This is to certify that the

dissertation entitled

NEW DIRECTIONS IN MACHINE SCHEDULING

presented by

Patchrawat Uthaisombut

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

[illegible]
Major professor

Date 8/21/2000

NEW DIRECTIONS IN MACHINE SCHEDULING

By

Patchrawat Uthaisombut

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2000

ABSTRACT

NEW DIRECTIONS IN MACHINE SCHEDULING

By

Patchrawat Uthaisombut

We explored several new directions in machine scheduling including bicriteria scheduling, extra-resource analysis, a new model of preemption, the k-client problem, and AND/OR linear programming.

First, we considered the problem of nonpreemptively scheduling jobs that arrive over time on one machine to simultaneously minimize both the makespan and the average completion time. Previous research focused on proving the existence of schedules that simultaneously approximate both criteria. We showed that optimal bicriteria schedules can be constructed in polynomial-time. By optimal, we mean that bicriteria schedules with a better bound for both criteria do not exist for some input instances.

Second, we applied extra-resource analysis to the load-balancing problem. Extra-resource analysis is a generalization of competitive analysis. Extra-resource analysis has been used to distinguish good and bad on-line algorithms whereas competitive analysis cannot. We used extra-resource analysis to derive a new type of result, namely a qualitative divergence between on-line and off-line load-balancing algorithms.

Third, we introduced a new model of preemption, the "preempt-decay" model. In this model, when a job is preempted, the work done on that job gradually decays

and has to be processed again. This model is a generalization of both the preempt-repeat and the preempt-resume models. We compared the optimal solution among the three models in the one machine environment.

Fourth, we studied the k-client problem which combines the ideas of multi-threaded environment and location-dependent processing time. In a multi-threaded environment, there are multiple chains of jobs. Jobs in each chain must be processed in order. In the environment where the processing time is location-dependent, the time required to process a job depends on the distance between the job and the server. When the underlying metric space in the k-client problem is a line, the problem becomes the multi-threaded disk scheduling problem. We analyzed two on-line greedy algorithms for the k-client problem, derived lower bounds in the line and the clique metric spaces, and considered the special case when $k = 2$.

Finally, we introduced AND/OR linear programming which is a generalization of ordinary linear programming. We devised an optimization scheme for AND/OR linear programs that has a small running time in practice. We outlined the application of AND/OR linear programs to the problem of finding a lower bound instance for an approximation algorithm. We demonstrated the technique on the problem of nonpreemptively scheduling jobs that arrive over time on one machine to minimize the total completion time.

To my parents

ACKNOWLEDGEMENTS

Many people contributed in many ways to make my dissertation possible. I take this opportunity to express my gratitude to them. First, I wish to express my deepest appreciation to Dr. Eric Torng, my advisor. Dr. Torng showed me the essence and the glory of theoretical computer science when I took his class. During my Ph.D. program, he always gave me ideas and new interesting problems to work on. I thank him for his encouragement, guidance, and support. I would like to thank Dr. Abdul Esfahanian, Dr. Matt Mutka, and Dr. Edgar Palmer, my committee members and also my teachers. I learned many things in and outside their classes. I also thank them for their encouragement.

I would like to thank my parents, Adun and Walaitip Uthaisombut, for their love, their support throughout my life, and their encouragement to persue higer education. I wish to thank my wife, Rujida (Leepipattanawit) Uthaisombut, for her love, support, and believe in me. Her presence gave me the courage to go forward. I would like to thank Dr. Robert Rasche and Mrs. Dorothy Rasche for their kindness throughout my stay at Michigan State University.

I would like to thank Mrs. Linda Moore for her help on all kinds of administrative work. Finally, I would like to thank all my friends for their help, their company, and the many interesting discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

Scheduling is the problem of allocating resources over time to perform a collection of tasks. Practical scheduling problems arise in a variety of situations. A problem could involve: jobs in a manufacturing plant, programs to be run in a computer system, bank customers waiting for services in front of tellers' windows, or airplanes waiting clearances to land or take off at an airport. Regardless of the application, there is a fundamental similarity among these problems. The vital elements in scheduling problems are resources, tasks, and objectives. Resources are typically characterized in terms of their qualitative and quantitative capacities, so that in a scheduling problem, each resource is described by its type and amount. Each task is typically described in terms of such information as its resource requirement, its duration, the time at which it may be started, and the time at which it is due. In addition, sometimes a collection of tasks is described in terms of precedence constraints that exist among the tasks. The objectives are some measure of goodness of solutions to a scheduling problem. Common goals in scheduling are high resources utilization, quick response to demands, close conformance to deadlines, and fairness.

1.2 Traditional Scheduling Problems

In this section, we describe a classification scheme of scheduling problems developed by Graham, Lawler, Lenstra, and Rinnooy Kan [38]. Suppose that m machines M_i ($i = 1, ..., m$) have to process n jobs J_j ($j = 1, ..., n$). A *schedule* is an allocation of one or more time intervals on one or more machines to each job. A schedule is *feasible* if at any time, there is at most one job on each machine, each job is run on at most one machine, and in addition, it meets a number of specific requirements concerning the machine environment and the job characteristics. A schedule is *optimal* if it minimizes (or maximizes) a given optimality criterion. A scheduling problem type can be specified using a three-field classification $\alpha|\beta|\gamma$ composed of the machine environment, the job characteristics, and the optimality criterion.

1.2.1 Job Data

First of all, the following data is specified for each job J_j:

- a *processing requirement* p_j in the case of single-operation models, or a collection of *processing requirements* p_{ij} in the case of multi-operation models;
- a *release date* r_j, on which J_j becomes available for processing;
- a non-decreasing real *cost function* f_j measuring the cost $f_j(t)$ incurred if J_j is completed at time t;

- a *due date* d_j, after which J_j is considered *late*, and which may be used in defining f_j;

- a *weight* w_j, which represents the importance of J_j relative to other jobs, and may be used in defining f_j.

1.2.2 Machine Environment

The first field $\alpha = \alpha_1\alpha_2$ specifies the machine environment. Let $\circ$ denote the empty symbol. If $\alpha_1 \in \{\circ, P, Q, R\}$, each job J_j consists of a single operation which can be processed on any machine M_i. Let p_{ij} denote the time to process J_j on M_i.

- $\alpha_1 = \circ$: *single machine*; there is only one machine. $p_{1j} = p_j$ for all J_j.

- $\alpha_1 = P$: *identical parallel machine*; there are multiple identical machine. $p_{ij} = p_j$ for all M_i and J_j.

- $\alpha_1 = Q$: *uniform machines*; there are multiple machines with different speeds. Each machine M_i has a *speed* s_i. $p_{ij} = p_j/s_i$ for all M_i and J_j.

- $\alpha_1 = R$: *unrelated machines*; there are multiple machines with different job-dependent speeds. Machine M_i runs job J_j with job-dependent speed s_{ij}. $p_{ij} = p_j/s_{ij}$ for all M_i and J_j.

If $\alpha_1 \in \{J, F, O\}$, each job J_j consists of a set of *operations* $\{O_{1,j}, O_{2,j}, ..., O_{m_j,j}\}$.

- $\alpha_1 = J$: *job shop*; the operations of each job J_j forms a chain $(O_{1,j}, O_{2,j}, ..., O_{m_j,j})$ which must be processed in that order, and O_{ij} must be processed on a given machine μ_{ij} requiring p_{ij} time units.

- $\alpha_1 = F$: *flow shop*; flow shop is a special case of job shop where $m_j = m$ and $\mu_{ij} = M_i$ for all jobs J_j.

- $\alpha_1 = O$: *open shop*; open shop is similar to flow shop except that the operations of any job J_j can be processed in any order.

If α_2 is a positive integer, then m is a constant, equal to α_2; it is specified as part of the problem *type*. If $\alpha_2 = \circ$, then m is a variable; the value of m is specified as part of the problem *instance*. Note that $\alpha_1 = \circ$ if and only if $\alpha_2 = 1$.

1.2.3 Job Characteristics

The second field $\beta \subseteq \{pmtn, r_j, \beta_{prec}\}$ indicates certain *job characteristics.*

- If *pmtn* is present, then *preemptions* are allowed; the processing of any job may be interrupted at no cost and resumed at a later time. Otherwise, no preemptions are allowed; once a jobs is started on a machine M_i, the job occupies the machine until it is completed.

- If r_j is present, then each job may have different *release dates.* Otherwise, all jobs arrive at time 0.

- If a *precedence constraint* β_{prec} is present, then there is a precedence relation $\prec$ among the jobs, *i.e.*, if $J_j \prec J_k$, then J_j must be completed before J_k can be started. If $\beta_{prec} = chain$, then $\prec$ forms chains. If $\beta_{prec} = tree$, then $\prec$ forms a tree. If $\beta_{prec} = prec$, then $\prec$ is an arbitrary partial order. If β_{prce} is not present, then jobs can be processed in any order.

The field β may contain some other job characteristics. They should be self-explanatory. It should be noted that there are many more types of precedence relations than those shown above.

1.2.4 Optimality Criteria

The third field γ specifies the *optimality criterion* or *the objective*, the value we wish to optimize. Given a feasible schedule, we can compute for each J_j

- the *completion time* C_j, the time at which the processing of job J_j is completed;
- the *flow time* $F_j = C_j - r_j$, the amount of time job J_j spends in the system;
- the *lateness* $L_j = C_j - d_j$, the amount of time by which the completion time of job J_j exceed its due date; lateness can be negative if job J_j finishes earlier than its due date;
- the *tardiness* $T_j = \max\{L_j, 0\}$, the lateness of job J_j if it fails to meet its due date, or zero otherwise;
- the *unit penalty* $U_j = 0$ if $C_j \leq d_j, U_j = 1$ otherwise, a unit penalty of job J_j if it fails to meet its due date.

The cost f_j for each job J_j usually takes on one of the variables described above or the product of the weight w_j with one of the variables. The optimality criterion can be any function of the costs $f_j, j = 1, ..., n$. Common optimality criteria are usually in the form $\sum_j f_j$ and $f_{\max} = \max_j f_j$. Example of common optimality criteria are the total weighted completion time $\sum w_j C_j$, the maximum completion time or the makespan $C_{\max}$, and the maximum lateness $L_{\max}$.

1.2.5 Examples of Scheduling Problems

We give a few examples on the three-field classification of scheduling problems. The problem $1|r_j|\sum w_jC_j$ is the problem of minimizing weighted completion time on one machine subject to non-trivial release dates. The problem $P3|pmtn, prec|L_{\max}$ is the problem of minimizing maximum lateness on three identical parallel machines subject to general precedence constraint, allowing preemption.

1.3 Approximation Algorithms and On-Line Algorithms

In this section, we describe approximation algorithms, on-line algorithms, common methods for evaluating their performance, and we compare and contrast approximation and on-line algorithms. First, we define some notations. The *supremum* of a non-empty set A of real numbers, denoted $\sup A$, is the smallest real number u such that $u \geq x$ for each $x \in A$. The *infimum* of A, denoted $\inf A$, is the greatest real number l such that $l \leq x$ for each $x \in A$. If A is finite, then both $\sup A$ and $\inf A$ belong to A. In this case, they are often called the *minimum* and *maximum* of A and are denoted by $\min A$ and $\max A$, respectively.

1.3.1 Approximation Algorithms

Many problems are NP-hard. It is unlikely that there exist efficient algorithms for solving NP-hard problems optimally. In this case, we should focus our effort to find efficient approximation algorithms. An *approximation algorithm* for a problem Π is an algorithm that, for any input instance I in Π, always give a feasible solution for I. We usually measure the performance of an approximation algorithm by its approximation ratio. Suppose we are considering a *minimization* problem. Let $\mathcal{A}(I)$ be the cost of the solution to input instance I produced by an algorithm $\mathcal{A}$. The *performance guarantee* or the *approximation ratio* of an approximation algorithm $\mathcal{A}$, denoted $R_{\mathcal{A}}$, is defined as

$$R_{\mathcal{A}} = \sup_I \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)} \tag{1.1}$$

where $\mathcal{OPT}$ is the optimal algorithm and the supremum is taken over all possible input instances of the problem. An algorithm $\mathcal{A}$ for a minimization problem is an *α-approximation algorithm* if $\alpha \geq R_{\mathcal{A}}$. For a *maximization* problem, the approximation ratio of an approximation algorithm $\mathcal{A}$ is defined as

$$R_{\mathcal{A}} = \inf_I \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)}. \tag{1.2}$$

An algorithm $\mathcal{A}$ for a maximization problem is an α-approximation algorithm if $\alpha \leq R_{\mathcal{A}}$. Note that the performance guarantee of algorithms is always greater than or equal to 1 for *minimization* problems and always smaller than or equal to 1 for *maximization* problems. In both cases, the performance guarantee is 1 for optimal algorithms.

1.3.2 On-Line Algorithms

In many real-life situations, we do not know all information about the input instance in advance. Yet we have to produce a partial solution to the partial input we have. For example, in an operating system, jobs arrive over time, and we only know of their existence when they arrive. Yet we have to construct a schedule over time without knowledge of the future.

To model this, *on-line* algorithms were introduced. An algorithm is *off-line* if it has all the knowledge about the input instance before it makes any decision. An algorithm is *on-line* if it must construct a partial solution to the currently known partial input without the knowledge of the future. In general, on-line algorithms cannot produce an optimal solution.

The performance of an on-line algorithm is usually evaluated by the *competitive analysis* technique introduced by Sleator and Tarjan [77]. The *competitiveness* or the *competitive ratio* of an on-line algorithm $\mathcal{A}$ for a minimization problem, denoted $c_{\mathcal{A}}$, is defined as

$$c_{\mathcal{A}} = \sup_{I} \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)} \tag{1.3}$$

where $\mathcal{OPT}$ is the optimal off-line algorithm and the supremum is taken over all possible input instances of the problem. An algorithm $\mathcal{A}$ for a minimization problem is *c-competitive* if $c \geq c_{\mathcal{A}}$. The competitive ratio of an on-line algorithm $\mathcal{A}$ for a *maximization* problem is defined as

$$c_{\mathcal{A}} = \inf_{I} \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)}. \tag{1.4}$$

An algorithm $\mathcal{A}$ for a maximization problem is c-competitive if $c \leq c_{\mathcal{A}}$.

1.3.3 Comparison Between Approximation Algorithms and On-Line Algorithms

Both approximation algorithms and on-line algorithms typically give suboptimal solutions because they have some forms of handicap. Approximation algorithms have a limited *computational power* They are allowed only polynomial time to produce a solution. In contrast, on-line algorithms have a limited *knowledge of the future.* At any time, they must produce a partial solution knowing only the information up to the current time.

Both approximation analysis and competitive analysis compare the performance of the algorithm of interest with that of the best off-line algorithm. Therefore, we can interpret the approximation ratio (competitive ratio) of an algorithm as its absolute performance measure. Alternatively, we can interpret the approximation ratio (competitive ratio) of an algorithm as a relative performance measure. By comparing the approximation ratios (competitive ratios) of different algorithms, we can differentiate the good ones from the bad ones.

1.4 New Directions for Scheduling Problems

In this section, we describe some recent research directions in the field of scheduling. We will discuss their motivations and some general results in the literature.

1.4.1 Multiple Objectives

In a long history of the scheduling theory, numerous algorithms have been designed to optimize many kinds of optimality criteria in many scheduling models. Typically, each criterion has been studied separately. Few studies considered multiple criteria together. Understanding interaction among multiple criteria is important because in real life, decision makers of scheduling problems usually have to consider multiple criteria before arriving at a decision [67]. Decision makers can gain useful insights from the trade-offs involving multiple criteria.

There were some bicriteria results in the literature which were byproducts of work on single criterion problems. For the problem of scheduling jobs on parallel identical machines, Graham showed that any list scheduling algorithm will produce a schedule with makespan at most twice the optimal makespan [36]. For the weighted completion time objective of the same model, Kawaguchi and Kyan showed that a list scheduling algorithm which orders jobs by non-increasing ratio of weight to processing time is an $(\sqrt{2}+1)/2$-approximation algorithm [51]. If all weights are equal, this algorithm is optimal for the average completion time [23]. Therefore, this algorithm performs well for both the makespan and the weighted completion time objectives.

An approach set out to explicitly study bicriteria scheduling is characterizing the set of "pareto-optimal" schedules. A set of schedules is *pareto-optimal* if there does not exist a schedule which is simultaneously better, in both criteria, than any of the schedules in the set. Many results on finding pareto-optimal sets are due to Van Wassenhove and Gelders [85], Nelson, Sarin, and Daniels [65], Garey, Tarjan, and Wilfond [32], McCormick and Pinedo [61], Hoogeveen [43, 42], Hoogeveen and Van de Velde [44].

A second approach to study a bicriteria scheduling problem is setting a constraint of the value of one criterion and optimizing the other criterion subject to the constraint. Smith studied the minimization of the average completion time on one machine subject to minimal maximum lateness [78]. Shmoys and Tardos studied the minimization of the average completion time on unrelated machines subject to the constraint that the makespan must be at most twice the optimal [76]. Hurkens and Coster showed that there exist instances for the problem of scheduling jobs on unrelated machines such that all optimal average completion time schedules have a makespan of $\Omega(\log n)$ times optimal [45].

A third approach to study a bicriteria scheduling problem is constructing a schedule that tries to optimize both criteria simultaneously. Chakrabarti *et al.* introduced a general technique for constructing algorithms that simultaneously optimize both the makespan and the average weighted completion time [18]. Wang studied the single machine schedule problem with release dates minimizing the makespan and the average weighted completion time [84]. Stein and Wein showed that for a very general class of scheduling problems, there exist schedules which are simultaneously at most 1.88-approximation for both the makespan and the average weighted completion time [80]. Recently Rasala extended this work and proved the existence of bicriteria schedules for several pairs of common scheduling criteria [70]. Her results apply to a very general class of scheduling problems. More literature in multiple criteria scheduling can be found in a survey paper by Nagar, Haddock, and Heragu [64].

Stein and Wein [80] introduced the following notation for bicriteria optimization problems. Suppose we have criteria (A, B), then a schedule S for an instance I is an *(α, β)-approximation schedule* or an *(α, β)-schedule* if the objective value of S for the criterion A is at most α times the optimal value for the criterion A and *simultaneously* the objective value of S for the criterion B is at most β times the optimal value for the criterion B. Similarly an *(α, β)-approximation algorithm* is an algorithm that always return a (α, β)-schedule.

In general, a schedule which is optimal in one criterion is not optimal in another criterion. Thus, the first step in studying bicriteria scheduling problems is to establish the values of α and β for which bicriteria (α, β)-schedules exist. The second step is to design algorithms to find such schedules.

1.4.2 Extra-Resource Analysis

The goal of competitive analysis is to evaluate the performance of on-line algorithms. The competitive ratio of an on-line algorithm can be interpreted as an absolute performance or a relative performance when compared to the competitive ratio of other on-line algorithms. However, in some scheduling problems, competitive analysis fails to offer useful information. The competitive ratios of good algorithms is extremely large, or the competitive ratios of good and bad algorithms are the same. For example, for the problem of on-line non-clairvoyant scheduling on single machine to minimize the average response time, Motwani, Phillips, and Torng showed that the deterministic competitive ratio is $\Omega(n^{1/3})$, and the randomized competitive ratio is $\Omega(\log n)$ [63].

Kalyanasundaram and Pruhs were the first to explicitly use *extra-resource analysis* of on-line algorithms [48] with the goal of identifying good on-line algorithms in settings where traditional competitive analysis fails to offer useful information. Extra-resource analysis is a relaxed notion of competitive analysis in which the on-line algorithm has more resources than the optimal off-line algorithm to which it is compared. Extra-resource analysis has been used to argue that certain on-line algorithms are good choices for solving specific problems as they perform well with respect to the optimal off-line algorithm when given extra resources [47, 69, 12, 49, 54, 57, 3, 27]. In scheduling problems, extra resources provided to on-line algorithms could be faster machines, extra machines, or a combination of both.

For the problem of on-line non-clairvoyant scheduling on single machine to minimize the average response time discussed above, Kalyanasundaram and Pruhs showed that there exists an on-line algorithm with the performance ratio of $1 + 1/\epsilon$ if it has a machine with speed $1 + \epsilon$ while the optimal off-line algorithm has a machine with speed 1 where $0 \leq \epsilon \leq 1$. Also, the same bound still holds if the on-line algorithm is equipped with a unit speed machine and an ϵ speed machine, instead of a $(1 + \epsilon)$ speed machine. This provides a practical way to improve the loss of system performance due to the on-line nature of the problem. By providing the on-line algorithm with either a faster machine or an extra machine the on-line algorithm can be constant competitive against the optimal off-line algorithm.

Extra-resource analysis is related to bicriteria competitive analysis. By thinking of the amount of resources used by algorithms as a parameter to be optimized, extra-resource analysis of a single-criterion problem can be thought of as a special case of bicriteria competitive analysis of a bicriteria problem.

1.4.3 New Models of Preemption

Traditional scheduling problems can be categorized into two models with respect to preemptions. In the *no-preemption* model, no preemptions are allowed; after a job is started, it must be executed to completion. An example of a scheduling problem in this model is the car rental problem. After a customer takes off with a car, the car cannot be recalled and rented to a second customer. The car can be rented to the second customer only when the first customer returns it. In the *preempt-resume* model, the execution of any job may be interrupted any number of times at no cost; preempted jobs resume execution from the point at which they were last preempted. An example for this model, is the scheduling of processes in a time-sharing operating system

Now let us consider a relaxation of the no-preemption model. In the *preempt-repeat* model, the execution of any job may be interrupted any number of times, but the work done on that job is completely lost. When the preempted jobs restart, they restart from the beginning. Any no-preemption schedule is itself a preempt-repeat schedule. Any preempt-repeat schedule can be converted into a no-preemption schedule by eliminating all preempted executions in the preempt-repeat schedule. Clearly, the elimination does not affect the completion of any job. For off-line problems, the no-preemption and the preempt-repeat models are equivalent because off-line algorithms have all the information about the input instance up front and can perform any conversion before producing the output.

However, in the on-line setting, the no-preemption and the preempt-repeat models are different. An on-line algorithm has more flexibility in the preempt-repeat model than it has in the no-preemption model. An on-line algorithm in the preempt-repeat model can decide to schedule a job and later decide to preempt that job to run another newly arrived job. In the no-preemption model, this type of action is prohibited. Scheduling a sound recording studio is an example for the preempt-repeat model. A recording of a song could be interrupted. However, the entire song must be recorded again.

Each of the preemption models described above realistically captures many real-life situations. However, there are still many practical situations for which none of these models accurately apply.

Consider a scenario in a metal casting factory where a piece of metal needs to be heated in a furnace for 60 minutes before it can be used. However, after the metal is heated for 40 minutes, there is another more urgent job; another piece of metal needs to be heated for 10 minutes. Since the furnace cannot accommodate both pieces of metal, the first piece has to be taken out before it is finished heating. After the second piece of metal is finished heating, the first piece of metal is inserted back into the furnace. The crucial point here is that during the time the metal is put outside of the furnace, it cools down. Suppose the rate that it cools down is one-half the rate that the furnace can heat up the metal. Then it takes 10/2=5 minutes to reheat the metal to the temperature just before it was taken out of the furnace. After that, it needs another 20 minutes to heat up to the desired temperature.

To capture time-dependent losses after preemptions, we propose a new model of preemption called the *preempt-decay* model. To facilitate the discussion, we define the following. The *effective remaining processing time* of job j at time t, denoted $p_j(t)$, is defined as follows. If job j starts or resumes its execution at time t, and there is no preemption while it is running, it will take exactly $p_j(t)$ time units to complete, *i.e.*, it will complete at time $t+p_j(t)$. The *effective completed processing time* $c_j(t)$ of job j at time t is defined as $c_j(t) = p_j - p_j(t)$. For example, $p_j(0) = p_j$ and $c_j(0) = 0$. If job j runs continuously from time t_1 to time t_2, then $c_j(t_2) = c_j(t_1) + (t_2 - t_1)$.

In the preempt-decay model, there is a *decay function* $d : Z^+ \to Z^+$. Preemptions are allowed with the following penalty. When a job J_j is idle for t time units as a result of a preemption, $d(t)$ units of work done on J_j are lost and have to be reprocessed. If job j is preempted at time t' and is idle for t time units, then $c_j(t'+t) = \max\{c_j(t') - d(t),\ 0\}$. It is natural to assume that $d(t)$ is a nondecreasing function because the longer a job is idle, the more the work done on that job should be lost. See Figure 1.1 for an example. This figures show a schedule S in the preempt-decay model for an instance I with a decay function $d(t) = t/2$. The graphs on the bottom half of the figure show $c_j(t)$.

The preempt-resume and the preempt-repeat models are special cases of the preempt-decay model. If $d(t) = 0$, the preempt-decay model specializes to the preempt-resume model. If $d(t) = \infty$, the preempt-decay model specializes to the preempt-repeat model. A preempt-decay problem with $d(t) = c$ where c is a positive finite constant models the operating system scheduling problem with a context switching cost of c.

1.4.4 Multi-Thread Environment

In traditional scheduling problems, there are 2 choices with respect to the arrival of jobs. In the first model, all jobs arrive at the same time, typically at time 0. Many scheduling problems are easy to solve when all jobs arrive at the same time. For example, $1||\sum w_jC_j$ can solved by the **Weighted Shortest Processing Time** (WSPT) algorithm [78], $1||L_{\max}$ can be solved by the **Earliest Due Date** (EDD) algorithm [9], and $1||\sum U_j$ can be solved by Moore's algorithm [62].

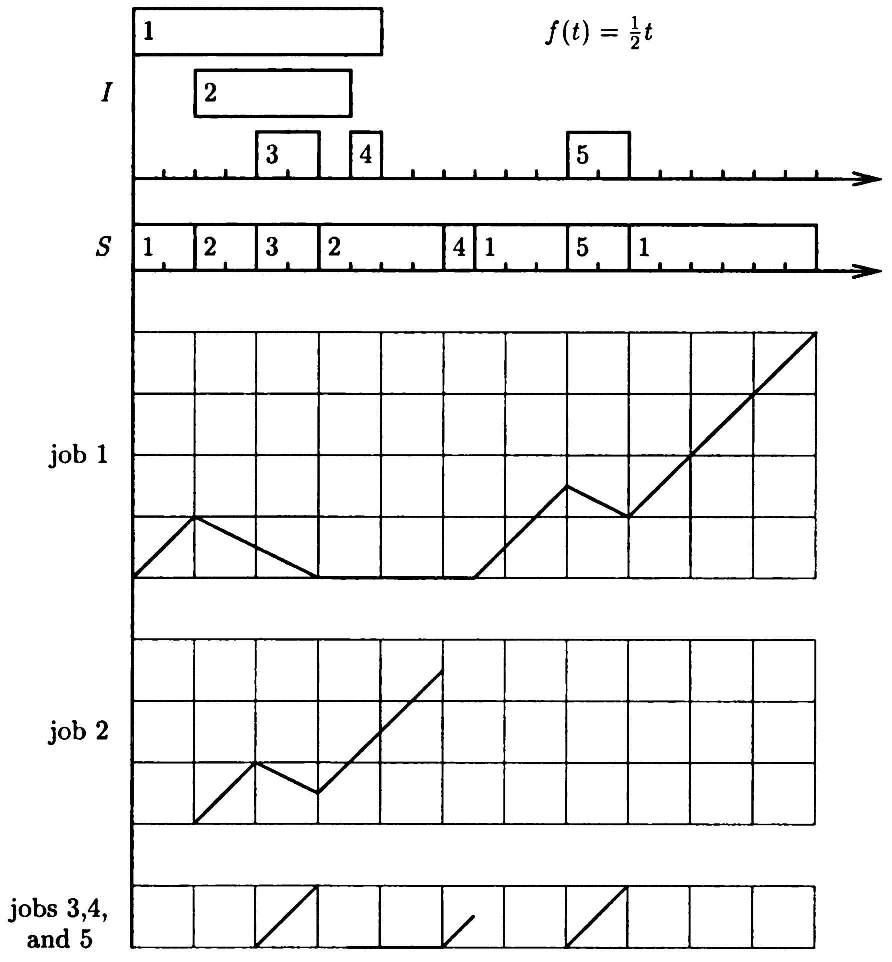


Figure 1.1: Example of a schedule in the preempt-decay model.

In the second model, jobs could arrive at different times. Each job has a fixed *release time*. Many scheduling problems become difficult when jobs arrive at different times especially those where preemptions are not allowed. Examples of NP-complete scheduling problems are $1|r_j|\sum C_j$, $1|r_j|L_{\max}$, and $1|r_j|\sum U_j$ [59]. Some preemptive scheduling problems remain easy to solve when jobs arrive at different times. Examples of polynomially solvable problems are $1|pmtn, r_j|\sum C_j$ [9], $1|pmtn, r_j|f_{\max}$ [10], and $1|pmtn, r_j|\sum U_j$ [58].

Single-Thread Scheduling Problems

In the past few years, another model emerged. In this model, there is an *arrival dependency* among jobs. One common model is a chain where jobs have a linear arrival dependency among them. In a chain, initially, only the first job is available to be scheduled. The next job will become available only after the current job is scheduled.

A representative problem for this model is the load balancing problem. In this model, there are m machines, and a list of jobs. Jobs become available one at a time and the next job will become available only after the current job is scheduled. The *load* of a machine is the sum of the processing time of the jobs on that machine. The objective is to schedule all jobs and minimize the makespan, the maximum load on all machine.

A note on availability of a job. In the schedule itself, *time* has no meaning. When a job becomes available for the algorithm to schedule, the job can be scheduled anywhere in the schedule.

Multi-Threaded Scheduling Problems

Significant work has been done in on-line scheduling problems. However, most of this work abstracts away the existence of multiple threads. In most previous studies of on-line algorithms, researchers have assumed that the system or algorithm must cope with a single request sequence; in particular, at any given time, there is at most one outstanding request in the system and future requests will not arrive until the current request is serviced (in some problems, the system is allowed to choose to not service the current request). Because of this assumption, the underlying problem addressed by most previous work in on-line algorithms has been deciding which system resource(s), if any, should be allocated to service the current request. Two of the many examples of interesting problems which fall into this single request sequence model are the paging problem [77, 60, 30] and its generalizations, the k-server and generic task system problems [60, 56, 14].

While the single request sequence model captures many important problems, there are many others which do not fall into this category, such as some operating system scheduling problems [48, 28, 29, 63] and some real-time scheduling problems [53, 11, 26]. In a typical problem, there is a single system resource such as a processor and, at any given time, there are multiple requests in the system waiting to be serviced. As a result, the underlying problem is deciding which current request should the system service rather than which system resource to use. Note that in many cases, there are multiple resources and multiple requests so the system needs to decide which requests to service as well as which system resources to use.

While this multiple request model captures the scheduling aspect of many systems, it loses the thread-based or transaction-based nature of the single request model where the arrival of requests is dependent on whether or not the system processed previous requests. For example, in practice, for relatively long stretches of time, there is a fixed number of users on an operating system making requests to the disk. Furthermore, each user or client generates a sequence of requests for service where each request is only generated after the previous request of the client has been serviced. This thread-based client model also describes database systems where users perform transactions which constitute a series of atomic operations in the database.

In a scheduling problem in a thread-based environment, each job is composed of a chain of operations Each operation has a processing time, the amount of time the operation requires to run on the machine. The first request from each job is released when the job is released. Other requests are released when the previous request from the same job runs to completion. Traditional scheduling problems assume that requests are independent and that each of them has a fixed release time. Examples of threads are the operating system scheduling problem and the disk scheduling problem. In those problems, for relatively long stretches of time, there is a fixed number of clients utilizing the system. Each client generates a sequence of requests for service where each request is only generated after the previous request of that client has been serviced.

In order to capture (i) the multiple client nature and (ii) the thread-based client nature of problems such as operating system scheduling and disk scheduling, we study the k-client problem.

1.4.5 Location-Dependent Processing Time

In a most general form of traditional scheduling, the processing time of a job depends on both the job and the machine. This model is not rich enough to model many practical problems. For example, in a disk scheduling problem, there is a disk, a server, and requests. The requests appear at different locations on the disk. The server can move around on the disk. To service a request, the server must move from its current position to the location of the request. The cost of servicing a request is largely dependent on the distance the server has to move. The distance the server has to move to service a request depends on how the server has serviced previous requests and thus varies from algorithm to algorithm. There is a large body of work analyzing disk scheduling algorithms [22, 81, 66, 21, 33, 75, 86, 5].

1.4.6 AND/OR Linear Programming

Previously, linear programming and integer programming have been used in the design and analysis of scheduling problems. We introduce AND/OR linear programs which are a generalization of ordinary linear programs. AND/OR linear programs can be used to find hard instances of scheduling problems against a fixed approximation algorithm.

1.5 Organization of the Dissertation

The rest of the proposal is organized as follows. In Chapter 2, our preliminary results on bicriteria single machine scheduling minimizing the makespan and the total completion time are presented. In Chapter 3, we present a new application of extra-resource analysis for the load-balancing problem. In Chapter 4, we present our preliminary results on single machine preempt-decay scheduling. We studied a relationship between the optimal total flow time among the preempt-resume, preempt-repeat, and the preempt-decay models. In Chapter 5, we present our on-line results on the k-client problem. The ideas of multi-threaded environment and location-dependent processing time are used to model this problem. In Chapter 6, we study the problem of nonpreemptively scheduling jobs that arrive over time on one machine minimizing the total completion time. We show that a class of approximation algorithms cannot guarantee an approximation ratio better than $e/(e-1) \approx 1.58$. In Chapter 7, we introduce AND/OR linear programs, a generalization of ordinary linear programs. We use AND/OR linear programs to find hard instance of the problem studied in Chapter 6. Each of the chapters above contains an introduction to the specific settings of the problems studied, some more specific related previous works, the details of the results, and related interesting open problems.

Chapter 2

Bicriteria Scheduling

2.1 Introduction

We study a bicriteria scheduling problem in a single machine environment with release times. The two criteria we want to minimize are the makespan, the maximum completion time of any job, and the average completion time. We denote this problem by $1|r_j|(C_{\max}, C_{\text{avg}})$.

Stein and Wein proved that for any instance and $0 < \alpha < 1$, there exists a $(1+\alpha, \frac{1}{\alpha})$-approximation schedule which is a schedule with the makespan at most $1+\alpha$ times the optimal makespan and the average completion time at most $\frac{1}{\alpha}$ times the optimal average completion time [80]. In fact, their results apply to a very large class of scheduling problems. They also showed that there exist instances for which there is no (x, y)-schedule with both x and y simultaneously smaller than $\frac{\sqrt{5}+1}{2} \approx 1.618$. Later, Aslam, Rasala, Stein, and Young [6] improved the upper bound to $(1+\alpha, \frac{e^\alpha}{e^\alpha - 1})$-approximation for $0 < \alpha < 1$. Their results also apply to a large class of scheduling problems.

A natural open problem is to close the gap between the upper bound and the lower bound of the quality of bicriteria approximation schedules. Another natural question is the time complexity of constructing such schedules. We answer these two questions for the single machine environment where jobs arrive over time $(1|r_j|(C_{\max}, C_{\text{avg}}))$. Our main results are the following. First, we show that the upper bound found by Aslam *et al.* [6] is best possible by constructing a family of matching lower bound instances. This is shown in Section 2.2. Secondly, we show that this bound can be achieved by a deterministic polynomial-time algorithm. This is shown in Section 2.3. Our algorithm, called $\mathcal{BEST}$-β, is simply a slightly generalized version of the $\mathcal{BEST}$-α algorithm by Chekuri, Motwani, Natarajan, and Stein [19]. Interestingly, $\mathcal{BEST}$-α is an $\frac{e}{e-1}$-approximation algorithm intended for the unicriteria problem of minimizing the average completion time on one machine with release times $(1|r_j|C_{\text{avg}})$. Based on our preliminary results, Rasala [70] analyzed the bicriteria performance guarantee of $\mathcal{BEST}$-β for several combinations of criteria in the single machine environment with release time. She also analyzed our lower bound instances in these settings.

