Spring 5-31-2005

Some topics on deterministic scheduling problems

Yumei Huo
New Jersey Institute of Technology

ABSTRACT

SOME TOPICS ON DETERMINISTIC SCHEDULING PROBLEMS

by
Yumei Huo

Sequencing and scheduling problems are motivated by allocation of limited resources over time. The goal is to find an optimal allocation where optimality is defined by some problem specific objectives.

This dissertation considers the scheduling of a set of n tasks, with precedence constraints, on $m \geq 1$ identical and parallel processors so as to minimize the makespan. Specifically, it considers the situation where tasks, along with their precedence constraints, are released at different times, and the scheduler has to make scheduling decisions without knowledge of future releases. Both preemptive and nonpreemptive schedules are considered. This dissertation shows that *optimal* online algorithms exist for some cases, while for others it is impossible to have one. The results give a sharp boundary delineating the possible and the impossible cases.

Then an $O(n \log n)$-time implementation is given for the algorithm which solves $P \mid p_j = 1, r_j, outtree \mid \sum C_j$ and $P \mid pmtn, p_j = 1, r_j, outtree \mid \sum C_j$.

A fundamental problem in scheduling theory is that of scheduling a set of n unit-execution-time (UET) tasks, with precedence constraints, on $m \geq 1$ parallel and identical processors so as to minimize the mean flow time. For arbitrary precedence constraints, this dissertation gives a 2-approximation algorithm. For intrees, a 1.5-approximation algorithm is given.

Six dual criteria problems are also considered in this dissertation. Two open problems are first solved. Both problems are single machine scheduling problems with the number of tardy jobs as the primary criterion and with the total completion time and the total tardiness as the secondary criterion, respectively. Both problems are shown to be NP-hard. Then it focuses on bi-criteria scheduling problems involving the number of tardy jobs, the

maximum weighted tardiness and the maximum tardiness. NP-hardness proofs are given for the scheduling problems when the number of tardy jobs is the primary criterion and the maximum weighted tardiness is the secondary criterion, or vice versa. It then considers complexity relationships between the various problems, gives polynomial-time algorithms for some special cases, and proposes fast heuristics for the general case.

SOME TOPICS ON DETERMINISTIC SCHEDULING PROBLEMS

by
Yumei Huo

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2005

APPROVAL PAGE

SOME TOPICS ON DETERMINISTIC SCHEDULING PROBLEMS

Yumei Huo

Joseph Leung, Dissertation Advisor — Date
Distinguished Professor of Computer Science, NJIT

Dr. Artur Czumaj, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Marvin K. Nakayama, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Michael Pinedo, Committee Member — Date
Julius Schlesinger Professor of Operations Management, New York University

BIOGRAPHICAL SKETCH

Author: Yumei Huo

Degree: Doctor of Philosophy

Date: May 2005

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, May 2005
- Master of Science in Computer Science,
  University of Science and Technology, Beijing, April 2000
- Bachelor of Science in Computer Science,
  University of Science and Technology, Beijing, July 1997

Major: Computer Science

Presentations and Publications:

Huo, Y. and J. Y-T. Leung, "Online Scheduling of Precedence Constrained Tasks", *SIAM J. on Computing*, accepted for publication.

Huo, Y. and J. Y-T. Leung, "Minimizing Total Completion Time for UET Tasks with Release Time and Outtree Precedence Constraints", *Mathematical Methods of Operations Research*, accepted for publication.

Huo, Y. and J. Y-T. Leung, "Minimizing Mean Flow Time for UET Tasks", *ACM Transactions on Algorithms*, accepted for publication.

Huo, Y., J. Y-T. Leung and H. Zhao, "Complexity of Two Dual Criteria Scheduling Problems", *submitted to Operations Research Letters.*

Huo, Y., J. Y-T. Leung and H. Zhao, "Bi-criteria Scheduling Problems: Number of Tardy Jobs and Maximum Weighted Tardiness", *submitted to European Journal of Operational Research.*

This dissertation is dedicated to my parents and my husband. Their constant love and caring are every reason for where I am and what I am. My gratitude and my love to them are beyond words.

ACKNOWLEDGMENT

I gratefully acknowledge all the people who gave me support and help in my Ph.D. program and my life during the past years of my stay in NJIT, although words here are too limited to express my sincere thanks. I also acknowledge the Department of Computer Science at New Jersey Institute of Technology for supporting my studies.

I deeply thank my advisor Dr. Joseph Y-T. Leung who led me to the research field of Computational Complexity and Scheduling Theory. With his constant guidance and encouragement, I was more confident in the work of this dissertation. He gave me many opportunities that stimulated my interests and enabled me to gain more experience in my research field. I have learned quite a lot from his extensive knowledge in scheduling areas and benefited from his many creative ideas. His kind help will make the past years an ever-good memory in my life. I also thank Dr. Joseph Y-T. Leung for his valuable advice on writing papers and strict corrections to them.

I am especially grateful to my committee – Dr. Artur Czumaj, Dr. Alexandros Gerbessiotis, Dr. Marvin K. Nakayama and Dr. Michael Pinedo for useful advice and constructive criticism at the proposal, at my defense, and in between.

I am very thankful to Dr. Artur Czumaj for helpful conversations and friendly advice on my study, research and life during the last four years.

I am deeply grateful to Dr. Michael A. Baltrush for much valuable advice on my presentations. Being his teaching assistant for four years, I really want to thank him for his kindness and patience.

I like to express my sincere appreciation to Hairong Zhao for the nice collaborations on the research of dual criteria scheduling problems and on the co-authored papers.

It has been a very nice time and a good memory for me to share the office with Hairong Zhao and Haibing Li who always gave me instant help. The pleasant atmosphere in our office made my work efficient and enjoyable.

I wish to express my great appreciation to Xiaofeng Wang, Hairong Zhao and Wenxin Mao for their warm help with my family. My husband and I will never forget their kindness.

I would also say many thanks to my good friend Yuting Zhang for her encouragement and help in the past years.

At last, I thank my family: my husband, Xin Wang, for giving me the most patient and loving support and encouragement to my work, and for his many good suggestions to my dissertation; my parents, for their ever-loving support and understanding during the years of my studies and work.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

Sequencing and scheduling problems are motivated by allocation of limited resources over time. The goal is to find an optimal allocation where optimality is defined by some problem specific objective. In the majority of the models studied, the resources consist simply of a set $P = \{P_1, \ldots, P_m\}$ of processors. Depending on the specific problem, they are either identical, identical in functional capability but different in speed, or different in both function and speed. The scheduling problems studied in this dissertation assume identical processors. Activities are modeled by tasks which can be executed by the processors.

Most of the early work on scheduling, starting from early 1950's, was motivated by production planning and manufacturing, and was primarily done in the operations research and management science community. The advent of computers and their widespread use had a considerable impact both on scheduling problems and solution strategies. A number of new problems and variations have been motivated by application areas in computer science such as parallel computing, databases, compilers, and time sharing. The advent of computers also initiated the formal study of efficiency of computation that led to the notion of NP-Completeness. Karp's seminal work [38] established the pervasive nature of NP-Completeness by showing that decision versions of several naturally occuring problems in combinatorial optimization are NP-Complete, and thus are unlikely to have efficient polynomial-time algorithms. Following Karp's work, many problems, including scheduling problems, were shown to be NP-Complete. Garey and Johnson [22] gave 18 basic NP-complete scheduling problems; since then many new variants were considered and shown to be NP-complete. It is widely believed that P (the set of languages that can be recognized by deterministic Turing machines in polynomial time) is a proper subset of NP (the set of languages that can be recognized by non-deterministic Turing machines in polynomial

time). Proving that $P \neq NP$ is the most outstanding problem in theoretical computer science today.

After the NP-Completeness results, the focus has shifted to designing approximation algorithms, often using quite non-trivial techniques and insights. Approximation algorithms are heuristics that provide provably good guarantees on the quality of the solutions they return. This approach is pioneered by the influential paper of Johnson [35] in which he showed the existence of good approximation algorithms for several NP-Hard optimization problems. He also remarked that the optimization problems that are all indistinguishable in the theory of NP-Completeness behave very differently when it comes to approximability. Remarkable work in the last couple of decades in both the design of approximation algorithms and proving inapproximability results has validated Johnson's remarks. The methodology of evaluating algorithms by the quality of their solutions is useful in comparing commonly used heuristics, and often the analysis suggests new and improved heuristics. In this dissertation two deterministic scheduling problems are considered. For each of these problems, an approximation algorithm is proposed and the approximation ratio for the corresponding algorithm is proved.

Over the past twenty years, online algorithms have received considerable research interest. An online algorithm receives the input incrementally, one piece at a time. In response to each input portion, the algorithm must generate output, not knowing future input. Online problems had been investigated already in the seventies and eighties, but an extensive, systematic study started only when Sleator and Tarjan [57] suggested comparing an online algorithm to an optimal offline algorithm, and Karlin, Manasse, Rudolph and Sleator [37] coined the term *competitive analysis*. In the late eighties and early nineties, three basic online problems were studied extensively, namely paging, the k-server problem [46] and metrical task systems [6]. During the past years, apart from the three basic problems, many online problems were investigated in application areas such as data structures, distributed data management, scheduling and load balancing, routing, robotics, financial

games, graph theory, and a number of problems arising in computer systems. This dissertation concentrates on some online scheduling problems. For each scheduling problem, either an optimal online algorithm is given or a proof is given to show that it is impossible to have an optimal online algorithm.

In the past, most of the research in scheduling has focussed on a single criterion. Numerous effective algorithms and heuristics have been developed for these single criterion problems; see Pinedo [54] and Brucker [7]. However, companies are usually faced with the problem of satisfying several different groups of people. According to Panwalkar *et al.* [53], managers actually develop schedules based on multiple criteria. Unfortunately, schedules that are optimal for one criterion usually perform quite poorly for other criteria. Thus, there is a need for further research in multi-criteria scheduling problems, and indeed, these problems have received more attention in the last three decades; see [10, 11, 15, 20, 21, 26, 44, 56, 58, 59]. This dissertation is concerned with scheduling problems with two criteria only.

The next section reviews relevant background in scheduling theory, approximation theory, online scheduling theory and multi-criteria scheduling. Section 1.2 outlines the contributions of the dissertation.

1.1 Background and Notation

1.1.1 Scheduling Theory

Scheduling theory encompasses a large and diverse set of models, algorithms, and results. Even a succinct overview of the field would take many pages. Hence only those concepts that are directly relevant to this dissertation are reviewed and the reader is referred to many excellent books and surveys available [54][12][43]. The problems this dissertation considers involve scheduling or allocating tasks to processors under various constraints. Unless otherwise stated, n denotes the number of tasks and m denotes the number of processors. Each task j is characterized by its processing time,release time, due date,

and weight (or 'value') which are denoted by p_j,r_j,d_j and w_j, respectively, and perhaps other characteristics as required by each variant of scheduling problems. The scheduling algorithm is asked to produce a schedule, which means that each task is assigned to one or more processors and one or more time slots, according to the variant of scheduling problems. Each processor is assigned to a single task at any time, and the processing of a task always takes at least its processing time. The tasks have precedence constraints, $\prec$, in that $i \prec j$ signifies that task j cannot start until task i has finished. The tasks and the precedence constraints are described by a directed acyclic graph $G = (V, A)$, where V is a set of vertices representing the tasks and A is a set of directed arcs representing $\prec$; there is a directed arc from task i to task j if $i \prec j$. Assume G has no transitive edges. The tasks can be scheduled preemptively or nonpreemptively. In preemptive scheduling, a task can be interrupted before it completes and later resumed on a possibly different processor. Assume that there is no time loss in preemption. By contrast, in nonpreemptive scheduling, a task once begun execution can not be interrupted until it completes. With respect to a schedule S, the completion time of task i is denoted by C_i, the makespan is denoted by $C_{\max} = \max\{C_i\}$, and the mean flow time is denoted by $\sum C_j$. If $C_j > d_j$, task j is defined to be tardy and $T_j = C_j - d_j$ denote its tardiness. In addition, the variable U_j is used as an indicator that task j is tardy; in this case U_j is set to 1. On the other hand, if $C_j \leq d_j$, task j is defined to be on time, and $U_j = 0$ and $T_j = 0$.

A task i is said to be an *immediate predecessor* of another task j if there is a directed arc (i, j) in G; j is said to be an *immediate successor* of i. Task i is said to be a *predecessor* of task j if there is a directed path from i to j; j is said to be a *successor* of i. Define G to be *intree* if every vertex, except the root, has exactly one immediate successor. G is an *outtree* if each vertex, except the root, has exactly one immediate predecessor. A *chain* is an outree where each vertex has at most one immediate successor. *prec* is used to denote an arbitrary directed acyclic graph.

In the past, research in scheduling theory has concentrated on these four classes of precedence constraints: prec, intree, outtree, and chains. A number of polynomial-time optimal algorithms have been developed. Among these algorithms, the Coffman-Graham algorithm [13] and Hu's algorithm [29] are two fundamental algorithms. The famous Coffman-Graham algorithm is optimal for $P2 \mid p_j = 1, prec \mid C_{\max}$, while the well-known Hu's algorithm is optimal for $P \mid p_j = 1, intree \mid C_{\max}$ and $P \mid p_j = 1, outtree \mid C_{\max}$. Since these two algorithms are used in both Chapter 2 and Chapter 4, the description of these two algorithms will be first given in the following.

The Coffman-Graham algorithm works by first assigning a label to each task which corresponds to the priority of the task; tasks with higher labels have higher priority. Once the labels are assigned, tasks are scheduled as follows: Whenever a processor becomes free, assign that task all of whose predecessors have already been executed and which has the largest label among those tasks not yet assigned.

Before the labeling algorithm, the definition of a linear order on decreasing sequences of positive integers is first described as follows.

Definition 1.1.1 *Let* $N = (n_1, n_2, \ldots, n_t)$ *and* $N' = (n'_1, n'_2, \ldots, n'_{t'})$ *be two decreasing sequences of positive integers. Define* $N < N'$ *if either*

1. *For some* i, $1 \leq i \leq t$, $n_j = n'_j$ *for all* j *satisfying* $1 \leq j \leq i-1$ *and* $n_i < n'_i$, *or*
2. $t < t'$ *and* $n_j = n'_j$ *for all* j *satisfying* $1 \leq j \leq t$.

Example: $(8, 6, 4, 3) < (8, 6, 5)$ and $(9, 8, 6) < (9, 8, 6, 4, 3)$.

