59] proved that if B is unbounded, then the real-time scheduling problem is polynomial-time solvable. The output of their dynamic program essentially converts the real-time scheduling problem to an interval scheduling problem by fixing the start and end times of every job.

Theorem 5. *[59] If B is unbounded, the real-time scheduling problem is polynomial-time solvable.*

First,we will need the following lemma (analogous to the proof given by Khandekar et al. [59]).

Lemma 9. *Suppose there exists a β-approximation for the busy time problem with unbounded B. Then there exists an algorithm that, given any busy time instance $(\mathcal{J}, B)$ for $B < \infty$, computes a feasible solution of cost at most $(\beta + 2)$ times that of the optimal solution for $(\mathcal{J}, B)$.*

Proof. We first use the β-approximation for unbounded B to compute a schedule $\mathcal{S}_\infty$ of cost at most $\beta \cdot \ OPT_\infty(\mathcal{J})$ for the given real-time instance $\mathcal{J}$. We then create a

[1]Khandekar et al. [59] refer to this problem as the real-time scheduling problem.

new interval job instance $\mathcal{J}'$ from the schedule $\mathcal{S}_\infty$. Let $[t_\infty(j), t_\infty(j) + p_j) \subseteq [r_j, d_j)$ be the interval where a job $j \in \mathcal{J}$ was scheduled in $\mathcal{S}_\infty$. Create a corresponding interval job $j' \in \mathcal{J}'$ with $r'_j = t_\infty(j)$ and $d'_j = t_\infty(j) + p_j$. The span of the job instance $\mathcal{J}'$ has size $|\mathrm{Sp}(\mathcal{J}')| = OPT_\infty(\mathcal{J}') \leq \beta \cdot OPT_\infty(\mathcal{J}) \leq \beta \cdot OPT(\mathcal{J})$.

Now apply GreedyTracking to $\mathcal{J}'$. From Lemma 8, we know that the total busy time of all machines except the first one is at most $\frac{2\ell(\mathcal{J}')}{B}$. Since we did not increase the processing times of any job in the creation of $\mathcal{J}'$ from $\mathcal{J}$, we have that $\ell(\mathcal{J}') = \ell(\mathcal{J})$. Furthermore, the busy time of the first machine is at most the span of $\mathcal{J}'$, i.e., $OPT_\infty(\mathcal{J}')$ which is at most $\beta \cdot OPT(\mathcal{J})$. Hence, combining the two, we get that the cost of the schedule produced by GreedyTracking on $\mathcal{J}'$ is at most $\beta\ OPT(\mathcal{J}) + \frac{2\ell(\mathcal{J})}{B}$. □

Theorem 6. *There exists a polynomial-time* 3*-approximation algorithm for the busy time scheduling problem.*

Proof. By Theorem 5, the output of GreedyTracking on $\mathcal{J}'$ gives a schedule with cost at most $OPT(\mathcal{J}) + \frac{2\ell(\mathcal{J})}{B}$. Since $OPT(\mathcal{J}) \geq \frac{\ell(\mathcal{J})}{B}$, the schedule of the tracking algorithm on $\mathcal{J}'$ is at most 3 times the cost of any optimal schedule for the original instance $\mathcal{J}$. □

Note that we do not claim that any α-approximation algorithm for the special case of interval jobs can be applied as a blackbox to yield an α-approximation algorithm for general busy time. The implication is specific to the analysis of the former algorithm (i.e., for interval jobs).

8.3 Busy Time for Preemptive Jobs

In this section, we study a variant of the general busy time problem (defined in Section 8.2 where jobs may or may not be interval jobs) in which jobs can be *preempted*. In particular, a job need not even be assigned to a single machine; the only constraints

on job J_j are that a total of p_j time units of it must be scheduled within the interval $[r_j, d_j)$ and at most one machine may be working on it at any given time.