2.2 Lower Bound of the Quality of Bicriteria Schedules

In the proof of the following theorem, we use Dirac's delta function $\delta(\cdot)$ which is defined as follows.

$$\begin{aligned} g_\Delta(t) &= \begin{cases} 1/\Delta & 0 < t < \Delta \\ 0 & \text{elsewhere} \end{cases} \\ \delta(\cdot) &= \lim_{\Delta \to 0^+} g_\Delta(\cdot) \end{aligned}$$

Note that

$$\int_u^v \delta(t)\,dt \;=\; \begin{cases} 0 & \text{if } 0 < u < v \\ 1 & \text{if } u \leq 0 < v. \end{cases}$$

Theorem 2.2.1. *For $0 < \beta < 1$, there exists an (infinite-size) instance such that there does not exist a (x, y)-schedule where $x < 1 + \beta$ and simultaneously $y < \frac{e^\beta}{e^\beta - 1}$.*

Proof. Fix β where $0 < \beta < 1$. The lower bound instance has the following structure. The number of jobs, n, approaches infinity. There is one job of size 1 released at time 0. All other jobs have size 0. Among all 0-size jobs, $e^{-\beta}$ fraction of them are released at time 1. The rest of the 0-size jobs are released independently in the interval $(0, \beta)$ with the density e^{-t} at time t. The release time of all 0-size jobs can be described as the following probability density function f:

$$f(t) \;=\; e^{-\beta}\,\delta(\beta - t) + \begin{cases} e^{-t} & \text{for } 0 < t < \beta \\ 0 & \text{elsewhere.} \end{cases}$$

Any reasonable schedule for this instance can be described by a single parameter s which represents the starting time of the unit-size job in that schedule. All 0-size jobs which are released before time s are run as soon as they arrive. All 0-size jobs which are released after time s are run as soon as the unit-size job is finished, *i.e.*, they are run at time $s + 1$. Since all 0-size jobs are released no later than time β, then we only need to consider the case $0 \leq s \leq \beta$. Let $C^s_{\max}$ and C^s_{avg} denote the makespan and the average completion time of the schedule which starts the unit-size job at time s respectively. Since all jobs finish before or at the same time as the unit-size job, then

$$C^s_{\max} = 1 + s.$$

Since the number of jobs approaches infinity, the completion time of the unit-size job is negligible. The average completion time can be computed as follows:

$$\begin{aligned}
C^s_{\text{avg}} &= \int_0^s tf(t)dt + \int_s^\beta (s+1)f(t)dt \\
&= \int_0^s t\left(e^{-t} + e^{-\beta}\delta(\beta - t)\right)dt + (s+1)\int_s^\beta \left(e^{-t} + e^{-\beta}\delta(\beta - t)\right)dt \\
&= \begin{cases} \int_0^s te^{-t}dt + (s+1)\left(\int_s^\beta e^{-t}dt + e^{-\beta}\right) & 0 \leq s < \beta \\ \int_0^\beta te^{-t}dt + \beta e^{-\beta} & s = \beta \end{cases} \\
&= \begin{cases} 1 & 0 \leq s < \beta \\ \frac{e^\beta - 1}{e^\beta} & s = \beta. \end{cases}
\end{aligned}$$

The two objectives are conflicting. When $s = \beta$, the optimal average completion time of $\frac{e^\beta-1}{e^\beta}$ is achieved, and the makespan is $1+\beta$. When $0 \leq s < \beta$, the average completion time is 1 which is exactly $\frac{e^\beta}{e^\beta - 1}$ times the optimal average completion time. When $s = 0$, the optimal makespan of 1 is achieved. Thus, there does not exist a value for s such that the average completion time is strictly less than $\frac{e^\beta}{e^\beta-1}$ times the optimal average completion time unless the makespan is exactly $1+\beta$ times the optimal makespan. Therefore, the result follows. $\square$

2.3 Constructing Bicriteria Schedules in Polynomial Time

We begin this section by stating some definitions from [19]. The **Shortest Remaining Processing Time** ($\mathcal{SRPT}$) algorithm is a preemptive algorithm which always runs a job with the shortest remaining processing time. Note that $\mathcal{SRPT}$ is optimal for the problem of preemptively scheduling jobs that arrive over time on one machine to minimize the average completion time [9]. Let P be the *preemptive* schedule produced by $\mathcal{SRPT}$. For $0 \leq \alpha \leq 1$, let $C^P_j(\alpha)$ be the time at which an α-fraction of

J_j is completed in schedule P. An α-schedule is a *nonpreemptive* schedule obtained by list scheduling jobs in order of increasing $C_j^P(\alpha)$. Randomized algorithm $\mathcal{RAND}_f$ chooses an α-schedule randomly according to the probability density function f over [0,1]. $\mathcal{RAND}$-β is $\mathcal{RAND}_f$ with the probability density function

$$f(t) \;=\; \begin{cases} \frac{e^t}{e^\beta - 1} & \text{for } 0 \le t \le \beta \\ 0 & \text{everywhere else.} \end{cases}$$

Theorem 2.3.1. *For $0 < \beta \le 1$, algorithm $\mathcal{RAND}$-β is a randomized $(1+\beta, \frac{e^\beta}{e^\beta-1})$-approximation algorithm.*

The theorem follows from the following two results by Chekuri *et al.*. [19].

Lemma 2.3.1. *[19] The makespan of any α-schedule is at most $1+\alpha$ times the optimal* preemptive *makespan.*

Lemma 2.3.2. *[19] For any probability density function f over [0,1], the expected average completion time of $\mathcal{RAND}_f$ is at most $1+\gamma$ times the optimal* preemptive *average completion time where*

$$\gamma \;=\; \max_{0<t\le 1} \int_0^t \frac{1+\alpha-t}{t} f(\alpha)d\alpha.$$

Observe that in the preemptive $\mathcal{SRPT}$ schedule, there are at most $n-1$ preemptions. Thus, given an instance, there are at most $n-1$ critical values of α, and n different α-schedules. We can try all α-schedules and choose the best one in polynomial time. Deterministic algorithm $\mathcal{BEST}$-β considers all α-schedules where $0 \le \alpha \le \beta$ and chooses the one with the minimum average completion time. Thus, the performance guarantee of $\mathcal{BEST}$-β is at least as good as the performance guarantee of $\mathcal{RAND}$-β. Therefore, the following corollary follows from Theorem 2.3.1.

Corollary 2.3.1. *For $0 \le \beta \le 1$, algorithm $\mathcal{BEST}$-β is a deterministic polynomial-time $(1+\beta, \frac{e^\beta}{e^\beta - 1})$-approximation algorithm and runs in $O(n^2 \log n)$ time.*

2.4 Open Problems in Bicriteria Scheduling

Bicriteria scheduling is a relatively new direction in machine scheduling. There are still many open problems. Examples of open problems are $1|r_j|(F_{\max}, \sum w_j C_j)$, $1|r_j|(F_{\max}, \sum(1-U_j))$, and $P2|r_j|(C_{\max}, \sum C_j)$.

Chapter 3

Applying Extra-Resource Analysis to Load Balancing

3.1 Introduction

In the past few years, the extra-resource analysis technique popularized by Kalyanasundaram and Pruhs [48] has been used for analyzing the performance of on-line algorithms for problems where traditional competitive analysis yields non-constant competitive ratios. In many cases, the traditional competitive analysis does not differentiate between good and bad algorithms. For these problems, extra-resource analysis has been used to argue that certain on-line algorithms are good choices for solving specific problems because they perform well with respect to the optimal off-line algorithm when given extra resources [47, 69, 13, 49, 54, 57, 3, 27].

In this chapter, we use extra-resource analysis to provide us with more insight into the behavior of specific load balancing algorithms. This leads us to a qualitative divergence between on-line and off-line algorithms for the load balancing problem. This result also reemphasizes the value of sorting by job size before performing list scheduling.

3.1.1 Problem Definitions

In the load balancing problem, we are given m identical machines and a job list I of n jobs. All jobs arrive at time 0. For $j = 1, ..., n$, job j has a processing time p_j. Let $p_{\min} = \min_{1 \le j \le n} p_j$. An assignment algorithm chooses a machine i for each job j, and that job runs on that machine until completion. Let $s_j^{\mathcal{A}}(m, I)$ and $C_j^{\mathcal{A}}(m, I)$ denote the starting time and the completion time, respectively, of job j in the schedule produced by algorithm $\mathcal{A}$ for job list I on m machines. Note that $C_j^{\mathcal{A}}(m, I) = s_j^{\mathcal{A}}(m, I) + p_j$. Let $C_{\max}^{\mathcal{A}}(m, I)$ denote the makespan, the maximum load on any machine, of the schedule produced by algorithm $\mathcal{A}$ for job list I on m machines, which is defined as $C_{\max}^{\mathcal{A}}(m, I) = \max_j C_j^{\mathcal{A}}(m, I)$. We will drop the arguments m and/or I if there is no ambiguity. The goal of the load balancing problem is to minimize the makespan.

An algorithm is *on-line* if it must permanently assign the current job to a machine before it is aware of the next job in the list. An algorithm is *off-line* if it is aware of the entire job list in advance. We say that an algorithm $\mathcal{A}$ is an (m, k)-*machine ρ-approximation* algorithm if

$$\sup_I \frac{C_{\max}^{\mathcal{A}}(m + k, I)}{C_{\max}^{\mathcal{OPT}}(m, I)} \le \rho$$

where $\mathcal{OPT}$ denotes the optimal off-line algorithm. We say that an algorithm $\mathcal{A}$ is *ρ-approximation* if $\mathcal{A}$ is an $(m, 0)$-machine ρ-approximation for all $m \ge 1$. We say that an algorithm $\mathcal{A}$ is (m, k)-*optimal* if $\mathcal{A}$ is an (m, k)-machine 1-approximation algorithm.

We study two algorithms in particular. The **List-Scheduling** algorithm ($\mathcal{LS}$) runs the next job in the list on the machine with the smallest load. The **Longest-**

Processing-Time algorithm ($\mathcal{LPT}$) runs the *longest* unscheduled job on the machine with the smallest load. $\mathcal{LPT}$ is an off-line algorithm while $\mathcal{LS}$ is an on-line algorithm.

3.1.2 Related Results

The load balancing problem is NP-hard [31]. Graham showed that $\mathcal{LS}$ is a $(2 - \frac{1}{m})$-approximation algorithm [36] and $\mathcal{LPT}$ is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm [37] for any $m \geq 1$. Hochbaum and Shmoys devised a polynomial time approximation scheme (PTAS) for this problem [41]. For the on-line setting, Albers introduced a deterministic on-line 1.923-approximation algorithm, and she proved a deterministic on-line lower bound of 1.852 [2]. Recently, the lower bound was improved to 1.85358 by Gormley, Reingold, Torng, and Westbrook [35].

Kalyanasundaram and Pruhs were the first to explicitly use *extra-resource analysis* of on-line algorithms [48] with the goal of identifying good on-line algorithms in settings where traditional competitive analysis fails to do so. The extra-resource analysis technique is a relaxed notion of competitive analysis in which the on-line algorithm has more resources than the optimal off-line algorithm to which it is compared. Following this paper, several more studies have used the extra-resource analysis technique in a wide variety of settings [47, 69, 13, 49, 54, 57, 27, 3].

Azar, Epstein, and van Stee have independently and in parallel considered the problem of load balancing with extra resources [8]. In contrast to our work, they only consider on-line algorithms. Their main result is an on-line load balancing algorithm with an approximation ratio which decays exponentially in $(m + k)/m$ as well as a nearly matching lower bound. They also considered other problem features such as allowing a job to be scheduled on multiple machines, temporary tasks, and related and unrelated processors. We only consider identical machines and permanent tasks which must be scheduled on only a single machine.

Another approach for identifying good on-line algorithms when competitive analysis fails to yield useful results is to restrict the set of legal input instances in some form. This has been proposed and used in a variety of settings [15, 55, 16, 17]. In contrast, extra-resource analysis compares the performance of on-line/off-line approximation algorithms to that of the optimal off-line algorithm on *all* possible input instances; however, the approximation algorithms are given more resources than the optimal off-line algorithm.

Our problem is also related to the well studied and NP-hard bin packing problem [20, 31]. In the bin packing problem, we have to pack items into a minimum number of bins of known size. An item can be placed into a bin only if it fits in the available space. Johnson *et al.* prove that two simple on-line bin packing algorithms, **First-Fit** and **Best-Fit**, are $(\frac{17}{10}m + 2)$-approximation algorithms [46]. Richey introduced an on-line algorithm **Harmonic+1** which has an asymptotic worst case ratio smaller than 1.5888 [71]. Karp and Karmarkar devised an asymptotic fully polynomial-time approximation scheme (asymptotic FPTAS) for off-line bin packing [50].

Dell'Olmo *et al.* studied off-line bin packing with extendable bins [25]. In this problem, we have to pack items into a fixed number of bins of known size. However, the size of each bin can be extended if needed. The "final" size of a bin is the maximum between the original bin size and the total size of items in that bin. The goal is to minimize the *sum* of the final size of all bins. In contrast, the goal of load balancing is to minimize the *maximum* load on any machine. Speranza and Tuza studied the on-line version of the bin-packing problem with extendable bins [79].

Azar and Regev studied the on-line bin-stretching problem [7]. In this problem, the number of bins is fixed, and we wish to pack all the items into the bins while we stretch the size of the bins as little as possible. The on-line algorithm is presented with one item at a time, and it has to pack the item before the next item is presented to it. This problem is equivalent to the load balancing problem except that the optimal makespan (maximum load) is known in advance. In contrast, we study the load balancing problem where approximation algorithms do not know the optimal makespan. Furthermore, in our study, approximation algorithms have more machines (bins) than the optimal off-line algorithm. In a way similar to the work by Azar and Regev, we compare the makespan (bin stretch) of the schedules produced by approximation algorithms and the optimal off-line algorithm.

The property of an algorithm being (m, k)-optimal for the load balancing problem with extra machines is similar to being an $\frac{m+k}{m}$-approximation algorithm for the bin packing problem. Both types of algorithms can pack items into $m + k$ bins when the optimal off-line algorithm needs m bins. Load balancing algorithms know the number of bins used, but they do not know the optimal makespan. In contrast, bin packing algorithms know the bin size but do not know the number of bins used by the optimal algorithm. Thus, an (m, k)-optimal load balancing algorithm can be thought of as an algorithm for solving a bin packing problem with unknown bin size but with a specified number of bins.

3.1.3 Summary of Results

The extra-resource analysis technique has been used to derive insight into the behavior of on-line and off-line algorithms. Previously, these insights have been used to identify good on-line algorithms in settings where traditional competitive analysis fails to do so. In this work, we show that these insights can also be used to derive a qualitative divergence between off-line and on-line load balancing algorithms.

In Section 3.2, we give on-line results. We begin by extending Graham's results [36] on the performance of $\mathcal{LS}$ for the load balancing problem to the case when $\mathcal{LS}$ has $m+k$ machines while $\mathcal{OPT}$ has m machines. We show that $\mathcal{LS}$ is a (m,k)-machine $(1+\frac{m-1}{m+k})$-approximation algorithm. We give a lower bound instance to show that this result is tight. The lower bound for $\mathcal{LS}$ can be generalized to apply to all on-line algorithms. In particular, we prove that no on-line algorithm is (m,k)-optimal for any k. It should be noted that although no on-line algorithm is (m,k)-optimal for any k, the makespan of the on-line schedule could be asymptotically close to the optimal makespan as can be seen from the performance guarantee of $\mathcal{LS}$.

In Section 3.3, we give off-line upper bound results. By extending Graham's result [37] on $\mathcal{LPT}$, we show that $\mathcal{LPT}$ is an (m,k)-machine $(\max\{\frac{4m+k-1}{3m+3k},1\})$-approximation algorithm. This implies that $\mathcal{LPT}$ is (m,k)-optimal when $k \geq \frac{m-1}{2}$; that is, if $\mathcal{LPT}$ has at least $\frac{3m-1}{2}$ machines and $\mathcal{OPT}$ has m machines, then $\mathcal{LPT}$ will produce a schedule with a makespan no larger than that of $\mathcal{OPT}$.

Next, we refine this result to show that $\mathcal{LPT}$ is (m,k)-optimal when $k \geq \frac{m-1}{3}$. The proof is based on a seemingly trivial but important fact that the sum of the processing times of jobs on each machine in the optimal schedule is no larger than the optimal makespan. To exploit this fact, new ideas are required which we now briefly summarize. The main idea is to not classify jobs by their absolute sizes, but to classify each job in a schedule by (1) the number of jobs on the same machine and (2) the order of its size among jobs on the same machine. The classification allows us to establish some crucial relationships between the optimal schedule and the $\mathcal{LPT}$ schedule. These relationships enable us to identify a most heavily loaded machine y in the $\mathcal{LPT}$ schedule and some machine z in the optimal schedule such that the ith largest job on machine y is no larger than the ith largest job on machine z. Therefore, the makespan of the $\mathcal{LPT}$ schedule is no larger than the makespan of the optimal schedule.

Results in Sections 3.2 and 3.3 imply a divergence between off-line and on-line algorithms because simple off-line algorithms such as $\mathcal{LPT}$ guarantees (m, k)-optimality with only a few extra machines whereas no on-line algorithm can guarantee (m, k)-optimality with any number of extra machines. This result also underscores the value of sorting before performing list scheduling. Namely, if the list is sorted in non-increasing order ($\mathcal{LPT}$), then we can achieve (m, k)-optimality with $k = \frac{m-1}{3}$. If, however, the list is not sorted in non-increasing order, (m, k)-optimality through list scheduling cannot be guaranteed for any value of k.

Results in Section 3.2 also imply a difference between load balancing and bin packing. Consider an (m, k)-optimal load balancing algorithm and an $\frac{m+k}{m}$-approximation bin packing algorithm. Both algorithms can pack all items into $m + k$ bins without overpacking any bin whereas the optimal algorithm needs m bins. However, each of them knows different pieces of information. A load balancing algorithm knows how many bins is needed by the optimal algorithm, but it does not know the optimal makespan (bin size). A bin packing algorithm knows the bin size but it does not know how many bins is needed by the optimal algorithm. The performance achievable by on-line algorithms in each of the problems are quite different. No on-line load balancing algorithm can guarantee not to overpack the bins. In contrast, simple on-line bin packing algorithms such as **First-Fit** and **Best-Fit** need only $\frac{17}{10}m + 2$ bins to pack all items [46]. This result indicates that the optimal makespan (bin size) is a more important piece of information than the optimal number of bins.

In Section 3.4, we describe a procedure to compute a good, though not necessarily optimal, lower bound of the performance of $\mathcal{LPT}$ for any m and k.

3.2 On-Line Results

We begin this section by proving the following lemma which is an extension of Graham's analysis for $\mathcal{LS}$ [36] and $\mathcal{LPT}$ [37] to the case when list scheduling algorithms have k extra machines. Note that the lemma applies to any list scheduling algorithm.

Lemma 3.2.1. *For any instance I and any $k \geq 0$, if job l is the last job to finish in the $\mathcal{LS}$ schedule,*

$$\textit{then} \quad C^{\mathcal{LS}}_{\max}(m+k, I) \leq \left(\frac{m}{m+k}\right) C^{\mathcal{OPT}}_{\max}(m, I) + \left(\frac{m+k-1}{m+k}\right) p_l.$$

Proof. All machines must be busy up to time $s^{\mathcal{LS}}_l$ when job l is scheduled, otherwise job l could have been started earlier. The starting time $s^{\mathcal{LS}}_l$ of job l is upper bounded by the average load on $m + k$ machines just before job l is scheduled. Thus,

$s_l^{\mathcal{LS}}(m+k) \leq \frac{1}{m+k}((\sum_{j=1}^{n} p_i) - p_l) = \frac{1}{m+k}\sum_{j=1}^{n} p_i - \frac{1}{m+k}p_l$. The optimal makespan on m machines is lower bounded by the average load on m machines after all jobs are scheduled. Therefore, $C_{\max}^{\mathcal{OPT}}(m) \geq \frac{1}{m}(\sum_{j=1}^{n} p_i)$.

$$\begin{aligned} C_{\max}^{\mathcal{LS}}(m+k, I) &= s_l^{\mathcal{LS}} + p_l \\ &\leq \frac{1}{m+k}\sum_{j=1}^{n} p_i - \frac{1}{m+k}p_l + p_l \\ &\leq \left(\frac{m}{m+k}\right) C_{\max}^{\mathcal{OPT}}(m, I) + \left(\frac{m+k-1}{m+k}\right) p_l. \end{aligned}$$

□

Theorem 3.2.1. *For $m \geq 1, k \geq 0$, and any job list I,*

$$C_{\max}^{\mathcal{LS}}(m+k, I) \leq \left(1 + \frac{m-1}{m+k}\right) C_{\max}^{\mathcal{OPT}}(m, I),$$

and this is tight.

Proof. Suppose job l is the last job to finish in the $\mathcal{LS}$ schedule. By Lemma 3.2.1 and the fact that no job has a processing time larger than $C_{\max}^{\mathcal{OPT}}$,

$$\begin{aligned} C_{\max}^{\mathcal{LS}}(m+k, I) &\leq \left(\frac{m}{m+k}\right) C_{\max}^{\mathcal{OPT}}(m, I) + \left(\frac{m+k-1}{m+k}\right) C_{\max}^{\mathcal{OPT}}(m, I) \\ &= \left(1 + \frac{m-1}{m+k}\right) C_{\max}^{\mathcal{OPT}}(m, I). \end{aligned}$$

The tightness of this result can be seen by considering an instance that begins with $(m-1)(m+k)$ jobs of size 1 and ends with a job of size $m+k$. $\mathcal{LS}$ would schedule $m-1$ jobs of size 1 on each of the $m+k$ machines and schedule the job of size $m+k$ on one of the machines. Thus, its makespan would be $2m+k-1$. The optimal schedule would schedule $m+k$ jobs of size 1 on $m-1$ machines and schedule the job of size $m+k$ on the remaining machine. Thus, its makespan would be $m+k$. $\square$

The lower bound for $\mathcal{LS}$ can be generalized to apply to all on-line algorithms.

Theorem 3.2.2. *No on-line algorithm is (m,k)-optimal for $m \geq 2$ and $k \geq 0$.*

Proof. The idea of the proof is that the adversary will keep generating new jobs until the on-line algorithm schedules a new job on a non-empty machine. Fix $m \geq 2, k \geq 0$, and an on-line algorithm $\mathcal{A}$. The size of jobs generated by the adversary can be described as follows. The first job has size 4. The size of each subsequent job is equal to the size of all the previous jobs combined. Therefore, $p_1 = 4$ and $p_j = 2^j$ for $j \geq 2$. If the adversary has generated l jobs, then the optimal schedule for these l jobs is to schedule job l by itself and to schedule the other jobs arbitrarily on the other machines. In fact, the adversary can schedule jobs 1 through $l-1$ on the same machine. The optimal makespan for these l jobs is 2^l which is the size of the latest job generated by the adversary and is also equal to the sum of the size of jobs 1 through $l-1$.

Suppose job l is the first job that the on-line algorithm schedules on a non-empty machine. Since the on-line algorithm has only $m+k$ machines, then $l \leq m+k+1$. The adversary would generate no more jobs after job l. From the argument above, the optimal makespan is 2^l, the size of job l. Since the on-line algorithm schedules job l on a non-empty machine, then the on-line makespan is strictly greater than 2^l. Thus, the result follows. $\square$

3.3 Off-Line Upper Bounds

We begin this section by extending Graham's analysis for $\mathcal{LPT}$ [37] to the case when $\mathcal{LPT}$ has k extra machines.

Theorem 3.3.1. *For $m \geq 1, k \geq 0$, and any job list I,*

$$\frac{C_{\max}^{\mathcal{LPT}}(m+k,I)}{C_{\max}^{\mathcal{OPT}}(m,I)} \leq \begin{cases} \frac{4m+k-1}{3m+3k}, & 0 \leq k \leq \frac{m-1}{2} \\ 1, & k \geq \frac{m-1}{2}. \end{cases}$$

Proof. The proof is very similar to the case where both $\mathcal{LPT}$ and $\mathcal{OPT}$ have m machines [37]. Suppose job l is the last job to *finish* in the $\mathcal{LPT}$ schedule. Thus, $C_{\max}^{\mathcal{LPT}} = s_l^{\mathcal{LPT}} + p_l$. We can assume that job l is the last job to *start* in the $\mathcal{LPT}$ schedule. If this is false, we can remove all jobs i such that $s_i^{\mathcal{LPT}} \geq s_l^{\mathcal{LPT}}$. This does not change the makespan of the $\mathcal{LPT}$ schedule because these jobs must have run on machines other than the one job l does. Moreover, this cannot increase the optimal makespan. Since $\mathcal{LPT}$ schedules job l last, then job l is the *smallest* job, *i.e.*, $p_l = p_{\min}$. There are two cases.

Case 1: $p_{\min} \leq \frac{1}{3}C_{\max}^{\mathcal{OPT}}$.

Since l is the last job to finish in the $\mathcal{LPT}$ schedule, then by Lemma 3.2.1,

$$C_{\max}^{\mathcal{LPT}} \leq \left(\frac{m}{m+k}\right) C_{\max}^{\mathcal{OPT}} + \left(\frac{m+k-1}{m+k}\right) \frac{1}{3}\, C_{\max}^{\mathcal{OPT}} = \left(\frac{4m+k-1}{3m+3k}\right) C_{\max}^{\mathcal{OPT}}.$$

Case 2: $p_{\min} > \frac{1}{3}C_{\max}^{\mathcal{OPT}}$.

Thus, $p_i > \frac{1}{3}C_{\max}^{\mathcal{OPT}}$ for all i. Therefore, in the optimal schedule, there are at most 2 jobs on each machine. If $n \leq m$, then the optimal algorithm schedules one job per machine. If $m+1 \leq n \leq 2m$, we claim that for $j = 1, ..., m$, the optimal algorithm schedules job j on the same machine as job $2m+1-j$ if $2m+1-j \leq n$, and schedules job j by itself otherwise. The optimality can be proven by an interchange argument. Furthermore, $\mathcal{LPT}$ would produce a similar schedule on $m+k$ machines. If $n \leq m+k$, then $\mathcal{LPT}$ schedules one job per machine. If $m+k+1 \leq n \leq 2(m+k)$, then for $j = 1, ..., m+k$, $\mathcal{LPT}$ schedules job j on the same machine as job $2(m+k)+1-j$ if $2(m+k)+1-j \leq n$, and schedules job j by itself otherwise. Thus, $C_{\max}^{\mathcal{LPT}} \leq C_{\max}^{\mathcal{OPT}}$.

From Cases 1 and 2,
$$C_{\max}^{\mathcal{LPT}} \leq \max\{ \frac{4m+k-1}{3m+3k}, 1 \} C_{\max}^{\mathcal{OPT}}$$
and
$$\frac{4m+k-1}{3m+3k} > 1 \iff k < \frac{m-1}{2}$$

□

Corollary 3.3.1. *$\mathcal{LPT}$ is (m,k)-optimal if $k \geq \frac{m-1}{2}$.*

Proof. Immediate from Theorem 3.3.1. □

We can refine the above result and get the following theorem.

Theorem 3.3.2. *$\mathcal{LPT}$ is (m,k)-optimal if $k \geq \frac{m-1}{3}$.*

Proof. Assume that the theorem is not true. Then there must be some values of k and m for which the theorem statement does not hold. Let us fix a value of k for which the theorem statement does not hold, and let us fix m to be the minimum possible m such that the theorem does not hold for m and k. Note that $m \leq 3k+1$ because otherwise the theorem is true. Finally, given k and m, define I to be a smallest counterexample; that is, an input instance with the minimum possible number of jobs such that $C_{\max}^{\mathcal{LPT}}(m+k, I) > C_{\max}^{\mathcal{OPT}}(m, I)$. We will now show that this counterexample cannot exist, and thus the theorem follows.

Suppose that jobs in I are labeled according to the order that they are scheduled by $\mathcal{LPT}$. Thus, $p_1 \geq p_2 \geq ... \geq p_n$. We first observe that job n is the only job in $\mathcal{LPT}$'s schedule that finishes later than the last job in $\mathcal{OPT}$'s schedule; that is, $C_{\max}^{\mathcal{LPT}}(m+k, I) = C_n^{\mathcal{LPT}}(m+k, I) > C_{\max}^{\mathcal{OPT}}(m, I)$ and $C_i^{\mathcal{LPT}}(m+k, I) \leq C_{\max}^{\mathcal{OPT}}(m, I)$ for $i = 1, ..., n-1$. Otherwise, there exists a job i such that $C_i^{\mathcal{LPT}}(m+k, I) > C_{\max}^{\mathcal{OPT}}(m, I)$ and $1 \leq i \leq n-1$. Create an instance I' from I by removing jobs $i+1$ through n from I. This does not affect the completion time of job i in the $\mathcal{LPT}$ schedule because jobs $i+1$ through n are scheduled after job i is. Moreover, this cannot increase the optimal makespan. Thus, $C_{\max}^{\mathcal{LPT}}(m+k, I') \geq C_i^{\mathcal{LPT}}(m+k, I') = C_i^{\mathcal{LPT}}(m+k, I) > C_{\max}^{\mathcal{OPT}}(m, I) \geq C_{\max}^{\mathcal{OPT}}(m, I')$, and I' has fewer jobs than I. This contradicts our assumption that I is a smallest counterexample.

We next observe that $\mathcal{OPT}$ cannot produce a schedule where some machine has only one job i. Suppose this is not the case. Create an instance I' from I by setting p_i to $C_{\max}^{\mathcal{OPT}}(m, I)$. This only possibly increases p_i. Clearly, $C_{\max}^{\mathcal{OPT}}(m, I') = C_{\max}^{\mathcal{OPT}}(m, I)$. Furthermore, $C_{\max}^{\mathcal{LPT}}(m+k, I') \geq C_{\max}^{\mathcal{LPT}}(m+k, I) > C_{\max}^{\mathcal{OPT}}(m, I) = C_{\max}^{\mathcal{OPT}}(m, I')$. Thus, I' is also a smallest counterexample. We can assume that, in the schedule created by $\mathcal{LPT}$ for instance I', job i is on a machine by itself. Create an instance I'' from I' by removing job i. Clearly, $C_{\max}^{\mathcal{OPT}}(m-1, I'') \leq C_{\max}^{\mathcal{OPT}}(m, I')$. Furthermore, $C_{\max}^{\mathcal{LPT}}(m-1+k, I'') = C_{\max}^{\mathcal{LPT}}(m+k, I') > C_{\max}^{\mathcal{OPT}}(m, I') \geq C_{\max}^{\mathcal{OPT}}(m-1, I'')$. Thus, I'' is a counterexample to the theorem statement where there are $m-1$ base machines and k extra machines. This contradicts the minimality of m.

We now observe that the schedule produced by $\mathcal{LPT}$ just before job n arrives also cannot have any machines with only one job i. Suppose this is not the case.

Assign job n to the machine with only job i. Since $\mathcal{OPT}$ has no machines with only one job, job i must be scheduled with some other job j in the schedule produced by $\mathcal{OPT}$. Since job n is the smallest job in the input instance, $p_n \le p_j$. This implies that $C_{\max}^{\mathcal{LPT}}(m+k, I) = C_n^{\mathcal{LPT}}(m+k, I) \le p_i + p_n \le p_i + p_j \le C_{\max}^{\mathcal{OPT}}(m, I)$.

We now proceed with the remainder of the proof. There are 2 cases based on the size of p_n. In both cases, we show that $C_{\max}^{\mathcal{LPT}}(m+k, I) \le C_{\max}^{\mathcal{OPT}}(m, I)$ which is a contradiction to the assumption that I is a counterexample.

Case 1: $p_n \le \frac{1}{4} C_{\max}^{\mathcal{OPT}}$.

Job n is the last to finish in the $\mathcal{LPT}$ schedule, and $p_n = p_{\min} \le \frac{1}{4} C_{\max}^{\mathcal{OPT}}$. Thus, from Lemma 3.2.1,

$$\begin{aligned} C_{\max}^{\mathcal{LPT}} &\le \left(\frac{m}{m+k}\right) C_{\max}^{\mathcal{OPT}} + \left(\frac{m+k-1}{m+k}\right) \frac{1}{4}\, C_{\max}^{\mathcal{OPT}} \\ &= \left(\frac{4m+m+k-1}{4m+4k}\right) C_{\max}^{\mathcal{OPT}} \\ &\le C_{\max}^{\mathcal{OPT}} \qquad \text{because } m \le 3k+1. \end{aligned}$$

Case 2: $p_n > \frac{1}{4} C_{\max}^{\mathcal{OPT}}$.

Before we continue, we provide some definitions.

- Let ϕ be an optimal schedule on m machines for instance I.
- Let σ be the schedule produced by $\mathcal{LPT}$ on $m+k$ machines for instance I.
- Let π be the partial schedule produced by $\mathcal{LPT}$ on $m+k$ machines for instance I when exactly the first $n-1$ jobs are scheduled.
- Let H and K be the set of machines in schedule ϕ with exactly 2 and 3 jobs, respectively.

- Let E and F be the set of machines in schedule π with exactly 2 and 3 jobs, respectively.

Since for $i = 1, ..., n$, $p_i \geq p_{\min} > \frac{1}{4}C^{\phi}_{\max}$, and $C^{\phi}_i \leq C^{\phi}_{\max}$, then there are at most 3 jobs per machine in schedule ϕ. Similarly, there are at most 3 jobs per machine in schedule π because $C^{\pi}_i \leq C^{\phi}_{\max}$ for $i = 1, ..., n-1$. From earlier observations, there are at least two jobs per machine in both schedule ϕ and schedule π.

For the remainder of this proof, we introduce the following notation. When describing a machine, we will use a distinct letter such as u or v. When describing jobs, we will use two different notations. First, we will still call the last job scheduled by $\mathcal{LPT}$ job n, and we will still refer to its length as p_n. However, we will also refer to jobs by how they are scheduled by the optimal algorithm. Specifically, if u is one of the m machines for the optimal schedule, u_i is the i'th largest job scheduled on machine u with ties broken arbitrarily. Also, we will use $p(u_i)$ to denote the processing time of job u_i. We now classify jobs according to

1. the number of jobs on the machine on which they are scheduled in ϕ and
2. the order of their size among jobs on the same machine in ϕ.

- Let $H_i = \{ u_i \mid u \in H \}$ for $i = 1, 2$.
- Let $H_{12} = H_1 \cup H_2$.
- Definitions of K with subscripts are analogous to the two definitions above.

We also classify jobs in a second manner with respect to the schedule π produced by $\mathcal{LPT}$.

- Let $E_i = \{ \, u_j \mid \text{job } u_j \text{ is the } i\text{th job on some machine } v \text{ in schedule } \pi \text{ and } v \in E\}$ for $i = 1, 2$.
- Let $E_{12} = E_1 \cup E_2$.
- Definitions of F with subscripts are analogous to the two definitions above.

Figure 3.1 illustrates the two job classification schemes just described. Note that jobs in the same class may have different sizes. To further clarify these definitions, consider the following statements which are implied by $u_i \in K_{23}$.

1. Either $u_i = u_2$ or $u_i = u_3$.
2. Job u_i is on machine u in schedule ϕ.
3. $u \in K$.
4. There are 3 jobs on machine u in schedule ϕ, namely u_1, u_2, and u_3.
5. $u_1 \in K_1$, $u_2 \in K_2$, $u_3 \in K_3$.
6. $p(u_1) \geq p(u_2) \geq p(u_3)$.
7. $p(u_1) + p(u_2) + p(u_3) \leq C^{\phi}_{\max}$.