Let n denote the number of tasks in *prec*. The labeling algorithm assigns to each task i an integer label $\alpha(i) \in \{1, 2, \ldots, n\}$. The mapping α is defined as follows. Let $IS(i)$ denote the set of immediate successors of task i and let $N(i)$ denote the decreasing sequence of integers formed by ordering the set $\{\alpha(j) \mid j \in IS(i)\}$.

1. An arbitrary task i with $IS(i) = \emptyset$ is chosen and $\alpha(i)$ is defined to be 1.

2. Suppose for some $k \leq n$ that the integers $1, 2, \ldots, k-1$ have been assigned. From the set of tasks for which α has been defined on all elements of their immediate successors, choose the task j such that $N(j) \leq N(i)$ for all such tasks i. Define $\alpha(j)$ to be k.

3. Repeat the assignment in 2 until all tasks of *prec* have been assigned some integer.

Example: Fig. 1.1 shows a set of tasks with their precedence constraints. The number inside the circle represents the task's index and the number next to the circle represents the label assigned to the task by the Coffman-Graham labeling algorithm. The schedule on two processors is also shown in Fig. 1.1.

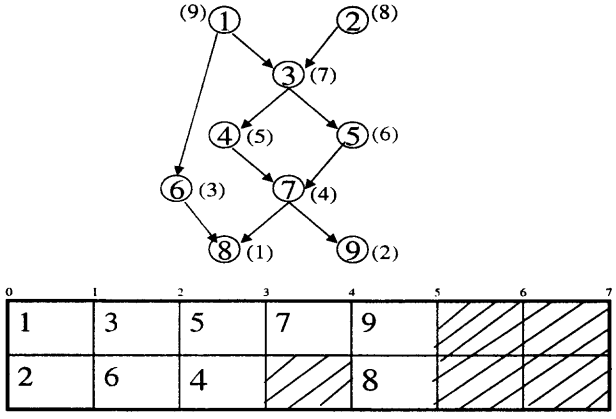


Figure 1.1 Example illustrating the Coffman-Graham algorithm

Like the Coffman-Graham algorithm, Hu's algorithm first assigns a label to each task which corresponds to the priority of the task; tasks with higher labels have higher priority. Once the labels are assigned, tasks are scheduled as follows: Whenever a processor becomes free, assign that task all of whose predecessors have already been executed and which has the largest label among those tasks not yet assigned. In Hu's algorithm, the label of a task is a function of the *level* of the task.

Definition 1.1.2 *The level of a task i with no immediate successor is its processing time p_i. The level of a task with immediate successor(s) is its processing time plus the maximum level of its immediate successor(s).*

Hu's labeling algorithm assigns higher labels to tasks at higher levels; ties are broken in an arbitrary manner.

Example: Fig. 1.2 shows a set of tasks with outtree precedence constraint. The number inside the circle is the task's index and the number next to the circle is the label given by Hu's algorithm. A schedule on four processors produced by Hu's algorithm is also shown.

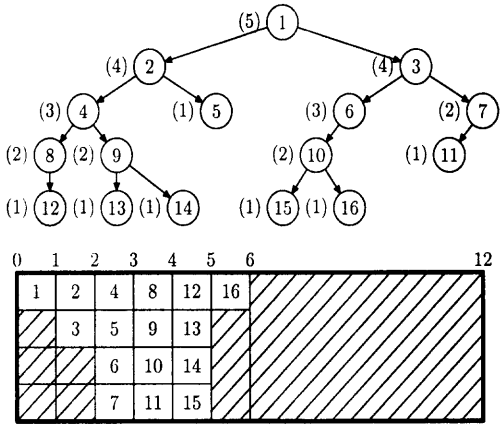


Figure 1.2 Example illustrating Hu's algorithm

1.1.2 Online Algorithms

For online scheduling the most important classification of online problems is according to which part of the problem is given. There are several very different possibilities [55].

Scheduling tasks one by one. In this paradigm the tasks are ordered in some list (sequence) and are presented one by one according to this list. Each of them has to be assigned to some processor(s) and time slot(s) before the next task is seen, consistent with other restrictions given by the problem. As soon as the task is presented, all of its

characteristics, including the processing time, are known. It is allowed to assign the tasks to arbitrary time slots (i.e., they can be delayed); however, once the successive tasks are seen, the assignment of the previous tasks cannot be changed.

Unknown processing time. Here the main online feature is the fact that the processing time of a task is unknown until the task finishes; an online algorithm only knows whether a task is still running or not. Unlike in the previous paradigm, at any time all currently available tasks are at the disposal of the algorithm; any of them can be started now on any processor(s) or delayed further. Also, if preemptions or restarts are allowed, the algorithm can decide to preempt or stop any task which is currently running. The tasks may become available over time according to their release time or precedence constraints, but the situation when all tasks are available at the beginning plays an important role in this paradigm, too. If there are other characteristics of a task than its processing time, they are known when the task becomes available, which has to be known to guarantee that the task is scheduled legally.

Tasks arrive over time. In this paradigm the algorithm has the same freedom as in the previous one, and in addition, the processing time of each task is also known when that task is available. Thus the only online feature is the lack of knowledge of tasks arriving in the future. Algorithms that know the running time of a task as soon as it arrives are called clairvoyant, in contrast to non-clairvoyant algorithms that correspond to the previous paradigm of unknown processing time.

Interval scheduling. All the previous paradigms assume that a task may be delayed. Contrary to that, the paradigm of interval scheduling assumes that each task has to be executed in a precisely given time interval; if this is impossible it may be rejected.

The online problems studied in this dissertation belong to the third paradigm: tasks arrive over time. At different release time, a group of tasks are released with their precedence constraints.

1.1.3 Approximation Algorithms

In this subsection, an NP-Hard optimization problem is defined and the notion of approximation are explored [49]. The following is a formal definition of a maximization problem; a minimization problem can be defined analogously.

Definition 1.1.3 *An optimization problem* $\prod$ *is characterized by three components:*

[Instance] D*: a set of input instances.*

[Solutions] SI*: the set of all feasible solutions for an instance* $I \in D$

[value] f*: a function which assigns a value to each solution, i.e.* $f : S(I) \rightarrow \Re$

A maximization problem $\prod$ *is: Given* $I \in D$*, find a solution* $\sigma^I_{opt} \in S(I)$ *such that*

$\forall \sigma \in S(I), f(\sigma^I_{opt}) \geq f(\sigma)$

The value of the optimal solution will be referred to as $OPT(I)$,i.e. $OPT(I) \triangleq f(\sigma^I_{opt})$.

Given an NP-Hard optimization problem $\prod$, it is impossible to find an algorithm which is guaranteed to compute an optimal solution in polynomial time for all input instances, unless $P = NP$. So it is necessary to relax the requirement of optimality and ask for an approximation algorithm. This is defined as follows.

Definition 1.1.4 *An approximation algorithm* A*, for an optimization problem* $\prod$*, is a polynomial-time algorithm such that given an input instance* I *for* $\prod$*, it will output some* $\sigma \in S(I)$*.* $A(I)$ *denotes the value* $f(\sigma)$ *of the solution obtained by* A*.*

Note that this dissertation is only interested in polynomial-time algorithms and so this is built into the definition of an approximation algorithm.

Some ways are needed to compare approximation algorithms and analyze the quality of solutions produced by them. Moreover, the "measure of goodness" of an approximation

algorithm must somehow relate the optimal solution to the solution produced by the algorithm. Such measures are referred to as performance guarantees. There are several notions of performance guarantees. Since the work of Graham [25] in 1966 on multiprocessor scheduling, relative performance guarantee becomes the mostly used performance guarantee for problems in scheduling. Some of these results can be found in the survey article by Lawler et al [43]. The following definition formalizes this notion.

Definition 1.1.5 *Let A be an approximation algorithm for an optimization problem $\prod$. The performance ratio $R_A(I)$ of the algorithm A on an input instance I is defined as*

$$R_A(I) = \frac{A(I)}{OPT(I)}$$

in the case where $\prod$ is a minimization problem. On the other hand when $\prod$ is a maximization problem, define the performance ratio as

$$R_A(I) = \frac{OPT(I)}{A(I)}$$

1.1.4 Dual Criteria Scheduling

When faced with a choice of schedules, a manager should pick the "best" one. However, defining "best" may be very difficult. Should it be the one that generates the most profit in the short term, or the one that makes the most customers happy? Unfortunately, schedules which perform well with respect to one measure often do poorly with respect to another. To alleviate this problem, it is necessary to consider two measures simultaneously.

In the literature, there are three general dual criteria approaches that are applicable to scheduling problems: secondary criterion, efficient set generation and weighting of criteria [10].

The secondary criterion approach is to have one criterion designated as the primary criterion and the other one designated as the secondary criterion. This approach seeks a schedule that minimizes the primary criterion and chooses, from among all the schedules that minimize the primary criterion, the one that also minimizes the secondary criterion.

Extending the notation of Graham *et al.* [24], $1 \mid\mid \gamma_2 \mid \gamma_1$ is used to denote the single machine scheduling problem, where γ_1 is the primary criterion and γ_2 is the secondary criterion. For example, $1 \mid\mid \sum C_j \mid T_{\max}$ denotes the problem where the primary criterion is maximum tardiness and the secondary criterion is total completion time. As another example, $1 \mid\mid \sum T_j \mid \sum U_j$ denotes the problem where the primary criterion is the number of tardy jobs and the secondary criterion is the total tardiness.

Efficient set generation approach is to efficiently generate the Pareto curve which enables the decision maker to make explicit trade-offs between these schedules. Extending the notation of Graham *et al.* [24], this approach is denoted by $1 \mid\mid \gamma_1, \gamma_2$, where the two criteria of interest are γ_1 and γ_2. $1 \mid\mid \sum C_j, T_{\max}$ denotes the problem to generate all non-dominated schedules considering total completion time and maximum tardiness simultaneously.

Weighting of criteria approach is to use a cost function which is a linear combination of the two criteria. Here the decision maker expresses a tradeoff which, once specified,allows the problem to be solved with a single criterion. A scheduling problem with two criteria, say γ_1 and γ_2, and a given weighting function f is denoted by $1 \mid\mid f(\gamma_1, \gamma_2)$, where f is a linear combination of γ_1 and γ_2, e.g. $f(\gamma_1, \gamma_2) = \lambda_1\gamma_1 + \lambda_2\gamma_2$.

This dissertation considers only the first approach. Although there are numerous work done under the second and the third approaches, this dissertation will not dwell into them.

1.2 Organization and Overview of Contribution

This section describes the problems and corresponding results considered in this dissertation.

1.2.1 Online Scheduling of Precedence Constrainted Tasks

Chapter 2 considers the problem of scheduling a set of tasks on $m \geq 1$ identical and parallel processors so as to minimize $C_{\max}$. In the three-field classification scheme introduced by

Graham *et al.* [24], the problems considered in this dissertation are $P \mid prec \mid C_{\max}$ and $P \mid pmtn, prec \mid C_{\max}$. A number of polynomial-time optimal algorithms have been developed for these problems. In nonpreemptive scheduling, the famous Coffman-Graham algorithm [13] is optimal for $P2 \mid p_j = 1, prec \mid C_{\max}$, while the well-known Hu's algorithm [28] is optimal for $P \mid p_j = 1, intree \mid C_{\max}$ and $P \mid p_j = 1, outtree \mid C_{\max}$. It is known that $P \mid p_j = 1, prec \mid C_{\max}$ is strongly NP-hard [22], although the complexity is still open for each fixed $m \geq 3$. If the tasks have arbitrary processing times, then the problem becomes NP-hard in the ordinary sense even if there are two processors and the tasks are independent; i.e. $P2 \mid\mid C_{\max}$ is NP-hard in the ordinary sense [22].

For preemptive scheduling, the Muntz-Coffman algorithm [50, 51] is optimal for $P2 \mid pmtn, prec \mid C_{\max}$, $P \mid pmtn, intree \mid C_{\max}$ and $P \mid pmtn, outtree \mid C_{\max}$. Again, $P \mid pmtn, prec \mid C_{\max}$ is strongly NP-hard [22], while the complexity is still open for each fixed $m \geq 3$.

All of the algorithms mentioned above assume that all tasks are available for processing at the beginning (i.e., at time $t = 0$).

This dissertation considers the situation where tasks, along with their precedence constraints, are released at different times, and the scheduler has to make scheduling decision without knowledge of future releases. In other words, the scheduler has to schedule tasks in an online fashion. An online scheduling algorithm is said to be *optimal* if it always produces a schedule with the minimum $C_{\max}$, i.e., a schedule as good as any schedule produced by any scheduling algorithm with full knowledge of future releases of tasks. Since an online scheduling algorithm has to schedule tasks in an online fashion, it is not clear that an optimal online scheduling algorithm necessarily exists. This dissertation shows that online scheduling algorithms exist for some cases, while for others it is impossible to have one. These results give a sharp boundary delineating the possible and the impossible cases.

The notation of Graham *et al.* [24] is extended to online scheduling problems in a natural way. For example, $P2 \mid p_j = 1, prec_i\ released\ at\ r_i \mid C_{\max}$ refers to the case where

tasks with arbitrary precedence constraint, $prec_i$, are released at time r_i. In this case, there are two processors, each task has unit processing time, and preemption is not allowed. As another example, $P \mid pmtn, outtree_i \ released \ at \ r_i \mid C_{\max}$ refers to the case where tasks with outtree precedence constraint, $outtree_i$, are released at time r_i. In this case, there are arbitrary number of processors, tasks have arbitrary processing times, and preemption is allowed.

Hong and Leung [27] have given an optimal online scheduling algorithm for a set of independent tasks on an arbitrary number of processors where preemption is allowed. The idea of their algorithm is to schedule tasks using a modified McNaughton's wrap-around rule. (It is known that McNaughton's wrap-around rule is optimal for $P \mid pmtn \mid C_{\max}$ [47].) Tasks will be executed according to the schedule until new tasks arrive, at which time the algorithm will reschedule, by the same rule, the remaining portions of the unfinished tasks along with the newly arrived tasks. This process is repeated until all tasks are finished and no new tasks arrive.

Note that for nonpreemptive scheduling, it can be shown that it is impossible to have an optimal online algorithm for a set of independent tasks with arbitrary processing times, even if there are only two processors.

This dissertation shows that optimal online scheduling algorithms exist for:

(1) $P2 \mid p_j = 1, prec_i \ released \ at \ r_i \mid C_{\max}$.

(2) $P \mid p_j = 1, outtree_i \ released \ at \ r_i \mid C_{\max}$.

(3) $P2 \mid pmtn, prec_i \ released \ at \ r_i \mid C_{\max}$.