Theorem 7. *For unbounded B and preemptive jobs, there is an exact algorithm to minimize busy time.*

Proof. The algorithm is a simple greedy one. Over the course of its execution, it maintains for each job J_j the amount $p'_j \leq p_j$ remaining to be done. Once p'_j becomes 0, J_j is considered to be satisfied. At each step, the algorithm considers the set $\mathcal{J}_1$ of unsatisfied jobs of earliest deadline d_1. It schedules as late as possible what remains of each job in $\mathcal{J}_1$, i.e. by opening the interval $[d_1 - C, d_1)$ to be busy, where

$$C = \max_{j \in \mathcal{J}_1} p'_j$$

is the maximum remaining length of any job in $\mathcal{J}_1$. Since B is unbounded, for all other unsatisfied jobs J_j (in particular, those not in $\mathcal{J}_1$), the algorithm schedules as much of J_j as possible in $[d_1 - C, d_1)$, updating p'_j to be $\max(0, p'_j - C)$. Finally, the algorithm contracts the interval $[d_1 - C, d_1)$, modifying the release times and deadlines of unsatisfied jobs appropriately. Then, it repeats on the contracted subproblem.

To see that this algorithm is correct, observe that without loss, the optimal solution will also open the interval $[d_1 - C, d_1)$; there exists some job of remaining length C whose deadline is d_1, and since d_1 is the earliest deadline, there is no reason for the optimal solution to schedule this job any earlier than $[d_1 - C, d_1)$. Furthermore, for each job, the algorithm schedules as much of it in $[d_1 - C, d_1)$ as possible. Thus, without loss, not only does the optimal solution also open $[d_1 - C, d_1)$, it schedules no more of any job in this interval than the greedy algorithm. It is irrelevant whether OPT also opens intervals earlier than $d_1 - C$ to be busy; it may be that it does – it still holds that without loss, OPT opens $[d_1 - C, d_1)$. Then, the correctness of the entire algorithm follows by induction on the number of iterations. □

As a consequence, one can approximate preemptive busy time scheduling for bounded B. First, solve the instance under the assumption that B is unbounded;

denote by $\mathcal{S}_\infty$ this (possibly infeasible) solution. The busy time of $\mathcal{S}_\infty$ is $OPT_\infty(\mathcal{J})$, and is a lowerbound on the optimal solution for bounded B. The algorithm for bounded B will commit to working on job J_j precisely in the time intervals where $\mathcal{S}_\infty$ had scheduled it. Partition the busy time of $\mathcal{S}_\infty$ into a set of maximal intervals $\{I_1, \ldots, I_k\}$, such that within each interval I_i, $\mathcal{S}_\infty$ neither starts nor stops working on any job. Per the greedy algorithm above, there are without loss a polynomial number of such intervals. We convert $\mathcal{S}_\infty$ into a feasible solution one interval I_i at a time. If $\mathcal{S}_\infty$ was working on $n(I_i)$ jobs in I_i, then assign these jobs (in any order) to $\lceil \frac{n(I_i)}{B} \rceil$ machines, filling the machines greedily such that there is at most one machine with strictly less than B jobs.

The following lower bounds immediately imply a 2-approximation: $OPT_\infty(\mathcal{J}) \leq OPT$ and $\frac{\ell(\mathcal{J})}{B} \leq OPT$. For each interval I_i, of all the machines that are busy in I_i, at most one contains less than B jobs; charge to $OPT_\infty(\mathcal{J})$ the cost of this machine (i.e. its span, that is, $|I_i|$). All other machines are at capacity, i.e., have exactly B jobs; charge their costs to $\frac{\ell(\mathcal{J})}{B}$. The approximation bound follows.

Theorem 8. *There is a preemptive algorithm whose busy time is at most twice that of the optimal preemptive solution, for bounded B.*

We note that the complexity of the preemptive busy time problem remains unresolved.

Chapter 9

CONCLUSIONS

This thesis introduces a simple and intuitive model for energy consumption in various operational contexts, most notably that of data centers. Current energy models either involve complex and/or a limited class of power functions [86, 54] or measure other relevant but different energy aspects, e.g., the number of gaps in a schedule [10]. The active time model is pertinent to settings in which the cost to operate at full capacity is equivalent to the cost for the system to be "on" at all, e.g., when overhead costs are high. We examine several special problems within this model, many of which admit polynomial-time solutions, and others for which we give approximations.

The other overarching contribution of this thesis is the *Cov-MECF* framework and with it, the *LPO* heuristic as well as an experimental examination of several important covering problems. Our results solidly establish *LPO* as a competitive (if not superior) alternative to Wolsey's greedy algorithm. In general, we find that freezing offers a significant boost, yielding a significant reduction in the number of flow computations, while very rarely harming the quality of the solution.