From earlier arguments, there are either 2 or 3 jobs per machine in schedule

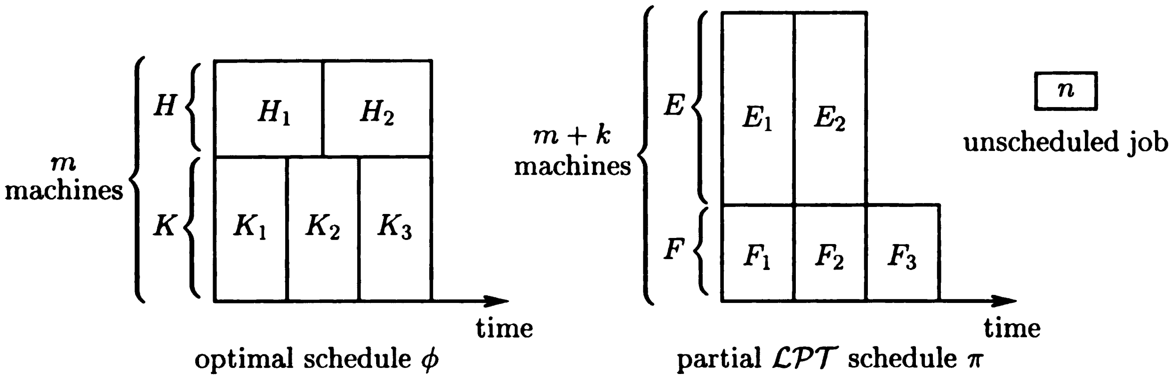


Figure 3.1: Job classification schemes.

ϕ and in schedule π. Therefore, we have the following equalities.

$$n = 2|E| + 3|F| + 1 \tag{3.1}$$

$$n = 2|H| + 3|K| \tag{3.2}$$

$$m = |H| + |K| \tag{3.3}$$

$$m + k = |E| + |F| \tag{3.4}$$

$$2|E| + 3|F| + 1 = 2|H| + 3|K| \qquad \text{from (3.1) and (3.2)} \tag{3.5}$$

$$4(|H| + |K|) \leq 3(|E| + |F|) + 1 \qquad \text{from (3.3),(3.4), and } m \leq 3k + 1 \tag{3.6}$$

We show that there exists a machine y in π and a machine z in ϕ such that if job n is scheduled on y, there is a pairing of jobs from the two machines such that the jobs from y are no larger than the jobs from z. We consider two cases.

Case 2.1: $E_1 \cap K_{23} \neq \emptyset$.

This case is shown pictorially in Figure 3.2. Relation $E_1 \cap K_{23} \neq \emptyset$ means that there exists a job v_i such that $v_i \in E_1$ and $v_i \in K_{23}$. Let job u_j be the job in E_2 on the same machine as v_i in schedule π. By definition of E_1 and E_2, we have that $p(u_j) \leq p(v_i)$. In the optimal schedule, job v_i is scheduled on the same machine with exactly two other jobs, and job v_i is not the largest job on the machine. Thus, $p(u_j) \leq p(v_i) \leq p(v_2) \leq p(v_1)$. Finally, since job n is the smallest job, then $p_n \leq p(v_3)$. Combining these facts, it follows that job n can be scheduled on the same machine as v_i and u_j in the $\mathcal{LPT}$ schedule so that the total load on the machine is no larger than the optimal makespan.

Algebraically, since $p(u_j) \leq p(v_i) \leq p(v_2) \leq p(v_1)$ and $p_n \leq p(v_3)$, then $C^{\sigma}_{\max} = C^{\sigma}_n \leq p(v_i) + p(u_j) + p_n \leq p(v_1) + p(v_2) + p(v_3) \leq C^{\mathcal{OPT}}_{\max}$.

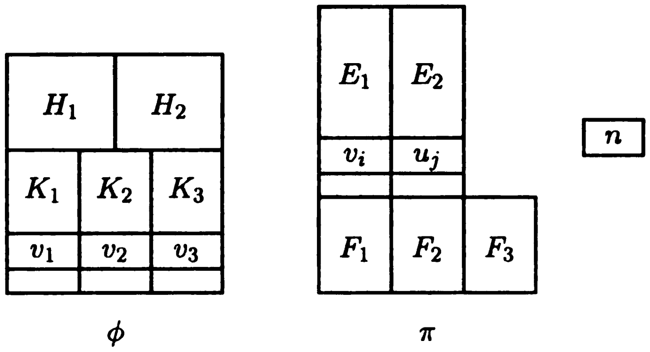


Figure 3.2: Case 2.1 of Theorem 3.3.2: $E_1 \cap K_{23} \neq \emptyset$.

Case 2.2: $E_1 \cap K_{23} = \emptyset$.

First, we establish some relations between schedule ϕ and schedule π.

$$E_1 \subseteq K_{23}^c = H_{12} \cup K_1 \qquad \text{because } E_1 \cap K_{23} = \emptyset \qquad (3.7)$$

$$2|H| + |K| \geq |E| \qquad \text{from (3.7)} \qquad (3.8)$$

$$2|H| + |K| \leq |E| \qquad \text{from (3.5)+(3.6)} \qquad (3.9)$$

$$E_1 = H_{12} \cup K_1 \qquad \text{from (3.7), (3.8) and (3.9)} \quad (3.10)$$

$$E_2 \cup F_{123} \cup \{n\} = E_1^c = (H_{12} \cup K_1)^c = K_{23} \quad \text{from (3.10)} \quad (3.11)$$

Equalities (3.10) and (3.11) are the critical equalities we need and are shown graphically in Figure 3.3.

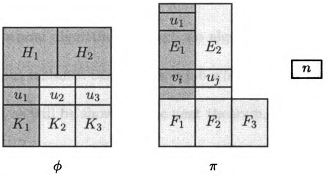


Figure 3.3: Case 2.2 of Theorem 3.3.2: $K_{23} = E_2 \cup F_{123} \cup \{n\}$ and $H_{12} \cup K_1 = E_1$.

Set E_2 is not empty because from equalities (3.10) and (3.3), $|E_2| = |E| = 2|H| + |K| \geq m \geq 1$. Let u_j be the first job in E_2 to be scheduled by $\mathcal{LPT}$. Suppose v_i is the job in E_1 which is on the same machine as u_j in schedule π. From relation (3.11), $u_j \in K_{23}$. Thus, there are 3 jobs on machine u in ϕ. If we can show that $p_n \leq p(u_3)$, $p(u_j) \leq p(u_2)$, and $p(v_i) \leq p(u_1)$, then $C^{\sigma}_{\max} = C^{\sigma}_n \leq p(v_i) + p(u_j) + p_n \leq p(u_1) + p(u_2) + p(u_3) \leq C^{\mathcal{OPT}}_{\max}$. Obviously, $p_n \leq p(u_3)$ because job n is the smallest job. Since $u_j \in K_{23}$, then job u_j is either u_2 or u_3. In any case, $p(u_j) \leq p(u_2)$.

We now prove $p(v_i) \leq p(u_1)$. By definition, job u_1 is in K_1 and u_1 is on the same machine as u_j in schedule ϕ. From relation (3.10), $u_1 \in E_1$. Since u_j is the first job to be scheduled in E_2, and $\mathcal{LPT}$ always schedules the next job on the least loaded machine, then the fact that u_j is scheduled on the same machine as v_i implies that v_i is a smallest job in E_1. In particular, $p(v_i) \leq p(u_1)$, and this completes case 2.2.

Since this is the last possibility to consider, we have proven that there are no counterexamples to the theorem and thus the theorem is true. $\square$

The result of the previous theorem is tight in 2 aspects. First, with $\frac{m-1}{3}$ extra machines, $\mathcal{LPT}$ does not always produce a schedule with makespan *strictly smaller* than that of the optimal algorithm. In fact, no algorithm can guarantee this due to the fact that an input instance can contain a job whose length is equal to the optimal makespan. Second, as will be shown in the next theorem, if $\mathcal{LPT}$ has fewer than $\frac{m-1}{3}$ extra machines, there exist instances such that $\mathcal{LPT}$ will produce a schedule with the makespan *strictly larger* than that of the optimal algorithm.

Theorem 3.3.3. *For any pair of non-negative integers m and k such that $k = \frac{m-2}{3}$, there exist instances such that $C_{\max}^{\mathcal{LPT}}(m+k) \geq \frac{8k+7}{8k+6} C_{\max}^{\mathcal{OPT}}(m) > C_{\max}^{\mathcal{OPT}}(m)$.*

Proof. For $k \geq 0$, the instance I_k with $8k+5$ jobs is defined as follows:

$2k+2$	jobs of size	$4k+3$
2	jobs of size	$2k+2+i$ for $i = 2k, ..., 1$ (a total of $4k$ jobs)
$2k+3$	jobs of size	$2k+2$

Figure 3.4 illustrates an instance with $k = 2$ and $m = 3k+2 = 8$, its optimal schedule on m machines, and its $\mathcal{LPT}$ schedule on $m+k$ machines. Each job in the figure is labeled with its size. The optimal schedule on $m = 3k+2$ machines can be described as follows:

$k+1$	machines with jobs of sizes	$4k+3$	$4k+3$		
2	machines with jobs of sizes	$4k+2$	$2k+2$	$2k+2$	
2	machines with jobs of sizes	$4k+2-i$	$2k+2+i$	$2k+2$	for i=1,...,k-1
1	machine with jobs of sizes	$3k+2$	$3k+2$	$2k+2$	

All machines in the optimal schedule have load $8k+6$. The schedule produced by $\mathcal{LPT}$ on $m+k = 4k+2$ machines can be described as follows:

1	machine with jobs of sizes	$4k+3$	$2k+2$	$2k+2$	
$2k+1$	machines with jobs of sizes	$4k+3$	$2k+2$		
2	machines with jobs of sizes	$4k+3-i$	$2k+2+i$		for i=1,...,k

The machine with 3 jobs in the $\mathcal{LPT}$ schedule have load $8k+7$. All other machines have load $6k+5$. Note that $\mathcal{LPT}$ could schedule the last job (of size $2k+2$) on any machine. □

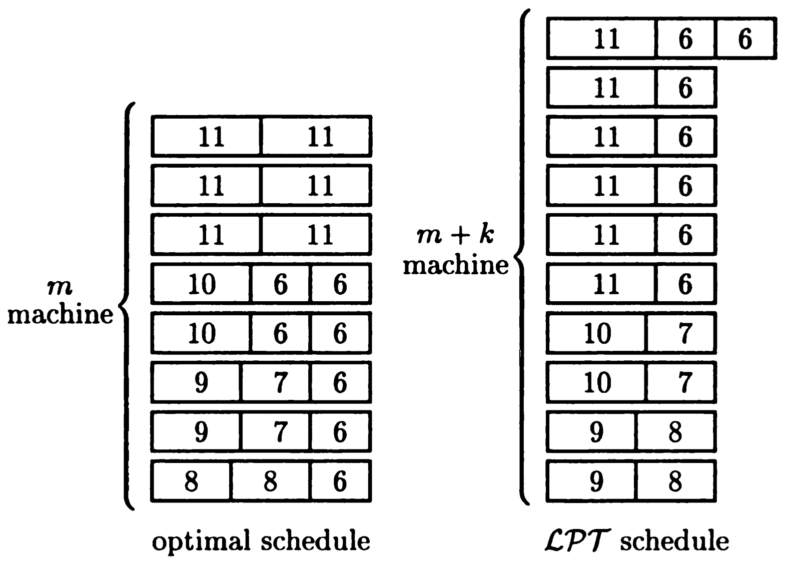


Figure 3.4: An instance with $m = 8$ and $m = 2$ showing tightness of Theorem 3.3.2.

3.4 Lower Bounds on $\mathcal{LPT}$

In this section, we describe a procedure to compute a good, though not necessarily optimal, lower bound of the performance of $\mathcal{LPT}$ for any m and k. Given m and k, we use a linear program that assumes that the optimal schedule and the $\mathcal{LPT}$ schedule have the specific structures shown in Figure 3.5 where jobs are ordered by non-increasing size. Note that each job in the figure is labeled with its job id. The variables of the linear program are the job sizes and the starting time of the last job in the $\mathcal{LPT}$ schedule. This procedure considers only a very restricted set of possible instances. The best lower bound instance from this restricted set can be obtained by optimizing the linear program.

We describe our assumptions in detail next. We assume that in the $\mathcal{LPT}$ schedule, there are 2 jobs on all machines except for one machine on which there are 3 jobs. Suppose $p_1 \leq p_2 \leq ... \leq p_n$. We assume that in the $\mathcal{LPT}$ schedule, job i is paired with job $n - i$, and job n is the last job to finish. We assume that the optimal makespan is no larger than 1. Finally, we assume that there are either 2 or 3 jobs on each machine in the optimal schedule, and the optimal schedule has a structure as illustrated in Figure 3.5.

The linear program can be described algebraically as follows. Let m_2 and m_3 be the number of machines in the optimal schedule with 2 and 3 jobs respectively.

From the above assumptions,

$$\begin{aligned} n &= 2(m+k)+1, & \\ n &= 2m_2+3m_3, & \text{and} \\ m &= m_2+m_3. & \\ \text{Thus,} \quad m_2 &= m-2k-1 & \text{and} \\ m_3 &= 2k+1. & \end{aligned}$$

We denote the starting time of job n in the $\mathcal{LPT}$ schedule by s. Given $m \geq 1$ and $0 \leq k \leq (m-1)/2$, we construct the linear program $LB(m,k)$ as follows.

$$\text{maximize } s + p_n \qquad \text{subject to}$$

$$\begin{aligned} p_i &\geq p_{i+1}, & i &= 1,\dots,n-1 \\ s &\leq p_i + p_{n-i}, & i &= 1,\dots,(n-1)/2 \\ p_i + p_j &\leq 1, & \forall (i,j) &\in H \\ p_i + p_j + p_l &\leq 1, & \forall (i,j,l) &\in K \end{aligned}$$

where

$$H = \{\, (i,\ 2m_2+1-i) \mid i = 1,\dots,m_2 \,\}$$

$$K = \{\, (2m_2+i,\ 2m_2+2m_3+2-i,\ 2m_2+2m_3+1+i) \mid i = 1,\dots,m_3-1 \,\} \cup \{\, (2m_2+m_3,\ 2m_2+m_3+1,\ 2m_2+m_3+2) \,\}$$

Notice the difference between the layout of jobs in the optimal schedules in Figures 3.4 and 3.5. In Theorem 3.3.3, we chose to use the layout in Figure 3.4 rather than the layout in Figure 3.5 because it is easier to describe. Table 3.1 shows the computed lower bounds for $m = 1, ..., 45$ and $k = 0, ..., 10$. Figure 3.6 shows the plot of this table for $m = 1, ..., 60$ and $k = 0, ..., 20$.

Figure 3.7 shows the plot of the lower bound when $m = 100$. Points in Figure 3.7 are the lower bounds obtained from our linear program when $m = 100$. The upper curve is the performance guarantee of $\mathcal{LPT}$ from Theorem 3.3.1. The upper curve is described by $y = \max\{\frac{4m+k-1}{3m+3k}, 1\}$ where y is the performance guarantee. It is important to note that the lower bounds found when $m \geq 1$ and $k = 0$ or $k \geq \frac{m-1}{3}$ match the upper bounds given by Graham [37] and Theorem 3.3.2, respectively. The lower curve is the performance guarantee of $\mathcal{LPT}$ when it is given m machines with speed $\frac{m+k}{m}$ instead of $m + k$ machines with unit speed. When an algorithm is given machines with speed s, it can guarantee a makespan which is $\frac{1}{s}$ times the guaranteed makespan using machines with unit speed. The lower curve is described by $y = \frac{4m-1}{3m}\frac{m}{m+k} = \frac{4m-1}{3m+3k}$ where y is the performance guarantee.

Intuitively, faster machines should be utilized more efficiently than extra machines. In Figure 3.7, the lower bounds obtained from the linear program mostly lie well above the lower curve. However, there is one point which lies below the lower curve. It is possible that this point is a counterexample to our result, but more likely we have not found an optimal extra machine lower bound for this particular combination of m and k.

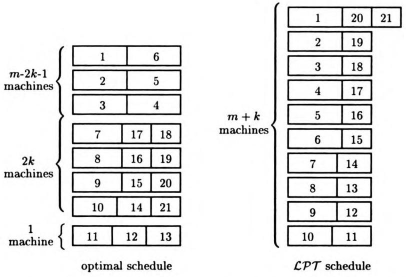


Figure 3.5: The lower bound instance with $m = 8$ and $k = 2$ obtained from the linear program.

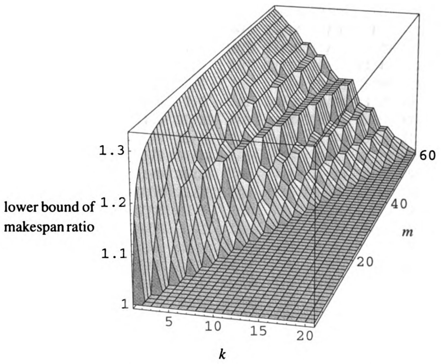


Figure 3.6: The plot of the lower bounds computed by the linear program.

$m\backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1	-	-	-	-	-	-	-	-	-	-
2	7/6	-	-	-	-	-	-	-	-	-	-
3	11/9	1	-	-	-	-	-	-	-	-	-
4	5/4	1	-	-	-	-	-	-	-	-	-
5	19/15	15/14	1	-	-	-	-	-	-	-	-
6	23/18	10/9	1	-	-	-	-	-	-	-	-
7	9/7	7/6	1	1	-	-	-	-	-	-	-
8	31/24	7/6	23/22	1	-	-	-	-	-	-	-
9	35/27	27/23	15/14	1	1	-	-	-	-	-	-
10	13/10	25/21	11/10	1	1	-	-	-	-	-	-
11	43/33	11/9	10/9	31/30	1	1	-	-	-	-	-
12	47/36	11/9	7/6	19/18	1	1	-	-	-	-	-
13	17/13	11/9	7/6	15/14	1	1	1	-	-	-	-
14	55/42	70/57	7/6	11/10	39/38	1	1	-	-	-	-
15	59/45	5/4	7/6	53/48	23/22	1	1	1	-	-	-
16	21/16	5/4	27/23	10/9	19/18	1	1	1	-	-	-
17	67/51	5/4	19/16	7/6	15/14	47/46	1	1	1	-	-
18	71/54	5/4	25/21	7/6	11/10	31/30	1	1	1	-	-
19	25/19	19/15	11/9	7/6	11/10	19/18	1	1	1	1	-
20	79/60	19/15	11/9	7/6	10/9	19/18	55/54	1	1	1	-
21	83/63	19/15	11/9	7/6	10/9	15/14	31/30	1	1	1	1
22	29/22	19/15	11/9	7/6	7/6	11/10	23/22	1	1	1	1
23	91/69	23/18	11/9	27/23	7/6	11/10	19/18	63/62	1	1	1
24	95/72	23/18	27/22	19/16	7/6	76/69	19/18	39/38	1	1	1
25	33/25	23/18	70/57	19/16	7/6	10/9	15/14	31/30	1	1	1
26	103/78	23/18	5/4	25/21	7/6	10/9	11/10	19/18	71/70	1	1
27	107/81	9/7	5/4	11/9	7/6	7/6	11/10	19/18	43/42	1	1
28	37/28	9/7	5/4	11/9	7/6	7/6	11/10	19/18	31/30	1	1
29	115/87	9/7	5/4	11/9	7/6	7/6	53/48	15/14	23/22	79/78	1
30	119/90	9/7	5/4	11/9	27/23	7/6	10/9	11/10	19/18	47/46	1
31	41/31	31/24	5/4	11/9	19/16	7/6	10/9	11/10	19/18	31/30	1
32	127/96	31/24	5/4	11/9	19/16	7/6	7/6	11/10	19/18	31/30	87/86
33	131/99	31/24	19/15	11/9	25/21	7/6	7/6	11/10	15/14	19/18	55/54
34	45/34	31/24	19/15	27/22	25/21	7/6	7/6	10/9	11/10	19/18	39/38
35	139/105	35/27	19/15	27/22	11/9	7/6	7/6	10/9	11/10	19/18	31/30
36	143/108	35/27	19/15	70/57	11/9	7/6	7/6	10/9	11/10	19/18	23/22
37	49/37	35/27	19/15	5/4	11/9	27/23	7/6	7/6	11/10	15/14	19/18
38	151/114	35/27	19/15	5/4	11/9	19/16	7/6	7/6	53/48	11/10	19/18
39	155/117	13/10	19/15	5/4	11/9	19/16	7/6	7/6	10/9	11/10	19/18
40	53/40	13/10	23/18	5/4	11/9	19/16	7/6	7/6	10/9	11/10	19/18
41	163/123	13/10	23/18	5/4	11/9	25/21	7/6	7/6	10/9	11/10	15/14
42	167/126	13/10	23/18	5/4	11/9	25/21	7/6	7/6	7/6	11/10	11/10
43	57/43	43/33	23/18	5/4	11/9	11/9	7/6	7/6	7/6	53/48	11/10
44	175/132	43/33	23/18	5/4	27/22	11/9	27/23	7/6	7/6	10/9	11/10
45	179/135	43/33	23/18	5/4	27/22	11/9	19/16	7/6	7/6	10/9	11/10

Table 3.1: The lower bounds computed by the linear program.

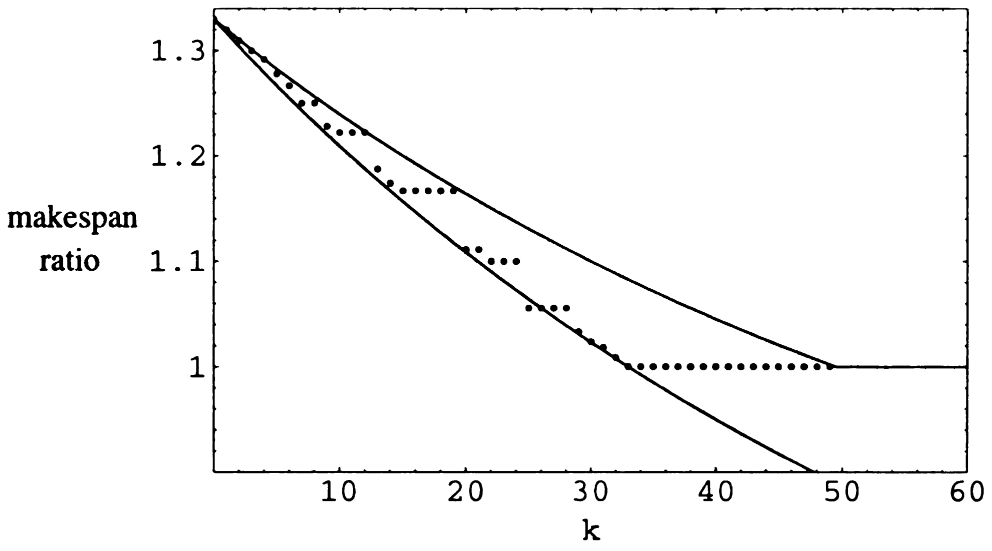


Figure 3.7: Summary of upper and lower bounds for $\mathcal{LPT}$ with extra resources.

3.5 Open Problems

Extra-resource analysis is a promising analysis technique which can be used to derive greater insight into the behavior of both on-line and off-line algorithms. Previously, these insights have been used to identify good on-line algorithms in settings where traditional competitive analysis fails to do so. In this work, we show that these insights can also be used to derive a divergence between off-line and on-line load balancing algorithms. We believe that future work should, in part, search for even more applications of the extra-resource analysis technique.

Chapter 4

Preempt-Decay Scheduling

4.1 Introduction

In this chapter, we study the single machine preempt-decay scheduling problem with release dates minimizing the total flow time. We denote this problem by $1|r_j, decay|\sum F_j$. We study its relation to the preempt-resume and preempt-repeat counterparts ($1|r_j, pmtn|\sum F_j$ and $1|r_j|\sum F_j$). A previous result along this line is the work by Kellerer, Tautenhahn, and Woeginger [52]. They showed that the optimal total flow time in the preempt-repeat model is at most $O(\sqrt{n})$ times that of the preempt-resume model, and there exist instances for which this is tight.

In Section 4.2, we show that the preempt-decay scheduling problem on one machine minimizing total flow time is NP-hard. Since the preempt-decay model is a middle ground between two extremes, the preempt-resume and the preempt-repeat models, we need to determine if the preempt-decay model is significantly different from the other two models. In Section 4.3, we show that the optimal total flow time between the preempt-decay model and the existing preempt-resume and preempt-repeat model could be a factor of $\Theta(\sqrt{n})$ apart. Thus, the total flow time of the optimal schedule in the preempt-repeat model could be a factor of $\Omega(\sqrt{n})$ from that of the preempt-decay model. To distinguish optimal algorithms in the three models, we use $\mathcal{POPT}$, $\mathcal{DOPT}_f$, and $\mathcal{NOPT}$ to denote an optimal algorithm in the preempt-resume model, the preempt-decay model where f is the decay function, and the preempt-repeat model, respectively. The summary of results in Section 4.3 is shown in in Table 4.1 and pictorially in Figure 4.1.

In Section 4.4, we analyze the performance of the Shortest Remaining Processing Time ($\mathcal{SRPT}$) algorithm in the preempt-decay model. Note that the $\mathcal{SRPT}$ algorithm always runs a job with smallest remaining processing time, and it is an optimal algorithm for the preempt-resume model [9]. Our analysis shows that $\mathcal{SRPT}$ performs poorly in the preempt-decay model; there exist instances for which the total flow time of the $\mathcal{SRPT}$ schedule is $\Omega(\sqrt{n})$ times the optimal total flow time in the preempt-decay model. The result is shown in Table 4.1 and pictorially in Figure 4.1. In section 4.5, we discuss some open problems in preempt-decay scheduling. Some open problems are shown in Table 4.1 and Figure 4.1.

4.2 NP-Hardness Result

First, we mention an NP-hardness result from the literature.

Theorem 4.2.1. *[59] The one machine scheduling problem with release dates minimizing the total flow time in the preempt-repeat model is NP-hard.*

Using a similar reduction from the 3-PARTITION, we prove that a similar problem in the preempt-decay model is NP-hard.

	Lower Bound	Upper Bound
$\frac{\mathcal{NOPT}}{\mathcal{POPT}}$	$\Omega(\sqrt{n})$ [52]	$O(\sqrt{n})$ [52]
$\frac{\mathcal{NOPT}}{\mathcal{DOPT}_f}$	$\Omega(\sqrt{n})$	$O(\sqrt{n})$
$\frac{\mathcal{DOPT}_f}{\mathcal{POPT}}$	$\Omega(\sqrt{n})$	$O(\sqrt{n})$

	Lower Bound	Upper Bound
$\frac{\mathcal{SRPT}}{\mathcal{DOPT}_f}$	$\Omega(\sqrt{n}), f(t) = ut, u \geq 1$ $\Omega(u\sqrt{n}), f(t) = ut, u \leq 1$	open
$\frac{\mathcal{A}}{\mathcal{DOPT}_f}$	open	open

Table 4.1: Summary of results on preempt-decay scheduling.

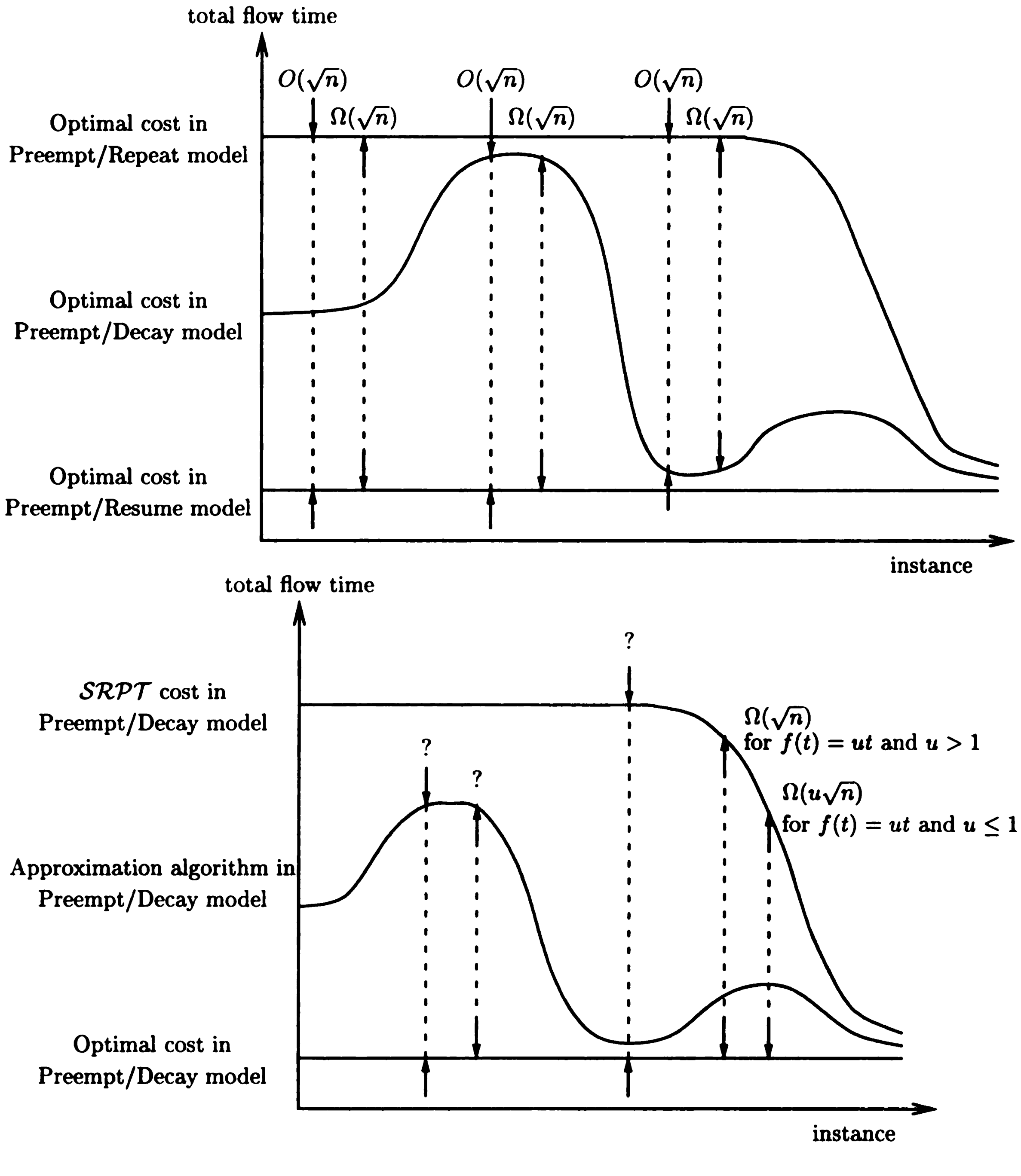

total flow time
$O(\sqrt{n})$
$\Omega(\sqrt{n})$
$O(\sqrt{n})$
$\Omega(\sqrt{n})$
$O(\sqrt{n})$
$\Omega(\sqrt{n})$
Optimal cost in Preempt/Repeat model
Optimal cost in Preempt/Decay model
Optimal cost in Preempt/Resume model
instance
total flow time
?
$\mathcal{SRPT}$ cost in Preempt/Decay model
$\Omega(\sqrt{n})$ for $f(t) = ut$ and $u > 1$
?
?
$\Omega(u\sqrt{n})$ for $f(t) = ut$ and $u \leq 1$
Approximation algorithm in Preempt/Decay model
Optimal cost in Preempt/Decay model
instance

Figure 4.1: Summary of results on preempt-decay scheduling.

3-PARTITION:

Instance: A bound $B \in Z^+$, a set $A = \{a_1, ..., a_{3m}\}$ of $3m$ integers such that $B/4 < a_j < B/2$ for all $a_j \in A$ and such that $\sum_{a_j \in A} a_j = mB$.

Question: Can A be partitioned into m disjoint sets $A_1, A_2, ..., A_m$ such that, for $1 \leq i \leq m$, $\sum_{a_j \in A_i} a_j = B$ (note that each A_i must therefore contain exactly three elements from A)?

Theorem 4.2.2. *The one machine scheduling problem with release dates minimizing the total flow time in the preempt-decay model is NP-hard.*

Proof sketch. The proof is very similar to the proof of Theorem 4.2.1. The reduction is from the 3-PARTITION problem. The idea is to translate elements of set A into jobs that released at time 0 and to release streams of small jobs so that there is a slot of size B between consecutive streams. After a last stream of small jobs, release a stream of large jobs. See Figure 4.2 for an illustration.

The optimal solution is to schedule 3 jobs in each slot according to the solution of the 3-PARTITION problem and to schedule all other jobs as soon as they arrive. Solutions with this structure are the only optimal solutions.

If there is no solution for the 3-PARTITION problem, then we cannot schedule 3 jobs in all slots. If a stream of small jobs are delayed, this would cause a large increase in the total flow time because there are so many of them. Similarly, if the stream of large jobs are delayed, this would cause a large increase in the total flow time. If a job is run after the stream of large jobs, its flow time would be very large.

Observe that preemption does not help either. This is because if a job is preempted by either a stream of small or large jobs, it completely decays before it resumes execution. □

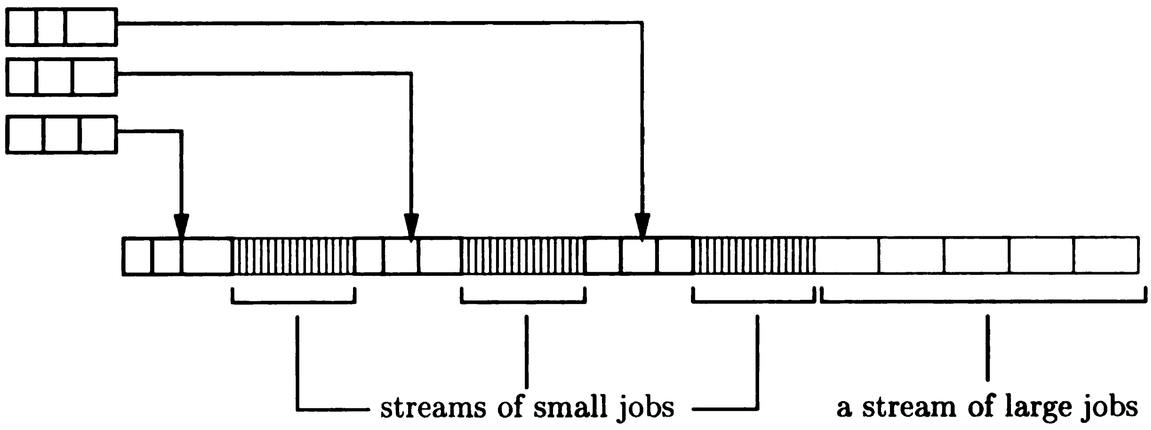


Figure 4.2: A reduction from 3-PARTITION to a preempt-decay scheduling problem.

4.3 Comparison of the Optimal Flow Time Among the Three Models

In this section, we compare the optimal flow time among the preempt-resume model, the preempt-decay model, and the preempt-repeat model to determine if the differences in the three models are significant. To distinguish optimal algorithms in the three models, we use $\mathcal{POPT}$, $\mathcal{DOPT}_f$, and $\mathcal{NOPT}$to denote an optimal algorithm in the preempt-resume model, the preempt-decay model where f is the decay function, and the preempt-repeat model, respectively. Note that the **Shortest Remaining Processing Time** ($\mathcal{SRPT}$) algorithm: always run a job with the smallest remaining processing time, is an optimal algorithm for the preempt-resume model [9].