(4) $P \mid pmtn, outtree_i \ released \ at \ r_i \mid C_{\max}$.

Using an adversary argument, it can be shown that it is impossible to have optimal online scheduling algorithms for:

(1) $P3 \mid p_j = 1, intree_i \ released \ at \ r_i \mid C_{\max}$.

(2) $P2 \mid p_j = p, chains_i\ released\ at\ r_i \mid C_{\max}$.

(3) $P3 \mid pmtn, p_j = 1, intree_i\ released\ at\ r_i \mid C_{\max}$.

In this dissertation, all of the optimal online scheduling algorithms follow the same format as the algorithm given in Hong and Leung [27]. For $P2 \mid p_j = 1, prec_i\ released\ at\ r_i \mid C_{\max}$, the Coffman-Graham algorithm is used to schedule tasks until new tasks arrive, at which time the remaining portions of the unfinished tasks along with the newly arrived tasks will be rescheduled. Hu's algorithm is used for $P \mid p_j = 1, outtree_i\ released\ at\ r_i \mid C_{\max}$, and the Muntz-Coffman algorithm for $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$.

1.2.2 Minimizing Total Completion Time for UET Tasks with Release Time and Outtree Precedence Constraints

Chaper 3 considers the problem of scheduling a set of n unit-processing-time tasks, with release time and outtree precedence constraints, on $m \geq 1$ identical and parallel processors so as to minimize the total completion time. The goal is to find a schedule such that the release time and precedence constraints are observed and $\sum C_j$ is minimized. In the notation introduced by Graham et al. [24], the problems considered in this chapter are $P \mid p_j = 1, r_j, outtree \mid \sum C_j$ and $P \mid pmtn, p_j = 1, r_j, outtree \mid \sum C_j$.

Recently, Brucker, Hurink and Knust gave an $O(n^2)$-time algorithm for solving $P \mid p_j = 1, outtree, r_j \mid \sum C_j$; see [9]. Furthermore, they showed that preemption will not reduce $\sum C_j$. Thus, the same algorithm solves the problem $P \mid pmtn, p_j = 1, outtree, r_j \mid \sum C_j$ as well. This dissertation shows that their algorithm admits an $O(n \log n)$-time implementation.

1.2.3 Minimizing Mean Flow Time for UET Tasks

Chapter 4 considers the problem of scheduling the set of n unit-execution-time(UET) tasks on $m \geq 1$ identical and parallel processors so as to minimize the mean flow time. In the

notation introduced by Graham et al. [24], the problem considered in this dissertation is $P \mid p_j = 1, prec \mid \sum C_j$. A number of polynomial-time algorithms and NP-hardness results have been obtained for these problems. For example, $P2 \mid p_j = 1, prec \mid \sum C_j$ can be solved by the Coffman-Graham algorithm [13]; $P \mid p_j = 1, outtree \mid \sum C_j$ can be solved by an algorithm due to Brucker et al. [9]; $Pm \mid p_j = 1, intree \mid \sum C_j$ can be solved in polynomial time for each fixed m [5]. On the other hand, $P \mid p_j = 1, prec \mid \sum C_j$ is NP-hard in the strong sense [22], although its complexity is still open for each fixed $m \geq 3$. The complexity of $P \mid p_j = 1, intree \mid \sum C_j$ has not yet been resolved.

If the tasks have arbitrary processing times, the problem becomes much more difficult. Lawler [40] has shown that $1 \mid prec \mid \sum C_j$ is NP-hard. $1 \mid intree \mid \sum C_j$ and $1 \mid outtree \mid \sum C_j$ can both be solved by an alogrithm due to Horn [28]. On the other hand, Du et al. [19] have shown that $Pm \mid chains \mid \sum C_j$ is NP-hard in the strong sense for each fixed $m \geq 2$. When there are no precedence constraints, the well-known SPT (shortest-processing-time first) rule solves the problem for any number of processors; i.e., $P \mid\mid \sum C_j$ can be solved by the SPT rule.

The complexity of the preemptive case is identical to that of the nonpreemptive case. Since preemption cannot reduce $\sum C_j$ on one processor, the complexity of preemptive scheduling on one processor is identical to that of the nonpreemptive case; i.e., $1 \mid pmtn, prec \mid \sum C_j$ is NP-hard while $1 \mid pmtn, intree \mid \sum C_j$ and $1 \mid pmtn, outtree \mid \sum C_j$ are both solvable in polynomial time. McNaughton [47] has shown that preemption cannot reduce $\sum C_j$ for a set of independent tasks. Thus, $P \mid pmtn \mid \sum C_j$ can also be solved by the SPT rule. Du et al. [19] have strengthened the result of McNaughton, showing that preemption cannot reduce $\sum C_j$ for a set of chains. Thus, $Pm \mid pmtn, chains \mid \sum C_j$ is NP-hard in the strong sense for each fixed $m \geq 2$.

This dissertation uses the Coffman-Graham algorithm as an approximation algorithm for $P \mid p_j = 1, prec \mid \sum C_j$ and shows that the Coffman-Graham algorithm has a worst-case bound of 2, which is also a tight bound. As noted above, the Coffman-Graham

algorithm is optimal for $P2 \mid p_j = 1, prec \mid \sum C_j$; it is optimal for the makespan objective as well. Lam and Sethi [39] have considered using the Coffman-Graham algorithm as an approximation algorithm for $P \mid p_j = 1, prec \mid C_{\max}$, and showed that it obeys a worst-case bound of $2 - 2/m$.

The algorithm for solving $Pm \mid p_j = 1, intree \mid \sum C_j$ [5] has running time $O(n^m)$, and hence it is impractical for large values of m. For this reason, this dissertation considers using approximation algorithms for the problem. In a search for reasonably good approximation algorithms for this problem, Hu's algorithm becomes a natural candidate since it is optimal for the makespan objective [29]. This dissertation shows that Hu's algorithm obeys a worst-case bound of 1.5, and that there are examples showing that the ratio can approach 1.308999.

1.2.4 Dual Criteria Scheduling Problems

Most of the single criterion scheduling problems are concerned with minimizing the total completion time, $\sum C_j$; the number of tardy jobs, $\sum U_j$; the maximum tardiness, $T_{\max} = \max\{T_j\}$; as well as the total tardiness, $\sum T_j$. Following the notation of Graham *et al.* [24], the above scheduling problems are denoted by $1 \mid\mid \sum C_j$, $1 \mid\mid \sum U_j$, $1 \mid\mid T_{\max}$, and $1 \mid\mid \sum T_j$, respectively.

It is well known that the SPT rule (shortest processing time first) gives a schedule with minimum total completion time. The SPT rule schedules jobs in ascending order of their processing times.

A schedule with minimum number of tardy jobs can be obtained by the Hodgson-Moore algorithm [48], which schedules jobs in ascending order of due dates. In the course of scheduling, if there is a job, say k, that completes after its due date, then the longest job currently in the schedule (including job k) will be deleted from the schedule. The deleted jobs will be scheduled after all the on-time jobs.

Maximum tardiness can be minimized by the EDD (earliest due date first) rule, which schedules jobs in ascending order of due dates. Maximum weighted tardiness can be solved by an algorithm due to Lawler [41], which actually solves a more general problem. Suppose each job j is subject to a nondecreasing penalty function $f_j(C_j)$ and the objective is to minimize $\max\{f_j(C_j)\}$. This problem can be solved as follows. For a single machine, there must be a job that completes at time $t = \sum p_j$. Choose the job j^* such that $f_{j^*}(t)$ is the smallest among all unscheduled jobs. Schedule job j^* to complete at time t. This reduces the problem to a set of $n - 1$ jobs to which the same rule applies. It can be shown that the schedule obtained has the smallest $\max\{f_j(C_j)\}$. Returning to the maximum weighted tardiness problem, define for each job j a penalty function $f_j(C_j)$, where $f_j(C_j)$ is defined as

$$f_j(C_j) = \begin{cases} 0 & \text{if } C_j \leq d_j \\ w_j(C_j - d_j) & \text{if } C_j > d_j \end{cases}$$

Clearly, $f_j(C_j)$ is a nondecreasing function. Thus, Lawler's algorithm can be applied to find a schedule with the minimum $\max\{w_jT_j\}$.

While the above three problems are solvable in polynomial time, unfortunately, minimizing total tardiness is binary NP-hard, as shown by Du and Leung [17].

So it is easy to see that the complexity of single criterion scheduling problems with these criteria have been solved. But when it comes to dual criteria scheduling problems with these criteria, which have more applications in the industrial areas, the complexity results are still open for some of them, where this dissertation is focused on.

As noted before, in this dissertation dual criteria scheduling problems are studied under the first approach, which is to have one criterion designated as the primary criterion and the other one designated as the secondary criterion. Extending the notation of Graham *et al.* [24], these problems can be expressed as $1 \mid\mid \gamma_2 \mid \gamma_1$, where γ_1 is the primary criterion and γ_2 is the secondary criterion.

As early as 1956, Smith [58] developed a polynomial-time algorithm for the problem $1 \mid\mid \sum C_j \mid T_{\max} = 0$. Heck and Roberts [26] extended the algorithm to solve $1 \mid\mid \sum C_j \mid T_{\max}$, while Emmons [21] further extended it to solve $1 \mid\mid \sum C_j \mid \max\{f_j(C_j)\}$, where $f_j(C_j)$ is an arbitrary nondecreasing penalty function for job j.

Many more results about primary and secondary criteria scheduling problems can be found in Chen and Bulfin [10], Dileepan and Sen [15] and Lee and Vairaktarakis [44]. The survey paper by Lee and Vairaktarakis [44] gave the complexity of many primary and secondary criteria scheduling problems. They noted that the complexity of the following problems remained open:

(1) $1 \mid\mid T_{\max} \mid \sum U_j$.

(2) $1 \mid\mid \sum U_j \mid T_{\max}$.

(3) $1 \mid\mid \max\{f_j(C_j)\} \mid \sum U_j$.

(4) $1 \mid\mid \sum U_j \mid \max\{f_j(C_j)\}$.

(5) $1 \mid\mid \sum C_j \mid \sum U_j$.

(6) $1 \mid\mid \sum T_j \mid \sum U_j$.

This dissertation mainly deals with the six dual criteria scheduling problems defined above.

Chapter 4 is concerned mainly with the dual criteria scheduling problems with the following criteria: the number of tardy jobs $\sum U_j$, the total completion time $\sum C_j$ and the total tardiness $\sum T_j$, which are problems (5) and (6) defined above. This dissertation shows that these two problems are NP-Hard.

Chapter 5 is concerned mainly with the dual criteria scheduling problems with the following criteria: the number of tardy jobs $\sum U_j$, the maximum tardiness $T_{\max}$ and the maximum weighted tardiness $\max\{w_j T_j\}$, which are the problem (1)-(4) defined above. This dissertation shows that problems (3) and (4) are NP-Hard even when the penalty

function f_j for each job j is simply the weighted tardiness of job j. Although much efforts have been invested in problems (1) and (2), their complexity remain open. Therefore, for problems (1) and (2), this dissertation considers complexity relationships between the various problems, gives polynomial-time algorithms for some special cases, and proposes fast heuristics for the general case. The effectiveness of the heuristics are measured by empirical study. The results show that one heuristic performs extremely well compared to optimal solutions.

CHAPTER 2

ONLINE ALGORITHMS

This chapter considers the problem of online scheduling a set of tasks on $m \geq 1$ identical and parallel processors so as to minimize $C_{\max}$. In these problems, tasks, along with their precedence constraints, are released at different times, and the scheduler has to make scheduling decision without knowledge of future releases.

It can be shown that optimal online scheduling algorithms exist for:

(1) $P2 \mid p_j = 1, prec_i \ released \ at \ r_i \mid C_{\max}$.

(2) $P \mid p_j = 1, outtree_i \ released \ at \ r_i \mid C_{\max}$.

(3) $P2 \mid pmtn, prec_i \ released \ at \ r_i \mid C_{\max}$.

(4) $P \mid pmtn, outtree_i \ released \ at \ r_i \mid C_{\max}$.

Using an adversary argument, one can show that it is impossible to have optimal online scheduling algorithms for:

(1) $P3 \mid p_j = 1, intree_i \ released \ at \ r_i \mid C_{\max}$.

(2) $P2 \mid p_j = p, chains_i \ released \ at \ r_i \mid C_{\max}$.

(3) $P3 \mid pmtn, p_j = 1, intree_i \ released \ at \ r_i \mid C_{\max}$.

These results give a sharp boundary delineating the possible and the impossible cases.

All of the optimal online scheduling algorithms follow the same format as the algorithm given in Hong and Leung [27]. For $P2 \mid p_j = 1, prec_i \ released \ at \ r_i \mid C_{\max}$, the Coffman-Graham algorithm is used to schedule tasks until new tasks arrive, at which time the remaining portions of the unfinished tasks, along with the newly arrived tasks, will

be rescheduled. Hu's algorithm is used for $P \mid p_j = 1, outtree_i \ released \ at \ r_i \mid C_{\max}$, and the Muntz-Coffman algorithm for $P2 \mid pmtn, prec_i \ released \ at \ r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i \ released \ at \ r_i \mid C_{\max}$.

The organization of this chapter is as follows. In Section 2.1, nonpreemptive scheduling is considered, while preemptive scheduling will be considered in Section 2.2. Finally, some conclusions will be drawn in the last section.

2.1 Nonpreemptive Schedules

In this section only nonpreemptive scheduling are considered. It is first shown that it is impossible to have optimal online algorithms for $P3 \mid p_j = 1, intree_i \ released \ at \ r_i \mid C_{\max}$ and $P2 \mid p_j = p, chains_i \ released \ at \ r_i \mid C_{\max}$. Then an optimal online algorithm is given for $P2 \mid p_j = 1, prec_i \ released \ at \ r_i \mid C_{\max}$ in Section 2.1.1 and an optimal online algorithm for $P \mid p_j = 1, outtree_i \ released \ at \ r_i \mid C_{\max}$ in Section 2.1.2.

Theorem 2.1.1 *It is impossible to have an optimal online algorithm for* $P3 \mid p_j = 1, intree_i \ released \ at \ r_i \mid C_{\max}$.

Proof: Adversary argument will be used to prove the theorem. Consider the intrees shown in Fig. 2.1: The number of processors is three, $intree_1$ is released at $r_1 = 0$ and $intree_2$ is released at $r_2 = 4$.