We suspect that the time bounds for the algorithms of Chapter 4 can be substantially improved. In fact, Koehler and Khuller [62] have already reduced the running time to $O(n^3)$ for the problem of minimizing the number of batches, via a non-DP approach. It is possible that a more sophisticated DP approach could improve this further. It also remains to find a strictly polynomial-time solution for the problem in which a budget on the number of batches is given.

Perhaps the most notable open problem is in determining whether the active time

problem with preemptive jobs of arbitrary length is in P (discussed in Chapter 6). Despite having no clear answer to this question, we provide a 5-approximation with a simple analysis. It is possible that this ratio can be decreased, e.g., via LP rounding approaches. In fact, it is even possible that the problem admits a polynomial-time solution. Resolving this issue in my opinion stands as the most interesting open question in this body of work.

Other directions for future work include closing the approximation gap for the busy time problem over interval jobs and considering more broadly the busy time problem with arbitrary demands. In this generalization, any point in time, the total sum of arbitrary demand of jobs on a given machine must never exceed B. All busy time problems studied in this thesis were for unit demand jobs. The problem of arbitrary demand was first studied by Khandekar et al. [59]. In addition, to the best of our knowledge, no lowerbounds on the best possible approximation ratio are known for the busy time problem, beyond NP-completeness.

One limitation of the active time model lies in its failure to account for latency. In particular, a schedule that frequently transitions between idle and active states may be considered high quality if the total active time is low. In some applications, this is acceptable, e.g. if the unit of time is large enough (e.g., several hours or even several days) that the time to turn on is negligible. However, in many situations where power is a precious commodity, including the ones discussed here, the transition from idle to active is hardly free; a time delay and also a power cost is almost always incurred. In that sense, the reality lies somewhere between the gap minimization model of Baptiste [10] and the active time model.

In the famous words of the late George E. P. Box, "all models are wrong, but some are useful" [14]. In spite of its limitations, the active time model remains a clean and natural way to capture important aspects of energy and operational costs. We anticipate its contribution to the collective understanding of power management and scheduling to be substantial.

BIBLIOGRAPHY

[1] R. K. Ahuja, Magnanti, T. L., and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., 1993.

[2] P. Ainsworth, M. Echenique, B. Padzieski, C. Villalobos, P. Walters, and D. Landon. Going green with IBM systems director active energy manager. *IBM Red Paper*, September 2008.

[3] S. Albers. Algorithms for energy saving. *Efficient Algorithms*, 5760:173–186, 2009.

[4] S. Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

[5] S. Albers and A. Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1266–1285, 2012.

[6] M. Alicherry and R. Bhatia. Line system design and a generalized coloring problem. In *Proceedings of the 11th Annual European Symposium on Algorithms*, pages 19–30, 2003.

[7] N. Bansal and K. Pruhs. The geometry of scheduling. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 407–414, 2010.

[8] N. Bansal and K. Pruhs. Weighted geometric set multi-cover via quasi-uniform sampling. In *Proceedings of the 20th Annual European Symposium on Algorithms*, pages 145–156, 2012.

[9] P. Baptiste. Batching identical jobs. *Mathematical Methods of Operations Research*, 52(3):355–367, 2000.

[10] P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 364–367, 2006.

[11] P. Baptiste, M. Chrobak, and C. Dürr. Polynomial time algorithms for minimum energy scheduling. In *Proceedings of the 15th Annual European Symposium on Algorithms*, pages 136–150, 2007.

[12] J. Bar-Ilan, G. Kortsarz, and D. Peleg. How to allocate network centers. *J. of Algorithms*, 15(3):385–415, 1993.

[13] H. L. Bodlaender and M. R. Fellows. W[2]-hardness of precedence constrained k-processor scheduling. *Operations Research Letters*, 18(2):93–97, 1995.

[14] G. E. P. Box and N. R. Draper. *Empirical Model Building and Response Surfaces*. John Wiley and Sons, 1987.

[15] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete and Computational Geometry*, 14(1):463–479, 1995.

[16] J. Chang, H. N. Gabow, and S. Khuller. A model for minimizing active processor time. In *Proceedings of the 20th Annual European Symposium on Algorithms*, pages 289–300, 2012.