Fact 4.3.1. *Suppose f and g are decay functions. If $f(t) \leq g(t)$ for all $t \geq 0$, then for any instance I, $\mathcal{DOPT}_f(I) \leq \mathcal{DOPT}_g(I)$.*

Fact 4.3.2. *For any instance I and any decay function f, $\mathcal{POPT}(I) \leq \mathcal{DOPT}_f(I) \leq \mathcal{NOPT}(I)$.*

4.3.1 Preempt-Repeat and Preempt-Resume models

The result in this section is due to Kellerer, Tautenhahn, and Woeginger [52]. We restate their result here for completeness.

Theorem 4.3.1. *For the total flow time cost metric, $\mathcal{NOPT}(I_n)/\mathcal{POPT}(I_n) = O(\sqrt{n})$ where I_n is any input instance with n jobs, and there exists an input instance where this is tight. [52]*

Proof. We omit the proof of the upper bound. The following lower bound instance is adapted from the one given in [52].

$$r_i = \begin{cases} in & \text{for } i = 1, .., n-1 \\ 0 & \text{for } i = n \end{cases} \quad \text{and} \quad p_i = \begin{cases} 1 & \text{for } i = 1, .., n-1 \\ n\sqrt{n} & \text{for } i = n \end{cases}$$

See Figure 4.3 for illustrations of the lower bound instance, the optimal schedule in the preempt-resume model, and possible schedules in the preempt-repeat model. The optimal schedule in the preempt-resume model can be described as follows. Jobs $J_1, ..., J_{n-1}$ are run as soon as they arrive, and they run to completion without interruption. Job J_n is run as soon as it arrives, but it is always preempted by other jobs. In particular, job J_n is preempted by jobs $J_1, ..., J_{\sqrt{n}}$. The optimal total flow time in the preempt-resume model is $O(n\sqrt{n})$ which is primarily the flow time of job J_n.

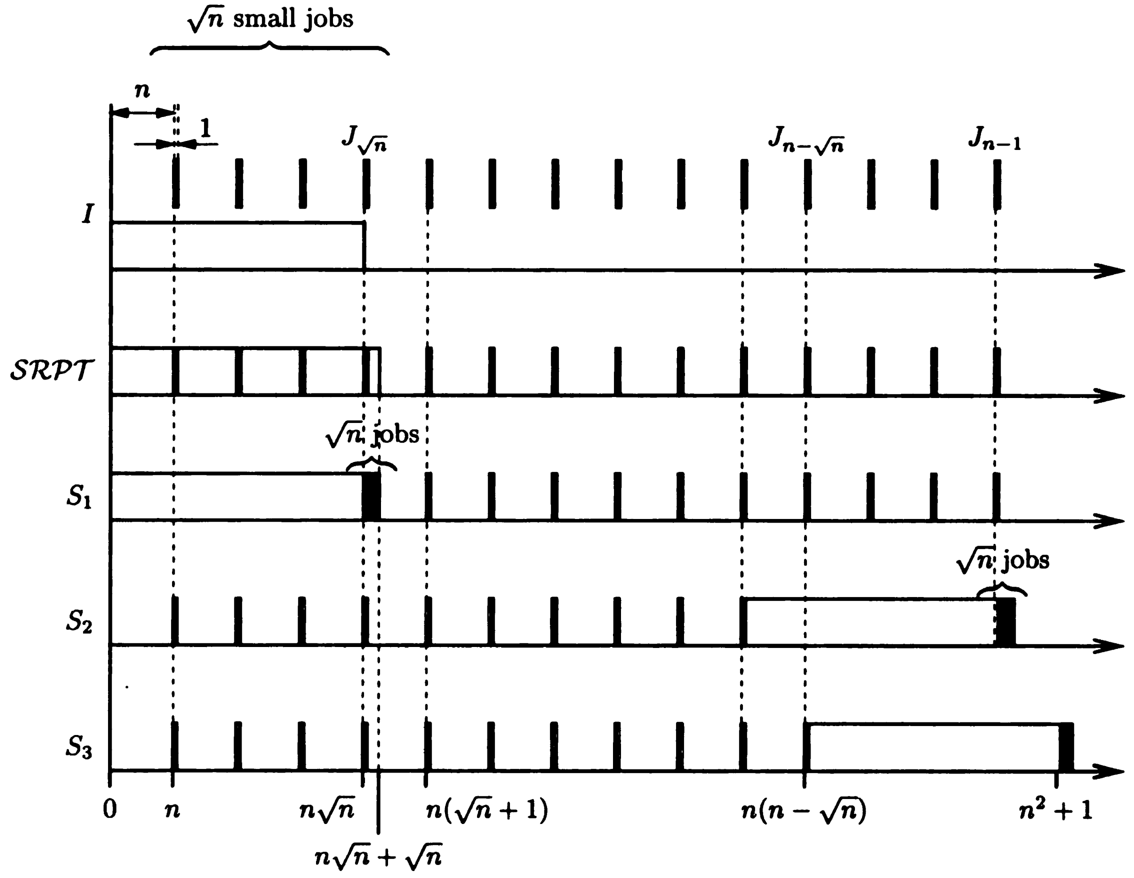


Figure 4.3: Lower bound instance between the preempt-repeat and preempt-resume models.

Consider a schedule in the preempt-repeat model. Without loss of generality, we assume that jobs $J_1, ..., J_{n-1}$ are run in that order. If job J_n is run before job $J_{n-\sqrt{n}}$, it would cause $\sqrt{n}$ other jobs to have an average flow time of $\Omega(n\sqrt{n})$ giving a total flow time of at least $\Omega(n^2)$. If job J_n is run after job $J_{n-\sqrt{n}}$, its flow time, which is a lower bound of the total flow time, would be at least $\Omega(n^2)$. Therefore, the ratio of the optimal total flow time in the preempt-repeat model and the preempt-resume model is $\Omega(\sqrt{n})$ for this input instance. □

4.3.2 Preempt-Repeat and Preempt-Decay models

Corollary 4.3.1. *For the total flow time cost metric, $\mathcal{NOPT}(I_n)/\mathcal{DOPT}_f(I_n) = O(\sqrt{n})$ where I_n is any input instance with n jobs, and f is any decay function, and there exists an input instance where this is tight.*

Proof. The upper bound follows from Theorem 4.3.1 and Fact 4.3.2.

$$\frac{\mathcal{NOPT}(I_n)}{\mathcal{DOPT}_f(I_n)} \leq \frac{\mathcal{NOPT}(I_n)}{\mathcal{POPT}(I_n)} = O(\sqrt{n})$$

Next, we show that an instance similar to that of Kellerer et al. [52] achieves a lower bound of $\Omega(\sqrt{n})$ between the optimal costs in the preempt-repeat model and the preempt-decay model. The lower bound instance is defined as follows.

$$r_i = \begin{cases} Din & \text{for } i = 1, ..., n-1 \\ 0 & \text{for } i = n \end{cases} \quad \text{and} \quad p_i = \begin{cases} 1 & \text{for } i = 1, ..., n-1 \\ Dn\sqrt{n} & \text{for } i = n \end{cases}$$

where $D = f(1) + 1$ and f is a fixed decay function in the preempt-decay model.

See Figure 4.4 for illustrations of the lower bound instance, the optimal schedule in the preempt-decay model, and possible schedules in the preempt-repeat model. The optimal schedule in the preempt-decay model can be described as follows. Jobs

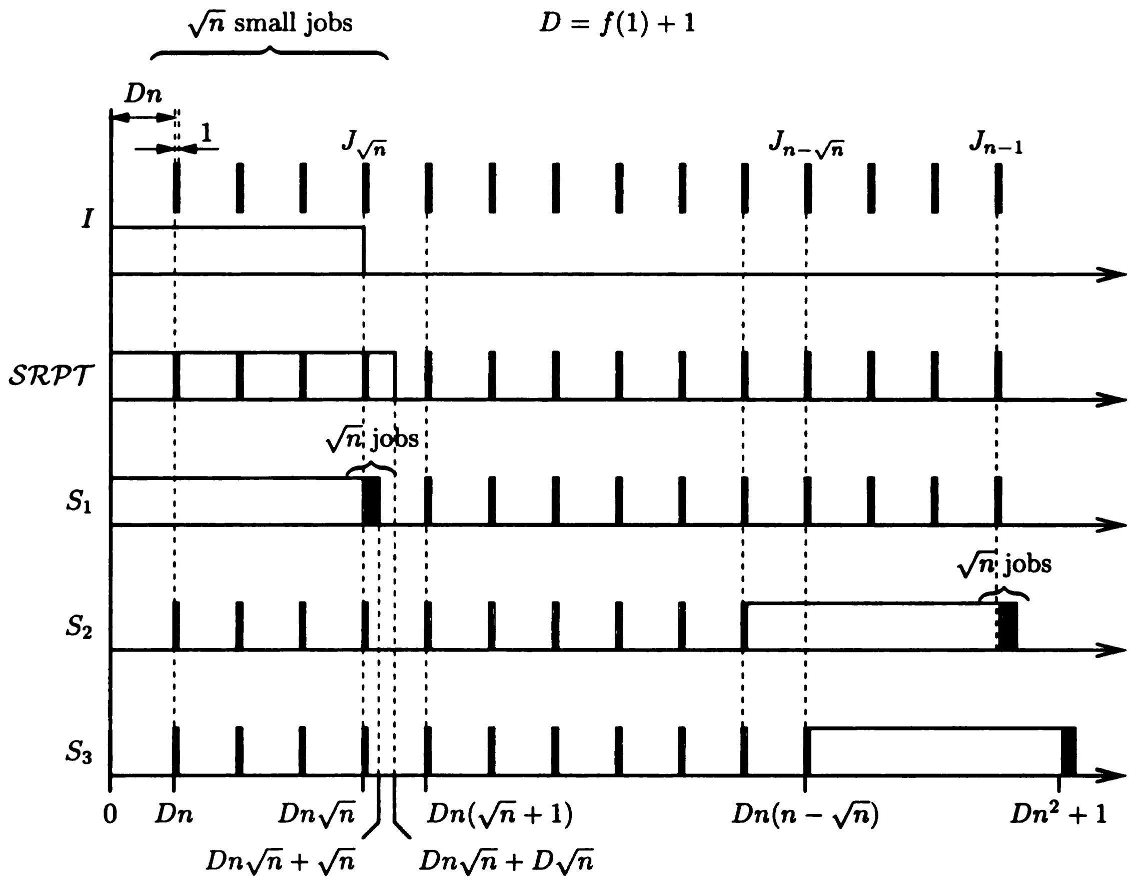


Figure 4.4: Lower bound instance between the preempt-repeat and preempt-decay models.

$J_1, ..., J_{n-1}$ are run as soon as they arrive, and they run to completion without interruption. Job J_n is run as soon as it arrives, but it is always preempted by other jobs. In particular, job J_n is preempted by jobs $J_1, ..., J_{\sqrt{n}}$.

$$\text{Thus,} \quad \mathcal{DOPT}(I_n) = O(Dn\sqrt{n}) = O((f(1)+1)n\sqrt{n})$$

This is primarily the flow time of job J_n. Note that even though job J_n has to make up for the amount of processing decayed during the time it waited for jobs $J_1, ..., J_{\sqrt{n}}$ to finish, job J_n can finish before job $J_{\sqrt{n}+1}$ arrives.

Consider a schedule in the no-preemption model. Without loss of generality, we assume that jobs $J_1, ..., J_{n-1}$ are run in that order. If job J_n is run before job $J_{n-\sqrt{n}}$, it would cause $\sqrt{n}$ other jobs to have an average flow time of $\Omega(Dn\sqrt{n})$ giving a total flow time of at least $\Omega(Dn^2)$. If job J_n is run after job $J_{n-\sqrt{n}}$, its flow time, which is a lower bound of the total flow time, would be at least $\Omega(Dn^2)$.

$$\text{Thus,} \quad \mathcal{NOPT}(I_n) = \Omega(Dn^2) = \Omega((f(1)+1)n^2)$$

Therefore, the ratio of the optimal total flow time in the preempt-repeat model and the preempt-decay model for this input instance is

$$\frac{\mathcal{NOPT}(I_n)}{\mathcal{DOPT}(I_n)} \geq \frac{\Omega(Dn^2)}{O(Dn\sqrt{n})} = \Omega(\sqrt{n}).$$

Note that the result apply as long as $f(t) \geq 0$. □

4.3.3 Preempt-Decay and Preempt-Resume models

Corollary 4.3.2. *For the total flow time cost metric, any decay function $f(t) = ut$ where u is a constant, and any input instance I_n with n jobs, $\mathcal{NOPT}(I_n)/\mathcal{DOPT}_f(I_n)$ $= O(\sqrt{n})$, and there exists an input instance where this is tight.*

Proof. Since for any instance I_n with n jobs and any decay function f, $\mathcal{DOPT}_f(I_n) \leq \mathcal{NOPT}(I_n)$, and from Theorem 4.3.1, $\mathcal{NOPT}(I_n)/\mathcal{POPT}(I_n) = O(\sqrt{n})$, then $\mathcal{DOPT}_f(I_n)/\mathcal{POPT}(I_n) = O(\sqrt{n})$.

Next, we show that there exist input instances that achieve a lower bound of $\Omega(\sqrt{n})$ when $u \geq 1$ and $\Omega(u\sqrt{n})$ when $u \leq 1$. Suppose $n = x + \sqrt{x}$. Thus, when n is large, $n = x + \sqrt{x} \sim x$. A lower bound instance is defined as follows.

$$r_i = \begin{cases} (i-1)(u+1), & i = 1, ..., x \\ (i-x-1)(u+1)\sqrt{x}, & i = x+1, ..., x+\sqrt{x} \end{cases}$$

$$p_i = \begin{cases} 1, & i = 1, ..., x \\ u\sqrt{x}, & i = x+1, ..., x+\sqrt{x} \end{cases}$$

See Figure 4.5 for illustration. Tick marks indicate the completion time of big jobs.

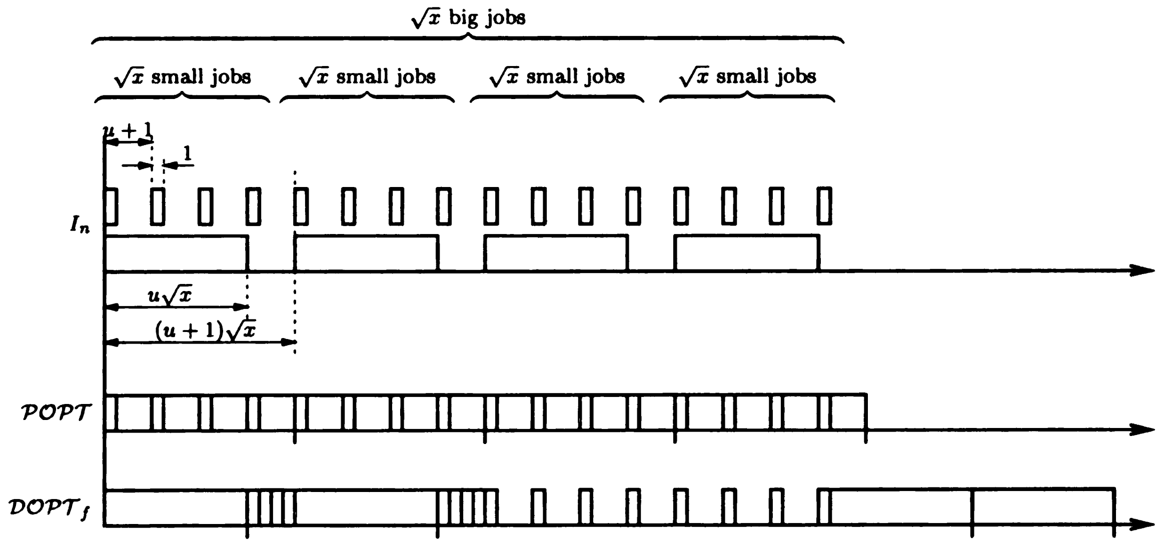


Figure 4.5: Lower bound instance between the preempt-decay and preempt-resume models.

Notice that the total gap time of $\sqrt{x}$ gaps between $\sqrt{x}+1$ small jobs in the input instance is exactly the length of a big job. Thus, in the preempt-resume model, for every $\sqrt{x}$ small jobs completed, one big job can be completed by running it between the gaps. The total flow time of such schedule is $(u+1)x + x = (u+2)x$. Note that if $u \leq 1$, this schedule is not optimal. An optimal schedule can be obtained by using the $\mathcal{SRPT}$ algorithm.

Now consider the preempt-decay model. The length of the gap between two consecutive small jobs in the input instance is exactly u times the length of a small job. Suppose we try to run big jobs between the gaps as we do in the preempt-resume model. When a big job is run in a gap, u units of work is completed. However, during the time the big job is suspended for 1 time unit because of a preemption by a small job, $f(1) = u$ units of the big job have decayed. In other words, the work of the big job done in a gap entirely decays when the big job is preempted by a small job following the gap. Thus, we can conclude that big jobs are never preempted in any reasonable schedule for this instance in the preempt-decay model.

The optimal solution in the preempt-decay model is in the following form. For some integer i where $0 \leq i \leq \sqrt{x}$. the first i big jobs are run as soon as they arrive. All the small jobs are run as soon as they arrive except when some of the first i big jobs are running. All the remaining $\sqrt{x} - i$ big jobs are run after the last small job. We will show that regardless of the value of i, the total flow time of these schedules can be bounded from below by $(u/2+1)x\sqrt{x}$. Each of the first i big jobs causes $\sqrt{x}$ small jobs to have an average flow time of at least $(u/2+1)\sqrt{x}$ giving a total flow time of at least $i(u/2+1)x$. The last $\sqrt{x} - i$ big jobs finish after time $(u+1)x$ giving a total flow time of at least $(\sqrt{x} - i)(u+1)x$. The total flow time from both cases is

$$
\begin{aligned}
i\left(\frac{u}{2}+1\right)x + (\sqrt{x}-i)(u+1)x &= (u+1)x\sqrt{x} - \frac{iux}{2} \\
&\geq (u+1)x\sqrt{x} - \frac{ux\sqrt{x}}{2} \qquad \text{for } 1 \leq i \leq \sqrt{x} \\
&= \frac{1}{2}(u+2)x\sqrt{x}
\end{aligned}
$$

Thus, the ratio between the optimal cost in the preempt-decay and preempt-resume models is at least $\sqrt{x}/2 = \Omega(\sqrt{n})$ for this input instance.

$\square$

Corollary 4.3.2 can be generalized to a larger class of decay functions. We need the following notation. A function f *dominates* another function f' in the interval $[t_1, t_2]$ if $f(t) \geq f'(t)$ for $t_1 \leq t \leq t_2$. See Figure 4.6 for examples. In both examples, f dominates f' in the interval $[0, t']$.

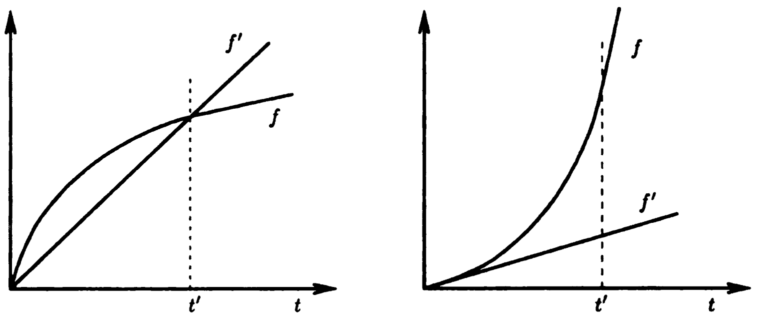


Figure 4.6: Examples of domination between decay functions.

Corollary 4.3.3. *For the total flow time cost metric, any decay function f which dominates some linear function $f'(t) = ut, u > 0$, in the interval $[0, t']$ for some $t' > 0$, and any input instance I_n with n jobs, $\mathcal{NOPT}(I_n)/\mathcal{DOPT}_f(I_n) = O(\sqrt{n})$, and there exists an input instance where this is tight.*

Proof. The upper bound proof is the same as those for Corollary 4.3.2. To proof the lower bound, beginning with a lower bound instance given in Corollary 4.3.2, we linearly scale the release time and the processing time of all jobs so that the processing time of no job is greater than $f'(t')$. From the fact that jobs decay faster with f than with f', and the argument in Corollary 4.3.2, it can be concluded that there should be no preemptions in any optimal schedule in the preempt-decay model. The rest of the argument is the same as those in Corollary 4.3.2, and the lower bound follows. □

4.4 Approximability and Inapproximability in the Three Models

In this section, we restate some approximability and inapproximability results in the preempt-repeat model from the literature. Next, we show that the $\mathcal{SRPT}$ algorithm which is optimal in the preempt-resume model performs poorly in the preempt-decay model.

The following results are due to Kellerer, Tautenhahn, and Woeginger [52].

Theorem 4.4.1. *There exists an $O(n^{3/2}\log n)$-time $O(\sqrt{n})$-approximation algorithm for the minimum total flow time scheduling problem in the preempt-repeat model, and there exist instances where this bound it tight. [52]*

Theorem 4.4.2. *Unless $P = NP$, no polynomial time approximation algorithm for the minimum total flow time scheduling problem in the preempt-repeat model can have an approximation ratio of $O(n^{\frac{1}{2}-\varepsilon})$ for any $\varepsilon > 0$. [52]*

Next, we show that the $\mathcal{SRPT}$ algorithm performs poorly in the preempt-decay model. We need the following notation. For any real number x, let $[x]_1 =$

$\lceil x \rceil - 1$, and let $\{x\}_1 = x - [x]_1$. For examples, $[2.71]_1 = 2$, $\{2.71\}_1 = 0.71$, $[3]_1 = 2$, and $\{3\}_1 = 1$. Note that $0 < \{x\}_1 \leq 1$ and $[x]_1 + \{x\}_1 = x$ for any real number x.

Theorem 4.4.3. *For the decay function $f(t) = ut = t/v$ where u is a constant and $v = 1/u$, there exist instances I_n with n jobs such that*

$$\frac{\mathcal{SRPT}_f(I_n)}{\mathcal{DOPT}_f(I_n)} > \frac{n}{3(v+4)} = \begin{cases} \Omega(n) & \text{for } u \geq 1, \text{ and} \\ \Omega(un) & \text{for } u \leq 1. \end{cases}$$

Proof. We prove the theorem by constructing a class of instances that achieves the stated lower bound. Let $\varepsilon > 0$ be a small positive real number. Let $v' = v + \varepsilon$ and $\bar{v} = [v']_1$. Suppose $n = c \cdot (\bar{v} + 3)$ for some positive integer c. For convenience, jobs will be labeled as $J_{i,j}$ where $i = 1, ..., c$ and $j = 1, ..., \bar{v} + 3$. We will also be refering to clusters of jobs. Cluster i is composed of jobs $J_{i,1}, ..., J_{i,\bar{v}+3}$, $1 \leq i \leq c$. A lower bound instance can be described as follows. Note that r_i's are just dummy variables.

$$\begin{aligned} r_i &= (i-1)(v+1)(v'+1) \qquad \text{for } i = 1, ..., c+1 \\ r_{i,j} &= \begin{cases} r_i + j(v+1) - v & \text{for } i = 1, ..., c \text{ and } j = 1, ..., \bar{v}+1 \\ r_i + [v'+1]_1(v+1) + \{v'+1\}_1 & \text{for } i = 1, ..., c \text{ and } j = \bar{v}+2 \\ r_i & \text{for } i = 1, ..., c \text{ and } j = \bar{v}+3 \end{cases} \\ p_{i,j} &= \begin{cases} v & \text{for } i = 1, ..., c \text{ and } j = 1, ..., \bar{v}+1 \\ v\{v'+1\}_1 & \text{for } i = 1, ..., c \text{ and } j = \bar{v}+2 \\ v'+1 & \text{for } i = 1, ..., c \text{ and } j = \bar{v}+3 \end{cases} \end{aligned}$$

See an illustration in Figure 4.7. Shaded blocks in the top figure are shown in detail in the middle and the bottom figures. The middle and the bottom figures show a cluster of jobs when $v \geq 1$ and $v \leq 1$ respectively. Shaded job pieces in the middle and the bottom figures signify that these job pieces are not run to completion. There are c clusters of $\bar{v} + 3$ jobs. The total processing time of jobs in each cluster is $v[v'+1]_1 + v\{v'+1\}_1 + (v'+1) = (v'+1)(v+1)$. Each cluster is also released

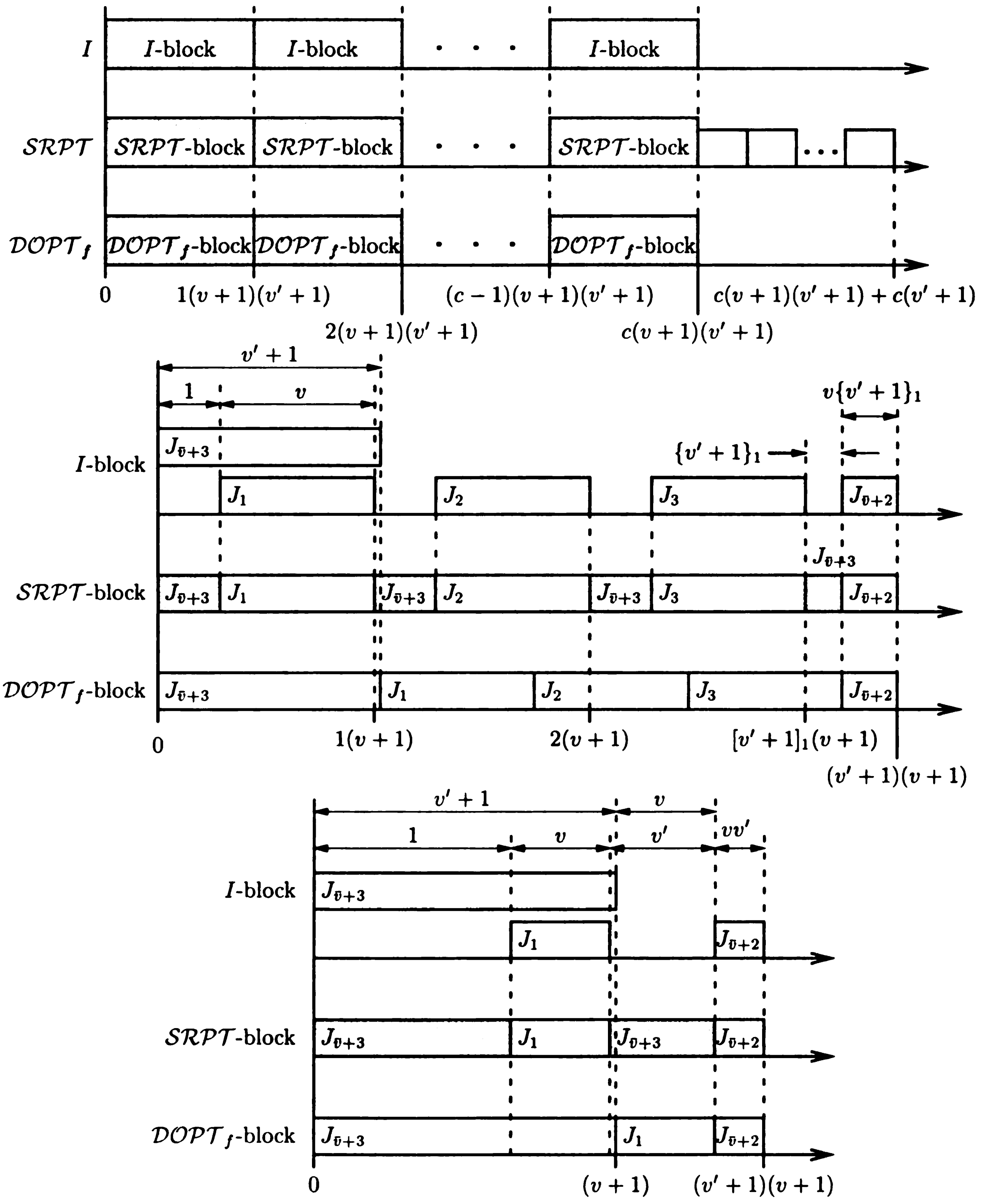


Figure 4.7: Lower bound instance for $\mathcal{SRPT}$ in the preempt-decay model.

$(v'+1)(v+1)$ time units apart. Thus, there is enough room to run all jobs in a cluster before jobs in the next cluster arrive. This can be done by running jobs in each cluster nonpreemptively and contiguously by the order of their release times. In fact, the whole optimal schedule for this instance is obtained by runing all jobs nonpreemptively and contiguously by the order of their release times. The completion time and the flow time of jobs in the optimal schedule are as follows.

$$C_{i,j}^{\mathcal{DOPT}_f} = \begin{cases} r_i + (v'+1) + jv & \text{for } i = 1,...,c \text{ and } j = 1,...,\bar{v}+1 \\ r_{i+1} & \text{for } i = 1,...,c \text{ and } j = \bar{v}+2 \\ r_i + (v'+1) & \text{for } i = 1,...,c \text{ and } j = \bar{v}+3 \end{cases}$$

$$F_{i,j}^{\mathcal{DOPT}_f} = \begin{cases} v'+1+v-j & \text{for } i = 1,...,c \text{ and } j = 1,...,\bar{v}+1 \\ v\{v'+1\}_1 & \text{for } i = 1,...,c \text{ and } j = \bar{v}+2 \\ v'+1 & \text{for } i = 1,...,c \text{ and } j = \bar{v}+3 \end{cases}$$

Thus, the optimal total flow time is

$$\begin{aligned} \mathcal{DOPT}_f(I) &= c\left((v'+1) + v\{v'+1\}_1 + \sum_{i=1}^{[v'+1]_1}(v'+1+v-j)\right) \\ &= c\Big((v'+1) + v\{v'+1\}_1 + v'[v'+1]_1 + [v'+1]_1 + v[v'+1]_1 \\ &\qquad -\frac{1}{2}[v'+1]_1[v'+2]_1\Big) \\ &\leq c(v'+1)(1+v+\frac{1}{2}[v'+1]_1) \end{aligned}$$

In the $\mathcal{SRPT}$ schedule, all jobs except the first job in each clusters (jobs $J_{i,\bar{v}+3}$ for $i = 1,...,c$) are run as soon as they arrive, and they are run to completion without interruptions. In contrast, the first job in each cluster (jobs $J_{i,\bar{v}+3}$ for $i = 1,...,c$) never completes its execution before time r_{c+1}. This can be explained below. Consider the input instance with jobs $J_{i,\bar{v}+3}$, $1 \leq i \leq c$, removed. Notice that the length of the gap in front of any job is exactly $1/v$ times the length of the job. For example, job $J_{i,1}$ has length v, and the gap in front of it has length 1. For another example, job $J_{i,\bar{v}+2}$

has length $v\{v'+1\}_1$, and the gap in front of it has length $\{v'+1\}_1$.

Consider jobs $J_{1,\bar{v}+3}$ and $J_{1,1}$. Job $J_{1,\bar{v}+3}$ is run from time 0 to time 1. At time 1, job $J_{1,1}$ arrives. Since $J_{1,1}$ has a shorter remaining time than $J_{1,\bar{v}+3}$, then $J_{1,\bar{v}+3}$ is preempted. At time $1+s$, job $J_{1,1}$ completes. By the argument above, the work done on job $J_{1,\bar{v}+3}$ has just entirely decayed. Job $J_{1,\bar{v}+3}$ has to start anew. By repeating the argument, jobs $J_{i,\bar{v}+3}$ for $i=1,...,c$ never complete their execution until all other jobs have completed. After all other jobs have completed (at time r_{c+1}), $\mathcal{SRPT}$ run jobs $J_{i,\bar{v}+3}$, $1 \le i \le c$, nonpreemptively in an arbitrary order. The flow time of jobs in the $\mathcal{SRPT}$ schedule is as follows.

$$\begin{aligned}
F_{i,j}^{\mathcal{SRPT}} &= p_{i,j} = \begin{cases} v & \text{for } i=1,...,c \text{ and } j=1,...,\bar{v}+1 \\ v\{v'+1\}_1 & \text{for } i=1,...,c \text{ and } j=\bar{v}+2 \end{cases} \\
\sum_{i=1}^{c} F_{i,\bar{v}+3}^{\mathcal{SRPT}} &= \sum_{i=1}^{c} i((v+1)(v'+1)+v'+1) \\
&= (v+2)(v'+1)c(c+1)/2 \\
\mathcal{SRPT}_f(I) &= c\left(v[v'+1]_1 + v\{v'+1\}_1\right) + \frac{c(c+1)}{2}(v+2)(v'+1) \\
&= cv(v'+1) + \frac{c(c+1)}{2}(v+2)(v'+1) \\
&= c(v'+1)\left(v + \frac{c+1}{2}(v+2)\right)
\end{aligned}$$

Thus, the ratio of the total flow time of the $\mathcal{SRPT}$ schedule and the optimal schedule of this input instance is the following.

$$
\begin{aligned}
\frac{\mathcal{SRPT}_f(I)}{\mathcal{DOPT}_f(I)} &\geq \frac{c(v'+1)\left(v+\frac{c+1}{2}(v+2)\right)}{c(v'+1)(1+v+\frac{1}{2}[v'+1]_1)} \\
&> \frac{\frac{n}{[v'+3]_1}(v+2)}{(2+2v+[v'+1]_1)} \\
&> \frac{n(v+2)}{(3v+4)(v+4)} \\
&> \frac{n}{3(v+4)} \\
&= \frac{nu}{3(4u+1)} \\
&= \begin{cases} \Omega(n) & \text{for } u \geq 1, \text{ and} \\ \Omega(un) & \text{for } u \leq 1. \end{cases}
\end{aligned}
$$

□

4.5 Open Problems in Preempt-Decay Scheduling

The results in section 4.3 show that the optimal flow time between the preempt-resume model and the existing preempt-decay and preempt-repeat model could be a factor of $\Theta(\sqrt{n})$ apart. Corollary 4.3.1 tells us that simply using an optimal algorithm in the preempt-repeat model to approximate a preempt-decay schedule could result in a factor of $\Theta(\sqrt{n})$ from the optimal flow time. Also, Theorem 4.4.3 tells us that simply using the $\mathcal{SRPT}$ algorithm, an optimal algorithm in the preempt-resume model, to approximate a preempt-decay schedule could result in a factor of $\Theta(n)$ from the optimal flow time. These cost differences are significant. The preempt-decay model cannot be approximated by either of the two existing models. Therefore, this problem deserves a further study. A logical next step to pursue this problem is to find a polynomial-time approximation algorithm with a performance guarantee better than $O(\sqrt{n})$, or to prove that there isn't one (unless P=NP).

To the best of our knowledge, there are no previous studies in the preempt-decay model. We study only one particular setting. The preempt-decay model can be applied to many other settings such as problems with due dates or with multiple machines. Examples are $1|r_j, decay|L_{max}$, $1|r_j, decay|(1-U_j)$, and $P|r_j, decay|\sum F_j$.

Chapter 5

The k-Client Problem

5.1 Introduction

In the basic k-client problem, there is one server, k clients, and a metric space (e.g. a plane or a line). Each client generates a sequence of requests in the metric space, and a request is serviced the moment the server, which moves at constant speed, moves to the location of the request; that is, we assume zero processing time for all requests. We define a move of the server to be a non-zero distance movement that takes it from one request to a second request with no intervening requests.

An input instance I of the basic k-client problem has clients 1 through k where client i for $1 \leq i \leq k$ has n_i requests. We define $n_{max}(I) = \max_{1 \leq i \leq k} n_i$ and $n(I) = \sum_{1 \leq i \leq k} n_i$. We represent I by the set of requests $\{r_{i,j}\}$ where $r_{i,j}$ is the j^{th} request of client i for $1 \leq i \leq k$ and $0 \leq j \leq n_i$ (for notational convenience, we assume that each client has a dummy request $r_{i,0}$ for $1 \leq i \leq k$ located at the initial server position). At any time, each client has at most one request in the system; more specifically, request $r_{i,j+1}$ arrives exactly when $r_{i,j}$ has been serviced for $1 \leq i \leq k$ and $0 \leq j \leq n_i - 1$. Thus, for $1 \leq i \leq k$, $r_{i,0}$, the dummy request of each client, is serviced at time 0 and $r_{i,1}$, the first real request of each client, arrives at time 0. Because of our zero processing time assumption, we can assume without loss of generality that consecutive requests of a client are not located at the same point. An algorithm $\mathcal{A}$ solves a specific input instance I by computing a schedule $\mathcal{A}(I)$ of server movement.