For $intree_1$, the length of the longest path is nine, so the makespan can not be smaller than nine. To obtain the minimum makespan, task 10 must finish by time $t = 4$, which means that all its predecessors must be finished by time $t = 3$. Since task 10 has nine predecessors and since there are only three processors, all of the predecessors of task 10 must be executed in the first three time units. This means that there must be an idle processor in the time interval $[3, 4]$. Now, if $intree_2$ is released at time $t = 4$, then the makespan must be larger than 10. As shown in Fig. 2.2, the optimal makespan is 10. On

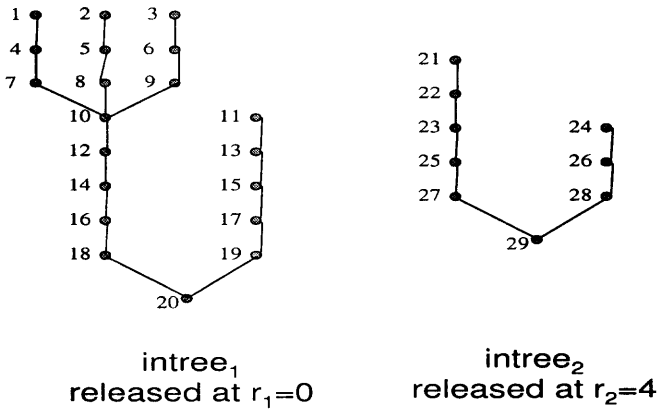


Figure 2.1 Example showing impossibility for $P3 \mid p_j = 1, intree_i \text{ released at } r_i \mid C_{\max}$.

the other hand, if task 10 is not completed by time $t = 4$, then the schedule is already not optimal for $intree_1$.

S1

0	1	2	3	4	5	6	7	8	9	10	11	12
1	4	7	10	12	14	16	17	18	20	**29**		
2	5	8	11	13	15	**24**	**25**	19	**28**			
3	6	9		**21**	**22**	**23**	**26**	**27**				

S2

0	1	2	3	4	5	6	7	8	9	10	11	12
1	3	5	7	10	12	14	16	18	20			
2	4	6	8	**21**	17	**23**	**25**	19	**29**			
11	13	15	9	**24**	**22**	**26**	**28**	**27**				

Figure 2.2 Schedule for the example in Fig. 2.1.

Thus, the adversary first releases $intree_1$ at time $t = 0$. If the online algorithm did not finish task 10 by time $t = 4$, then the schedule produced by the online algorithm is already not optimal for $intree_1$. On the other hand, if the online algorithm completes task 10 by time $t = 4$, then the adversary releases $intree_2$ at time $t = 4$. The online algorithm cannot finish both intrees by time $t = 10$, but the optimal makespan is 10. Again, the online algorithm did not produce an optimal schedule. ■

Theorem 2.1.2 *It is impossible to have an optimal online algorithm for* $P2 \mid p_j = p, chains_i$ *released at* $r_i \mid C_{\max}$.

Proof: Consider the chains shown in Fig. 2.3: Two chains released at $r_1 = 0$, one chain released at $r_2 = 5$, each task in the chains has two units of processing time, and the number of processors is two. The minimum makespan for the first two chains (released at time $t = 0$) is six, which can be attained only if both chains execute continuously from time $t = 0$ until time $t = 6$. Now, if the second chain is released at time $t = 5$, then the makespan will be 16. However, as shown in Fig. 2.3, the optimal makespan is 15.

Thus, the adversary first releases the two chains at time $t = 0$. If the online algorithm leaves a processor idle in the time interval $[0, 5]$, then the schedule is already not optimal for the two chains. On the other hand, if the online algorithm keeps both processors busy during the interval $[0, 5]$, then the adversary releases the second chain at time $t = 5$. The online algorithm cannot finish all the chains by time $t = 15$, but the optimal makespan is 15. Again, the online algorithm did not produce an optimal schedule. ■

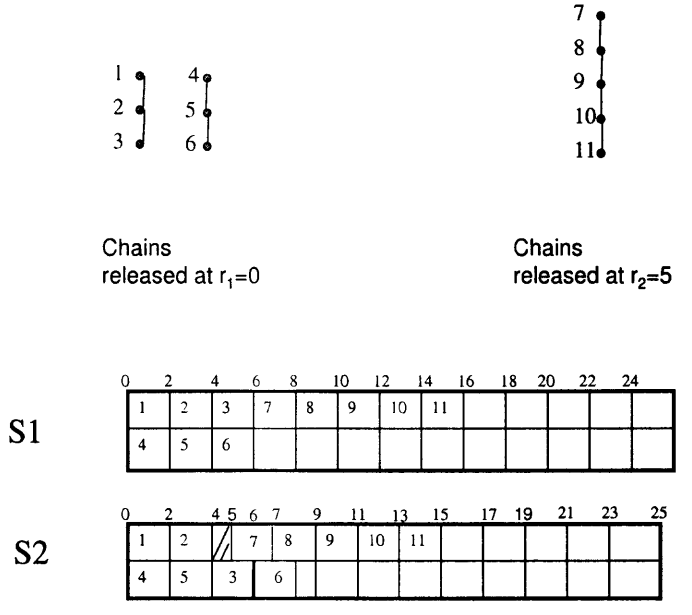


Figure 2.3 Example showing impossibility for $P2 \mid p_j = p, chains_i \text{ released at } r_i \mid C_{\max}$.

2.1.1 UET Tasks, Arbitrary Precedence Constraint and Two Processors

The online algorithm utilizes the Coffman-Graham algorithm to schedule tasks. When new tasks arrive, the new tasks along with the unexecuted portion of the unfinished tasks will be rescheduled by the Coffman-Graham algorithm again.

Algorithm A

Whenever new tasks arrive, do {

$t \leftarrow$ the current time;

$U \leftarrow$ the set of tasks active (i.e., not finished) at time t;

Call the Coffman-Graham algorithm to reschedule the tasks in U;

}

Example: Fig. 2.4 shows another set of tasks released at time $r_2 = 2$, after the tasks in Fig. 1.1 were released at time $r_1 = 0$. Note that tasks 4, 5, 7, 8, and 9 from the first release are unfinished at time $t = 2$. They are rescheduled, along with the new tasks from the second release, by the Coffman-Graham algorithm. The final schedule obtained by Algorithm A is also shown.

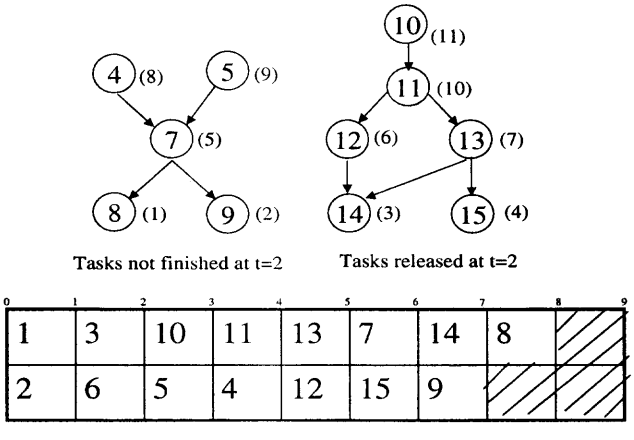


Figure 2.4 Example illustrating Algorithm A.

The next lemma, whose proof will be omitted, is instrumental in proving that Algorithm A is optimal.

Lemma 2.1.3 *Let S be a schedule for a set of tasks with arbitrary precedence constraints, where each task has unit processing time.*

1. *If S has the largest number of tasks completed at any time instant t, then S must be optimal for $C_{\max}$.*
2. *If S has the minimum idle processor time at any time instant t, then S must be optimal for $C_{\max}$.*
3. *For two processors, the schedule S' produced by the Coffman-Graham algorithm has the largest number of tasks completed at any time instant t.*

Theorem 2.1.4 *Algorithm A is optimal for $P2 \mid p_j = 1, prec_i$ released at $r_i \mid C_{\max}$. Moreover, the schedule produced by Algorithm A has the largest number of tasks completed at any time instant t.*

Proof: The theorem is proved by induction on the number, i, of release times. The basis case of $i = 1$ follows from Lemma 2.1.3. Assume the theorem is true for $i = k - 1$ release times, the following will show that the theorem is true for $i = k$ release times.

Let S_{k-1} denote the schedule obtained by Algorithm A after the first $k - 1$ releases. By the induction hypothesis, S_{k-1} is optimal for the tasks in the first $k - 1$ releases and it has the largest number of tasks completed at any time instant t. The release time r_k divides the tasks into two groups: (1) τ_1 — tasks completed by r_k in S_{k-1} and (2) τ_2 — tasks completed after r_k in S_{k-1}. Let S_k denote the schedule obtained by Algorithm A after the k^{th} release. By the nature of Algorithm A, S_k is identical to S_{k-1} from time 0 until r_k. Thus, every task in τ_1 is completed by r_k in S_k as well.

Let $\hat{S}_k$ be an optimal schedule for k releases. The release time r_k divides the tasks into two groups: (1) $\hat{\tau}_1$ — tasks completed by r_k in $\hat{S}_k$ and (2) $\hat{\tau}_2$ — tasks completed after r_k in $\hat{S}_k$. Let $prec_k$ denote all the tasks in the k^{th} release. It is clear that the tasks in $\tau_1 \cup \tau_2$ are the same as the tasks in $\hat{\tau}_1 \cup \hat{\tau}_2 \setminus prec_k$.

Now construct another schedule $\bar{S}_k$ from $\hat{S}_k$ as follows: (1) Delete all the tasks in $\tau_1 \cup \tau_2$ from $\hat{S}_k$; (2) Schedule all the tasks in τ_1 exactly as in S_{k-1}. The schedule $\bar{S}_k$ is identical to S_{k-1} from time 0 until r_k. After r_k, it has tasks in $prec_k$ scheduled exactly as in $\hat{S}_k$ and idle processor times due to the deletion of the tasks in $\tau_1 \cup \tau_2$.

It will be shown that the tasks in τ_2 can be scheduled into the idle processor times in $\bar{S}_k$ in such a way that the number of tasks completed at each time instant t is not smaller than that in $\hat{S}_k$. By Lemma 2.1.3, $\bar{S}_k$ has the same makespan as $\hat{S}_k$.

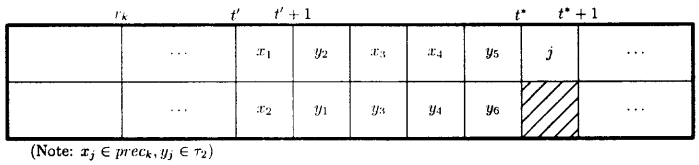
Let L be the list of tasks in τ_2 in ascending order of their completion times in S_{k-1}. The tasks in τ_2 are scheduled as follows. Whenever there is an idle processor time, scan the list L and assign the first ready task encountered in the scan to the idle time.

Keep assigning tasks by the above method until a time t^* is encountered such that both processors are idle in the time interval $[t^*, t^* + 1]$, j is the only task from τ_2 that can be assigned in the interval, and the number of tasks completed by $t^* + 1$ in $\bar{S}_k$ is smaller than that completed at the same time in $\hat{S}_k$. Since no tasks can be assigned in the interval, the remaining unassigned tasks in τ_2 must all be successors of task j. There are two cases to consider.

Case I: There is a time t', $r_k \leq t' < t^*$, such that both processors are executing some tasks in $prec_k$ in the time interval $[t', t' + 1]$. If there are several such times, let t' be the largest.

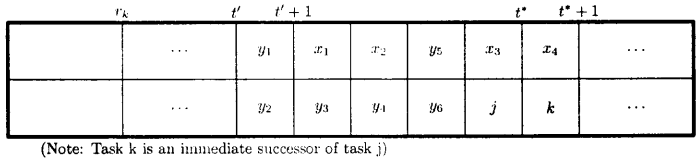
Shown in Fig. 2.5 (a) is an example of Case I. In this figure, x_j denotes a task in $prec_k$ and y_j denotes a task in τ_2. The schedule is transformed to the one shown in Fig. 2.5 (b). After the transformation, task j is completed by t^* and an immediate successor of j (task k shown in the figure) can now be scheduled in the time interval $[t^*, t^* + 1]$. It is clear that there is no precedence constraint violation in the transformed schedule.

Case II: In every time interval $[t, t + 1]$, $r_k \leq t < t^*$, at most one processor is executing some task in $prec_k$.



(Note: $x_j \in prec_k, y_j \in \tau_2$)

Fig(a) An example of Case I



(Note: Task k is an immediate successor of task j)

Fig(b) The transformed schedule

Figure 2.5 Example illustrating the proof of Case I.

In this case pull out all the tasks in $prec_k$ that were scheduled in the time interval $[r_k, t^* + 1]$ and reschedule all the tasks in τ_2 as in S_{k-1}. By the induction hypothesis, S_{k-1} has the largest number of tasks completed at any time instant and hence it can complete all the tasks by t^*. The schedule will be rearranged so that in every time interval $[t, t + 1]$, $r_k \leq t < t^*$, at least one processor is executing a task in τ_2; i.e., there is no time interval $[t, t + 1]$ such that both processors are idle. Now schedule the tasks in $prec_k$ into the idle processor times and an immediate successor of j in the time interval $[t^*, t^* + 1]$. It is clear that the schedule has no precedence constraint violation.

After the above operations are performed, the number of tasks completed by time $t^* + 1$ in $\bar{S}_k$ is identical to that in $\hat{S}_k$. Continue this operation until all tasks in τ_2 have been scheduled. Thus, the number of tasks completed at each time instant t is not smaller than that in $\hat{S}_k$.

Observe that S_k is identical to S_{k-1} from time 0 until r_k. Thus, it has the largest number of tasks completed at each time instant t up until r_k. After r_k, the tasks are scheduled by Coffman-Graham algorithm and hence S_k has the largest number of tasks completed at each time instant t. ■

2.1.2 UET Tasks, Outtrees and Arbitrary Number of Processors

The online algorithm utilizes Hu's algorithm to schedule tasks. When new tasks arrive, the new tasks along with the unexecuted portion of the unfinished tasks will be rescheduled by Hu's algorithm again.

Algorithm B

Whenever new tasks arrive, do {

$t \leftarrow$ the current time;

$U \leftarrow$ the set of tasks active (i.e., not finished) at time t;

Call Hu's algorithm to reschedule the tasks in U;

}

Example: Fig. 2.6 shows another outtree released at time $r_2 = 3$, after the outtree shown in Fig. 1.2 was released at time $r_1 = 0$. The schedule produced by Algorithm B is also shown.