[17] J. Chang and S. Khuller. A min-edge cost framework for capacitated covering problems. In *Proceedings of the 15th Annual Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 14–25, 2013.

[18] J. Chang, S. Khuller, and K. Mukherjee. Scheduling by groups: Minimizing busy time on multiple machines. *In submission*, 2013.

[19] J. Chuzhoy and J. Naor. Covering problems with hard capacities. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 481–489, 2002.

[20] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[21] E. G. Coffman, Jr. and M. R. Garey. Proof of the 4/3 conjecture for preemptive vs. nonpreemptive two-processor scheduling. *J. ACM*, 40(5):991–1018, 1993.

[22] A. Condotta, S. Knust, and N. V. Shakhlevich. Parallel batch scheduling of equal-length jobs with release and due dates. *J. of Scheduling*, 13(5):463–477, 2010.

[23] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2001.

[24] E. D. Demaine, M. Ghodsi, M. T. Hajiaghayi, A. S. Sayedi-Roshkhar, and M. Zadimoghaddam. Scheduling to minimize gaps and power consumption. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 46–54, 2007.

[25] E. D. Demaine and M. Zadimoghaddam. Scheduling to minimize power consumption using submodular functions. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–29, 2010.

[26] U. S. Department of Energy. Saving energy in data centers. `http://www1.eere.energy.gov/manufacturing/datacenters/about.html`.

[27] J. Du, Y.-T. Leung, and G. H. Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92:219–236, 1991.

[28] U. S. Environmental Protection Agency. Report to Congress on server and data center energy efficiency, Public Law 109–431, 2007.

[29] G. Even, G. Kortsarz, and W. Slany. On network design problems: fixed cost flows and the covering steiner problem. *Transactions on Algorithms*, 1(1):74–101, 2005.

[30] G. Even, D. Rawitz, and S. M. Shahar. Hitting sets when the VC-dimension is small. *Information Processing Letters*, 95(2):358–362, 2005.

[31] S. Even and R. Tarjan. Network flow and testing graph connectivity. *J. on Computing*, 4(4):507–518, 1975.

[32] U. Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[33] M. Flammini, G. Monaco, L. Moscardelli, H. Shachnai, M. Shalom, T. Tamir, and S. Zaks. Minimizing total busy time in parallel scheduling with application to optical networks. In *Proceedings of Interational Parallel Distributed Processing Symposium*, pages 1–12, 2009.

[34] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *J. on Applied Mathematics*, 17(4):784–789, 1969.

[35] H. N. Gabow. An almost-linear algorithm for two-processor scheduling. *J. ACM*, 29(3):766–780, 1982.

[36] H. N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 448–456, 1983.

[37] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. of Computer and System Sciences*, 30(2):209 – 221, 1985.

[38] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM*, 38(4):815–853, 1991.

[39] A. Gandhi, M. Harchol-Balter, and M. Kozuch. Are sleep states effective in data centers? In *Green Computing Conference (IGCC), 2012 International*, pages 1–10, 2012.

[40] R. Gandhi, E. Halperin, S. Khuller, G. Kortsarz, and A. Srinivasan. An improved approximation algorithm for vertex cover with hard capacities. *J. of Computer and System Sciences*, 72(1):16 – 33, 2006.

[41] M. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *J. on Computing*, 6(3):416–426, 1977.

[42] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freemand and Company, 1979.

[43] M. R. Garey and D. S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *J. ACM*, 23(3):461–467, 1976.

[44] A. Gerber, S. Sen, and O. Spatscheck. A call for more energy-efficient apps. `http://www.research.att.com/articles/featured_stories/2011_03/201102_Energy_efficient`, 2011.

[45] J. Gergov. Algorithms for compile-time memory optimization. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 907–908, 1999.

[46] O. Gerstel, R. Ramaswami, and G. Sasaki. Cost-effective traffic grooming in WDM rings. *Transactions on Networking*, 8(5):618–630, 2000.

[47] R. A. Giri and A. Vanchi. Increasing data center efficiency with server power measurements. *Intel Technical Report*, January 2010.

[48] A. Goyel, W. Lu, and L. V. Lakshmanan. CELF++: Optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 19th International World Wide Web Conference*, pages 47–48, 2011.