We will evaluate the quality of $\mathcal{A}(I)$ using two cost functions. We first consider a *server* oriented cost function which measure the "length" of $\mathcal{A}(I)$: the total distance cost function measures the total distance moved by the server. The total distance cost function is used in the k-server problem. We next consider a *client* oriented cost function which focus on the quality of service provided to each client: the average completion time cost function measures the average completion time of any *entire* client.

For any algorithm $\mathcal{A}$ and any input instance I, we use $\mathcal{A}^D(I)$ to denote the total distance moved in schedule $\mathcal{A}(I)$ and $\mathcal{A}^{\text{ACT}}(I)$ to denote the average completion time of all clients in schedule $\mathcal{A}(I)$. For any cost function CF, the competitive ratio $c_{\mathcal{A}}^{\text{CF}}$ of an on-line algorithm $\mathcal{A}$ is defined as

$$c_{\mathcal{A}}^{\text{CF}} = \sup_I \frac{\mathcal{A}^{\text{CF}}(I)}{\mathcal{OPT}^{\text{CF}}(I)}$$

where $\mathcal{OPT}$ denotes the optimal offline algorithm. When the cost function is not ambiguous, we will drop the CF superscript. That is, we will abuse notation by using $\mathcal{A}(I)$ to represent both the schedule produced by $\mathcal{A}$ as well as its cost for the given cost function, and we will use $c_{\mathcal{A}}$ to represent its competitive ratio for the given cost function.

To simplify later proofs, we define the following notation to represent some commonly used features of input instance I. Let $I_i \subseteq I$ denote the input instance consisting only of requests from client i of I. Let $D_i(I) = \mathcal{OPT}^D(I_i)$; that is, the minimum distance the server must move to service only client i. Let $D(I) = \sum_{i=1}^{k} D_i(I)$ and $D_{max}(I) = \max_{1 \leq i \leq k} D_i(I)$. Let $d_{i,j}(I) = \delta(r_{i,j}, r_{i,j+1})(I)$ denote the distance between the j^{th} and $j + 1^{st}$ requests of client i of I. Clearly $D_i(I) = \sum_{j=0}^{n_i - 1} d_{i,j}(I)$. Note we will typically omit I when the input instance I is not ambiguous.

One of the goals of this work is to study the performance of several commonly used disk scheduling algorithms in the context of multithreaded systems. In particular, we wish to determine how the performance of these algorithms is related to the number of threads in the system. There are two main types of algorithms: greedy algorithms which seek to optimize some short term objective and "fair" algorithms which seek to insure all threads receive service in a reasonably timely fashion.

We first define two commonly studied greedy algorithms. The Shortest Distance First ($\mathcal{SDF}$) algorithm moves the server to the nearest request. In disk scheduling, this algorithm is often referred to as Shortest Seek Time First. The Sequential ($\mathcal{SEQ}$) algorithm services all the requests for one client, then services all the requests of a second client, and so on. Note $\mathcal{SEQ}$ will service requests from other clients if it happens to pass over them while servicing the current client.

Our upper bound results apply to any metric space. For lower bound proofs, we consider two primary metric spaces: $line(\infty)$ and K_i. The first metric space, $line(\infty)$, represents an unbounded continuous line. The K_i metric space is a clique with i nodes where each node is distance 1 from any other node.

Surprisingly, nearly all of our results are identical for these two different cost functions. The summary is shown in Figure 5.1. In section 5.2, we prove an upper bound of $2k-1$ for the average completion time cost function; we show that $\mathcal{SDF}$ and $\mathcal{SEQ}$ algorithms are $(2k-1)$-competitive on any metric space for the average completion time cost function, and this bound is tight for any metric space containing $line(\infty)$. In section 5.3, we prove lower bounds of $\frac{\lg k}{2}+1$ for the total distance and average completion time cost functions when the metric space is K_∞. In section 5.4.1, we prove lower bounds of $\frac{9}{5}$ for both the total distance and average completion time cost functions when $k=2$ and K_∞ is the metric space. In section 5.4.2, we show that when $k=2$, the lower bound for the average completion time cost function improves to 3 for any metric space containing $line(\infty)$. In section 5.5, we discuss some open questions regarding the k-client problem.

5.2 Upper Bounds

In this section, we prove that the greedy $\mathcal{SDF}$ and $\mathcal{SEQ}$ algorithms are $(2k-1)$-competitive for the average completion time cost function in any metric space. We also show that this bound is tight for any metric space containing $line(\infty)$. We begin by proving some a basic fact about the optimal off-line algorithm.

Fact 5.2.1. *For all input instances I, $\mathcal{OPT}^{ACT}(I) \geq \frac{D(I)}{k}$.*

Proof. Consider any input instance I with k clients. For $1 \leq i \leq k$, $\mathcal{OPT}$ clearly cannot complete client i before $D_i(I)$, the time it takes to complete client i if client i is the only client. Therefore, $\mathcal{OPT}^{\text{ACT}}(I)$, the average completion time incurred by $\mathcal{OPT}$, is lower bounded by $\frac{1}{k}\sum_{i=1}^{k} D_i(I) = \frac{D(I)}{k}$. □

	k clients		
Cost Function	Upper Bound	Lower Bound (K_∞)	Lower Bound ($line(\infty)$)
total distance	$2k-1$ [4]	$\frac{\lg k}{2}+1$	$\frac{\lg k}{2}+1$ [4]
average completion time	$2k-1$	$\frac{\lg k}{2}+1$	$\frac{\lg k}{2}+1$ [4]

	2 clients		
Cost Function	Upper Bound	Lower Bound (K_∞)	Lower Bound ($line(\infty)$)
total distance	3 [4]	9/5	25/9 [4]
average completion time	3	9/5	3

Table 5.1: Summary of results for the k-client problem.

5.2.1 Upper bounds for the Total Distance Cost Function

For completeness, we include some upper bound results in [4].

Theorem 5.2.1. *[4] For the total distance cost function,* $c_{\mathcal{SDF}} = 2k - 1$.

Theorem 5.2.2. *[4] For the total distance cost function,* $c_{\mathcal{SEQ}} = 2k - 1$.

The lower bound instance against both $\mathcal{SDF}$ and $\mathcal{SEQ}$ is illustrated in Figure 5.1 [4]. The optimal solution for this instance is to move the server to the far right to point $k-1$ and then to the far left to point $-n_1$ resulting in a cost of $n_1 + (2k-2)$. Assuming ties are broken to $\mathcal{SDF}$'s detriment, $\mathcal{SDF}$ will move left to service all of the requests of client 1, then return to the right and service all of the requests of client 2, etc., resulting in a total cost of $(2k-1)n_1 + O(k^2)$. For any $\epsilon > 0$, there exists an n_1 such that the competitive ratio will exceed $2k - 1 - \epsilon$. Note, this input instance can easily be adjusted to eliminate ties.

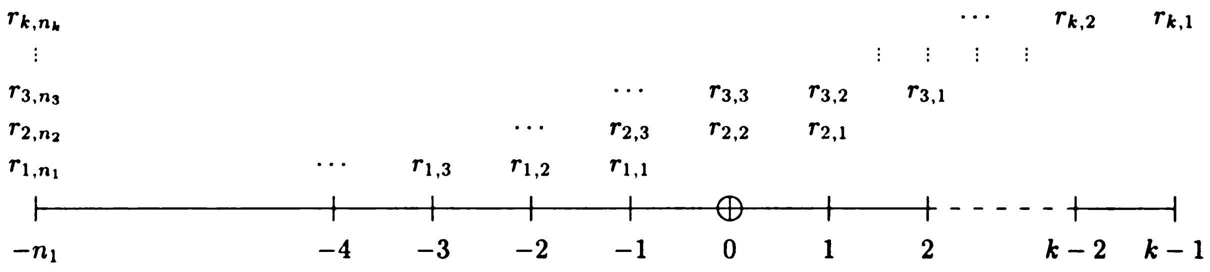


Figure 5.1: Lower bound instance for $\mathcal{SDF}$ and $\mathcal{SEQ}$ with the total distance cost function.

5.2.2 Upper bounds for the Average Completion Time Cost Function

Theorem 5.2.3. *For the average completion time cost function,* $c_{\mathcal{SDF}} = 2k - 1$.

Proof. Without loss of generality, we assume the clients are labeled according to the order that $\mathcal{SDF}$ will service their last requests. That is, the completion time of client i is no greater than the completion time of client j in $\mathcal{SDF}(I)$ for $1 \le i \le j \le k$.

We bound $\mathcal{SDF}^{\mathrm{ACT}}(I)$ by bounding the cost of each move of $\mathcal{SDF}$. Every move begins from a request $r_{i,j}$ for $1 \le i \le k$ and $0 \le j \le n_i$.

If a move starts from request $r_{i,j}$ where $j < n_i$, i.e. $r_{i,j}$ is not a terminal request of client i, then this move adds a cost of at most $d_{i,j}$ to each unfinished client's completion time. This is true because either the server moves to request $r_{i,j+1}$, or it moves to a closer request from another client. Clearly such a move adds a cost of at most $d_{i,j}$ to the average completion time. The total cost of moves starting from nonterminal requests of all clients is at most

$$\sum_{i=1}^{k} \sum_{j=0}^{n_i-1} d_{i,j} = \sum_{i=1}^{k} D_i = D$$

If a move starts from request r_{i,n_i}, the terminal request of client i, then the cost of the move can be upper bounded by $\left(\frac{k-i}{k}\right)(D_i + \min_{i<j\le k} D_j)$. This is true for the following reasons. Clients 1 to i have finished. There are only $(k-i)$ unfinished clients. Consider the unfinished client $j > i$ such that D_j is minimized. Consider the distance between request r_{i,n_i} and the current request of client j. This distance is no more than $D_i + D_j$ because in the worst case, the server must move from r_{i,n_i} back to $r_{i,0} = r_{j,0}$ and then to the current request of client j. Since SDF moves to the closest request, the distance it moves in this case can be no larger than $D_i + D_j$. Thus the completion time of the at most $(k-i)$ unfinished clients is increased by at most $D_i + D_j$ and the average completion time is increased by at most $\left(\frac{k-i}{k}\right)(D_i + D_j)$.

The total cost of moves starting from terminal requests of all clients is at most

$$\begin{aligned}
\sum_{1\le i\le k}\left(\frac{k-i}{k}\right)(D_i+\min_{i<j\le k} D_j) &\le \sum_{1\le i\le k}\left(\left(\frac{k-i}{k}\right)D_i+\sum_{i<j\le k}\frac{D_j}{k}\right)\\
&= \sum_{1\le i\le k}\left(\frac{k-i}{k}\right)D_i+\sum_{1\le i\le k}\sum_{i<j\le k}\frac{D_j}{k}\\
&= \sum_{1\le i\le k}\left(\frac{k-i}{k}\right)D_i+\sum_{1\le j\le k}\sum_{1\le i<j}\frac{D_j}{k}\\
&= \sum_{1\le j\le k}\left(\frac{k-j}{k}\right)D_j+\sum_{1\le j\le k}\left(\frac{j-1}{k}\right)D_j\\
&= \sum_{1\le j\le k}\left(\frac{k-1}{k}\right)D_j\\
&= \left(\frac{k-1}{k}\right)D
\end{aligned}$$

The cost of moves of $\mathcal{SDF}$ starting from nonterminal requests is at most D. The cost of moves of $\mathcal{SDF}$ starting from terminal requests is at most $\left(\frac{k-1}{k}\right)D$. The combined cost is at most $\left(\frac{2k-1}{k}\right)D$. Thus, from Fact 5.2.1, $\mathcal{SDF}^{\mathrm{ACT}}(I) \le (2k-1)\mathcal{OPT}^{\mathrm{ACT}}(I)$. Note this upper bound is true for any metric space.

This bound is tight in any metric space that includes $line(\infty)$. More specifically, $c_{SDF} > 2k-1-\epsilon$ for all $\epsilon > 0$ on $line(\infty)$. The lower bound instance is illustrated in Figure 5.2. Note that client 1 has several requests. All other clients have exactly one request. Assuming that ties are broken to $\mathcal{SDF}$'s detriment, $\mathcal{SDF}$ will first move to the left and service all requests of client 1. It will then service the requests of the remaining clients. The completion time of client 1 will be n_1. The completion time of client j will be $2n_1+j-1$. The average completion time is $\frac{1}{k}((2k-1)n_1+O(k^2))$. The optimal solution is to move all the way to the right first. The optimal average completion time is $\frac{1}{k}(n_1+O(k))$. For any $\epsilon > 0$, there exists an n_1 such that the competitive ratio will exceed $2k-1-\epsilon$. Note that this input instance can easily be adjusted to eliminate ties. □

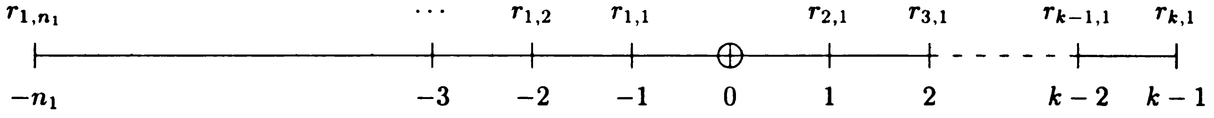


Figure 5.2: Lower bound instance for $\mathcal{SDF}$ and $\mathcal{SEQ}$ with the average completion time cost function.

Theorem 5.2.4. *For the average completion time cost function, $c_{\mathcal{SEQ}} = 2k - 1$.*

Proof. We bound $\mathcal{SEQ}^{\text{ACT}}(I)$ by bounding the completion time of each client of I. For $1 \leq i \leq k$, the completion time of client i is at most $D_i + \sum_{j=1}^{i-1} 2D_j$. This bound assumes a worst-case scenario where the server must service client 1, return to its initial position, then service client 2, return to its initial position, etc., before finally servicing client i. Thus, we obtain the following upper bound on $\mathcal{SEQ}^{\text{ACT}}(I)$.

$$\begin{aligned}
\mathcal{SEQ}^{\text{ACT}}(I) &\leq \left(\frac{1}{k}\right)\left(\sum_{1\leq i\leq k}\left[\left(\sum_{1\leq j<i} 2D_j\right) + D_i\right]\right) \\
&= \left(\frac{1}{k}\right)\left(\left(\sum_{1\leq i\leq k}\sum_{1\leq j<i} 2D_j\right) + \sum_{1\leq i\leq k} D_i\right) \\
&= \left(\frac{1}{k}\right)\left(\left(\sum_{1\leq j\leq k}\sum_{j<i\leq k} 2D_j\right) + \sum_{1\leq i\leq k} D_i\right) \\
&= \left(\frac{1}{k}\right)\left(\sum_{1\leq j\leq k} (k-j)2D_j + \sum_{1\leq j\leq k} D_j\right) \\
&= \left(\frac{1}{k}\right)\left(\sum_{1\leq j\leq k} (2k-2j+1)D_j\right) \\
&\leq \left(\frac{1}{k}\right)\left(\sum_{1\leq j\leq k} (2k-1)D_j\right) \\
&= (2k-1)\left(\frac{D}{k}\right)
\end{aligned}$$

From Fact 5.2.1 we can conclude

$$\mathcal{SEQ}^{\text{ACT}}(I) \leq (2k-1)\mathcal{OPT}^{\text{ACT}}(I)$$

Again, this upper bound holds for any metric space [4], and this bound is tight in any metric space that includes $line(\infty)$ as can be seen by the same lower bound input instance for $\mathcal{SDF}$ (Figure 5.2). □

5.3 General Lower Bounds

We now prove that no on-line algorithm has a competitive ratio better than $\frac{\lg k}{2} + 1$ for either the total distance cost function or for the average completion time cost function when k is a power of 2. These lower bounds drop to $\frac{\lg k}{2}$ when k is not a power of 2. These results apply in clique metric spaces.

This section is organized as follows. In Section 5.3.1, we define a restricted adversary strategy that will be used in the proof of both lower bounds. We then prove some basic properties about this adversary in Section 5.3.2. We then prove the actual lower bound results in Sections 5.3.3 and 5.3.4. In Section 5.3.5, we state some results from [4], which translate these lower bounds to line metric spaces.

5.3.1 Definition of Adversary Strategy $A(N, k)$

The k-client problem can be described as a game between the adversary and the on-line algorithm as follows. The adversary begins the game by generating a single request for each of the k clients in the metric space. The on-line algorithm then responds by moving the server to a location where at least one request resides and servicing all requests at that position. Consider any client whose request has just been serviced. The adversary must respond by either generating another request for this client or informing the on-line algorithm that this client has no more requests. The game continues in this fashion until the on-line algorithm has been informed that all clients have no more requests.

We define an adversary strategy $A(N, k)$ that is parameterized by two integers N and k, both of which must be at least 1. It will utilize a clique with $Nk + 1$ nodes partitioned into N subcliques of size k plus an extra node which is the initial server position. We number the subcliques from 1 to N. On a first reading, it is helpful to focus on adversary strategy $A(1, k)$ which uses only subclique 1. Before we can define adversary $A(N, k)$, we first define the following notation and concepts. Also, for the remainder of this section, when we say "at any time during the game" or "at the end of the game", it will be understood that the game is taking place between adversary strategy $A(N, k)$ and any on-line algorithm $\mathcal{A}$.

Definition 5.3.1. *At any time during the game, let σ_i be the sequence of positions occupied by the requests of client i in the clique of size Nk. Define $\sigma_i(h)$ to be the sequence of positions occupied by the requests of client i in subclique h. Define $I(h)$ to be the restricted input instance where client i has only the sequence of requests $\sigma_i(h)$ for $1 \leq i \leq k$.*

Definition 5.3.2. *At any time during the game, for $1 \leq i \leq k$ and $1 \leq h \leq N$, client i has h-*length *n if $\sigma_i(h)$ currently has length n. Note the h-length of a client may increase over time as the adversary continues to generate requests for client i in subclique h.*

Definition 5.3.3. *At any time during the game, client i* subsumes *client j in subclique h if $\sigma_j(h)$ is a proper subsequence of $\sigma_i(h)$.*

Definition 5.3.4. *Let $I_\phi = \bigcup_{i \in \phi} I_i$ where $\phi \subseteq \{1, \ldots, k\}$.*

Thus, $I(h)_\phi$ is the restricted input instance consisting only of the clients in ϕ and their requests that occur in subclique h.

Fact 5.3.1. *At any time during the game, $\mathcal{OPT}^{ACT}(I(h)_\phi) = D_i(I(h))$ if $i \in \phi$ and $\forall j \in \phi$, i subsumes j in subclique h.*

Proof. The cost of servicing all of the requests of client i in subclique h is $D_i(I(h))$. In order to service client i in subclique h, the server must visit all the locations in $\sigma_i(h)$ in order. It follows that the server will also visit all of locations of $\sigma_j(h)$ in order for any client j subsumed by i in subclique h. □

Definition 5.3.5. *At any time during the game, client i* covers *client j in subclique h if client i subsumes client j in subclique h and there is no client l that is simultaneously subsumed by client i in subclique h and subsumes client j in subclique h. A client j is* uncovered *in subclique h if there is no client i such that i covers j in subclique h.*

We now define the notion of a client i being dead/alive/critically alive in subclique h.

Definition 5.3.6. *At any time during the game, for $1 \leq i \leq k$, and for $1 \leq h \leq N$,*

- *Let l_i be the most recent request of client i that has appeared. Since the game begins with the adversary generating one request for each client, l_i must be defined.*

- *Let c_i be the subclique in which l_i appears.*

- *Client i is* dead *if l_i has been serviced,* **and** *the adversary has made known to the on-line algorithm that client i has no more requests.*

- *Client i is* alive *if it is not dead. An alive client* lives *in subclique h if $c_i = h$. Note that if $c_i = h$, it is not always the case that client i lives in subclique h because client i may be dead.*

- *Client i is* critically alive *if it is alive, request l_i has just been serviced, and the adversary has not yet responded.*

Definition 5.3.7. *At any time during the game, client i is* graftable *in subclique h if it has been planted in subclique h, it is uncovered in subclique h, and it no longer lives in subclique h.*

Definition 5.3.8. *At any time during the game, if we focus on a single subclique h, our adversary is restricted to making only the following three types of moves.*

- *The adversary can* plant *client i in subclique h by placing the next request of client i on a node in subclique h that has not yet been the location of any other request.*

- *The adversary can* terminate *a critically alive client i which lives in subclique h by informing the on-line algorithm that client i has no more requests in subclique h. Note, the adversary may or may not choose to plant the next request of client i in subclique $h + 1$ for $1 \leq h \leq N - 1$.*

- *The adversary can* concatenate *a graftable client j in subclique h with h-length n to a critically alive client in subclique h with the same h-length by making the $n + l^{th}$ request of client i in subclique h be on the same node as the l^{th} request of client j in subclique h for $l = 1, ..., n$. Note that j is no longer graftable in subclique h.*

Definition 5.3.9. *Adversary strategy $A(N, k)$ is defined as follows.*

1. *The adversary begins the game by planting clients* 1 *through k in subclique 1.*
2. *In each later turn, the adversary responds only if there is a critically alive client i. If there is such a client i, let h be the subclique client i lives in, and let n be the h-length of client i. The adversary responds as follows.*
   (a) *If there is a graftable client j in subclique h with h-length n, the adversary* concatenates *client j to client i.*
   (b) *If there is no graftable client j in subclique h with h-length n, $A(N, k)$ responds as follows.*
      i. *The adversary* terminates *client i in subclique h (making client i graftable in subclique h).*
      ii. *If $\mathcal{A}$ has serviced at most $N - 1$ requests, the adversary* plants *client i into subclique $h + 1$. Otherwise, client i is dead.*

5.3.2 Properties of the Adversary Strategy

We now prove some properties about the adversary strategy $A(N, k)$.

Lemma 5.3.1. *For $1 \leq i \leq k$, $1 \leq h \leq N$, and at any time during the game, the h-length of client i is 0 or is 2^l for some integer $l \geq 0$.*

Proof. We first observe that the adversary can only increase the h-length of a client using the planting or concatenation operations. After planting, client i has h-length $1 = 2^0$. Assuming the lemma holds before a concatenation occurs, it must also hold after concatenation since concatenation can only occur between two clients of equal length. □

Lemma 5.3.2. *For $1 \leq i \leq k$, $1 \leq h \leq N$, and any time during the game, client i is covered by at most one other client in subclique h.*

Proof. We first observe that when a client is planted in subclique h, it is not covered since its single request is placed on an otherwise unoccupied node $v(i)$ of subclique h. Afterwards, no other client will contain request $v(i)$ unless client i is concatenated to the end of some other client j. Thus, the only client which can cover client i at this time is client j, and client j may not exist. Assuming such a client j does exist, client i is no longer graftable which means that the only clients which can contain node $v(i)$ are those which subsequently subsume client j. However, by the definition of cover, these clients will not cover client i and the result follows. □

To help understand the adversary strategy, it is useful to consider the graphical representation of the covers relation in each of the subcliques.

Definition 5.3.10. *For $1 \leq h \leq N$ and any time during the game, the* cover graph *H_h of subclique h is a directed graph with k nodes labeled 1 through k. An arc exists from node i to node j in H_h if client i covers client j in subclique h.*

In a general case, the N cover graphs may not help convey the structure of the adversary strategy and the input instance, but for $A(N, k)$, the N cover graphs do help. In particular, our adversary $A(N, k)$ will construct an input instance such that at any time during the game, each H_h will be a collection of directed binomial trees; furthermore, at the end of the game, each H_h will be a directed binomial heap [83, 24].

Corollary 5.3.1. *For $1 \leq h \leq N$ and any time during the game, the cover graph H_h generated by adversary $A(N, k)$ will consist of a set of directed trees.*

Proof. This follows immediately from Lemma 5.3.2. □

Definition 5.3.11. *At any time during the game, let T be a tree in H_h. Define $C(T)$ to be the set of clients which have nodes in tree T, and define the root client $r(T)$ to be the client corresponding to the root node of T.*

Lemma 5.3.3. *For $1 \leq h \leq N$, $1 \leq i \leq k$, and any time during the game, the following statements are true.*

1. *H_h consists of a collection of directed binomial trees [83, 24].*

2. *If client i has h-length $2^l > 0$ for some integer l, then the maximal subtree in H_h rooted at node i is a directed binomial tree with height l containing 2^l nodes.*

3. *For any tree $T \in H_h$, there is at most 1 client in $C(T)$ which lives in subclique h. Furthermore, if there is such a client, it is the root client $r(T)$.*

4. *If client i is in $C(T)$ and client j is in $C(T')$ where $T \neq T'$ are both trees in H_h, then $\sigma_i(h)$ and $\sigma_j(h)$ occupy disjoint sets of nodes in subclique h.*

Proof. Most of these properties follow trivially from the definition of our adversary, the three operations our adversary can perform, the k-client game, and the definition of binomial trees. The key observation is that when client j is concatenated to client i in subclique h, the only change to H_h is the addition of an arc from node i to node j. □

Corollary 5.3.2. *At any time during the game, the on-line algorithm $\mathcal{A}$ can service at most one request per turn.*

Proof. The proof follows from properties 3 and 4 of Lemma 5.3.3. □

Lemma 5.3.4. *For $1 \leq h \leq N$, $1 \leq i \leq k$, and any time during the game, there is at most 1 graftable client with h-length 2^j in subclique h for $j \geq 0$.*

Proof. Fix j and h. Initially, there are no graftable clients in subclique h so there are no graftable clients with h-length 2^j in subclique h. The termination step of the adversary is the only place where the number of graftable clients in subclique h can increase. However, for the termination step to be executed, the number of graftable clients with h-length 2^j must be 0. Otherwise, a concatenation would have been performed instead. Furthermore, after the execution of this step, the number of graftable clients with h-length 2^j is exactly 1. Therefore, the result follows. □

Definition 5.3.12. *Let $B(n)$ be the set of 1-valued bits of the binary representation of n where bit 0 is the least significant bit.*

$$\begin{aligned} \text{For example,} \qquad B(15) &= B(1111_2) = \{3,2,1,0\} \\ B(16) &= B(10000_2) = \{4\}. \end{aligned}$$

$$\text{Note that} \qquad \lfloor n/2^b \rfloor \bmod 2 = \begin{cases} 0 & \text{if } b \notin B(n) \\ 1 & \text{if } b \in B(n). \end{cases}$$

$$\begin{aligned} \text{For example, } \lfloor 1000100_2/2^2 \rfloor \bmod 2 &= 1 \text{ since } B(1000100_2) = \{6,2\} \ni 2 \\ \lfloor 1101111_2/2^4 \rfloor \bmod 2 &= 0 \text{ since } B(1101111_2) = \{6,5,3,2,1,0\} \not\ni 4. \end{aligned}$$

Lemma 5.3.5. *For any subclique h and any $i \geq 1$, if there are exactly i clients with h-length greater than 0 at the end of the game, then*

1. *there is exactly 1 uncovered client with h-length 2^b in subclique h for each $b \in B(i)$, and*

2. *there are no uncovered clients with h-length b in subclique h for $b \notin B(i)$; that is, the underlying undirected graph of H_h is a binomial heap [83, 24] with i nodes.*

Proof. Fix i and h. Let x_b be the number of alive clients with h-length 2^b *created* by the adversary during the course of its operation. Let y_b be the number of uncovered clients with h-length 2^b *left* when the adversary terminates its operation. From the operation of the adversary, in particular steps (2.a) and (2.b), it follows that that for any nonnegative integer b,

$$\begin{aligned} x_b &= \lfloor i/2^b \rfloor \qquad \text{and} \\ y_b &= x_b \mod 2 = \begin{cases} 0 & \text{if } b \notin B(i) \\ 1 & \text{if } b \in B(i). \end{cases} \end{aligned}$$

Thus, the result follows. □

Lemma 5.3.6. *For any subclique h and $1 \leq i \leq k$, if there are exactly i clients with h-length greater than 0 at the end of the game, then the number of requests in subclique h generated by $A(N,k)$ is*

$$\sum_{b \in B(i)} 2^{b-1}(b+2) \geq \begin{cases} \frac{i}{2}\lg i + i & \textit{if } i \textit{ is a power of 2} \\ \frac{i}{2}\lg i & \textit{otherwise.} \end{cases}$$

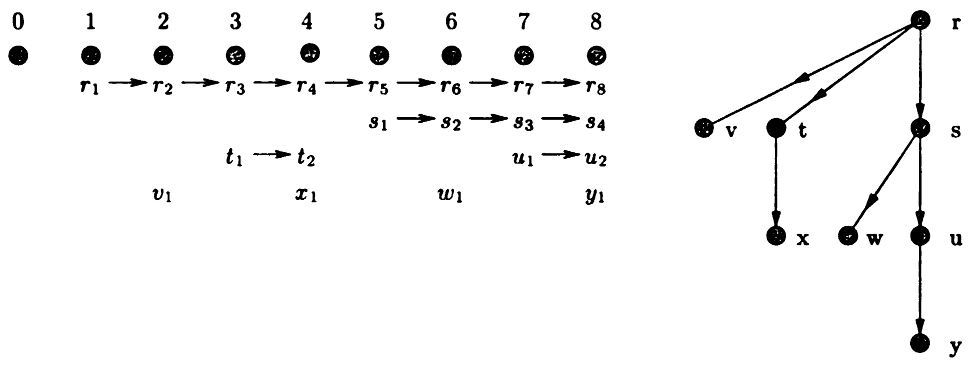
Proof. From Lemma 5.3.3, for $1 \leq h \leq N$, we can determine the number of requests generated in subclique h at the end of the game by summing the number of nodes in the binomial subtree rooted at each node in the final cover graph H_h. From Lemma 5.3.5, H_h will contain exactly $|B(i)|$ disjoint binomial trees, one of size 2^b for each $b \in B(i)$.

Fix b in $B(i)$. Applying well known properties of binomial trees, the binomial subtree of size 2^b in H_h will contain $2^{b-1}/2^j$ nodes which are the roots of binomial trees of size 2^j for $j = 0, \ldots, b-1$. Thus, there are a total of $2^{b-1}(b+2)$ requests represented by this binomial tree. Therefore, the number of requests represented by all binomial trees in H_h is $\sum_{b \in B(i)} 2^{b-1}(b+2)$.

If i is a power of 2, then $i = 2^g$ for some integer $g \geq 0$. Thus, there are $2^{g-1}(g+2)$ requests which is $\frac{i}{2}\lg i + i$. If i is not a power of 2, then from Lemma A.2, the number of requests is lower bounded by $\frac{i}{2}\lg i$. □

We will be particularly interested in the input instances generated by adversary $A(1,k)$ where k is a power of 2. These input instances have the following characteristics. The initial request of each of the k clients is on a different node, no client has more than one request on any node, and there are no requests on node 0. There is a single client with k requests, and for each i, $0 \leq i < (\lg k) - 1$, there are

$k/2^{i+1}$ clients with 2^i requests. Figure 5.3 shows a possible input instance and its corresponding cover graph for $k = 8$. Note, all edges in the clique graph of size 9 have been removed. Also, the nodes have been displayed in a linear fashion to emphasize the structure of the clients.



0 1 2 3 4 5 6 7 8
$r_1 \to r_2 \to r_3 \to r_4 \to r_5 \to r_6 \to r_7 \to r_8$
$s_1 \to s_2 \to s_3 \to s_4$
$t_1 \to t_2$
$u_1 \to u_2$
v_1 x_1 w_1 y_1
r
v
t
s
x
w
u
y

Figure 5.3: An example $A(1, 8)$ lower bound instance and its cover graph.

5.3.3 General Lower Bound for the Total Distance Cost Function

Theorem 5.3.1. *For the total distance cost function and any on-line algorithm $\mathcal{A}$,*

$$c_{\mathcal{A}}^{D} \geq \begin{cases} \frac{1}{2}\lg k + 1 & \textit{if } k \textit{ is a power of 2} \\ \frac{1}{2}\lg k & \textit{otherwise} \end{cases}$$

on the K_{k+1} metric space.

Proof. We use adversary $A(1, k)$ to prove this theorem. In particular, we use K_{k+1} which contains only one subclique. In this case, the adversary description can be simplified by making clause (2.b) simply terminate the client with no option to plant the client in the next subclique.

We first bound the on-line cost incurred by $\mathcal{A}$. From Corollary 5.3.2, the cost incurred by $\mathcal{A}$ will be the number of requests in the input instance. From Lemma 5.3.6, there are a total of $\frac{k}{2}\lg k + k$ requests if k is a power of 2. If k is not a power of 2, the number of requests is lower bounded by $\frac{k}{2}\lg k$.

We now show the optimal off-line cost is k. At the end of the game, H_1 consists of a collection of disjoint binomial trees. Let T be a tree in H_1 at the end of the game. The root client $r(T)$ subsumes all clients in $C(T)$. Thus, from Fact 5.3.1, the optimal off-line algorithm can service all requests of all clients in $C(T)$ by servicing the requests of $r(T)$ in order. If the optimal off-line algorithm does this for each tree in H_1, it will service all requests by visiting each node of subclique 1 exactly once for a total cost of k. No algorithm can do better than this since there are k distinct positions containing requests, so the off-line cost follows.

Dividing the on-line cost by the off-line cost, we get the final result. □

5.3.4 General Lower Bound for the Average Completion Time Cost Function

Theorem 5.3.2. *For the average completion time cost function, any on-line algorithm $\mathcal{A}$, and any integer $N \geq 1$,*

$$c_{\mathcal{A}}^{ACT} \geq \begin{cases} (\frac{\lg k}{2}+1)(\frac{2N+k-1}{2N+2k^2-2}) & \textit{if } k \textit{ is a power of 2} \\ (\frac{\lg k}{2})(\frac{2N+k-1}{2N+2k^2-2}) & \textit{otherwise} \end{cases}$$

on the K_{Nk+1} metric space.

Proof. Adversary $A(1,k)$ used in Section 5.3.3 was concerned with forcing the on-line algorithm $\mathcal{A}$ to traverse nodes as many times as possible while still allowing the optimal off-line algorithm to service all requests by traversing each node only once. This adversary strategy is not appropriate for the average completion time cost function because this strategy allows $\mathcal{A}$ to complete many clients in a short amount of time. We need an adversary strategy which

- forces the on-line algorithm $\mathcal{A}$ to traverse the nodes many times while the off-line algorithm can service all requests by traversing each node only once, and
- prevents the on-line algorithm $\mathcal{A}$ from completing any client too quickly.

Thus, we use $A(N, k)$ which may terminate a client in a subclique but keeps the client alive by planting it in the next subclique.

We first bound the on-line cost incurred by $\mathcal{A}$. From Corollary 5.3.2, $\mathcal{A}$ services at most one request when it visits a node. From step (2.b.ii), after the first $N-1$ requests have been serviced, no client has been completed. Servicing the first $N-1$ requests requires at least $N-1$ time steps, and this contributes $N-1$ to the average completion time. After the first $N-1$ requests are serviced, each client has at least one more request. The cost of servicing these requests contributes at least $\frac{1}{k}\sum_{i=1}^{k} i = \frac{k+1}{2}$ to the average completion time. Thus, the cost incurred by $\mathcal{A}$ is lower bounded by $N-1+\frac{k+1}{2}$.

Next, we compute an upper bound of the optimal cost. We will calculate the average completion time incurred by the obvious algorithm which services the subcliques in order. Clearly this is an upper bound on the cost of the optimal solution.