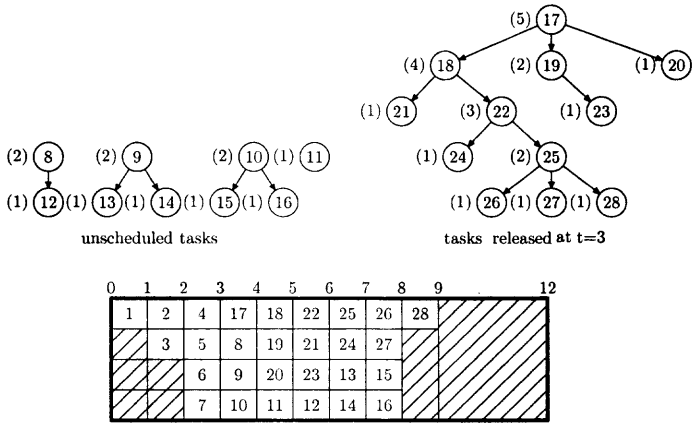


Figure 2.6 Example illustrating Algorithm B.

Theorem 2.1.5 *Algorithm B is optimal for* $P \mid p_j = 1, outtree_i$ *released at* $r_i \mid C_{\max}$. *Moreover, the schedule produced by Algorithm B has the largest number of tasks completed at any time instant t.*

Proof: Let S be the schedule produced by Algorithm B for an instance of the $P \mid p_j = 1, outtree_i\ released\ at\ r_i \mid C_{\max}$ problem and let $\hat{S}$ be an optimal schedule. It will be shown, by contradiction, that the number of tasks completed in S at each time instant t is not smaller than that in $\hat{S}$. By Lemma 2.1.3, S is an optimal schedule. Suppose not. Let t' be the first time instant such that the number of tasks completed in S is less than that in $\hat{S}$. Then there must be an idle processor in the time interval $[t'-1, t']$ in S.

Consider the schedule $\hat{S}$. Let k be the earliest completed task executed in $\hat{S}$ in the time interval $[0, t']$ that is not executed in S, and let k be completed at time t^* in $\hat{S}$. Assume that k is in $outtree_i$. It will be shown that k can be scheduled in the time interval $[t'-1, t']$ in S as well. Suppose not. Then there must be a predecessor of k, say j, executing in the time interval $[t'-1, t']$ in S. Let t be the first time instant such that predecessors of k are continuously executing from time t until t' in S, but that no predecessor of k is executing in the time interval $[t-1, t]$. There are two cases to consider.

Case I: $t = r_i$.

In this case, it is clear that k must be completed after t' in any schedule whatsoever, contradicting the assumption that k is completed by t' in $\hat{S}$.

Case II: $t > r_i$.

Let l be the predecessor of k executed in the time interval $[t, t+1]$ in S. According to Hu's algorithm, l was not executed in the time interval $[t-1, t]$ because the tasks executed in that time interval in S all have levels greater than or equal to that of l and that all processors are busy in the time interval. Since outtrees are considered, every processor must be busy from time $t-1$ until t', contradicting the fact that there is an idle processor in the time interval $[t'-1, t']$.

Repeating the above argument, it can be shown that the number of tasks completed by t' in S is not smaller than that in $\hat{S}$. ■

2.2 Preemptive Schedules

In this section only preemptive scheduling will be considered. It will first be shown that it is impossible to have an optimal online algorithm for $P3 \mid pmtn, p_j = 1, intree_i \; released \; at \; r_i \mid C_{\max}$. Then an optimal online algorithm for $P2 \mid pmtn, prec_i \; released \; at \; r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i \; released \; at \; r_i \mid C_{\max}$ will be given.

Theorem 2.2.1 *It is impossible to have an optimal online algorithm for* $P3 \mid pmtn, p_j = 1, intree_i \; released \; at \; r_i \mid C_{\max}$.

Proof: The proof given in Theorem 2.1.1 also proves this theorem since preemption won't help. ■

The Muntz-Coffman algorithm is optimal for the problems $P2 \mid pmtn, prec \mid C_{\max}$, $P \mid pmtn, intree \mid C_{\max}$, and $P \mid pmtn, outtree \mid C_{\max}$. It is essentially a highest-level-first strategy; see Section 1.1.1 for the definition of level.

Muntz-Coffman algorithm: Assign one processor each to the tasks at the highest level. If there is a tie among y tasks (because they are at the same level) for the last x ($x < y$) processors, then assign $\frac{x}{y}$ processor to each of these y tasks. Whenever either of the two events below occurs, reassign the processors to the unexecuted portion of the unfinished tasks according to the above rule. These are

1. A task is completed.

2. A point is reached where, if the present assignment were to continue, some tasks at a lower level would be executing at a faster rate than other tasks at a higher level.

The schedule produced by the Muntz-Coffman algorithm is a processor-sharing schedule. It can be converted to a preemptive schedule by marking the time instants where processor assignment change, and rescheduling the tasks executed between two adjacent time instants by McNaughton's wrap-around rule.

Example: Fig. 2.7 shows a set of tasks with their precedence constraints. Inside each circle is the name of the task and its processing time. The number next to each circle is the level of the task. The processor-sharing schedule on two processors produced by the Muntz-Coffman algorithm is shown. Finally, the preemptive schedule constructed from the processor-sharing schedule is also shown.

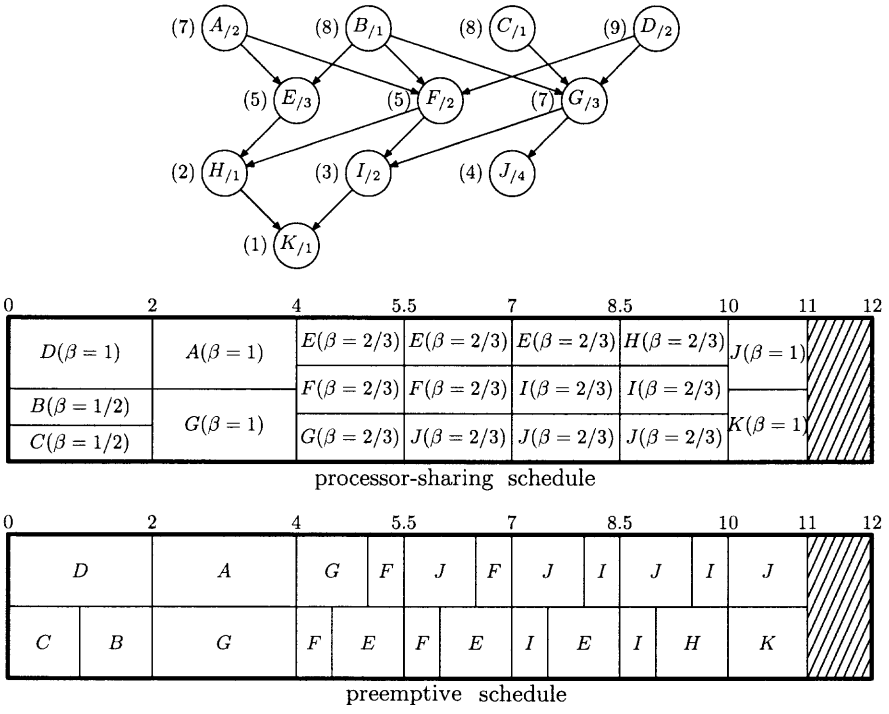


Figure 2.7 Example illustrating the Muntz-Coffman algorithm.

The online algorithm utilizes the Muntz-Coffman algorithm to schedule tasks. When new tasks arrive, the new tasks along with the unexecuted portion of the unfinished tasks will be rescheduled by the Muntz-Coffman algorithm again.

Algorithm C

Whenever new tasks arrive, do {

$t \leftarrow$ the current time;

$U \leftarrow$ the set of tasks active (i.e., not finished) at time t;

Call the Muntz-Coffman algorithm to reschedule the tasks in U;

}

Fig. 2.8 shows another set of tasks released at time $r_2 = 6$, after the tasks in Fig. 2.7 were released at time $r_1 = 0$. Algorithm C reschedules the unfinished portion of the unfinished tasks along with the new tasks. The processor-sharing schedule and the preemptive schedule are also shown.

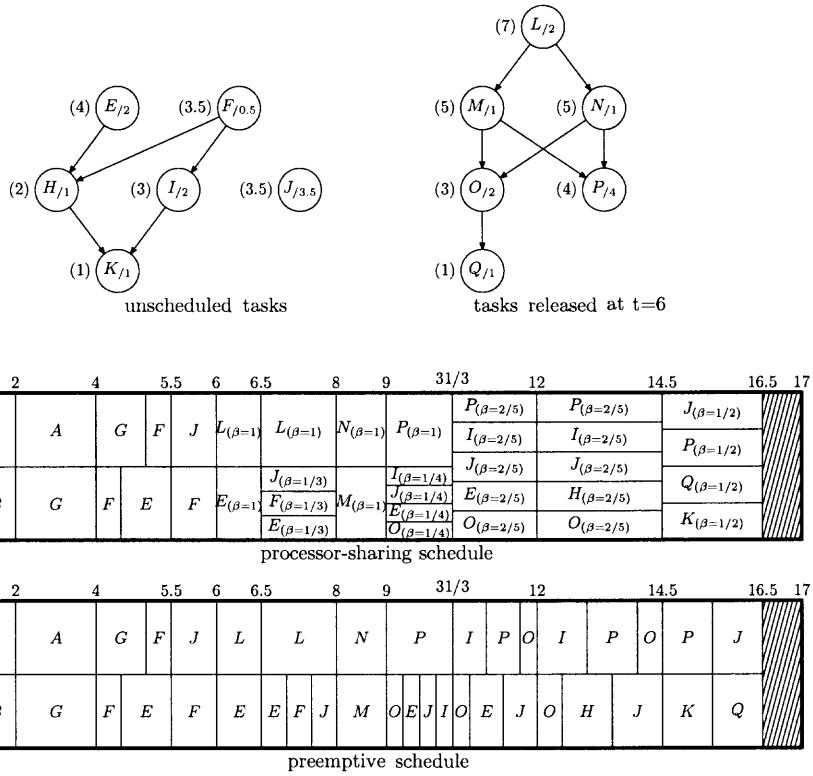


Figure 2.8 Example illustrating Algorithm C.

Before the proofs that Algorithm C is optimal for $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$ are given, an optimal offline algorithm for these two cases will be given.

Algorithm D: Assign one processor each to the tasks at the highest level. If there is a tie among y tasks (because they are at the same level) for the last x ($x < y$) processors, then

assign $\frac{x}{y}$ processor to each of these y tasks. Whenever one of the three events below occurs, reassign the processors to the unexecuted portion of the unfinished tasks according to the above rule. These are

1. A task is completed.
2. A point is reached where, if the present assignment were to continue, some tasks at a lower level would be executing at a faster rate than other tasks at a higher level.
3. New tasks arrive.

Algorithm D is essentially the Muntz-Coffman algorithm, except that another new event — when new tasks arrive — is added. In Sections 2.2.1 and 2.2.2, it will be shown that Algorithm D is an optimal offline algorithm for $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$, respectively. Shown in Fig. 2.9 are the processor-sharing schedule and the preemptive schedule constructed by Algorithm D for the instance given in Fig. 2.8.

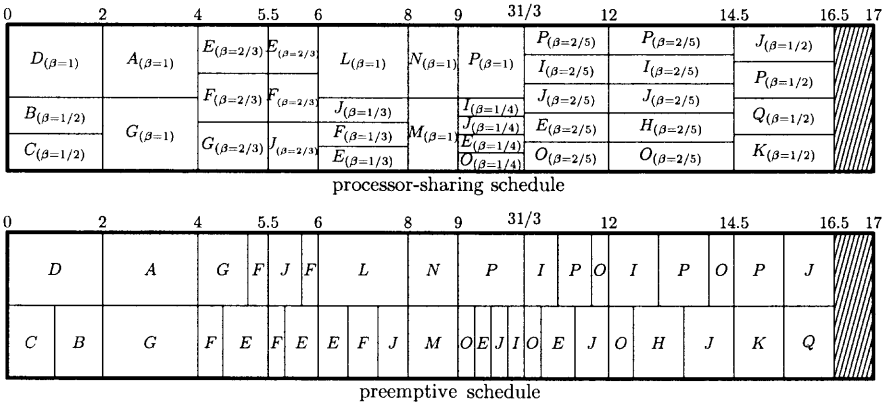


Figure 2.9 Example illustrating Algorithm D.

The proofs that Algorithm C is optimal for $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$ and $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$ are based on the fact that Algorithm D is an optimal offline algorithm for these two cases.

2.2.1 Arbitrary Precedence Constraint and Two Processors

Algorithm D will first be shown to be an optimal offline algorithm for $P2 \mid pmtn, prec_i$ *released at* $r_i \mid C_{\max}$, and then Algorithm C will be shown to be an optimal online algorithm for the same case.

Lemma 2.2.2 *Let S be the processor-sharing schedule produced by the Muntz-Coffman algorithm for an instance of $P2 \mid pmtn, prec \mid C_{max}$. Then, S has the minimum idle processor time at any time instant t.*

Proof: The lemma will be proved by contradiction. Let S be the processor-sharing schedule produced by the Muntz-Coffman algorithm for an instance of $P2 \mid pmtn, prec \mid C_{max}$, and let $\hat{S}$ be an optimal schedule for the same instance. Let t' be the first time instant such that the idle processor time in S is larger than that in $\hat{S}$. Then, there must be an idle processor in the time interval $[t', t' + \epsilon]$ in S, for some small positive number ϵ. According to the Muntz-Coffman algorithm, the reason that a processor is idle in $[t', t' + \epsilon]$ is that no task is ready in the interval other than those that are already executing in the interval. Let k be the earliest task executed in $\hat{S}$ in the time interval $[0, t' + \epsilon]$ but not in S, and let k started its execution at time t^* in $\hat{S}$. It will be shown that k can be scheduled in the time interval $[t', t' + \epsilon]$ in S as well. Suppose not. Then, there must be a predecessor of k, say j, executing in the time interval $[t', t' + \epsilon]$ in S. Let t be the first time instant such that predecessors of k are continuously executing from time t until $t' + \epsilon$ in S, but that no predecessor of k is executing in the time interval $[t - \delta, t]$ for some small positive number δ. There are two cases to consider.

Case I: $t = 0$.

If the predecessors of k are continuously executed either by one full processor or without sharing any processors with jobs that are not predecessors of k from time t until $t' + \epsilon$, then it is clear that k cannot be scheduled before $t' + \epsilon$ in any schedule whatsoever, contradicting the assumption that k is scheduled by $t' + \epsilon$ in $\hat{S}$. On the other hand, if some

predecessors of k are sharing processors with a set of other tasks, say U, then there must be at least one ready task at time t' that is either a task in U or a successor of task(s) in U, and hence there will be no idle processor in $[t', t' + \epsilon]$, contradicting the assumption that there is at least one idle processor.