[49] S. Guha, R. Hassin, S. Khuller, and E. Or. Capacitated vertex covering with applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 858–865, 2002.

[50] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[51] IEEE. *Proceedings of the 3rd International Green Computing Conference*, 2012.

[52] Y. Ikura and M. Gimple. Efficient scheduling algorithms for a single batch processing machine. *Operations Research Letters*, 5(2):61 – 65, 1986.

[53] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005.

[54] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *Transactions on Algorithms*, 3(4), 2007.

[55] A. Java, P. Kolari, T. Finin, and T. Oates. Modeling the spread of influence on the blogosphere. In *Proceedings of the 14th International World Wide Web Conference*, 2006.

[56] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. of Computer and System Sciences*, 9(3):256–278, 1974.

[57] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. AMS, 1991.

[58] D. Kempe, J. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[59] R. Khandekar, B. Schieber, H. Shachnai, and T. Tamir. Minimizing busy time in multiple machine real-time scheduling. In *Proceedings of the 2010 Foundations of Software Technology and Theoretical Computer Science Conference*, pages 169 – 180, 2010.

[60] S. Khuller, J. Li, and B. Saha. Energy efficient scheduling via partial shutdown. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1360–1372, 2010.

[61] S. Khuller and Y. Sussmann. The capacitated k-center problem. *J. on Discrete Mathematics*, 13(3):403–418, 2000.

[62] F. Koehler and S. Khuller. Optimal batch schedules for parallel machines. In *Proceedings of the 13th Annual Algorithms and Data Structures Symposium*, pages 475–486, 2013.

[63] S. O. Krumke, H. Noltemeier, S. Schwarz, H.-C. Wirth, and R. Ravi. Flow improvement and network flows with fixed costs. In *Proceedings of the 1998 International Conference of Operations Research (OR'98)*, pages 158–167, 1998.

[64] V. Kumar and A. Rudra. Approximation algorithms for wavelength assignment. In *Proceedings of the 2005 Foundations of Software Technology and Theoretical Computer Science Conference*, pages 152–163, 2005.

[65] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity results for scheduling chains on a single machine. *European J. of Operation Research*, 4(4):270–275, 1980.

[66] J. Leskovec. SNAP library. `http://snap.stanford.edu/snap/license.html`.

[67] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429, 2007.

[68] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004.

[69] M. Li and F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *J. on Computing*, 35(3):658–671, 2005.

[70] A. López-Ortiz and C.-G. Quimper. A fast algorithm for multi-machine scheduling problems with jobs of equal processing times. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, pages 380–391, 2011.

[71] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[72] R. Lupton, M. Maley, and N. E. Young. Data collection for the Sloan Digital Sky Survey: A network-flow heuristic. *J. of Algorithms*, 27(2):339–356, 1998.

[73] S. Micali and V. V. Vazirani. An $o(\sqrt{|V||e|})$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[74] M. P. Michael. Energy awareness for mobile devices. *Research seminar on Energy Awareness*, 2005.

[75] N. H. Mustafa and S. Ray. PTAS for geometric hitting set problems via local search. In *Proceedings of the 25th Annual Symposium on Computational Geometry*, pages 17–22, 2009.

[76] G. L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[77] M. E. Newman. Coauthorships in network science. *Physical Review E*, 74, 2006.

[78] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3G networks. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pages 137–150, 2010.

[79] B. Saha and S. Khuller. Set cover revisited: hypergraph cover with hard capacities. In *Proceedings of the 39th International Conference on Automata, Languages and Programming*, pages 762–773, 2012.

[80] B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *J. on Computing*, 12(2):294–299, 1983.

[81] B. Simons and M. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *J. on Computing*, 18(4):690–710, 1989.

[82] K. Varadarajan. Weighted geometric set cover via quasi-uniform sampling. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, pages 641–648, 2010.

[83] P. Winkler and L. Zhang. Wavelength assignment and generalized interval graph coloring. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 830–831, 2003.

[84] L. Wolsey. An analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2(4):385–393, 1982.

[85] H. Wu and J. Jaffar. Two processor scheduling with real release times and deadlines. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 127–132, 2002.

[86] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 374–382, 1995.