In an analogous fashion to the previous section, there exists a server path which visits each occupied node of a subclique exactly once and services all the requests on that subclique. The server simply services the requests of the root of each tree in H_h in order. Thus, if H_h has i nodes, then the time required to service all requests in subclique h is at most i. During this time, only these i clients can still have requests waiting to be serviced, so the average completion time increases by at most $\frac{i^2}{k}$. Thus, we can upper bound the cost of the optimal algorithm as follows:

$$\mathcal{OPT}(I) \leq \frac{1}{k}\sum_{i=1}^{k} i^2 p_i \tag{5.1}$$

where p_i is the number of cover graphs with exactly i clients with h-length greater than 0 for $1 \leq h \leq N$.

We will upper bound inequality (5.1) by upper bounding p_i for $i = 1, ..., k$. First, we give an upper bound of the total number of requests. Consider the time just after the last planting operation. At most $N - 1$ requests have been serviced by the on-line algorithm $\mathcal{A}$ at that time. Each client never changes subclique after that. Hence, each client has at most k unserviced requests. Therefore, there are at most k^2 unserviced requests. Thus, there are at most $N - 1 + k^2$ requests.

Let R_i be the total number of requests in a subclique with i clients. The value of R_i is given in Lemma 5.3.6. Since there are at most $N - 1 + k^2$ requests in I, this implies $\sum_{i=1}^{k} R_i p_i \leq N - 1 + k^2$. Thus $\mathcal{OPT}$(I) is upper bounded by the maximization of

$$\frac{1}{k}\sum_{i=1}^{k} i^2 p_i \qquad \text{subject to} \qquad \sum_{i=1}^{k} R_i p_i \leq N - 1 + k^2.$$

Since i^2 grows faster than $\frac{i}{2}\lg i + i$, and from Lemma A.2, $\frac{i}{2}\lg i + i$ grows at least as fast as R_i, then our upper bound on $\mathcal{OPT}$(I) is maximized when

$$\begin{aligned} p_i &= 0 \qquad\qquad \text{for } i = 1, ..., k-1 \quad \text{and} \\ p_k &= \frac{N - 1 + k^2}{R_k}. \end{aligned}$$

$$\text{Thus,}\quad \mathcal{OPT}(I) \;\le\; \frac{1}{k}\sum_{i=1}^{k} i^2 p_i \;=\; \frac{1}{k}k^2 p_k \;=\; k\,\frac{N-1+k^2}{R_k}$$

$$\le \begin{cases} \frac{N-1+k^2}{\frac{1}{2}\lg k+1} & \text{if } k \text{ is a power of } 2 \\ \frac{N-1+k^2}{\frac{1}{2}\lg k} & \text{otherwise.} \end{cases}$$

Dividing the on-line cost by the off-line cost, we get the final result. □

Corollary 5.3.3. *For the average completion time cost function, any on-line algorithm $\mathcal{A}$, and any integer $N \ge 2$, it is the case that on K_{N+k-1} metric space,*

$$c_{\mathcal{A}}^{ACT} \ge \begin{cases} (\frac{\lg k}{2}+1)(\frac{2N+k-1}{2N+2k^2-2}) & \textit{if } k \textit{ is a power of } 2 \\ (\frac{\lg k}{2})(\frac{2N+k-1}{2N+2k^2-2}) & \textit{otherwise.} \end{cases}$$

Proof. Nodes are assigned to the subcliques on demand. A new node is needed when the adversary plants a client. The adversary plants k clients in the initial step. The only other place the adversary plants a client is in step (2.b.ii). The adversary can plant at most 1 client after each request is serviced. The adversary plants no more clients when $\mathcal{A}$ has serviced more than $N-1$ requests. Therefore, the adversary does at most $k+N-1$ plantings. Hence, $k+N-1$ nodes are needed for placing requests.

Since $N \ge 2$, then $k+N-1 \ge k+1$. Thus, other than the k nodes on which the adversary makes k initial plantings, there is 1 other node that can be used as the initial position of the server. □

Corollary 5.3.4. *For the average completion time cost function, any on-line algorithm $\mathcal{A}$, and any $\varepsilon > 0$, there exists an integer M such that, on K_M metric space,*

$$c_{\mathcal{A}}^{ACT} \ge \begin{cases} \frac{\lg k}{2}+1-\varepsilon & \textit{if } k \textit{ is a power of } 2 \\ \frac{\lg k}{2}-\varepsilon & \textit{otherwise.} \end{cases}$$

Proof. From Corollary 5.3.3, by choosing N large enough and setting $M = N+k-1$, the result follows. □

5.3.5 General Lower Bound on the Line

It was shown in [4] that lower bound results on cliques can be transformed into lower bound results on lines. In particular, any lower bound result on K_l using a "finite adversary strategy" can be extended to hold on $line(2l-1)$ where a *finite adversary strategy* is one in which the maximum number of requests ever generated by the adversary is upper bounded by some known constant.

Theorem 5.3.3. *[4] Suppose we have a lower bound of c for either the total distance or average completion time cost functions on the K_l metric space for $l \geq 1$ as a result of a finite adversary strategy. Then this implies there exists a $c - \varepsilon$ lower bound for the same cost function on the $line(2l-1)$ metric space as the result of another finite adversary strategy.*

Corollary 5.3.5. *[4] For the total distance cost function, any on-line algorithm $\mathcal{A}$, and any $\varepsilon > 0$,*

$$c_{\mathcal{A}}^{ACT} \geq \begin{cases} \frac{\lg k}{2} + 1 - \varepsilon & \text{if } k \text{ is a power of 2} \\ \frac{\lg k}{2} - \varepsilon & \text{otherwise} \end{cases}$$

on the $line(2k-1)$ metric space.

Proof. This follows from Theorems 5.3.1 and 5.3.3. □

Corollary 5.3.6. *[4] For the average completion time cost function, any on-line algorithm $\mathcal{A}$, any integer $N \geq 2$, and any $\varepsilon > 0$,*

$$c_{\mathcal{A}}^{ACT} \geq \begin{cases} (\frac{\lg k}{2} + 1)(\frac{2N+k-1}{2N+2k^2-2}) - \varepsilon & \text{if } k \text{ is a power of 2} \\ (\frac{\lg k}{2})(\frac{2N+k-1}{2N+2k^2-2}) - \varepsilon & \text{otherwise} \end{cases}$$

on the $line(2N+2k-3)$ metric space.

Proof. This follows from Corollary 5.3.3 and Theorem 5.3.3. □

Corollary 5.3.7. *[4] For the average completion time cost function, any on-line algorithm $\mathcal{A}$, and any $\varepsilon' > 0$, there exists integer N' such that,*

$$c_{\mathcal{A}}^{ACT} \geq \begin{cases} \frac{\lg k}{2} + 1 - \varepsilon' & \textit{if } k \textit{ is a power of 2} \\ \frac{\lg k}{2} - \varepsilon' & \textit{otherwise} \end{cases}$$

on the line(N') metric space.

Proof. From Corollary 5.3.6, by choosing $\varepsilon < \varepsilon'$, choosing N large enough, and setting $N' = 2N + 2k - 3$, the result follows. □

5.4 General Lower bounds when $k = 2$

5.4.1 General Lower Bound on the Clique when $k = 2$

In this section, we improve our general lower bounds of $\frac{\lg k}{2} + 1$ to $\frac{9}{5}$ for the case where $k = 2$ on K_∞ for both the total distance and average completion time cost functions.

Theorem 5.4.1. *For the total distance cost metric, no on-line algorithm is $(\frac{9}{5} - \epsilon)$-competitive for the 2-client problem on the K_∞ metric space for all $\epsilon > 0$.*

Proof. Label the vertices with positive integers. Let D be a large integer. The adversary operates in rounds. At the beginning of round 1, a client is called the *leader*, and its initial request is on node $D+1$. The other client is called the *follower*, and its initial request is on node 1. Subsequent requests are generated according to the following rule. If the current request of both clients are on different nodes, then when the request, say on node x, of either of the clients is serviced, the next request of that client is on node $x + 1$. If the current request of both clients are on the same nodes, say on node x, then after these two requests are serviced, the new round begins. In the new round, the leader becomes the follower, and its next request is on node 1. The follower becomes the leader, and its next request is on node $x + 1$.

Suppose the adversary stops just after round n. The optimal solution is to service the client that is the follower in round n picking up requests of the other client along the way.

We show that any on-line algorithm which hopes to be better than 2-competitive on this form of input sequence can be described by an infinite monotonically increasing sequence of finite numbers α_i for $i \geq 1$ where $\alpha_1 = (x_1 - D)/D$, $\alpha_i = x_i/D$ for $i \geq 2$, and x_i is the last node on which the clients place their last requests in round i. Suppose α_1 is unbounded; that is, at any time, if the current request of the follower and the leader are on node x and y respectively, then $x < y$. In this case, the adversary will choose the input instance to be one where the final request of both clients is on node xD for a large integer x. The on-line cost will be $xD + xD - D$ while the off-line cost will be xD giving a competitive ratio of $2 - \frac{1}{x}$. Therefore, α_1 must be bounded to have a ratio better than 2. Clearly, this argument generalized which means that all α_i must be bounded for all $i \geq 1$.

Definition 5.4.1. *For $n \geq 1$, let o_n be the sum of α_i where i is an odd number between 1 and n inclusively.*

Definition 5.4.2. *For $n \geq 1$, let e_n be the sum of α_i where i is an even number between 1 and n inclusively.*

Definition 5.4.3. *Let $\lambda_0 = 0$. For $n \geq 1$, let $\lambda_n = \alpha_n - \lambda_{n-1}$.*

Definition 5.4.4. *For $n \geq 1$, let*

$$\gamma_n = \begin{cases} o_n & n \text{ is odd} \\ e_n & n \text{ is even} \end{cases}$$

Fact 5.4.1. *For $n \geq 0, \lambda_n \geq 0$.*

This is true because α_n's is a non-decreasing sequence.

Fact 5.4.2. *For $n \geq 1$,*

$$\lambda_n = \begin{cases} o_n - e_n = o_n - e_{n-1} & n \text{ is odd} \\ e_n - o_n = e_n - o_{n-1} & n \text{ is even} \end{cases}$$

Fact 5.4.3. *For $n \geq 3, \gamma_n = \gamma_{n-2} + \alpha_n$.*

This is true by Definition 5.4.4, 5.4.1, and 5.4.2.

Fact 5.4.4. *For $n \geq 2, \gamma_n = \gamma_{n-1} + \lambda_n$.*

This is true by Definition 5.4.4 and Fact 5.4.2.

The cost of the on-line algorithm is,

$$\begin{aligned} \forall n \geq 2 \qquad A(I_n) &= (\gamma_n + \gamma_{n-1} + \alpha_n)D - n \\ &= (\gamma_n + \gamma_{n-1} + \lambda_{n-1} + \lambda_n)D - n. \end{aligned}$$

The optimal cost is,

$$\begin{aligned} \forall n \geq 2 \qquad OPT(I_n) &= (\gamma_{n-1} + \alpha_n)D \\ &= (\gamma_{n-1} + \lambda_{n-1} + \lambda_n)D. \end{aligned}$$

Therefore, if the adversary stops just after round n, then the ratio of the cost of the on-line algorithm and the adversary is

$$\frac{(\gamma_{n-1} + \gamma_n + \lambda_{n-1} + \lambda_n)D - n}{(\gamma_{n-1} + \lambda_{n-1} + \lambda_n)D}, \qquad n \geq 2.$$

If D is sufficiently large, the ratio is arbitrarily close to

$$\frac{\gamma_{n-1}+\gamma_n+\lambda_{n-1}+\lambda_n}{\gamma_{n-1}+\lambda_{n-1}+\lambda_n} = \frac{2\gamma_{n-1}+\lambda_{n-1}+2\lambda_n}{\gamma_{n-1}+\lambda_{n-1}+\lambda_n} = 2 - \frac{\lambda_{n-1}}{\gamma_{n-1}+\lambda_{n-1}+\lambda_n}, n \geq 2. \quad (5.2)$$

We can write the equation (5.2) as $2 - c_n$ where

$$c_n = \frac{\lambda_{n-1}}{\gamma_{n-1}+\lambda_{n-1}+\lambda_n}, \qquad n \geq 2. \quad (5.3)$$

We now use the property that λ_i is finite for all $i \geq 1$ to define two important quantities of any sequence of finite values $\langle\lambda_i\rangle$. First, we define $X(\langle\lambda_i\rangle) = \inf_{n\geq 2} c_n$ where c_n is defined in equation (5.3). Note that $0 < X(\langle\lambda_i\rangle) < 1$. For the remainder of this proof, we will use X to denote $X(\langle\lambda_i\rangle)$. Next we define $R(\langle\lambda_i\rangle) = \inf_{n\geq 2}\frac{\lambda_n}{\gamma_{n-1}}$. For the remainder of this proof, we will use R to denote $R(\langle\lambda_i\rangle)$.

To derive the best possible lower bound on the competitive ratio of any on-line algorithm against this adversary, we need to find the set of values $\langle\lambda_i\rangle$ that minimize $2 - X$. We will show that the maximum possible value of X is $1/5$ which is achieved when $\frac{\lambda_n}{\gamma_{n-1}} = 1$ for all $n \geq 2$; that is, when $R = 1$.

We first observe that

$$\forall n \geq 2 \quad \lambda_n \leq \frac{1-X}{X}\lambda_{n-1} - \gamma_{n-1}. \quad (5.4)$$

This follows from equation (5.3) and the definition of X.

We next observe that

$$\forall n \geq 3 \quad \lambda_n \geq R(R+1)\gamma_{n-2}. \quad (5.5)$$

This can be derived as follows given the definitions of R and Fact 5.4.4.

$$\lambda_n \geq R\gamma_{n-1} = R(\lambda_{n-1}+\gamma_{n-2}) \geq R(R\gamma_{n-2}+\gamma_{n-2}) = R(R+1)\gamma_{n-2}.$$

From inequalities (5.4) and (5.5), we obtain

$$\forall n \geq 3 \qquad R(R+1)\gamma_{n-2} \leq \frac{1-X}{X}\lambda_{n-1} - \gamma_{n-1}$$

which can be rewritten as

$$\forall n \geq 2 \qquad (1-2X)\frac{\lambda_n}{\gamma_{n-1}} \geq X(R^2+R+1) \tag{5.6}$$

Combining inequality (5.6) and the definition of R as the infimum of $\frac{\lambda_n}{\gamma_{n-1}}$ for $n \geq 2$, we obtain

$$(1-2X)R \geq X(R^2+R+1)$$

which can be rewritten as

$$XR^2 + (3X-1)R + X \leq 0 \tag{5.7}$$

Since R must be a real number, we apply the quadratic equation and the constraint $0 < X < 1$ to find the maximum value of X that results in R being a real number.

$$(3X-1)^2 - 4 \cdot X \cdot X \geq 0$$

The maximum value of $X = 1/5$ when $R = 1$. Therefore, we achieve the desired lower bound of $2 - 1/5 = 9/5$ on the competitive ratio of any on-line algorithm for the 2-client problem on K_∞. □

Theorem 5.4.2. *For the average completion time cost metric, no on-line algorithm is* $(\frac{9}{5} - \epsilon)$*-competitive for the 2-client problem on the* K_∞ *metric space for all* $\epsilon > 0$.

Proof. The proof is the same as that of Theorem 5.4.1 and because both clients complete at the same time in both the optimal off-line schedule and the on-line schedule. □

5.4.2 General Lower Bound on the Line when $k = 2$

In this section, we improve our general lower bound of $\frac{\lg k}{2} + 1$ to 3 for average completion time cost function for the case when $k = 2$ and the metric space is $line(\infty)$. For completeness, we state a lower bound of $\frac{25}{9}$ for total distance cost function for the case when $k = 2$ and the metric space is $line(\infty)$ [4].

Theorem 5.4.3. *For the total distance cost function, no on-line algorithm is* $(\frac{25}{9} - \epsilon)$*-competitive for the 2-client problem on* $line(\infty)$ *for all* $\epsilon > 0$ *[4].*

Theorem 5.4.4. *For the average completion time cost function, no on-line algorithm is* $(3 - \epsilon)$*-competitive for the 2-client problem on* $line(\infty)$ *for all* $\epsilon > 0$.

Proof. We consider an adversary that constructs inputs of the following form. The server is initially located at the origin. The first request from client A is located one unit to the left of the origin. All the remaining requests from A will each be δ to the left of the previous one where δ tends to 0. The requests from client B will be arranged to the right of the origin in a similar fashion.

Without loss of generality, we assume that the on-line algorithm initially moves the server to the left to service client A. Any on-line algorithm that hopes to be better than 3-competitive against this adversary can be described by two finite positive numbers α and β. The first parameter, α, is the maximum distance the server is willing to move to the left before it changes direction for the first time. The second parameter, β, is the maximum distance the server is willing to move to the right before it changes direction for the second time.

Suppose α is unbounded; that is, the on-line algorithm will move the server to the left until there are no more requests to the left. In this case, the adversary will choose the input instance to be one where the final request of client A is at position $-x$ for a large value of x and client B will have only 1 request at position 1. The on-line cost will be $\frac{1}{2}((x) + (2x+1)) = \frac{1}{2}(3x+1)$ while the off-line cost will be $\frac{1}{2}((1) + (2+x)) = \frac{1}{2}(x+3)$ giving a competitive ratio of $3 - \frac{8}{x+3}$. Therefore, α must be bounded to have a competitive ratio better than 3. We can show that β is bounded with a similar argument.

Next, we show that the best an on-line algorithm can do is to make $\alpha = \beta$ in which case the on-line algorithm will be 3-competitive at best. For any given α and β, define the instance I as follows. Requests from client A appear progressively from position -1 to position $-\alpha - \delta$. Requests from client B appear progressively from position 1 to position $\beta + \delta$. Figure 5.4 illustrates both input instance I as well as how the on-line algorithm behaves on I. Clearly, the on-line algorithm will move left to position $-\alpha$, move right to position β, move left to position $-\alpha - \delta$ (completing client A), and then move right to position $\beta + \delta$ (completing client B). Thus, the average completion time incurred by the on-line algorithm on input instance I is $\frac{1}{2}((3\alpha + 2\beta + \delta) + (4\alpha + 3\beta + 3\delta)) = \frac{1}{2}(7\alpha + 5\beta + 4\delta)$. Meanwhile, $\mathcal{OPT}$ services the shorter client first which means $\mathcal{OPT}^{\text{ACT}}(I) = \min(\frac{1}{2}((\alpha+\delta) + (2\alpha+\beta+3\delta)), \frac{1}{2}((\beta+\delta) + (2\beta+\alpha+3\delta))) = \frac{1}{2}\min(3\alpha+\beta+4\delta, \alpha+3\beta+4\delta)$. Since we can make δ arbitrarily small, then this implies a lower bound of $\frac{7\alpha+5\beta}{\min(3\alpha+\beta,\alpha+3\beta)}$ on the competitive ratio of the on-line algorithm.

We now show that this lower bound is minimized to be 3 when $\alpha = \beta$. There

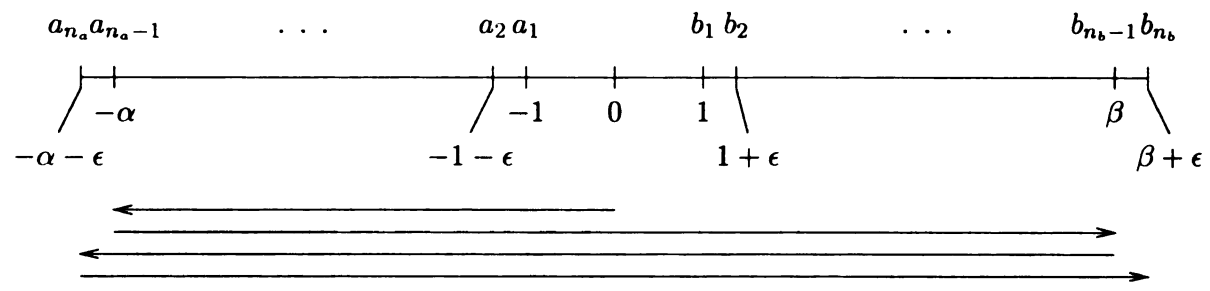


Figure 5.4: Lower bound instance for the average completion time cost function and $k = 2$.

are 2 cases to consider.

Case 1: $\alpha \leq \beta$

In this case, $\mathcal{OPT}^{\text{ACT}}(I) = \frac{1}{2}(3\alpha + \beta)$ which means that the lower bound on the competitive ratio of the on-line algorithm is $\frac{7\alpha+5\beta}{3\alpha+\beta}$ which is strictly larger than 3 when $\alpha < \beta$ and which equals 3 when $\alpha = \beta$.

Case 2: $\beta \leq \alpha$

In this case, $\mathcal{OPT}^{\text{ACT}}(I) = \frac{1}{2}(\alpha + 3\beta)$ which means that the lower bound on the competitive ratio of the on-line algorithm is $\frac{7\alpha+5\beta}{\alpha+3\beta}$ which is strictly larger than 3 when $\beta < \alpha$ and which equals 3 when $\alpha = \beta$.

Thus the result follows. □

5.5 Open Problems on the k-Client Problem

Obvious open problems are closing the gap between the lower bound of $\frac{\lg k}{2} + 1$ and the upper bound of $2k - 1$ for the total distance and average completion time cost functions. For the total distance cost function and when $k = 2$, there is still a gap between the lower bound of $\frac{25}{9}$ and the upper bound of 3.

In [4], the *k-client l-server problem* is considered. In this problem, there are l servers instead of only a single server. This model is a natural generalization of both the k-client problem and the classic l-server problem. Some preliminary results on this problem can be found in [4]. The question of how to schedule multiple servers efficiently in this model remains open.

Chapter 6

Minimizing Total Completion Time on One Machine

6.1 Introduction

In this chapter, we consider the problem of non-preemptive scheduling with release dates on one machine to minimize total completion time ($1|r_j|\sum_j C_j$). Many approximation algorithms for this problem and similar problems use the idea of "α-schedules" [40, 73, 39, 34, 74, 19]. Previously, the best approximation algorithm was based on this idea. This algorithm is an $\frac{e}{e-1}$-approximation algorithm called $\mathcal{BEST}$-α proposed by Chekuri, Motwani, Natarajan, and Stein [19] ($\frac{e}{e-1} \approx 1.58$). However, it was not known whether the bound of $e/(e-1)$ for $\mathcal{BEST}$-α was tight.

We generalize the idea of "α-schedules" and characterize a class of approximation algorithms which includes $\mathcal{BEST}$-α. We show that no algorithm in this class has an approximation ratio better than $e/(e-1)$. This implies that the performance guarantee of $\mathcal{BEST}$-α proven by Chekuri *et al.*[19] is tight. To find a better approximation algorithm, new ideas are needed. Afrati *et al.* introduced some new ideas beyond "α-schedule" and devised a polynomial time approximation scheme (PTAS) for this problem [1].

6.2 $\mathcal{SRPT}$-Subsequence Algorithms

6.2.1 Definition of α-Schedules and the $\mathcal{BEST}$-α Algorithm

For any algorithm $\mathcal{A}$ and any instance I, let $\mathcal{A}(I)$ denote the schedule for I given by $\mathcal{A}$. When there is no ambiguity, we also use $\mathcal{A}(I)$ to denote $C^{\mathcal{A}}(I)$, the total completion time of the schedule produced by $\mathcal{A}$. The **Shortest Remaining Processing Time** algorithm ($\mathcal{SRPT}$) is an optimal algorithm for the *preemptive* version of the problem we consider in this chapter. $\mathcal{SRPT}$ always schedules a job with the smallest remaining processing time. For any instance I and $\alpha \in [0,1]$, let $C_j^{\mathcal{SRPT}}(I,\alpha)$ be the time at which α-fraction of J_j is completed in schedule $\mathcal{SRPT}(I)$.

In the $\mathcal{SRPT}$ schedule of an input instance, a preemption can occur only when a job is running on the machine, and another job has just arrived. Therefore, there are at most $n-1$ preemptions in the $\mathcal{SRPT}$ schedule. When a preemption occurs, α-fraction of the preempted job is completed for some value of α in the interval $(0,1)$. Since there are at most $n-1$ preemptions in the $\mathcal{SRPT}$ schedule, then there are at most $n-1$ distinct values of α at which the preemptions occur. These $n-1$ critical values of α can subdivides the interval $[0,1]$ into at most n sub-intervals.

An α-schedule is a non-preemptive schedule obtained by list scheduling jobs in order of increasing $C_j^{\mathcal{SRPT}}(I,\alpha)$ for some α in the interval $[0,1]$. Different values of α chosen in the same sub-interval lead to the same ordering of $C_j^{\mathcal{SRPT}}(I,\alpha)$ which implies the same α-schedule. Since there are at most n sub-intervals, then there are at most n distinct α-schedules. For each instance I, $\mathcal{BEST}$-α chooses an α-

schedule which gives the smallest total completion time. $\mathcal{BEST}$-α was shown to be an $e/(e-1)$-approximation algorithm for $1|r_j|\sum_j C_j$ [19].

6.2.2 Definition of $\mathcal{SRPT}$-Subsequence Algorithms

We define the *$\mathcal{A}(I)$-sequence* to be the sequence of job identities of job pieces in schedule $\mathcal{A}(I)$. A schedule S for I is called an *$\mathcal{A}(I)$-subsequence schedule* if the S-sequence is a subsequence of the $\mathcal{A}(I)$-sequence. For any algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, if for any instance I, the $\mathcal{A}_1(I)$-sequence is an $\mathcal{A}_2(I)$-subsequence, then we call $\mathcal{A}_1$ an *$\mathcal{A}_2$-subsequence algorithm.* See Figure 6.1 for an example. The figure shows 3 schedules for instance I with corresponding sequences on the right. S_1 is the $\mathcal{SRPT}$-schedule. Since 23451 is a subsequence of 123214151 while 13452 is not, then S_2 is a non-preemptive $\mathcal{SRPT}$-subsequence schedule while S_3 is not. In this chapter, we will focus on non-preemptive $\mathcal{SRPT}$-subsequence algorithms. Note that $\mathcal{BEST}$-α belongs to this class.

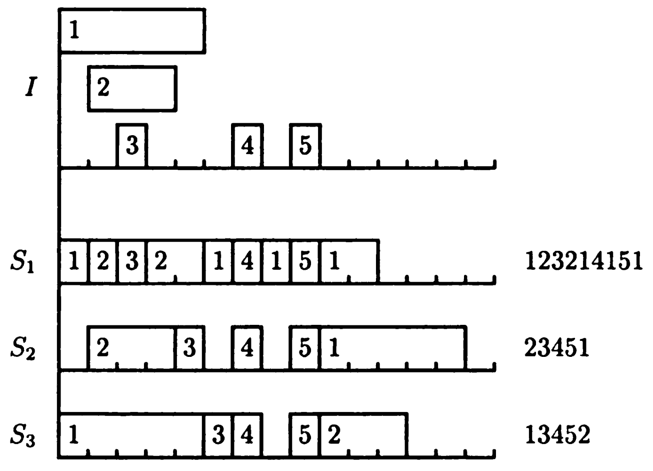


Figure 6.1: Examples of non-preemptive $\mathcal{SRPT}$-subsequence schedule.

6.3 Lower Bound of Non-Preemptive $\mathcal{SRPT}$-Subsequence Algorithms

Our main result is that no non-preemptive $\mathcal{SRPT}$-subsequence algorithm including $\mathcal{BEST}$-α has an approximation ratio better than $\frac{e}{e-1}$.

We need the following notations. For any positive integer n, let n_e denote $\lfloor n/e \rfloor$, and let

$$\begin{aligned} H_n &= \sum_{k=1}^{n} \frac{1}{k} \\ \widehat{H}_n &= H_n - H_{n_e} = \sum_{k=n_e+1}^{n} \frac{1}{k}. \end{aligned}$$

Note that $\widehat{H}_n = H_n - H_{n_e} \approx \ln n - \ln \frac{n}{e} = \ln e = 1$. Next, we describe lower bound input instances.

Definition 6.3.1. *For any integer $n \geq 3$, and $\varepsilon > 0$, we define the instance $I_n(\varepsilon)$ with n jobs as follows:*

$$\begin{aligned} r_i &= \begin{cases} 0 & \text{for } i = 1 \\ 1 + (n+1-i)\varepsilon & \text{for } i = 2, ..., n_e \\ H_n - H_{i-1} = \sum_{k=i}^{n} \frac{1}{k} + (n-i)\varepsilon & \text{for } i = n_e + 1, ..., n \end{cases} \\ p_i &= \begin{cases} 1 + \varepsilon & \text{for } i = 1 \\ \varepsilon & \text{for } i = 2, ..., n \end{cases} \end{aligned}$$

See Figure 6.2 for example of instances from Definition 6.3.1. The figure shows (1) the instance $I_9(\varepsilon)$ where $\varepsilon \to 0^+$, (2) the $\mathcal{SRPT}$ schedule, (3) all possible non-preemptive $\mathcal{SRPT}$-subsequence schedules, and (4) the optimal non-preemptive schedule for $I_9(\varepsilon)$. Note that $\widehat{H}_9 < 1$. We will only be interested in instances $I_n(\varepsilon)$ such that $\widehat{H}_n < 1$ and $0 < \varepsilon < 1/n$. Among these instances, we will show that as n tends to infinity and ε tends to 0, the total completion time of any non-preemptive

$\mathcal{SRPT}$-subsequence schedule will tend to $e/(e-1)$ times the optimal total completion time.

The intuitive idea behind instances from Definition 6.3.1 is the following. Consider any non-preemptive $\mathcal{SRPT}$-schedule. No matter where the large job is scheduled in the non-preemptive $\mathcal{SRPT}$-subsequence schedule, the total completion time remains constant. Note, the latest the large job can be scheduled is between jobs J_{n_ϵ} and $J_{n_\epsilon+1}$ which means it will delay at least about one-third of the small jobs no matter where it is run. On the other hand, in the optimal non-preemptive schedule, the large job is run last. Each of the small jobs is run at the earliest possible moment. Only the large job incurs a large delay which is negligible when there is a large number of jobs.

In [82], we proved the following theorem.

Theorem 6.3.1. *For any non-preemptive $\mathcal{SRPT}$-subsequence algorithm $\mathcal{A}$ and for all $\delta > 0$, there exist an integer n and $\varepsilon > 0$ such that*

$$\frac{\mathcal{A}(I_n(\varepsilon))}{\mathcal{SRPT}(I_n(\varepsilon))} > \frac{\mathcal{A}(I_n(\varepsilon))}{\mathcal{OPT}(I_n(\varepsilon))} \geq \frac{e}{e-1} - \delta.$$

However, we will omit the proof of Theorem 6.3.1 because it is rather lengthy and complicated. Instead, we borrow a technique used in the proof of Theorem 2.2.1 in Chapter 2 and prove a similar theorem. Again, we will need Dirac's delta function $\delta(\cdot)$ which is defined as follows.

$$\begin{aligned} g_\Delta(t) &= \begin{cases} 1/\Delta & 0 < t < \Delta \\ 0 & \text{elsewhere} \end{cases} \\ \delta(\cdot) &= \lim_{\Delta \to 0^+} g_\Delta(\cdot) \end{aligned}$$

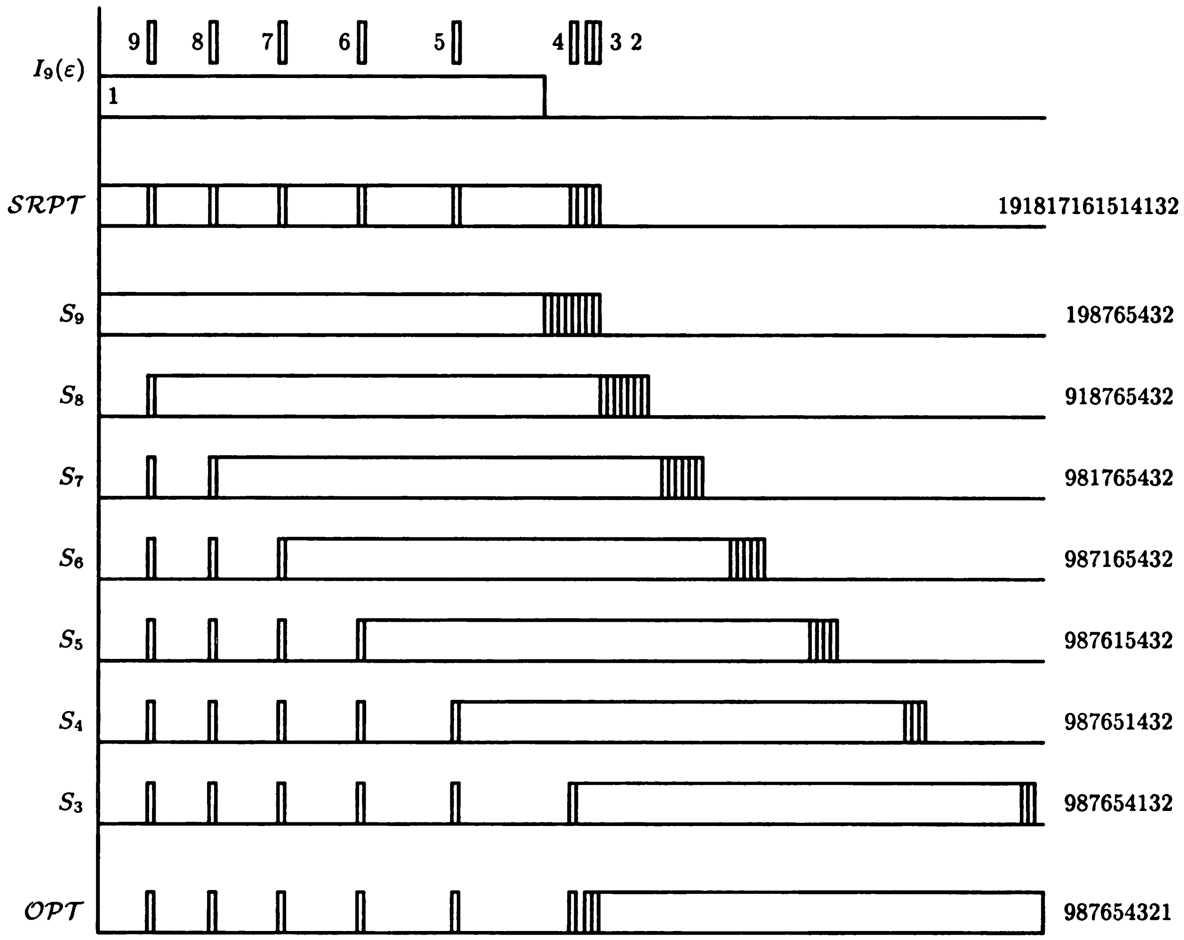


Figure 6.2: Lower bound instance for non-preemptive $\mathcal{SRPT}$-subsequence algorithm.

Note that

$$\int_u^v \delta(t)\,dt = \begin{cases} 0 & \text{if } 0 < u < v \\ 1 & \text{if } u \le 0 < v. \end{cases}$$

Theorem 6.3.2. *For any non-preemptive $\mathcal{SRPT}$-subsequence algorithm $\mathcal{A}$, there exists an (infinite-size) instance I such that*

$$\frac{\mathcal{A}(I)}{\mathcal{OPT}(I)} = \frac{e}{e-1}.$$

Proof. The lower bound instance has the following structure. The number of jobs, n, approaches infinity. There is one job of size 1 released at time 0. All other jobs have size 0. Among all 0-size jobs, e^{-1} fraction of them are released at time 1. Let's call them type-A jobs. The rest of the 0-size jobs are released independently in the interval (0,1) with the density e^{-t} at time t. Let's call them type-B jobs. The release time of 0-size jobs can be described as the following probability density function f:

$$f(t) = e^{-1}\,\delta(1-t) + \begin{cases} e^{-t} & \text{for } 0 < t < 1 \\ 0 & \text{everywhere else} \end{cases}$$

In the $\mathcal{SRPT}$ schedule of this instance, the unit-size job starts at time 0. However, it is always preempted by type-B jobs when they arrive. The pieces of the unit-size job are processed between type-B jobs. The unit-size job completes at time 1. Type-A jobs arrive at time 1 and do not preempt the unit-size job. Note that, to be precise, type-A jobs arrive in the interval $(1 - \lim_{t\to 0^+} t, 1)$, and we should have defined the size of the unit-size job to be $1 - \lim_{t\to 0^+} t$ so that, in the strict sense, the unit-size job finishes before any of type-A jobs arrives.