Case II: $t > 0$.

Let l be the predecessor of k executed at time t in S. According to the Muntz-Coffman algorithm, l was not executed in the time interval $[t - \delta, t]$ because the tasks executed in that interval all have levels greater than that of l and that all processors are busy in the time interval. Since the tasks in the time interval $[t - \delta, t]$ are not predecessors of k, there must be at least one job other than l that is ready at time t. But this means that both processors are busy from time t until $t' + \epsilon$, contradicting the fact that there is an idle processor in the time interval $[t', t' + \epsilon]$.

Repeating the above argument, one can show that the idle processor time in S at each time instant t is less than or equal to that in $\hat{S}$. ■

Using the same technique as in Theorem 2.1.4 and the property given in Lemma 2.2.2, it can be shown that Algorithm D is an optimal offline algorithm for $P2 \mid pmtn, prec_i$ $released\ at\ r_i \mid C_{\max}$. This is stated in the next theorem whose proof will be omitted.

Theorem 2.2.3 *Algorithm D is an optimal offline algorithm for* $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$.

Theorem 2.2.4 *Algorithm C is an optimal online algorithm for* $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$.

Proof: Let S be the schedule produced by Algorithm C for an instance of $P2 \mid pmtn, prec_i\ released\ at\ r_i \mid C_{\max}$ and let $\hat{S}$ be the schedule produced by Algorithm D for the same instance. It will be shown, by induction on the number of release times, i,

that $\hat{S}$ can be converted into S without increasing the makespan and without violating any precedence constraints. Thus, S is an optimal schedule as well.

The basis case, $i = 1$, is obvious, since S and $\hat{S}$ are identical schedules. Assuming that the hypothesis is true for all $i \leq k - 1$, it will be shown that the hypothesis is true for $i = k$. Let $t_1 < t_2 < \ldots < t_l$ be the time instants where Event 1 or Event 2 occurs in $\hat{S}$. Let r_1 and r_2 denote the first and second release times, respectively. There are two cases to consider.

Case I: $r_2 = t_j$ for some $1 \leq j \leq l$.

From time r_1 until r_2, S is identical to $\hat{S}$. At time r_2, the remaining portions of the unfinished tasks in both schedules are identical. There are only $k - 1$ releases from r_2 onwards. By the induction hypothesis, $\hat{S}$ can be converted into S without violating any precedence constraints.

Case II: r_2 is in the time interval $[t_j, t_{j+1}]$ for some $1 \leq j < l$.

There are two cases to consider.

Case II(a): Every task executing in the time interval $[t_j, t_{j+1}]$ is executing on a full processor.

The proof of this case is identical to that of Case I.

Case II(b): Some tasks are sharing processor(s) in the time interval $[t_j, t_{j+1}]$.

Let x tasks be sharing y processor(s) $(x > y)$ in the time interval $[t_j, t_{j+1}]$. Let $i_1, i_2, \ldots, i_x$ be the tasks sharing the y processor(s). Since there are only two processors, there must be at most one task, say j_1, executing on a full processor. It is easy to see that from time r_1 until t_j, S and $\hat{S}$ are identical schedules, but from t_j until r_2, S and $\hat{S}$ may not be the same.

The second release time, r_2, divides each task $i_1, i_2, \ldots, i_x$, in S into two parts: those that were executed before r_2 and those that were executed after r_2. Let the level of a task, j, at time r_2 be denoted by $level(j)$. Then $level(j_1) > level(i_l)$ for each $1 \leq l \leq x$,

but the tasks $i_1, i_2, \ldots, i_x$ may have different levels. Consider now the schedule $\hat{S}$. then $level(j_1) > level(i_1) = level(i_2) = \ldots = level(i_x)$. Note that j_1 may not exist if the tasks $i_1, i_2, \ldots, i_x$ share two processors.

The schedule $\hat{S}$ is now converted into one such that it is identical to S in the interval $[r_1, r_2]$ and such that the makespan is not increased and no precedence constraints are violated. From time r_2 onward, there are $k-1$ releases. Thus, by the induction hypothesis, $\hat{S}$ can be converted into S from time r_2 onward. Hence, S is an optimal schedule as well.

From time r_1 until t_j, $\hat{S}$ and S are identical schedules, but they may not be the same in the time interval $[t_j, r_2]$. It will be shown that $\hat{S}$ can be converted in the interval $[t_j, r_2]$ to be identical to S in the same interval, without increasing the makespan and without violating any precedence constraints. There are two cases to consider.

Case (i): Several tasks are sharing two processors in the time interval $[t_j, t_{j+1}]$.

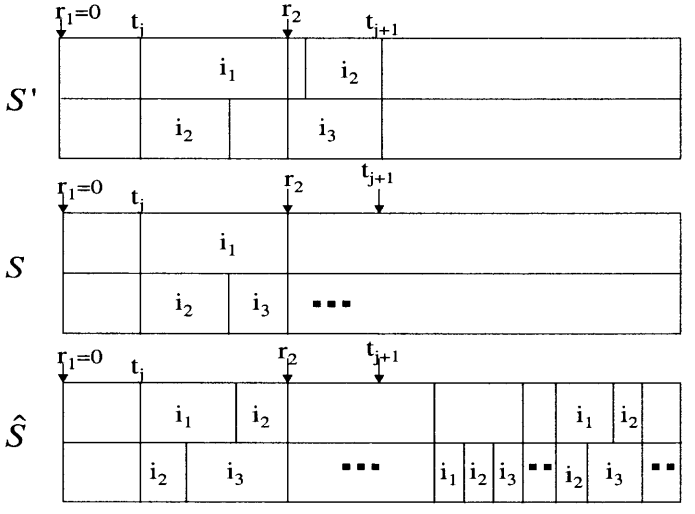


Figure 2.10 Example illustrating Case (i).

An example of this case is shown in Fig. 2.10: S' is the schedule produced by Algorithm C before tasks were released at r_2, S is the schedule obtained by Algorithm C after tasks were released at r_2, and $\hat{S}$ is the schedule produced by Algorithm D. It will be shown that the portions of the tasks scheduled in S' in the interval $[r_2, t_{j+1}]$ can be

rescheduled after time r_2 in $\hat{S}$, without increasing the makespan of $\hat{S}$ and without violating any precedence constraints.

Let $\hat{t}_1 < \hat{t}_2 < \ldots < \hat{t}_k$ be the time instants where Event 1, Event 2, or Event 3 occurs in $\hat{S}$. Define I_i to be the time interval $[\hat{t}_i, \hat{t}_{i+1}]$ in $\hat{S}$ for each $1 \leq i < k$. Consider how the tasks i_1, i_2 and i_3 are scheduled after r_2 in $\hat{S}$. There are three cases to consider: (a) i_1, i_2 and i_3 share one processor in the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$, but they are not scheduled in any other intervals; (b) i_1, i_2 and i_3 share two processors in the intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_q}$, but they are not scheduled in any other intervals; and (c) i_1, i_2 and i_3 share one processor in the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$, share two processors with other tasks in the intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_q}$, and they are not scheduled in any other intervals.

In Case (a), it is clear that the tasks can be scheduled in the interval $[r_2, t_{j+1}]$ in S' into the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$ in $\hat{S}$ without increasing the makespan and without violating any precedence constraints. In Case (b), the schedule can be divided in the interval $[r_2, t_{j+1}]$ in S' into q subintervals so that each subinterval will be scheduled into one of the intervals I_{j_x}, $1 \leq x \leq q$, in $\hat{S}$. Again, this will not increase the makespan or violate any precedence constraints. In Case (c), let l_0 be the total length of all the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$, and let l_x, $1 \leq x \leq q$, be the total execution time of the tasks i_1, i_2 and i_3 scheduled in I_{j_x}. The tasks can be scheduled in the interval $[r_2, r_2 + \frac{l_0}{2}]$ in S' into the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$ in $\hat{S}$. Then the interval $[r_2 + \frac{l_0}{2}, t_{j+1}]$ in S' will be divided into q subintervals: the x^{th} subinterval, $1 \leq x \leq q$, has length $\frac{l_x}{2}$. Take the tasks executed in S' in the x^{th} subinterval and schedule them in I_{j_x} in $\hat{S}$, along with the other tasks executed in the same interval. It is easy to see that the tasks can be rescheduled without increasing the makespan or violating any precedence constraints.

Using this approach, $\hat{S}$ in $[t_j, r_2]$ can always be converted to be identical to S in the same interval, without increasing the makespan of $\hat{S}$ and without violating any precedence constraints, no matter how many jobs are sharing the two processors in the time interval $[t_j, t_{j+1}]$.

Case (ii): One task is executing on a full processor while other tasks are sharing one processor in the time interval $[t_j, t_{j+1}]$.

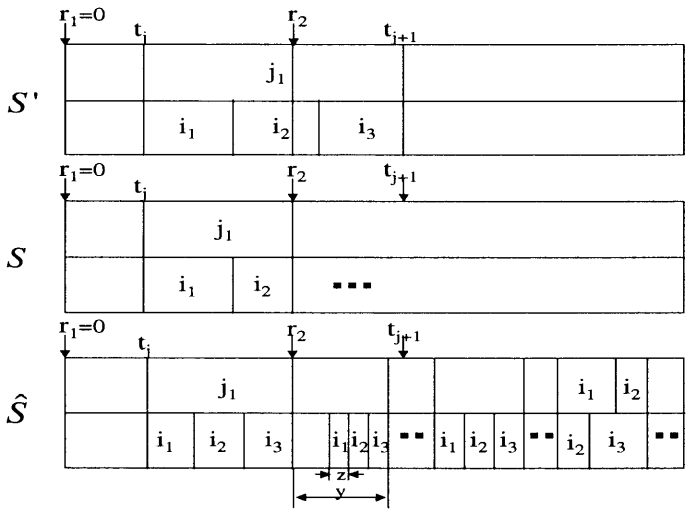


Figure 2.11 Example illustrating Case (ii).

An example of this case is shown in Fig. 2.11: S' is the schedule constructed by Algorithm C before new tasks were released at r_2, S is the schedule constructed by Algorithm C after new tasks were released at r_2, and $\hat{S}$ is the schedule produced by Algorithm D. Again, it will be shown that the portions of the tasks scheduled in S' in the interval $[r_2, t_{j+1}]$ can be rescheduled after time r_2 in $\hat{S}$, without increasing the makespan of $\hat{S}$ and without violating any precedence constraints.

As before, let $\hat{t}_1 < \hat{t}_2 < \ldots < \hat{t}_k$ be the time instants where Event 1, Event 2, or Event 3 occurs in $\hat{S}$. Define I_i to be the time interval $[\hat{t}_i, \hat{t}_{i+1}]$ in $\hat{S}$ for each $1 \leq i < k$. Consider how the tasks i_1, i_2 and i_3 are scheduled after r_2 in $\hat{S}$. There are two cases to consider: (a) i_1, i_2 and i_3 share one processor in the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$, but they are not scheduled in any other intervals; (b) i_1, i_2 and i_3 share one processor in the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$, share two processors with other tasks in the intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_q}$, and they are not scheduled in any other intervals.

In Case (a), it is clear that the tasks on the second processor in S' in the interval $[r_2, t_{j+1}]$ can be scheduled into the intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_p}$ in $\hat{S}$ without increasing the

makespan of $\hat{S}$ and without violating any precedence constraints. In Case (b), let $y = t_{j+1} - r_2$. Assume that in the time interval $[r_2, t_{j+1}]$, i_1, i_2, i_3 have each executed z units in $\hat{S}$, $0 \leq z \leq \frac{y}{3}$. This means that $y - 3z$ units of processor time are used by the newly released tasks in the same interval. It is easy to see that in $\hat{S}$, the remaining portions of i_1, i_2, i_3 at time t_{j+1} are all $\frac{y}{3} - z$. On the other hand, the lengths of the remaining portions of i_1, i_2, i_3 in S' in the interval $[r_2, t_{j+1}]$ are all different. Thus, after r_2, when the tasks i_1, i_2, i_3 are scheduled into the intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_q}$, it is quite possible that overlaps be created. It is clear that the total length of all the overlaps is no more than $\frac{y-3z}{2}$.

Assume that both processors are used by a task, say i_3, in the time interval $[\lambda_1, \lambda_2]$. From λ_1 back to r_2, the following method is used to eliminate the overlap: Find the first time interval $[t^*, t^* + \epsilon]$ such that: (1) one processor is used by i_1, i_2, i_3, j_1, or the successors of j_1, and the other processor, say the second processor, is used by other jobs, or (2) both processors are not used by i_1, i_2, i_3, j_1, or the successors of j_1. If (1) holds, then interchange the schedule on the second processor in $[t^*, t^* + \epsilon]$ with the schedule on the second processor in $[\lambda_2 - \epsilon, \lambda_2]$, so the overlap is reduced by ϵ. (Note that if i_3 is executing in the interval $[t^*, t^* + \epsilon]$, the interchange does not reduce the overlap but it has the effect of pushing backwards the time where overlap occurs.) If (2) holds, then interchange the schedule on the second processor in $[t^*, t^* + \epsilon]$ with the schedule on the second processor in $[\lambda_2 - \epsilon, \lambda_2]$, and interchange the schedule on the first processor in $[t^*, t^* + \epsilon]$ with the schedule on the second processor in $[\lambda_2 - 2 * \epsilon, \lambda_2 - \epsilon]$. Again, the overlap is reduced by ϵ and no precedence constraints are violated.

The above operation will be repeated until all of the overlaps are eliminated. Since in the time interval $[r_2, t_{j+1}]$, one processor has $y - 3z$ processor time used by the newly released tasks, the overlap can always be eliminated. Thus, the schedule in $[t_j, r_2]$ in $\hat{S}$ can be converted to be identical to S in the same interval, without increasing the makespan of $\hat{S}$ and without violating any precedence constraints.

Using this approach, the schedule in $[t_j, r_2]$ in $\hat{S}$ can always be converted to be identical to S in the same interval, without increasing the makespan and without violating any precedence constraints, no matter how many jobs are sharing one processor in the time interval $[t_j, t_{j+1}]$. ■

2.2.2 Outtrees and Arbitrary Number of Processors

Algorithm D is first proved to be an optimal offline algorithm for $P \mid pmtn, outtree_i$ *released at* $r_i \mid C_{\max}$, and then Algorithm C is proved to be an optimal online algorithm for the same case.

Theorem 2.2.5 *Algorithm D is an optimal offline algorithm for* $P \mid pmtn, outtree_i$ *released at* $r_i \mid C_{\max}$.

Proof: Let S be the schedule produced by Algorithm D for an instance of the $P \mid pmtn, outtree_i$ *released at* $r_i \mid C_{\max}$ problem and let $\hat{S}$ be an optimal schedule. It will be shown, by contradiction, that the idle processor time in S at each time instant t is less than or equal to that in $\hat{S}$. Thus, S is also an optimal schedule. Suppose not. Let t' be the first time instant such that the idle processor time in S is larger than that in $\hat{S}$. Then there must be an idle processor in the time interval $[t', t' + \epsilon]$ in S, for some small positive number ϵ. According to Algorithm D, the reason that a processor is idle in $[t', t' + \epsilon]$ is that no task is ready in the interval other than those that are already executing in the interval.

Consider the schedule $\hat{S}$. Let k be the task executed in $\hat{S}$ in the time interval $[0, t' + \epsilon]$ but not in S, and let k started its execution at time t^* in $\hat{S}$. Assume that k is in $outtree_i$. It will be shown that k can be scheduled in the time interval $[t', t' + \epsilon]$ in S as well. Suppose not. Then there must be a predecessor of k, say j, executing in the time interval $[t', t' + \epsilon]$ in S. Let t be the first time instant such that predecessors of k are continuously executing from time t until $t' + \epsilon$ in S, but that no predecessor of k is executing in the time interval $[t - \delta, t]$ for some small positive number δ. There are two cases to consider.

Case I: $t = r_i$.

If the predecessors of k are continuously executed by one full processor from time t until $t' + \epsilon$, then it is clear that k cannot be scheduled before $t' + \epsilon$ in any schedule whatsoever, contradicting the assumption that k is scheduled by $t' + \epsilon$ in $\hat{S}$. On the other hand, if some predecessors of k are sharing processors with other tasks, then there must be more ready tasks at time t' than the number of processors, and hence there will be no idle processor in $[t', t' + \epsilon]$, contradicting the assumption that there is at least one idle processor.

Case II: $t > r_i$.

Let l be the predecessor of k executed at time t in S. According to Algorithm D, l was not executed in the time interval $[t - \delta, t]$ because the tasks executed in that interval all have levels greater than that of l and that all processors are busy in the time interval. Since outtrees are being considered, every processor must be busy from time $t - \delta$ until $t' + \epsilon$, contradicting the fact that there is an idle processor in the time interval $[t', t' + \epsilon]$.

Repeating the above argument, one can show that the idle processor time in S at each time instant t is less than or equal to that in $\hat{S}$. Thus, S is also an optimal schedule. ■

Theorem 2.2.6 *Algorithm C is an optimal online algorithm for* $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$.

Proof: Let S be the schedule produced by Algorithm C for an instance of $P \mid pmtn, outtree_i\ released\ at\ r_i \mid C_{\max}$ and let $\hat{S}$ be the schedule produced by Algorithm D for the same instance. It will be shown, by induction on the number of release times, i, that the idle processor time in S is less than or equal to that of $\hat{S}$ at each time instant t. Thus, S is an optimal schedule as well.

The basis case, $i = 1$, is obvious, since S and $\hat{S}$ are identical schedules. Assuming that the hypothesis is true for all $i \leq k - 1$, it will be shown that the hypothesis is true for $i = k$. Let $t_1 < t_2 < \ldots < t_l$ be the time instants where Event 1 or Event 2 occurs in $\hat{S}$.

Let r_1 and r_2 denote the first and second release times, respectively. There are two cases to consider.

Case I: $r_2 = t_j$ for some $1 \leq j \leq l$.

From time r_1 until r_2, S is identical to $\hat{S}$. At time r_2, the remaining portions of the unfinished tasks in both schedules are identical. There are only $k - 1$ releases from r_2 onwards. By the induction hypothesis, the idle processor time in S is less than or equal to that of $\hat{S}$ at each time instant t after r_2.

Case II: r_2 is in the time interval $[t_j, t_{j+1}]$ for some $1 \leq j < l$.

There are two cases to consider.

Case II(a): Every task executing in the time interval $[t_j, t_{j+1}]$ is executing on a full processor.

The proof of this case is identical to that of Case I.

Case II(b): Some tasks are sharing processor(s) in the time interval $[t_j, t_{j+1}]$.

Let x tasks be sharing y processor(s) ($x > y$) in the interval $[t_j, t_{j+1}]$. Since tasks are sharing processors, the number of available tasks at time t_j must be greater than the number of processors. Let $i_1, i_2, \ldots, i_x$ be the tasks sharing the y processor(s) and let $j_1, j_2, \ldots, j_z$ be the tasks executing on a full processor. Clearly, $m = y + z$. It is easy to see that from time r_1 until t_j, S and $\hat{S}$ are identical schedules, but from t_j until r_2, S and $\hat{S}$ may not be the same.

The second release time, r_2, divides each task $i_1, i_2, \ldots, i_x$, in S into two parts: those that were executed before r_2 and those that were executed after r_2. Let the level of a task j at time r_2 be denoted by $level(j)$. Without loss of generality, assume that $level(j_1) \geq level(j_2) \geq \ldots \geq level(j_z)$. Then, $level(j_z) > level(i_l)$ for each $1 \leq l \leq x$, but the tasks $i_1, i_2, \ldots, i_x$ may have different levels. Without loss of generality, assume that $level(i_1) \geq level(i_2) \geq \ldots \geq level(i_x)$. Then, $level(j_1) \geq level(j_2) \geq \ldots \geq$

$level(j_z) > level(i_1) \geq level(i_2) \geq \ldots \geq level(i_x)$. Consider now the schedule $\hat{S}$. Then $level(j_1) \geq level(j_2) \geq \ldots \geq level(j_z) > level(i_1) = level(i_2) = \ldots = level(i_x)$.

Consider the schedule S' obtained from S by scheduling the remaining portions of the unfinished tasks at r_2 by Algorithm D. Since there are only $k-1$ release times (including r_2), by the induction hypothesis, the idle processor time in S is less than or equal to that of S' at each time instant t after r_2. Thus, if one can show that the idle processor time in S' is less than or equal to that of $\hat{S}$ at each time instant t after r_2, then the theorem is proved. This assertion will be proved by contradiction.

Let t^* be the first time instant where S' has more idle processor time than $\hat{S}$. Clearly, $t^* \geq r_2$. Let $[t^*, t^* + \epsilon]$ be the time interval where S' has more idle processor time than $\hat{S}$, for some small positive number ϵ. Clearly, S' must have some idle processors in the interval $[t^*, t^* + \epsilon]$. Let i^* be the earliest executed task that is executed in the time interval $[r_1, t^* + \epsilon]$ in $\hat{S}$ but not in S'. It can be shown that i^* can also be executed in the interval $[t^*, t^* + \epsilon]$ in S', contradicting the definition of i^*.

If i^* cannot be executed in the interval $[t^*, t^*+\epsilon]$ in S', then it must have a predecessor executed in the interval. Let i^* be a task in $outtree_l$, released at time r_l. There are two cases to consider.

Case (i): Task i^* is not the successor of any of the task i_h, $1 \leq h \leq x$.

Since i^* is not the successor of any of the task i_h, $1 \leq h \leq x$, the remaining portions of the predecessors of i^* after r_2 in S' must be identical to those in $\hat{S}$. Since $\hat{S}$ schedules i^* by $t^* + \epsilon$ while S' did not, there must be a time interval $[t', t' + \delta]$ such that either: (1) in S' the processors are all busy in the interval but none of the predecessors of i^* are executing in the interval, or (2) a predecessor of i^* is assigned less processor in the interval in S' than in $\hat{S}$. In both cases there must be more than m tasks ready for execution in the interval $[t', t'+\delta]$ in S'. This means that the processors are all busy from t' until $t^*+\epsilon$, contradicting the assumption that S' has some idle processors in the interval $[t^*, t^* + \epsilon]$.

Case (ii): Task i^* is the successor of the task i_h for some $1 \leq h \leq x$.

In this case, the remaining portions of the predecessors of i^* after r_2 in S' may not be the same as those in $\hat{S}$. If there is a time interval $[t', t' + \delta]$ such that either: (1) in S' the processors are all busy in the interval but none of the predecessors of i^* are executing in the interval, or (2) a predecessor of i^* is assigned less processor in the interval in S' than in $\hat{S}$, then one can resort to the same argument as in Case (i). Thus, one may assume that at each time instant after r_l, the predecessors of i^* are assigned the same or more processors in S' than in $\hat{S}$. In this case the only reason that i^* is executed in $\hat{S}$ but not in S' is that the remaining portion of task i_h after r_2 is larger in S' than in $\hat{S}$. Let i_h be finished at time $\bar{t}$ in S'. It is clear that every one of the tasks $i_1, i_2, \ldots, i_x$ must also be finished at $\bar{t}$. Furthermore, the tasks $j_1, j_2, \ldots, j_z$ are either finished at $\bar{t}$ or still active at $\bar{t}$, since they have higher levels than i_h. Thus, there are more active tasks than the number of processors. But this means that the processors are all busy from $\bar{t}$ until $t^* + \epsilon$, contradicting our assumption that S' has some idle processors in the interval $[t^*, t^* + \epsilon]$. ■

2.3 Concluding Remarks

In this chapter optimal online algorithms are given for the problems:

(1) $P2 \mid p_j = 1, prec_i \; released \; at \; r_i \mid C_{\max}$.

(2) $P \mid p_j = 1, outtree_i \; released \; at \; r_i \mid C_{\max}$.

(3) $P2 \mid pmtn, prec_i \; released \; at \; r_i \mid C_{\max}$.

(4) $P \mid pmtn, outtree_i \; released \; at \; r_i \mid C_{\max}$.

It is also shown that it is impossible to have optimal online algorithms for the problems:

(1) $P3 \mid p_j = 1, intree_i \; released \; at \; r_i \mid C_{\max}$.

(2) $P2 \mid p_j = p, chains_i \; released \; at \; r_i \mid C_{\max}$.

(3) $P3 \mid pmtn, p_j = 1, intree_i\ released\ at\ r_i \mid C_{\max}$.

Instead of the makespan objective, one wonders whether there are optimal online algorithms for the mean flow time objective. In this regard, it can be shown that Algorithm A is an optimal online algorithm for $P2 \mid p_j = 1, prec_i\ released\ at\ r_i \mid \sum C_j$, while Algorithm B is an optimal online algorithm for $P \mid p_j = 1, outtree_i\ released\ at\ r_i \mid \sum C_j$. The proof in Theorem 2.1.1 also shows that it is impossible to have optimal online algorithms for $P3 \mid p_j = 1, intree_i\ released\ at\ r_i \mid \sum C_j$ and $P3 \mid pmtn, p_j = 1, intree_i\ released\ at\ r_i \mid \sum C_j$. Relatively little is known about preemptive scheduling. For example, is it possible to have an optimal online algorithm for $P2 \mid pmtn, p_j = 1, prec_i\ released\ at\ r_i \mid \sum C_j$? Recently, Coffman *et al.* [14] gave an algorithm that simultaneously minimizes the makespan and the mean flow time for $P2 \mid pmtn, p_j = 1, prec_i\ released\ at\ r_i \mid \sum C_j$. Is it possible to adapt their algorithm to yield an optimal online algorithm for this case?

CHAPTER 3

FAST IMPLEMENTATION OF ALGORITHM

In the notation introduced by Graham et al. [24], the problems considered in this chapter are $P \mid p_j = 1, r_j, outtree \mid \sum C_j$ and $P \mid pmtn, p_j = 1, r_j, outtree \mid \sum C_j$.

Brucker, Hurink and Knust gave an $O(n^2)$-time algorithm for solving both $P \mid p_j = 1, outtree, r_j \mid \sum C_j$ and $P \mid pmtn, p_j = 1, outtree, r_j \mid \sum C_j$. This chapter shows that their algorithm admits an $O(n \log n)$-time implementation.

In the next section the algorithm of Brucker, Hurink and Knust [9] will be described, and $O(n \log n)$-time implementation will be given. In the last section some concluding remarks will be drawn.

3.1 The Algorithm

The algorithm of Brucker et al. assumes that the release times r_j are integers and compatible with the outtree precedence constraints; i.e., $r_i + 1 \leq r_j$ for all $i \prec j$. If the release times are not compatible with the outtree precedence constraints, one can modify the release times, without changing the problem, to satisfy the compatibility. This can be done in linear time by walking over the vertices of the outtree in a systematic way.

Their algorithm considers two relaxations of the problem $P \mid p_j = 1, r_j, outtree \mid \sum C_j$. In the first all precedence constraints are relaxed and a schedule S_1 is obtained for this version of the problem. In other words, the tasks have only release time constraints and the precedence constraints are ignored. The processor profile of S_1 is recorded in $m(t)$; i.e., at each time instant t, $m(t)$ records the number of processors used to schedule tasks at time t. In the second relaxation the number of processors (m) is replaced by the number of tasks (n) and a schedule S_2 is obtained for this version of the problem. It is clear that in the

second version every task can be executed as soon as it is released since there are always enough processors at each time instant. The processor profile of S_2 is recorded in $m'(t)$; i.e., at each time instant t, $m'(t)$ records the number of processors used in S_2 to schedule tasks. The algorithm then converts S_2 to fit the processor profile of S_1 (i.e., at each time instant t, exactly $m(t)$ processors will be used) in such a way that precedence constraints are observed. Brucker et al. showed that this can always be done since each task in an outtree has at most one immediate predecessor.

Let $\hat{r}_1 < \hat{r}_2 < \cdots < \hat{r}_x$ be the distinct release times of the n tasks and let $\hat{r}_{x+1} = \infty$. For each $1 \leq k \leq x$, let $\hat{S}_k$ denote the set of tasks with release time $\hat{r}_k$. If one sort the tasks in ascending order of the release times, one can compute $\hat{S}_k$ for all k in linear time. Clearly, the schedule S_2 will schedule all the tasks in $\hat{S}_k$ at the time instant $t = \hat{r}_k$. Therefore, $m'(\hat{r}_k) =\mid \hat{S}_k \mid$ for all $1 \leq k \leq x$ and $m'(t) = 0$ for all other t. The schedule S_1 can be obtained by the following algorithm.