A non-preemptive $\mathcal{SRPT}$-subsequence schedule for this instance can be described by a single parameter s which represents the starting time of the unit-size job in that schedule. All 0-size jobs which are released before time s are run as soon as they arrive. All 0-size jobs which are released after time s are run as soon as the unit-size job is finished, *i.e.*, they are run at time $s+1$. In the $\mathcal{SRPT}$ schedule, the first piece of the unit-size job is run at time 0, and its last piece is run before time 1. Thus, the value of s corresponding to a non-preemptive $\mathcal{SRPT}$-subsequence schedule must satisfy the relation $0 \leq s < 1$. The optimal non-preemptive schedule is to schedule the unit-size job after all 0-size jobs are completed (at time 1). Thus, the optimal schedule corresponds to setting s to 1.

We will consider the *average* completion time of jobs rather than their *total* completion time. Let C^s_{avg} denote the average completion time of non-preemptive schedule which starts the unit-size job at time s. Since the number of jobs approaches infinity, the completion time of the unit-size job is negligible. The average completion time can be computed as follows.

$$\begin{aligned}
C^s_{\text{avg}} &= \int_0^s tf(t)dt + \int_s^1 (s+1)f(t)dt \\
&= \int_0^s t\left(e^{-t} + e^{-1}\delta(1-t)\right)dt + (s+1)\int_s^1 \left(e^{-t} + e^{-1}\delta(1-t)\right)dt \\
&= \begin{cases} \int_0^s te^{-t}dt + (s+1)\left(\int_s^1 e^{-t}dt + e^{-1}\right) & 0 \leq s < 1 \\ \int_0^1 te^{-t}dt + e^{-1} & s = 1 \end{cases} \\
&= \begin{cases} 1 & 0 \leq s < 1 \\ \frac{e-1}{e} & s = 1 \end{cases}
\end{aligned}$$

Since $0 \leq s < 1$ for any non-preemptive $\mathcal{SRPT}$-subsequence schedule, then the average completion time of such a schedule is 1. The optimal non-preemptive schedule starts the unit-size job at time 1, and has the average completion time of $(e-1)/e$. Thus, the result follows. □

Since $\mathcal{BEST}$-α is an $\frac{e}{e-1}$-approximation algorithm [19] and a non-preemptive $\mathcal{SRPT}$-subsequence algorithm, we have the following corollary.

Corollary 6.3.1. *$\mathcal{BEST}$-α has an approximation ratio of* $\frac{e}{e-1} \approx 1.58$.

6.4 Summary

We generalized the idea of α-schedule and described the class of non-preemptive $\mathcal{SRPT}$-subsequence algorithms. We proved that no algorithm in this class including $\mathcal{BEST}$-α, has an approximation ratio better than $\frac{e}{e-1}$. This implies that the performance guarantee of $\mathcal{BEST}$-α proven by Chekuri *et al.*[19] is tight.

Chapter 7

AND/OR Linear Programs and Scheduling Problems

7.1 Introduction

In the literature, mathematical programming has been used to design approximation algorithms for scheduling problems [40, 73, 39, 34, 74, 69, 68]. In this chapter, we do the opposite; we use mathematical programming to identify hard input instances of a scheduling problem.

This chapter is organized as follows. In Section 7.2, we define the "lower bound problems". In Section 7.3, we give an introduction to linear and integer programming, the two simplest types of mathematical programming. In Section 7.3.1, we briefly mention previous applications of linear and integer programming for scheduling problems. In Section 7.4, we introduce AND/OR linear programming, a generalization of linear programming. In Section 7.4.1, we discuss the expressions of the underlying objective functions of algorithms for optimization problems. In Section 7.4.2, we describe how lower bound problems can be modeled as AND/OR linear programs. In Section 7.4.3, we describe some methods for solving AND/OR linear programs. In

Section 7.5, we describe an application of AND/OR linear programming for finding hard instances of a scheduling problem.

7.2 Lower Bound Problems

In this section we define "lower bound problems". Suppose we want to analyze the approximation ratio of an approximation algorithm $\mathcal{A}$ for an NP-hard minimization problem Π. Suppose $\mathcal{OPT}$ is an optimal algorithm for Π. Let $\mathcal{OPT}(I)$ and $\mathcal{A}(I)$ be the objective values of $\mathcal{OPT}$ and $\mathcal{A}$ for instance I in problem Π. We define the *lower bound problem* $LB(\Pi, \mathcal{A})$ as the problem of finding a lower bound of the approximation ratio $R_{\mathcal{A}}$ of $\mathcal{A}$ for problem Π. The optimal value of the lower bound problem $LB(\Pi, \mathcal{A})$ is the approximation ratio $R_{\mathcal{A}}$ of $\mathcal{A}$ itself. We overload the notation by using $LB(\Pi, \mathcal{A})$ to denote the optimal value of the lower bound. Thus,

$$LB(\Pi, \mathcal{A}) \;=\; R_{\mathcal{A}} \;=\; \sup_{I \in \Pi} \frac{\mathcal{A}(I)}{OPT(I)}.$$

We call Π the *underlying problem* of $LB(\Pi, \mathcal{A})$. We call $\mathcal{A}$ the *underlying approximation algorithm* of $LB(\Pi, \mathcal{A})$. We call $\mathcal{OPT}$ an *underlying optimal algorithm* of $LB(\Pi, \mathcal{A})$. We call $\mathcal{A}(I)$ and $\mathcal{OPT}(I)$ the *underlying approximation objective value* and the *underlying optimal objective value* of $LB(\Pi, \mathcal{A})$ for I.

We will be interested in underlying problems Π whose instances consist of only real-valued parameters and nothing else (no graphs, no precedence constraints, etc).

7.3 Linear and Integer Programming

A *linear programming problem* or a *linear program* is an algebraic formulation of an optimization problem which has the general form

$$\text{max or min } \; f(x_1, ..., x_n) \tag{7.1}$$

$$\text{subject to } \; g_i(x_1, ..., x_n) \begin{Bmatrix} \leq \\ = \\ \geq \end{Bmatrix} b_i, \qquad i = 1, ..., m \tag{7.2}$$

where x_i are scalar variables, b_i are scalar constants, and f and g_i are linear functions. When a linear program is used to model a decision problem, the variables $x_1, ..., x_n$ represent possible decision choices, the relations (7.2) represent the constraints, and the function f is the measure of effectiveness.

An assignment of values to the variables $x_1, ..., x_n$ is called a *solution* to the problem. The n-dimensional Euclidean space R^n is called the *solution space.* A solution is *feasible* if it satisfies all of the constraints. Otherwise, it is *infeasible.* The collection of all feasible solutions is called the *feasible region.*

The function f is called the *objective* or *objective function.* For a minimization (maximization) problem, a feasible solution is an *optimal solution* or an *optimum* if it yields a value of the objective function which is smaller (larger) than or equal to that of any other feasible solution. An optimal solution need not be unique, but the optimal value of the objective function must be.

An *integer programming problem* or an *integer program* is an algebraic formulation which has the same form as a linear program except that the values of the variables must be integers. Note that linear programs can be solved in polynomial time [72]. In contrast, the integer programming problem is NP-hard [72].

7.3.1 Previous Use of Linear and Integer Programs in Scheduling Problems

Linear and integer programming have been used in the design and analysis of polynomial-time scheduling algorithms [40, 73, 39, 34, 74, 69, 68]. Most of these works use the technique called "LP relaxation". The first step of this technique is to model a scheduling problem with an integer program. The integer program is then solved as if it were a linear program; that is, the integrality of the solution is being "relaxed". Note that a linear program can be solved in polynomial time. Then some polynomial time rounding scheme is applied to the solution to get an approximate integral solution. The approximate integral solution is then translated back to a feasible solution of the original scheduling problem. Since all computations involved are polynomial-time, this computational procedure is a polynomial-time approximation algorithm.

LP relaxation schemes use an algorithm for solving linear programs as a *subroutine* in an *approximation algorithm* for a scheduling problem. In contrast, we use mathematical programming as a *tool* to identify *hard instances* against an approximation algorithm for a scheduling problem. The mathematical program is constructed from the description of the scheduling problem, the optimal off-line algorithm, and the approximation algorithm being considered. However, ordinary linear programs are not powerful enough to represent all possible cases in a single program. Therefore, we introduce AND/OR linear programs which are a generalization of linear programs.

7.4 AND/OR Linear Programming

A solution of an ordinary linear program is feasible if the solution satisfies all the equalities and inequalities in (7.2). In other words, the constraints of an ordinary linear program is a conjunction of linear equalities and inequalities. To generalize the constraints of an ordinary linear program, we define the "AND/OR linear programming problem". In an *AND/OR linear programming problem* or an *AND/OR linear program*, the constraints can be any arbitrary boolean expression of linear equalities and inequalities. An AND/OR linear program has the general form

$$\text{max or min} \quad f(x_1, ..., x_n) \tag{7.3}$$

$$\text{subject to} \quad C(e_1, ..., e_m) \quad \text{where} \tag{7.4}$$

$$C \text{ is a boolean expression of } e_i\text{'s},$$

$$e_i \text{ is the relation } g_i(x_1, ..., x_n) \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} b_i, \qquad i = 1, ..., m,$$

x_i are scalar variables, b_i are scalar constants, and f and g_i are linear functions in variables x_i. The feasibility of a solution is generalized naturally from the case of ordinary linear programs; that is, a solution is feasible if it satisfies C.

7.4.1 Expressions of the Underlying Objective Functions

Suppose we are considering an algorithm $\mathcal{A}$ for a minimization problem Π. In this section, we will take a closer look at $\mathcal{A}(I)$, the objective value of $\mathcal{A}$ on input I. The expression of $\mathcal{A}(I)$ derived in this section will be used in the construction of an AND/OR linear program from a lower bound problem in Section 7.4.2. Note that the algorithm $\mathcal{A}$ considered could be any algorithm for Π that is, it could be an approximation algorithm for Π or an optimal algorithm for Π.

Without loss of generality, we assume that the only type of conditional branches in $\mathcal{A}$ are if-then-else statements. The *if-clauses* of the if-then-else statements are boolean expressions of relations of input parameters and/or temporary variables (which are derived from input parameters). An example of an if-clause is (($a > c$ and $b > c$) or $a + b < c$) where a, b, and c are input parameters. Each if-then-else statement executed will lead to a *branch result.* The *underlying combinatorial computation* of $\mathcal{A}$ for I is the sequence of branch results made during the execution of $\mathcal{A}$ on input I. Upon termination, $\mathcal{A}$ outputs its solution for I. We assume that $\mathcal{A}$ also outputs the objective value of the solution along with the solution itself. The objective value is a function of input parameters. We will call this function the *underlying combinatorial objective function* of $\mathcal{A}$ for I. In general, different underlying combinatorial computations will have different underlying combinatorial objective functions.

Note that two distinct input instances may have the same underlying combinatorial computation; that is the sequence of branch results made by $\mathcal{A}$ are the same for both instances. In this case, we say that the two input instance are *combinatorially equivalent* with respect to $\mathcal{A}$. They will also have the same underlying combinatorial objective function from which the objective values of their solutions are calculated.

For any positive integer n, let Π_n be the problem Π with an additional constraint that all input instances have size n. Let $c_1, c_2, ..., c_N$ be all possible underlying combinatorial computations of $\mathcal{A}$ for Π_n. Note that if $\mathcal{A}$ runs in polynomial time, N could be exponential in n. For $1 \leq i \leq N$, let $b_{i,1}, b_{i,2}, ..., b_{i,k_i}$ be the sequence of boolean expressions that must hold so that when $\mathcal{A}$ executes, c_i is the resulting computation. These boolean expressions are either the if-clauses or the negation of the if-clauses of the if-then-else statements encountered during the execution of $\mathcal{A}$. Note that if $\mathcal{A}$ has time complexity $\Theta(f(n))$, then $k_i = O(f(n))$. For $1 \leq i \leq N$, let o_i be the underlying combinatorial objective function corresponding to c_i. Note that o_i is a function of input parameters. For any input instance I in Π, we can represent $\mathcal{A}(I)$ as follows:

$$\begin{array}{rlll}
\text{If} & b_{1,1} \text{ and } b_{1,2} \text{ and } \ldots \text{ and } b_{1,k_1} \text{ holds,} & \text{then } \mathcal{A}(I) = o_1 \\
\text{else if} & b_{2,1} \text{ and } b_{2,2} \text{ and } \ldots \text{ and } b_{2,k_2} \text{ holds,} & \text{then } \mathcal{A}(I) = o_2 \\
& \cdots & \\
\text{else if} & b_{N,1} \text{ and } b_{N,2} \text{ and } \ldots \text{ and } b_{N,k_N} \text{ holds,} & \text{then } \mathcal{A}(I) = o_N.
\end{array}$$

Given an instance I of Π_n, I will fall into exactly one of the N cases. Some of the expressions $b_{i,j}$'s may contain inequalities "$<$" and "$>$". We will relax them by replacing all inequalities "$<$" and "$>$" by the corresponding inequalities "$\leq$" and "$\geq$". The replacement will cause some input instances of Π_n to fall into more than 1 case (underlying combinatorial computation). In Section 7.4.3, these input instances will be used as "bridges" to shift our attention from one underlying combinatorial computation to another.

7.4.2 Modeling Lower Bound Problems as AND/OR LPs

In this section, we explain how a lower bound problem $LB(\Pi, \mathcal{A})$ can be modeled as AND/OR linear programs. We make the following assumptions.

- An input instance of the underlying problem consists only of real-valued parameters.

- All underlying combinatorial objective functions of $\mathcal{A}$ and $\mathcal{OPT}$ are *linear* functions of the parameters of the input instance.

- All boolean expressions $b_{i,j}$'s supporting any underlying combinatorial computation c_i of $\mathcal{A}$ and $\mathcal{OPT}$ are composed of *linear* inequalities of the parameters of the input instance.

The second assumption implies that $\mathcal{A}(cI) = c\mathcal{A}(I)$ where c is a positive real constant and cI denotes the instance created from I by multiplying all parameters of I by c.

Next we show how to transform a lower bound problem $LB(\Pi, \mathcal{A})$ into an AND/OR linear programming problem. By definition, the value of $LB(\Pi, \mathcal{A})$ is defined as the following mathematical program $MP1$.

$$\max_I \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)}.$$

The solution space of $MP1$ is the set of all input instances I of Π. A feasible solution of $MP1$ is a lower bound instance for $\mathcal{A}$. The optimal solution of $MP1$ is the best lower bound instance against $\mathcal{A}$. The optimal value of the objective of $MP1$ is also the approximation ratio of $\mathcal{A}$.

We will show that $MP1$ can be transformed into a series of AND/OR linear programs. Since $\mathcal{A}(cI) = c\mathcal{A}(I)$ and $\mathcal{OPT}(cI) = c\mathcal{OPT}(I)$ for any instance I and positive number c, then it is sufficient to consider those instances I such that $\mathcal{OPT}(I) = 1$. Furthermore, if $\mathcal{OPT}(I) = 1$, then $\mathcal{A}(I)/\mathcal{OPT}(I) = \mathcal{A}(I)$. Thus,

$MP1$ is equivalent to the following program $MP2$.

$$\max_{I} \ \mathcal{A}(I) \qquad \text{subject to}$$

$$\mathcal{OPT}(I) = 1.$$

Since I can have any number of jobs, the number of variables of this mathematical program is unbounded. We can divide this mathematical program into subprograms according to the number of jobs in I. Thus, $MP2$ is equivalent to the following program $MP3$.

$$\max_{n \geq 1} \ MP3(n) \qquad \text{where } MP3(n) \text{ is defined as}$$

$$\max_{I:\,|I|=n} \ \mathcal{A}(I) \qquad \text{subject to}$$

$$\mathcal{OPT}(I) = 1.$$

Until now we have been omitting the clauses "$\mathcal{OPT}(I)$ is the optimal objective value for I" and "$\mathcal{A}(I)$ is the objective value of the solution produced by $\mathcal{A}$ for I". The values of $\mathcal{OPT}(I)$ and $\mathcal{A}(I)$ as functions of I can be derived as explained in Section 7.4.1. Let $b_{i,j}$ and o_i be subexpressions of $\mathcal{A}(I)$ as explained in Section 7.4.1. Let $b'_{i,j}$ and o'_i be corresponding subexpressions of $\mathcal{OPT}(I)$. By plugging the expressions of $\mathcal{OPT}(I)$ and $\mathcal{A}(I)$ into $MP3(n)$, we obtain the following mathematical program $MP4(n)$.

$$\max_{I:\,|I|=n} \quad A \qquad \text{subject to}$$

$$O = 1$$

and

$$\Big(\quad (O = o'_1 \text{ and } b'_{1,1} \text{ and } b'_{1,2} \text{ and } \ldots \text{ and } b'_{1,k'_1}) \text{ or}$$

$$(O = o'_2 \text{ and } b'_{2,1} \text{ and } b'_{2,2} \text{ and } \ldots \text{ and } b'_{2,k'_2}) \text{ or}$$

$$\ldots$$

$$(O = o'_{N'} \text{ and } b'_{N',1} \text{ and } b'_{N',2} \text{ and } \ldots \text{ and } b'_{N',k'_{N'}}) \quad \Big)$$

and

$$\Big(\quad (A = o_1 \text{ and } b_{1,1} \text{ and } b_{1,2} \text{ and } \ldots \text{ and } b_{1,k_1}) \text{ or}$$

$$(A = o_2 \text{ and } b_{2,1} \text{ and } b_{2,2} \text{ and } \ldots \text{ and } b_{2,k_2}) \text{ or}$$

$$\ldots$$

$$(A = o_N \text{ and } b_{N,1} \text{ and } b_{N,2} \text{ and } \ldots \text{ and } b_{N,k_N}) \quad \Big)$$

For $n \geq 1$, $MP4(n)$ is an AND/OR linear program. The optimal solution of $MP4(n)$ is the best lower bound instance with size n against $\mathcal{A}$. The approximation ratio of $\mathcal{A}$ is the maximum of $MP4(n)$ for all $n \geq 1$.

$$\max_{n \geq 1} \; MP4(n)$$

7.4.3 Solving an AND/OR Linear Program

Consider an AND/OR linear program with C as its constraint. We can assume that C is in disjunctive normal form (OR's of AND's), *i.e.*,

$$\begin{array}{ll} C = C_1 \vee C_2 \vee ... \vee C_h & \text{where} \\ C_i = d_{i,1} \wedge d_{i,2} \wedge ... \wedge d_{i,l_i} & \text{for } i = 1, ..., h, \text{ and} \\ d_{i,j} \in \{e_1, ..., e_m\} & \text{for } i = 1, ..., h \text{ and } j = 1, ..., l_i. \end{array}$$

Then each clause C_i together with the objective function of the AND/OR linear program defines an ordinary linear program. In theory, we can optimize an AND/OR linear program by simply optimizing the ordinary linear subprogram corresponding to each C_i and choosing the best solution.

However, this method potentially has exponential running time. This problem stems from two reasons. Firstly, there could be exponentially many linear subprograms to optimize. As shown in Section 7.4.1, there could be exponentially many clauses in the expressions of $\mathcal{A}(I)$ and $\mathcal{OPT}(I)$ in disjunctive normal form. Secondly, each clause C_i corresponding to a linear program could have exponential size. That is because the underlying problem Π is NP-hard. There is no known optimal algorithm for Π which runs in polynomial time (if it does indeed exist). Since the known optimal algorithm runs in exponential time, as shown in Section 7.4.1, each clause could have exponential size.

To evade the second problem, we could use a super-optimal algorithm $\mathcal{SOPT}$ which runs in polynomial time instead of $\mathcal{OPT}$. A super-optimal algorithm is an algorithm that always produces a solution whose objective value is better than or equal to the optimal solution with a relaxation that the solution produced may or may not be feasible. Since our goal is to find a lower bound for $\mathcal{A}$, the value of the lower bound found using $\mathcal{SOPT}$ is a valid lower bound for $\mathcal{A}$. However, the value of the lower bound found using $\mathcal{SOPT}$ may not be as tight as $\mathcal{OPT}$. For example, the problem of nonpreemptively scheduling jobs on one machine to minimize the total completion time ($1|r_j|\sum C_j$) is NP-hard. There is no known optimal algorithm for this problem which runs in polynomial time. A solution with preemptions will not be a feasible solution for this problem. However, preemptive solutions are super-optimal and can be constructed in polynomial-time.

To evade the first problem of exponential running time, we will use a local search scheme instead of the global one. This scheme does not guarantee to find a *global* optimum, but it guarantees to find a *local* optimum. Before we describe the scheme, we point out some observations. Suppose an AND/OR linear program $MP4(n)$ derived in Section 7.4.1 is given. Then the following tasks can be performed.

- Given an instance I of the underlying problem, we can determine all underlying computations of $\mathcal{OPT}$ (or $\mathcal{SOPT}$) and $\mathcal{A}$ for I. Note that multiple computations are the results of relaxing inequalities "$<$" and "$>$" to "$\leq$" and "$\geq$". Equivalently, in term of AND/OR linear program, given a solution of the AND/OR linear program, we can find all ordinary linear subprograms whose feasible regions contain the given solution.

- We can easily find some initial ordinary linear subprogram of the AND/OR linear program. This can be done by choosing a random input instance of the underlying problem and applying the previous observation.

The general procedure for finding a local optimum of an AND/OR linear program can be described as follows:

1. Choose an initial linear subprogram.
2. Compute an optimal solution of the linear subprogram.
3. Choose a new linear subprogram whose feasible region contains the solution from in step 2.
4. Repeat steps 2 and 3 until all linear subprograms whose feasible regions contain the current solution have been considered and the solution cannot be improved.

This procedure does not need a complete boolean expression for the constraints of the AND/OR linear program up front. We only need to construct the required linear subprograms as we proceed.

The procedure will eventually terminate because for a fixed AND/OR linear program, there are only finitely many linear subprograms to consider. We do not know the theoretical upper bound of the number of iterations of this procedure other than the obvious one, the number of all linear subprograms. However, in our examples in the next section, we need only a few iterations before the procedure terminates.

Next, we talk about the optimality of the solution found when the procedure terminates. After we obtain a solution in step 2, we find a new linear subprogram in step 3. The key observation is that the feasible region of the new linear subprogram not only contains the solution in step 2, it also potentially contains a better solution.

Thus, by solving the new linear subprogram, we guarantee to find a solution that is as good as or better than the previous solution. By repeating steps 2 and 3, we will increasingly find better and better solutions for the AND/OR linear program. The procedure will eventually find a local optimum because otherwise it would not terminate. However, the procedure cannot guarantee to find a global optimum. The final solution found by the procedure depends on the initial solution chosen in step 1 and the selection of new linear subprogram in each iteration in step 3. Note that if the solution space has only one local optimum, it is also the global optimum.

Although this procedure can be fully automated, a partially manual execution is more productive. It is too time-consuming to implement some of the steps in the procedure. Moreover, they are probably executed only a few times. Specifically, choosing an initial linear subprogram in step 1 and choosing a new linear subprogram in step 3 should be done manually. From our example in the next section, we only need to execute the procedure for a few iterations. The task of constructing linear subprograms from the sequence of execution of the optimal algorithm and the approximation algorithm could be automated or manually done depending on the problem. In our example, this task is automated. Computing an optimal solution of a linear subprogram should be automated because there are software libraries for this task and they are easily obtained. In addition, viewing the solutions of linear subprograms in some appropriate graphical form together with their numerical values can help us gains intuition about the problem faster and better than reading the numerical values alone. Thus, developing a program for showing a graphical representation of the solution should be considered.

7.5 Applying AND/OR LP to a Scheduling Problem

In this section, we illustrate our technique using the problem of nonpreemptively scheduling jobs which arrive over time on one machine to minimize the total completion time ($1|r_j|\sum C_j$). This problem has been considered in Chapter 6. A class of algorithms called the non-preemptive $\mathcal{SRPT}$-subsequence algorithms was considered. In Chapter 6, we proved that no algorithm in this class has an approximation ratio better than $e/(e-1) \approx 1.58$ against a class of lower bound instances. In this section, we will show how we came up with these input instances using the technique of transforming a lower bound problem into an AND/OR linear programming problem.

7.5.1 Program Formulation

Let $\mathcal{SS}$ be the class of all non-preemptive $\mathcal{SRPT}$-subsequence algorithms defined in Chapter 6 with an additional property that they do not insert unnecessary idle time in their schedules. Let $\mathcal{SS}(I)$ be the set of all non-preemptive $\mathcal{SRPT}$-subsequence schedules with no unnecessary idle time. We want to find an instance I such that $\mathcal{A}(I)/\mathcal{OPT}(I)$ is as large as possible for all $\mathcal{A}$ in $\mathcal{SS}$. This can be represented as the following mathematical program:

$$\max_{I} \min_{\mathcal{A}\in\mathcal{SS}} \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)}.$$

Given an instance I, any non-preemptive $\mathcal{SRPT}$-subsequence algorithm will produce an $\mathcal{SRPT}$-subsequence schedule for I. Thus, given an instance I, to consider all non-preemptive $\mathcal{SRPT}$-subsequence "algorithms", we only need to consider all non-preemptive $\mathcal{SRPT}$-subsequence "schedules". Therefore, our AND/OR linear program becomes

$$\max_{I} \min_{S \in SS(I)} \frac{S}{\mathcal{OPT}(I)}.$$

We can eliminate the "min" by replacing S with a new variable R, and adding a new constraint asserting that R is no larger than the minimum S in $SS(I)$. In other words, R is no larger than any S in $SS(I)$. Thus, our AND/OR linear program becomes

$$\begin{array}{ll} \max\limits_{I} \dfrac{R}{\mathcal{OPT}(I)} & \text{subject to} \\ R \le S & \forall S \in \mathcal{SS}(I). \end{array}$$

By using $\mathcal{SRPT}$ as a super-optimal algorithm and by following the transformation in Section 7.4.2, we obtain the following mathematical program.

$$\max_{n \ge 1} MP(n) \qquad \text{where } MP(n) \text{ is defined as}$$

$$\max_{I:\,|I|=n} R \qquad \text{subject to}$$

$$\mathcal{SRPT}(I) = 1 \tag{7.5}$$

$$R \le S \qquad \forall S \in \mathcal{SS}(I). \tag{7.6}$$

In the next section, we will derive the expression of $\mathcal{SRPT}(I)$ and all schedules S in $\mathcal{SS}(I)$. Note that an input instance I with n jobs of the underlying problem $1|r_j|\sum C_j$ consists of the release time r_j and the processing time p_j of each job j

where $j = 1, ..., n$. Thus, instances of the underlying problem consist only of real-valued parameters.

7.5.2 Expressions of the Objective Functions

In this section, we will derive the expression of $\mathcal{SRPT}(I)$ and all schedule S in $\mathcal{SS}(I)$. For $\mathcal{SRPT}$(I), we need to derive the condition when an underlying combinatorial computation of $\mathcal{SRPT}$ prevails over other computations, and the expression of the corresponding underlying combinatorial objective function. For each S in $SS(I)$, we only need to derive its expression. A schedule prevails when it has a minimum total completion time among all schedules in $\mathcal{SS}(I)$. This condition has already been enforced in $MP(n)$.

First, we give some intuition about the structure of the $\mathcal{SRPT}$ schedule. Given an instance I, we can assume that $r_i = s_i^{\mathcal{SRPT}}$ for all i where $s_i^{\mathcal{SRPT}}$ is the starting time of job i in the $\mathcal{SRPT}$ schedule. This is because $\mathcal{SRPT}$ cannot take advantage of having r_i smaller than $s_i^{\mathcal{SRPT}}$; that is, $\mathcal{SRPT}$ cannot produce a schedule with a smaller total completion time even though r_i is smaller than $s_i^{\mathcal{SRPT}}$. In contrast, having smaller r_i might allow non-preemptive $\mathcal{SRPT}$-subsequence algorithms to produce a schedule with smaller total completion time.

We next consider how jobs overlap in the $\mathcal{SRPT}$ schedule. Let v_i be the interval $[s_i^{\mathcal{SRPT}}, C_i^{\mathcal{SRPT}}]$. For any pair of jobs i and j, it is the case that either v_i and v_j are disjoint or one of them includes the other. To show this, suppose there exist two intervals v_i and v_j that overlap, *i.e.*, $s_i^{\mathcal{SRPT}} < s_j^{\mathcal{SRPT}} < C_i^{\mathcal{SRPT}} < C_j^{\mathcal{SRPT}}$. Since $s_i^{\mathcal{SRPT}} < s_j^{\mathcal{SRPT}} < C_i^{\mathcal{SRPT}}$, then job j is run while job i is ready; that is, job

j has a higher priority than job i. However, job j completes after job i because $C_i^{\mathcal{SRPT}} < C_j^{\mathcal{SRPT}}$. This is a contradiction.

The decisions $\mathcal{SRPT}$ has to make during its computation are (1) whether to preempt the running job when a new job arrives and (2) which job to run when a job is completed. The underlying combinatorial computation of $\mathcal{SRPT}$ is the sequence of starts and stops of jobs on the machine. The sequence of these events can be determined from, and, in fact, is equivalent to the sequence of job pieces in the $\mathcal{SRPT}$ schedule. They are also equivalent to the "order of inclusion and succession" among v_i in the $\mathcal{SRPT}$ schedule.

To help visualize the order of inclusion and succession, we introduce the "interval inclusion graph". Roughly, the interval inclusion graph $G(I)$ of an instance I can be constructed from the $\mathcal{SRPT}$ schedule of I as follows. First, replace each job piece in the $\mathcal{SRPT}$ schedule by a node. Make sure that the nodes still line up on a straight line. Second, for each pair of successive nodes of the same jobs, add an edge connecting them. The edge drawn should look like the upper half of a circle. Note that $G(I)$ is not exactly a graph because it retains information about the ordering of job pieces in the $\mathcal{SRPT}$ schedule.

In what follows, given an $\mathcal{SRPT}$ schedule of a fixed instance I with n jobs, we inductively construct (1) the interval inclusion graph $G(I)$ and (2) the constraints of the linear subprogram $LP(I, n)$ of the AND/OR linear program $MP(n)$.

During the construction of $G(I)$, we will maintain an ordered list of "clusters" which are non-empty sets of jobs. Their definitions will be given later. One of the jobs in each cluster will be referred to as the dominating job. Suppose U_j is a cluster with job i as its dominating job. The interval v_i will include the interval of all jobs in U_j. Each cluster U_j will have the following parameters:

- $r(U_j)$ is the release time of cluster U_j and is defined as $r(U_j) = r_i$.
- $p(U_j)$ is the processing time of the dominating job in cluster U_j, *i.e.*, $p(U_j) = p_i$.
- $q(U_j)$ is the processing time of all jobs in cluster U_j, *i.e.*, $q(U_j) = \sum_{l \in U_j} p_l$.
- $C(U_j)$ is the completion time of cluster U_j and is defined as $C(U_j) = C_i$.

Relabel jobs so that $C_1^{\mathcal{SRPT}} \leq C_2^{\mathcal{SRPT}} \leq ... \leq C_n^{\mathcal{SRPT}}$. For the convenience of the discussion, we assume that there is a dummy job 0 with $r_0 = 0$ and $p_0 = 0$. Initially, cluster U_0 is the only cluster in $G(I)$, and job 0 is its only member. Thus,

$$r(U_0) = r_0 = 0$$

$$q(U_0) = p(U_0) = p_0 = 0 \qquad \text{and}$$

$$C(U_0) = r(U_0) + q(U_0) = 0.$$

Suppose jobs $1, ..., l$ are already added into $G(I)$ where $0 \leq l \leq n-1$. Suppose further that there are k clusters in $G(I)$ at this time. We add job $l+1$ to $G(I)$ as follows. Relabel clusters in $G(I)$ so that $C(U_i) \leq r(U_{i+1})$ for $i = 1, ..., k-1$. Let i_1 be the maximum index such that $C(U_{i_1}) \leq r_{l+1}$. Note that $C(U_k) \leq C_{l+1}^{\mathcal{SRPT}}$ since jobs are ordered by non-decreasing $C_i^{\mathcal{SRPT}}$. Job $l+1$ has $k - i_1 + 1$ pieces in the $\mathcal{SRPT}$ schedule, one piece just after cluster j for $j = i_1, ..., k$. Let $p_{l+1}^{(j)}$ denote the length of each piece, $j = i_1, ..., k$. Some of the pieces possibly have zero length. The total length of job $l+1$ is p_{l+1} which is defined as the sum of the length of all of its pieces.

In $G(I)$, add a node $p_{l+1}^{(j)}$ to the right of cluster U_j for $j = i_1, ..., k$. These nodes represent the pieces of job $l+1$ in the $\mathcal{SRPT}$ schedule. Add an edge between node $p_{l+1}^{(j)}$ and $p_{l+1}^{(j+1)}$ for $j = i_1, ..., k-1$. Replace clusters $U_{i_1}, ..., U_k$ in the list of clusters by cluster U_{l+1} where the members of U_{l+1} consist of job $l+1$ and jobs in clusters $U_{i_1}, ..., U_k$. Let job $l+1$ be the dominating job of U_{l+1}.

The following constraints are added into $LP(I,n)$.

$$r(U_{l+1}) = r_{l+1} \tag{7.7}$$

$$p(U_{l+1}) = p_{l+1} = p_{l+1}^{(i_1)} + ... + p_{l+1}^{(k)} \tag{7.8}$$

$$q(U_{l+1}) = \Big(q(U_{i_1+1}) + ... + q(U_k)\Big) + p_{l+1} \tag{7.9}$$

$$C(U_{l+1}) = C_{l+1}^{\mathcal{SRPT}} = r(U_{l+1}) + q(U_{l+1}) \tag{7.10}$$

$$r(U_{l+1}) \geq C(U_{i_1}) \tag{7.11}$$

$$r(U_{l+1}) \leq r(U_{i_1+1}) \qquad \text{if } i_1 + 1 \leq k \tag{7.12}$$

$$p_{l+1}^{(j)} \geq 0 \qquad \text{for } j = i_1, ..., k \tag{7.13}$$

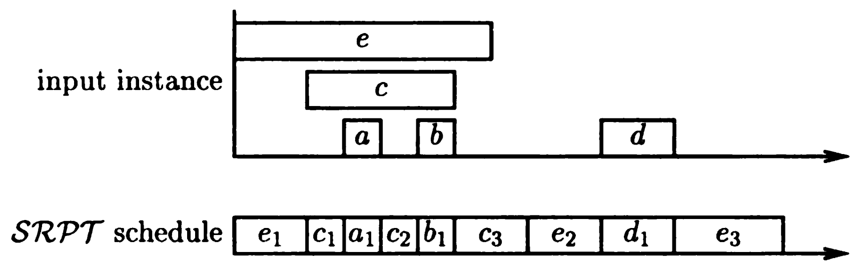
$$p(U_j) \leq p_{l+1}^{(j)} + ... + p_{l+1}^{(k)} \qquad \text{for } j = i_1 + 1, ..., k \tag{7.14}$$

Equalities (7.7) to (7.10) simply equate the parameters of cluster U_{l+1} with their values according to their definitions. Equalities (7.11) and (7.12) enforce the fact that i_1 is the maximum index such that $C(U_{i_1}) \leq r_{l+1}$. Equalities (7.13) enforces that all pieces of job $l+1$ must have non-negative size. For $j = i_1 + 1, ..., k$, the first job of cluster U_j preempts job $l+1$. Inequality (7.14) ensures that the $\mathcal{SRPT}$ rule is being followed; that is, when the first job of cluster U_j arrives and preempts job $l+1$, the remaining processing time of job $l+1$ is larger than or equal to that of the first job of cluster U_j.

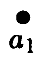
Figure 7.1 illustrates the construction of the interval inclusion graph and the constraints of the corresponding linear subprogram. The figure includes (1) an input instance, (2) the $\mathcal{SRPT}$ schedule of the instance, (3) the total completion time of the $\mathcal{SRPT}$ schedule, (4) the interval v_i of each job i in the $\mathcal{SRPT}$ schedule, (5) each step of the construction of the interval inclusion graph, and (6) constraints of the corresponding linear subprogram created in each step. There are 5 jobs, a, b, c, d, and e. In the $\mathcal{SRPT}$ schedule, $C_a \leq C_b \leq C_c \leq C_d \leq C_e$. Interval v_b succeeds interval v_a. Interval v_c includes intervals v_a and v_b. Interval v_d succeeds interval v_c. Interval v_e includes intervals v_c and v_d. In this figure, the length of the i'th piece of job a is represented as a_i. Furthermore, the clusters' parameters in the constraints are replaced by their values (as functions of jobs parameters). The total completion time of the $\mathcal{SRPT}$ schedule is $C_a + C_b + C_c + C_d + C_e$.

Next, we will determine the expression of all S in $\mathcal{SS}(I)$. Let's continue with the same example in Figure 7.1. To construct a non-preemptive $\mathcal{SRPT}$-subsequence schedule, there are 3 choices to place job e and 3 choices to place job b. Thus, there are $3 \times 3 = 9$ non-preemptive $\mathcal{SRPT}$-subsequence schedules for I. They are listed in Figure 7.2. The formula for the total completion time of each schedule is given to the left of each schedule.

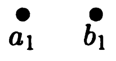
We have described the construction of the interval inclusion graph and the linear subprogram from the order of inclusion and succession of job intervals. Note



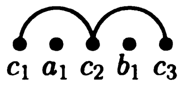
$$3r_a+3p_a+3r_b+3p_b+2r_c+2p_c+2r_d+2p_d+r_e+p_e$$



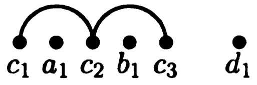
$$\begin{array}{ll} p_a = a_1 & 0 \leq r_a \\ a_1 \geq 0 & C_a = r_a + p_a \end{array}$$



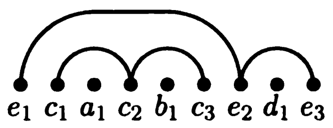
$$\begin{array}{ll} p_b = b_1 & C_a \leq r_b \\ b_1 \geq 0 & C_b = r_b + p_b \end{array}$$



$$\begin{array}{ll} p_c = c_1 + c_2 + c_3 & 0 \leq r_c \leq r_a \\ c_1 \geq 0 & C_c = r_c + p_a + p_b + p_c \\ c_2 \geq 0 & p_b \leq c_3 \\ c_3 \geq 0 & p_a \leq c_2 + c_3 \end{array}$$



$$\begin{array}{ll} p_d = d_1 & C_c \leq r_d \\ d_1 \geq 0 & C_d = r_d + p_d \end{array}$$



$$\begin{array}{ll} p_e = e_1 + e_2 + e_3 & 0 \leq r_e \leq r_c \\ e_1 \geq 0 & C_e = r_e + p_a + p_b + p_c + p_d + p_e \\ e_2 \geq 0 & p_d \leq e_3 \\ e_3 \geq 0 & p_c \leq e_2 + e_3 \end{array}$$

Figure 7.1: Construction of the interval inclusion graph and the linear subprogram

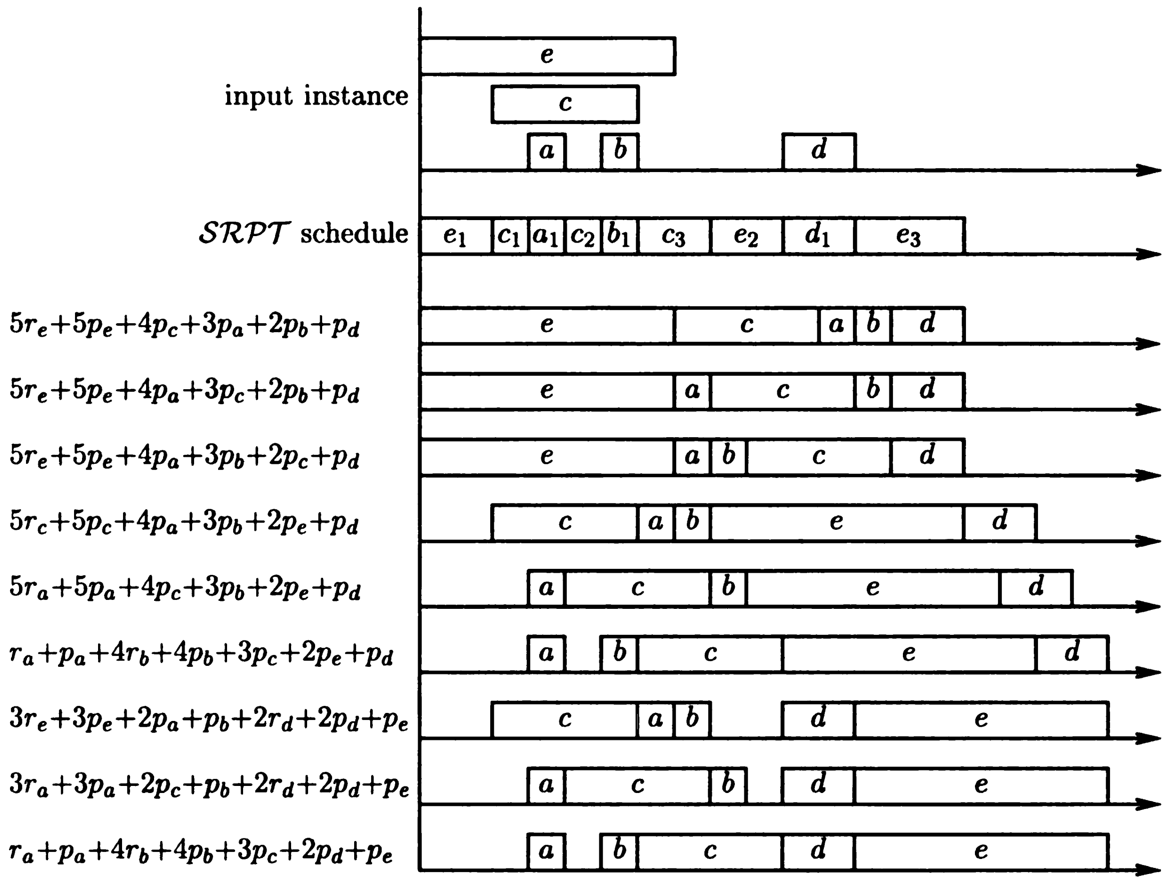

input instance
e
c
a
b
d
$\mathcal{SRPT}$ schedule
e_1 c_1 a_1 c_2 b_1 c_3 e_2 d_1 e_3
$5r_e+5p_e+4p_c+3p_a+2p_b+p_d$
$5r_e+5p_e+4p_a+3p_c+2p_b+p_d$
$5r_e+5p_e+4p_a+3p_b+2p_c+p_d$
$5r_c+5p_c+4p_a+3p_b+2p_e+p_d$
$5r_a+5p_a+4p_c+3p_b+2p_e+p_d$
$r_a+p_a+4r_b+4p_b+3p_c+2p_e+p_d$
$3r_e+3p_e+2p_a+p_b+2r_d+2p_d+p_e$
$3r_a+3p_a+2p_c+p_b+2r_d+2p_d+p_e$
$r_a+p_a+4r_b+4p_b+3p_c+2p_d+p_e$

Figure 7.2: Non-preemptive $\mathcal{SRPT}$-subsequence schedules.

that all three things are equivalent. So, for the rest of this section, we will treat the interval inclusion graph as the specification of the linear subprogram.

7.5.3 Optimizing AND/OR Linear Programs

Now, let us apply the procedure described in the previous section to identify good lower bound input instances for the scheduling problem we are considering. We will especially focus on step 3 of the procedure.

Example 1

Figure 7.3 illustrates the execution of the procedure. There are 6 graphs in the figure. The first graph is the initial interval inclusion graph which is the same example used in Figures 7.1 and 7.2 After optimizing the linear subprogram corresponding to the first graph, the solution is shown in the second graph. In the second graph, a job piece with size 0 is represented as a thick "dot", and a job piece with non-zero size is represented as a thick "line". The length of each thick line is proportional to the size of the corresponding job piece.

Observe that piece a_1, the first piece of job a, has zero length. Removing a_1 from the schedule essentially does not change the total completion time of the $\mathcal{SRPT}$ schedule. The third graph is the new interval inclusion graph obtained. In the third graph, the release time of job a is between the completion time of job piece b_3 and job c. Furthermore, there are now only 6 possible non-preemptive $\mathcal{SRPT}$-subsequence schedules. Note that other job pieces with zero length cannot be removed from the second graph because removing them will change the total completion time of the $\mathcal{SRPT}$ schedule as well as those of non-preemptive $\mathcal{SRPT}$-subsequence schedules.

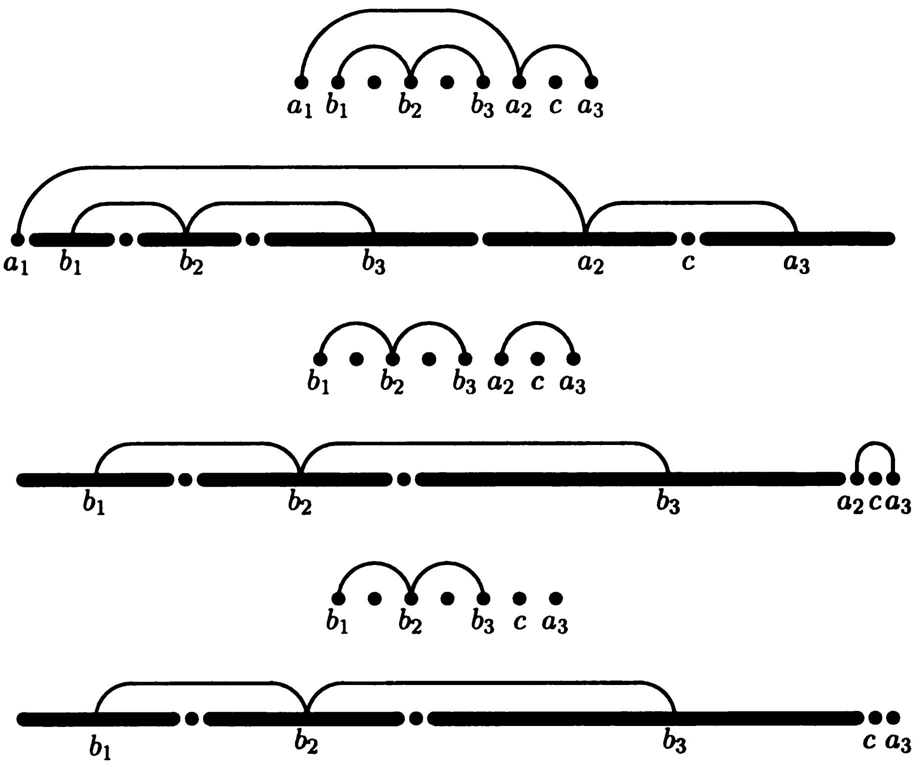
The optimal solution of the linear subprogram corresponding to the third graph is shown in the fourth graph. Similarly to the previous step, piece a_2 has zero length and can be removed. The fifth graph is the new interval inclusion graph obtained. The optimal solution of the linear subprogram corresponding to the fifth graph is shown in the sixth graph. At this point, we cannot modify the interval inclusion graph anymore. Thus, we have found a local optimum. This instance give us a lower bound of 1.370.

Example 2

Let us look at another example in Figure 7.4. The first graph in the figure is the initial interval inclusion graph. The second graph is the solution obtained from optimizing the linear subprogram corresponding to the first graph. Observe that all job pieces in front of piece b_1 have zero size. Therefore, all of them can be removed without affecting the total completion time of the $\mathcal{SRPT}$ schedule. Job pieces c_1, c_2, c_3, and c_4 can also be removed. Other job pieces with zero size cannot be removed because otherwise the total completion time of the $\mathcal{SRPT}$ schedule and the non-preemptive $\mathcal{SRPT}$-subsequence schedules will be affected. The third graph is the new interval inclusion graph obtained. The optimal solution of the corresponding linear subprogram of the third graph is shown in the fourth graph. We cannot modify the interval inclusion graph anymore. Thus, a local optimum is reached. This instance gives us a lower bound of 1.403.

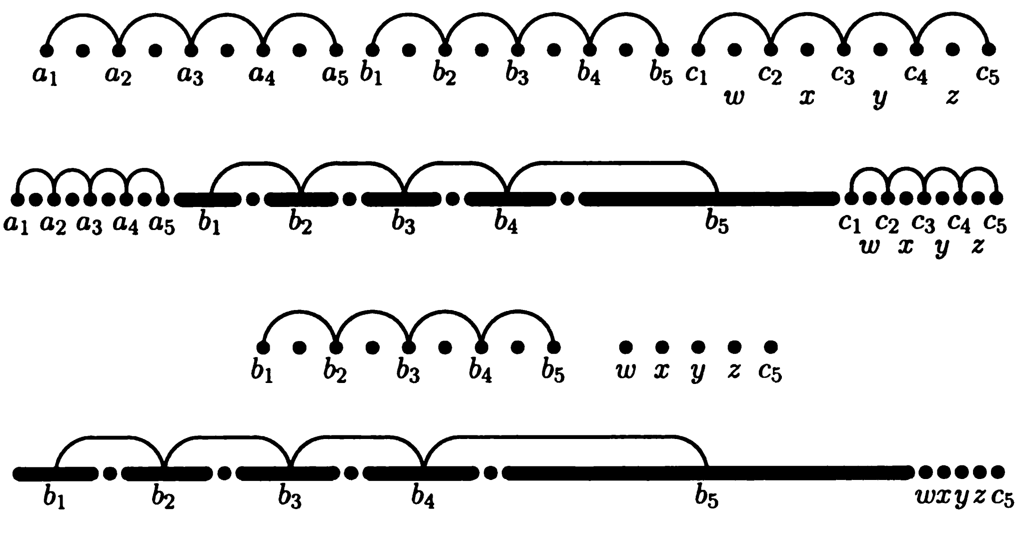
Example 3

Figure 7.5 is our third and final example. The first graph is the initial inter-



a_1 b_1 b_2 b_3 a_2 c a_3
a_1 b_1 b_2 b_3 a_2 c a_3
b_1 b_2 b_3 a_2 c a_3
b_1 b_2 b_3 $a_2 c a_3$
b_1 b_2 b_3 c a_3
b_1 b_2 b_3 c a_3

Figure 7.3: The first example of optimizing an AND/OR linear program.



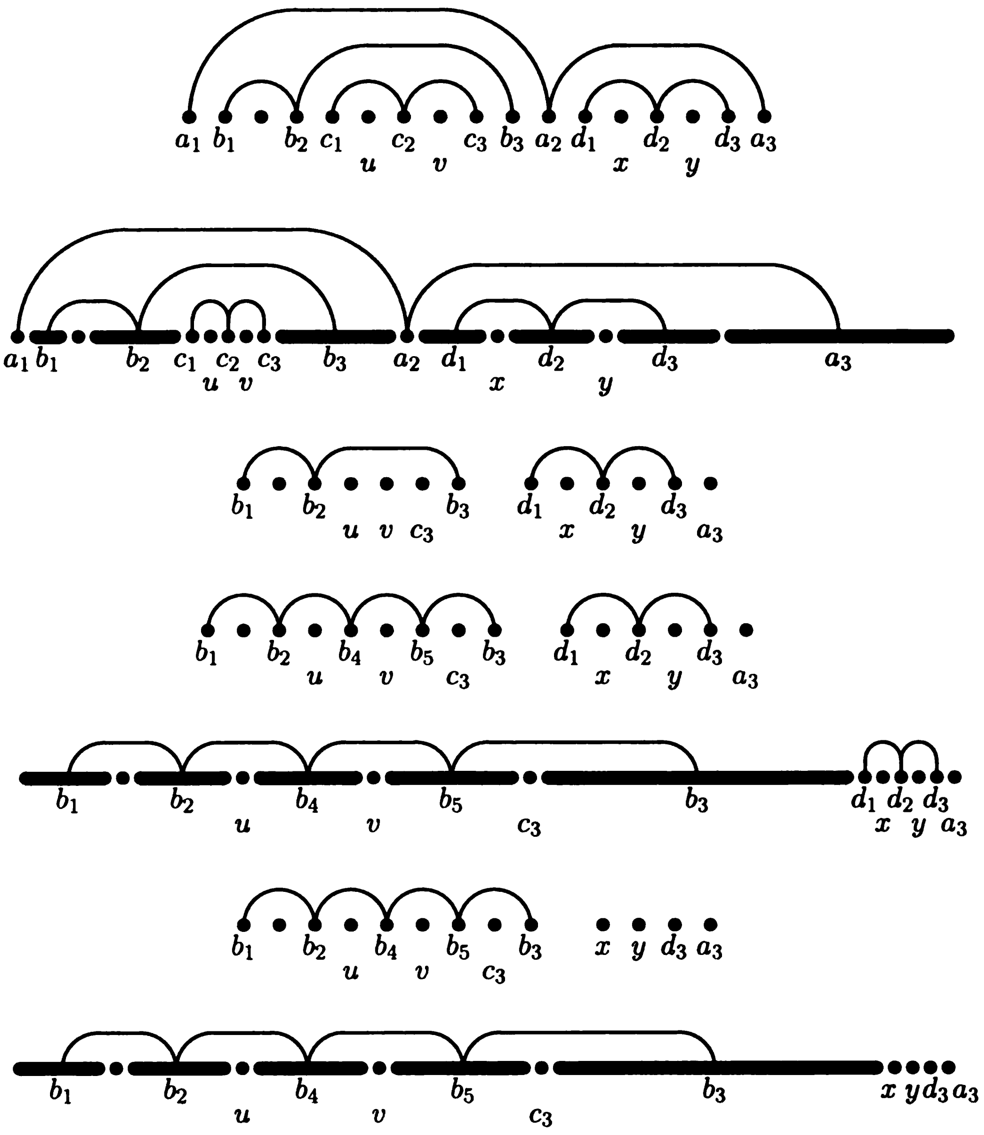
a_1 a_2 a_3 a_4 a_5 b_1 b_2 b_3 b_4 b_5 c_1 c_2 c_3 c_4 c_5
w x y z
a_1 a_2 a_3 a_4 a_5 b_1 b_2 b_3 b_4 b_5 c_1 c_2 c_3 c_4 c_5
w x y z
b_1 b_2 b_3 b_4 b_5 w x y z c_5
b_1 b_2 b_3 b_4 b_5 $wxyzc_5$

Figure 7.4: The second example of optimizing an AND/OR linear program.

val inclusion graph. We progress similarly. The second graph is the solution from optimizing the first graph. However, there is something different from before. The third graph is obtained by removing job pieces a_1, a_2, c_1, and c_2. In this graph, among others, job b includes jobs u, v, and c. However, there are no pieces of job b between u and v, and v and c. This does not follow our construction of interval inclusion graph. The fourth graph is obtained by adding 2 pieces of job b between jobs u, v, and c. There is a side effect of adding these pieces; there will be more non-preemptive $\mathcal{SRPT}$-subsequence schedules which are potentially better than the existing ones. However, since our goal is to gain intuition, not to prove anything, then this modification is acceptable. Moreover, for this particular instance, it turns out that new non-preemptive $\mathcal{SRPT}$-subsequence schedules associated with adding two new pieces for job b do not have a better total completion time than the existing ones. Then we proceed as before until a local optimum is reached.

Structure of the Lower Bound Instance Obtained

By inspecting the final solutions in all 3 examples, the structure of (local) optimal lower bound instances is now evident. There is one long job released at time 0. There are a number of zero length jobs. Some are released so that they preempt the long job. Some are released when the long job has just finished. With a closer look at the optimal solution obtained and with some more analysis, we found the lower bound instances as defined in Chapter 6, and that concludes this section.



a_1 b_1 b_2 c_1 c_2 c_3 b_3 a_2 d_1 d_2 d_3 a_3
u v x y
a_1 b_1 b_2 c_1 c_2 c_3 b_3 a_2 d_1 d_2 d_3 a_3
u v x y
b_1 b_2 b_3 d_1 d_2 d_3
u v c_3 x y a_3
b_1 b_2 b_4 b_5 b_3 d_1 d_2 d_3
u v c_3 x y a_3
b_1 b_2 b_4 b_5 b_3 d_1 d_2 d_3
u v c_3 x y a_3
b_1 b_2 b_4 b_5 b_3 x y d_3 a_3
u v c_3
b_1 b_2 b_4 b_5 b_3 x y d_3 a_3
u v c_3

Figure 7.5: The third example of optimizing an AND/OR linear program.

7.6 Discussion

In the literature, linear programming algorithm has been used as subroutines in polynomial-time approximation algorithms. In contrast we use AND/OR linear programs to find hard instances of a scheduling problem. An interesting question is how well our technique can be applied to other problems. Another interesting question is whether we can apply this idea to help design and analyze approximation algorithms.

APPENDIX

Appendix A

Bit Summation Inequalities

Definition A.1. *For $n \geq 1$, let $B(n)$ be the set of non-negative integers such that $\sum_{b \in B(n)} 2^b = n$.*

Lemma A.1. *For $n \geq 1$, $\sum_{b \in B(n)} 2^b(b' - b) < n$ where $b' = \max B(n)$.*

Proof. Let $B_n = |B(n)|$, and let $b_1 < b_2 < \ldots < b_{B_n}$ be the elements of $B(n)$.

$$\begin{aligned}
\sum_{b \in B(n)} 2^b(b' - b) &= \sum_{1 \leq i \leq B_n} 2^{b_i}(b_{B_n} - b_i) \\
&= \sum_{1 \leq i \leq B_n - 1} 2^{b_i}(b_{B_n} - b_i) \\
&= \sum_{1 \leq i \leq B_n - 1} 2^{b_i} \sum_{i+1 \leq j \leq B_n} (b_j - b_{j-1}) \\
&= \sum_{1 \leq i \leq B_n - 1} \sum_{i+1 \leq j \leq B_n} 2^{b_i}(b_j - b_{j-1}) \\
&= \sum_{1 \leq i < j \leq B_n} 2^{b_i}(b_j - b_{j-1}) \\
&= \sum_{2 \leq j \leq B_n} \sum_{1 \leq i \leq j-1} 2^{b_i}(b_j - b_{j-1}) \\
&\leq \sum_{2 \leq j \leq B_n} \sum_{1 \leq i \leq j-1} 2^{b_i} 2^{(b_j - b_{j-1} - 1)} \quad \text{because } x \leq 2^{x-1} \text{ for all integers } x \\
&= \sum_{2 \leq j \leq B_n} \sum_{1 \leq i \leq j-1} 2^{b_j + b_i - b_{j-1} - 1} \\
&= \sum_{2 \leq j \leq B_n} \left(2^{b_j} \cdot 2^{-(b_{j-1}+1)} \cdot \sum_{1 \leq i \leq j-1} 2^{b_i} \right)
\end{aligned}$$

$$
\begin{aligned}
&< \sum_{2\le j\le B_n} 2^{b_j}\cdot 2^{-(b_{j-1}+1)}\cdot 2^{b_{j-1}+1}\\
&= \sum_{2\le j\le B_n} 2^{b_j}\\
&< \sum_{1\le j\le B_n} 2^{b_j}\\
&= \sum_{b\in B(n)} 2^{b}\\
&= n
\end{aligned}
$$

□

Lemma A.2. *For* $n \ge 1$, $n\lg n \;<\; \sum_{b\in B(n)} 2^b(b+2) \;\le\; n\lg n + 2n$.

Proof.

$$
\begin{aligned}
n\lg n &= \left(\sum_{b\in B(n)} 2^b\right)\left(\lg \sum_{b\in B(n)} 2^b\right)\\
&< \left(\sum_{b\in B(n)} 2^b\right)\left(\lg 2^{b'+1}\right) \qquad \text{where } b' = \max B(n)\\
&= \sum_{b\in B(n)} 2^b(b'+1)\\
&= \sum_{b\in B(n)} 2^b(b+1+b'-b)\\
&= \sum_{b\in B(n)} 2^b(b+1) + \sum_{b\in B(n)} 2^b(b'-b)\\
&< \sum_{b\in B(n)} 2^b(b+1) + \sum_{b\in B(n)} 2^b \qquad \text{by Lemma A.1}\\
&= \sum_{b\in B(n)} 2^b(b+2)\\
\sum_{b\in B(n)} 2^b(b+2) &\le \sum_{b\in B(n)} 2^b(b'+2) \qquad \text{where } b' = \max B(n)\\
&= (b'+2)\sum_{b\in B(n)} 2^b\\
&= (b'+2)n\\
&\le (\lg n+2)n \qquad \text{because } 2^{b'} \le \sum_{b\in B(n)} 2^b = n\\
&= n\lg n + 2n
\end{aligned}
$$

□

BIBLIOGRAPHY

Bibliography

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proceedings of the 40th Symposium on the Foundations of Computer Science (FOCS)*, pages 32–44, 1999.

[2] S. Albers. Better bounds for online scheduling. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 130–139, 1997.

[3] S. Albers, S. Arora, and S. Khanna. Page replacement for generalized caching problems. In *Proceedings of the 10th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 31–40, 1999.

[4] Houman Alborzi, Eric Torng, Patchrawat Uthaisombut, and Stephen Wagner. The k-client problem. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 73–82, 1997.

[5] M. Andrews, M. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proceedings of IEEE FOCS*, 1996.

[6] Javed Aslam, April Rasala, Cliff Stein, and Neal Young. Improved bicriteria existence theorems for scheduling. In *Proceedings of the 10th annual ACM-SIAM Symposium on Discrete Algorithms*, pages S846–S847, 1999.

[7] Y. Azar and O. Regev. On-line bin-stretching. In *Proceedings of the 2nd RANDOM*, pages 71–81, 1998.

[8] Yossi Azar, Leah Epstein, and Rob van Stee. Resource augmentation in load balancing, 1999. manuscript, 14 pages.

[9] K.R. Baker. *Introduction to Sequencing and Scheduling*, chapter 2. Wiley, New York, 1974.

[10] K.R. Baker, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnooy Kan. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints. *Operations Research*, pages 381–386, 1982.

[11] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, pages 101–110, 1991.

[12] Piotr Berman and Chris Coulston. Speed is more powerful than clairvoyance. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 255-263, 1998.

[13] Piotr Berman and Chris Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, pages 181–193, 1999.

[14] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task system. *Journal of the ACM*, 39:745–763, 1992.

[15] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[16] Joan Boyar and Kim S. Larsen. The seat reservation problem. Technical report, University of Southrn Denmark, 1996. PP-1996-24, September 1996, 12 pages.

[17] Joan Boyar, Kim S. Larsen, and Morten N. Nielsen. The accommodating function – a generalization of the competitive ratio. Technical report, University of Southrn Denmark, 1998. PP-1998-24, December 22, 1998, 30 pages.

[18] S. Chakrabarti, C.A. Phillips, A.S. Schulz, D.B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming, Lecture Notes in Computer Science 1099*, pages 646–657, Berlin, 1996. Springer. Proceedings of the 23rd International Colloquium (ICALP'96).

[19] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 609–618, 1997.

[20] E. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2, pages 46–93. PWS Publishing Company, 1997.

[21] E. Coffman and M. Hofri. On the expected performance of scanning disks. *SIAM Journal on Computing*, 11:60–70, 1982.

[22] E. Coffman, L. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing*, 1(3):269–279, 1972.

[23] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[24] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[25] Paolo Dell'Olmo, Hans Kellerer, Maria Grazia Speranza, and Zsolt Tuza. A 13/12 approximation algorithm for bin packing with extendable bins. *Information Processing Letters*, 65:229–233, 1998.

[26] M. Dertouzos and A. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15:1497–1506, 1989.

[27] Jeff Edmonds. Scheduling in the dark. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 179–188, 1999.

[28] A. Feldmann, M.-Y. Kao, J. Sgall, and S-H. Teng. Optimal online scheduling of parallel jobs with dependencies. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 642–651, 1993.

[29] A. Feldmann, J. Sgall, and S-H. Teng. Dynamic scheduling on parallel machines. In *Proc. of 32nd IEEE Symp. on Foundations of Computer Science*, pages 111–120, 1991.

[30] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[31] M.R. Garey and D.S. Johnson. Strong NP-completeness results: motivation, examples and implications. *Journal of the ACM*, 25:499–508, 1978.

[32] M.R. Garey, R.E. Tarjan, and G.T. Wilfong. One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research*, 13:330–348, 1988.

[33] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Scheduling*, 5(1):77–92, 1987.

[34] M.X. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[35] T. Gormley, N. Reingold, E. Torng, and J. Westbrook. Generating adversaries for request-answer games. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 564–565, 2000.

[36] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[37] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, 17:263–269, 1969.

[38] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math*, 5:287–326, 1979.

[39] L.A. Hall, A. Schulz, D.B. Shmoys, and J. Wein. Scheduling to minimize average completion time: off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–549, 1997.

[40] L.A. Hall, D.B. Shmoys, and J. Wein. Scheduling to minimize weighted completion time: off-line and on-line algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, 1996.

[41] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: practical and theoretical results. *Journal of the ACM*, 34(1):144–162, 1987.

[42] J.A. Hoogeveen. Minimizing maximum promptness and maximum lateness on a single machine. *Mathematics of Operations Research*, 21:100–114, 1996.

[43] J.A. Hoogeveen. Single-machine scheduling to minimize a function of two or three maximum cost criteria. *Journal of Algorithms*, 21(2):415–433, 1996.

[44] J.A. Hoogeveen and S.L. van de Velde. Minimizing total completion-time and maximum cost simultaneously is solvable in polynomial-time. *Operations Research Letters*, 17(5):205–208, 1995.

[45] C.A.J. Hurkens and M.J. Coster. On the makespan of a schedule minimizing total completion time for unrelated parallel machines, 1996. Unpublished manuscript.

[46] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974.

[47] B. Kalyanasundaram and K. Pruhs. The online transportation problem. In *Lecture Notes in Computer Science 979*, pages 484–493, 1995. European Symposium on Algorithms (ESA). Also to appear in SIAM Journal on Discrete Mathematics.

[48] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *Proceedings of the 36th Annual IEEE Foundations of Computer Science*, pages 214–221, 1995. Also to appear in Journal of the ACM.

[49] Bala Kalyanasundaram and Kirk Pruhs. Maximizing job completions online. In *Lecture Notes in Computer Science 1461*, pages 235–246, 1998. European Symposium on Algorithms (ESA).

[50] R.M. Karp and N. Karmarkar. An efficient approximation scheme for the one-dimensional bin-packing problem. In *Proc. of the 23th IEEE Symp. Foundations of Computer Science*, pages 312–320, 1982.

[51] Tsuyoshi Kawaguchi and Seiki Kyan. Worst case bound of an LRF schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15(4):1119–1129, 1986.

[52] H. Kellerer, T. Tautenhahn, and G.J. Woeginger. Approximability and non-approximability results for minimizing total flow time on a single machine. In *STOC'96*, pages 418–426, 1996.

[53] G. Koren, D. Shasha, and S.-C. Huang. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. In *Proc. 14th Real-Time Systems Symposium*, pages 172–181, 1993.

[54] Madhukar R. Korupolu, C. Greg Plaxton, and Rajmohan Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 1998.

[55] E. Koutsoupias and C. Papadimitriou. Beyond competitive analysis. In *Proceedings of the 35th Annual IEEE Foundations of Computer Science*, pages 394–400, 1994.

[56] E. Koutsoupias and C. Papadimitriou. On the k–server conjecture. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 507–511, 1994.

[57] Tak Wah Lam and Kar Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 1999.

[58] E.L. Lawler. A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. *Annals of Operations Research*, 26:125–133, 1990.

[59] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[60] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1988.

[61] S.T. McCormick and M.L. Pinedo. Scheduling n independent jobs on m uniform machines with both flow time and makespan objectives: A parametric approach. *ORSA Journal on Computing*, 7:63–77, 1992.

[62] J.M. Moore. Sequencing n jobs on one machine to minimize the number of tardy jobs. *Management Science*, 17(1), 1968.

[63] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[64] A. Nagar, J. Haddock, and S. Heragu. Multiple and bicriteria scheduling: A literature survey. *European Journal of Operational Research*, 81(1):88–104, 1995.

[65] Rosser T. Nelson, Rakesh K. Sarin, and Richard L. Daniels. Scheduling with multiple performance measures: the one-machine case. *Management Science*, 32(4):464–479, 1986.

[66] W. Oney. Queuing analysis of the scan policy for moving-head disks. *Journal of the ACM*, 22(3):397–412, 1975.

[67] S.S. Panwalkar, R.A. Dudek, and M.L. Smith. Sequencing research and the industrial scheduling problem. In S.E. Elmaghraby, editor, *Symposium on the Theory of Scheduling and its Applications*, New York, 1973. Springer.

[68] Cynthia Phillips, Andreas Schulz, David Shmoys, Clifford Stein, and Joel Wein. Improved bounds on relaxations of a parallel machine scheduling problem. *Journal of Combinatorial Optimization*, 1(4):413–426, 1998.

[69] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 140–149, 1997.

[70] April Rasala. Existence theorems for scheduling to meet two objectives. Technical report, Dartmouth College, 1999. Computer Science Technical Report PCS-TR99-347.

[71] M.B. Richey. Improved bounds for harmonic-based bin packing algorithms. *Discrete Applied Mathematics*, 34:203–227, 1991.

[72] Alexander Schrijver. *Theory of Linear and Integer Programming.* John-Wiley & sons, 1986.

[73] A.S. Schulz. *Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds*, pages 301–315. Springer, Berlin, 1996. Proceedings of the 5th International Conference on Integer Programming and Combinarotial Optimization.

[74] A.S. Schulz and M. Skutella. Scheduling-LPs bear probabilities: Randomized approximations for min-sum criteria. Technical report, Technical University of Berlin, Germany, 1996. Technical Report 533/1996.

[75] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX*, pages 313–324, 1990.

[76] D.B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[77] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[78] W.E. Smith. Various optimizers for single-state production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[79] M.G. Speranza and Z. Tuza. On-line approximation algorithms for scheduling tasks on identical machines with extendable working time. *Annals of Operations Research*, 86:491–506, 1999.

[80] Cliff Stein and Joel Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operations Research Letters*, 21(3):115–122, 1997.

[81] T.J. Teorey and T.B. Pinkerton. A comparative analysis of disk scheduling policies. *Comm. ACM*, 15:177–194, 1972.

[82] Eric Torng and Patchrawat Uthaisombut. A tight lower bound for the BEST-alpha algorithm. *Information Processing Letters (IPL)*, 71(1):17–22, 1999.

[83] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 24:309–315, 1978.

[84] Yaoguang Wang. Bicriteria job sequencing with job release dates. Technical report, Max Plank Institute, Germany, 1997. MPI-I-97-1-005.

[85] Luc N. Van Wassenhove and Ludo F. Gelders. Solving a bicriterion scheduling problem. *European Journal of Operational Research*, 4:42–48, 1980.

[86] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS*, pages 146–156, 1995.