Algorithm E

$S = \emptyset$;

For $i = 1$ to x do begin

$t \leftarrow \hat{r}_i$; $S \leftarrow S \cup \hat{S}_i$;

While $t < \hat{r}_{i+1}$ and $S \neq \emptyset$ do begin

$m(t) \leftarrow \min\{m, \mid S \mid\}$;

Schedule a set $S_t \subseteq S$ of $m(t)$ tasks at time t;

$S \leftarrow S \setminus S_t$; $t \leftarrow t + 1$;

End

End

It is clear that Algorithm E runs in linear time, assuming that the sets $\hat{S}_i$ have already been computed. Fig. 3.1 shows an instance of $P \mid p_j = 1, r_j, outtree \mid \sum C_j$. Fig. 3.2 shows the schedule S_1 obtained by Algorithm E and Fig. 3.3 shows the schedule S_2.

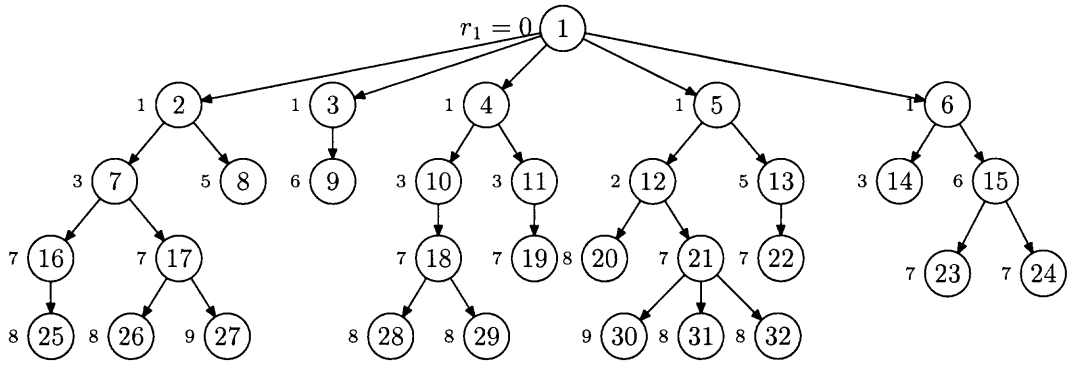


Figure 3.1 An instance of $P \mid p_j = 1, r_j, outtree \mid \sum C_j$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	6	7		8	9	16	21	20	28	30		
	3	12	10		13	15	17	22	25	29			
	4		11				18	23	26	31			
	5		14				19	24	27	32			

Figure 3.2 Schedule S_1 for the instance in Fig. 3.1.

The algorithm of Brucker et al. then transforms S_2 by iteratively moving jobs from left to right, using the processor profile of S_1 as a guide. The reader is referred to [9] for a description of the transformation. The transformation takes $O(n^2)$ time.

One can show that the transformation can be implemented in $O(n \log n)$ time. Assume that there is an array $SUCCNT$ such that $SUCCNT(i)$ stores the number of immediate successors of task i; $SUCCNT$ can be built as the outtree is input. Assume further that there is an array $PRED$ such that $PRED(i)$ gives the immediate predecessor of task i; i.e., $PRED(i) = j$ if task j is the immediate predecessor of task i. Again, $PRED$ can be built as the outtree is input. Let $t_1 < t_2 < \cdots < t_y$ be all the time instants such that $m(t_k) > 0$, $1 \leq k \leq y$. Note that $m(t_k)$ can be obtained as a byproduct of Algorithm E.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	12	7		8	9	16	20	27				
	3		10		13	15	17	25	30				
	4		11				18	26					
	5		14				19	28					
	6						21	29					
							22	31					
							23	32					
							24						

Figure 3.3 Schedule S_2 for the instance in Fig. 3.1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	6	7		8	9	16	21	20	29	27		
	3	12	10		13	15	17	22	25	30			
	4		11				18	23	26	31			
	5		14				19	24	28	32			

Figure 3.4 Transformed schedule using Algorithm B for the instance in Fig. 3.1.

The tasks will be scheduled backward, starting from t_y until t_1. Maintain a data structure, called $HEAP$, of available jobs (i.e., jobs that are ready to be scheduled).

A $HEAP$ is a data structure that implements a priority queue efficiently. Let S be a set of objects. Associated with each object is a KEY field that can be used for comparison purpose. A $HEAP$ of the set S is a binary tree in which every leaf is of depth d or $d-1$. Furthermore, every interior node has its KEY greater than or equal to that of its immediate successors. Thus, the root of the $HEAP$ is the largest element. For a set S of n elements, A $HEAP(S)$ can be built in linear time. Furthermore, one can insert an element into a $HEAP$ and still maintain the properties of a $HEAP$ in $O(\log n)$ time. One can also delete the largest element from the $HEAP$ and still maintain the $HEAP$ property in $O(\log n)$ time. For more details about $HEAP$, see [2].

A $HEAP(S)$ of available jobs S will be maintained. The KEY field of every job is its release time. Initially, S consists of all the jobs with $SUCCNT$ equal to zero. Then delete $m(t_y)$ jobs from $HEAP(S)$ and schedule them in the time unit t_y. For each deleted job j,

the $SUCCNT$ of its immediate predecessor k will be decremented. If the $SUCCNT$ of k becomes zero, insert k into $HEAP(S)$. The algorithm then move to time t_{y-1} to schedule jobs. This process is repeated until time t_1. The full algorithm is described below.

Algorithm F

// Assume that $t_1 < t_2 < \cdots < t_y$ are all the time instants such that $m(t_k) > 0$ for all $1 \leq k \leq y$. //

1. Let S be the set of tasks with $SUCCNT$ equal to zero; Build $HEAP(S)$;
2. $i = y$;
3. If $i = 0$, then stop;
4. Delete $m(t_i)$ tasks from $HEAP(S)$ and schedule the tasks in the time unit t_i;
5. For each task j scheduled in the time unit t_i, let k be the predecessor of j; $SUCCNT(k) \leftarrow SUCCNT(k) - 1$; If $SUCCNT(k) = 0$, then insert k into $HEAP(S)$;
6. $i \leftarrow i - 1$; goto 3;

Now examine the running time of Algorithm F. Step (1) takes $O(n)$ time. Inside the loop, each job gets inserted and deleted from the $HEAP$ exactly once. Since it takes $O(\log n)$ time to insert or delete, the entire algorithm takes $O(n \log n)$ time. Fig. 3.4 shows the transformed schedule using algorithm F.

3.2 Concluding Remarks

The algorithm of Brucker, Hurink and Knust gives a schedule that simultaneously minimizes both the $C_{\max}$ and the $\sum C_j$; such a schedule will be called an *ideal* schedule. The Coffman-Graham scheduling algorithm also yields an ideal schedule for two processors,

unit-processing-time, and arbitrary precedence constraints; see [13]. However, for three processors, unit-processing-time, and intree precedence constraints, there are instances for which no ideal schedule could possibly exist; see [32]. Coffman, Sethuraman and Timkovsky [14] recently gave an algorithm that produces ideal preemptive schedules for two processors, unit-processing-time, and arbitrary precedence constraints. It would be interesting to characterize the exact class that has ideal schedules.

CHAPTER 4

APPROXIMATION ALGORITHMS

In this chapter, the Coffman-Graham algorithm is used as an approximation algorithm for $P \mid p_j = 1, prec \mid \sum C_j$. It will be shown that the Coffman-Graham algorithm has a worst-case bound of 2, which is also a tight bound. As noted above, the Coffman-Graham algorithm is optimal for $P2 \mid p_j = 1, prec \mid \sum C_j$; it is optimal for the makespan objective as well. Lam and Sethi [39] have considered using the Coffman-Graham algorithm as an approximation algorithm for $P \mid p_j = 1, prec \mid C_{\max}$, and showed that it obeys a worst-case bound of $2 - 2/m$.

The algorithm for solving $Pm \mid p_j = 1, intree \mid \sum C_j$ [5] has running time $O(n^m)$, and hence it is impractical for large values of m. For this reason, approximation algorithms are used for the problem. In a search for reasonably good approximation algorithms for this problem, Hu's algorithm becomes a natural candidate since it is optimal for the makespan objective. It will be shown that Hu's algorithm obeys a worst-case bound of 1.5, and that there are examples showing that the ratio can approach 1.308999.

Recently, there have been some interests in schedules that simultaneously minimize both the makespan and the mean flow time; such a schedule will be called an *ideal* schedule. An interesting question is that for which type of precedence constraint and for which number of processors can one have ideal schedules? It is known that for two processors and arbitrary precedence constraints, the schedules produced by the Coffman-Graham algorithm are ideal schedules. For outtrees and arbitrary number of processors, the schedules produced by the algorithm of Brucker et al. [9] are also ideal schedules. On the other hand, the example given in Section 4.1 shows that there are no ideal schedules for intrees and three processors. All of the above assume that the tasks are unit-processing-time tasks. For preemptive scheduling, Coffman et al. [14] recently showed that there are ideal schedules

for two processors and arbitrary precedence constraints, assuming that the tasks are all unit-processing-time tasks.

The organization of this chapter is as follows. In Section 4.1, intrees are considered and Hu's algorithm is shown to have a worst-case bound no more than 1.5. In Section 4.2, it will be shown that the Coffman-Graham algorithm has a worst-case bound no more than 2. Finally, some concluding remarks are drawn in the last section.

For convenience, the following notation will be used throughout this chapter. In any schedule, a time unit is defined to be a *column*. A column during which all processors are busy will be called a *full column*. Columns that are not full columns will be called *partial columns*. If it is not clear from the context, $\sum C_j(S)$ is used to denote the mean flow time of the schedule S.

4.1 Intree Precedence Constraints

In this section, Hu's algorithm is considered as an approximation algorithm for $P \mid p_j = 1, intree \mid \sum C_j$. It will be shown that there are no ideal schedules for intree precedence constraints and three processors. Then Hu's algorithm will be shown to produce schedules with worst-case bound no more than 1.5, and examples are given showing that the ratio can approach 1.308999.

Example: Fig. 4.1 shows a set of tasks with intree precedence constraints. Shown in Fig. 4.2 is the schedule (S1) produced by Hu's algorithm on three processors. The mean flow time of S1 is 87. But the optimal mean flow time is 86, which is given by the schedule S2 in Fig. 4.2. From this example, one can see that there are no ideal schedules for intrees and three processors.

Although Hu's algorithm is not optimal for $P \mid p_j = 1, intree \mid \sum C_j$, it can still be used as an approximation algorithm. It will be shown that it obeys a worst-case bound no more than 1.5.

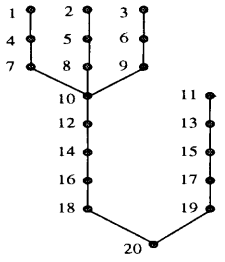


Figure 4.1 Example illustrating Hu's algorithm is not optimal.

S1

0	1	2	3	4	5	6	7	8	9	10	11	12
1	4	7	10	12	14	16	18	20				
2	5	8	11	13	15	17	19					
3	6	9										

Mean flow time: 3*1+3*2+3*3+4*2+5*2+6*2+7*2+8*2+9=87

S2

0	1	2	3	4	5	6	7	8	9	10	11	12
1	3	5	7	10	12	14	16	18	20			
2	4	6	8	17	19							
11	13	15	9									

Mean flow time: 3*1+3*2+3*3+4*3+5*2+6*2+7+8+9+10=86

Figure 4.2 Schedule for the example in Fig. 1.2.

For any instance of $P \mid p_j = 1, intree \mid \sum C_j$, let S denote the schedule produced by Hu's algorithm and let S^* denote an optimal schedule. Let $C_{\max}$ denote the makespan of S. First, some obvious characterizations of S will be given.

Property 4.1.1 *In S, the number of tasks scheduled in each time slot $[t, t+1]$ is non-increasing in t.*

Property 4.1.2 *In S, if the first column $[0, 1]$ is a partial column, then S is optimal.*

Property 4.1.3 *In S, if the first $C_{\max} - 2$ columns are full columns, then S is optimal.*

Suppose that the first t ($t \geq 1$) columns in S are all full columns but the $(t+1)^{st}$ column is a partial column. Then, from property 4.1.1, all the columns after the $(t+1)^{st}$ column are also partial columns. For any task in the time interval $[t, t+1]$, if it has no predecessor in the first t columns, then it can be moved backward to the time interval $[0, 1]$

and the moving distance of this task is t, which is the largest possible. Any task in the time interval $[t+i+1, t+i+2]$ $(i \geq 0)$ has at least one predecessor in the time interval $[t+i, t+i+1]$ and it must be scheduled after its predecessors, so it can be moved backward by at most t as well.

The improvement that can be made to S will be bounded in terms of the mean flow time. Clearly, the only improvement that can be made to S is to move the tasks scheduled in the $(t+1)^{st}$ column and thereafter to earlier columns. For each $1 \leq j \leq m$, let n_j be the number of tasks scheduled on the j^{th} processor in S. Without loss of generality, assume that for any processor, say the j^{th} processor, the task scheduled in S in the time interval $[t+i, t+i+1]$, $0 \leq i \leq n_j - t - 2$, is the predecessor of the task scheduled in the time interval $[t+i+1, t+i+2]$. If the tasks scheduled in the time interval $[t, n_j]$ on the j^{th} processor have no predecessors scheduled on any other processors, then these tasks can be moved backward to the time interval $[0, n_j - t]$. There are two cases to consider.

Case I: $n_j > 2t$.

If the tasks in the time interval $[t, n_j]$ were moved backward to the time interval $[0, n_j - t]$, then t tasks scheduled in the first t columns must be moved out of the first t columns to accommodate these tasks. The best place to which these t tasks are moved will be the idle processors in the time interval $[t, t+1]$. The net effect of the move is that the last t tasks on the j^{th} processor are moved to the idle processors in the time interval $[t, t+1]$.

Case II: $t \leq n_j \leq 2t$.

If the tasks in the time interval $[t, n_j]$ were moved backward to the time interval $[0, n_j - t]$, then $n_j - t$ tasks scheduled in the first t columns must be moved out of the first t columns to accommodate these tasks. The best place to which these $n_j - t$ tasks are moved will be the idle processors in the time interval $[t, t+1]$. The net effect of the move is that the last $n_j - t - 1$ tasks on the j^{th} processor are moved to the idle processors in the time interval $[t, t+1]$. Note that in this case the number of tasks moved is less than t, since $n_j \leq 2t$.

In both cases, the net effect is that at most t tasks will be moved to the idle processors in the time interval $[t, t+1]$.

Theorem 4.1.4 *For any instance of* $P \mid p_j = 1, intree \mid \sum C_j$, *let* S *denote the schedule produced by Hu's algorithm and let* S^* *denote an optimal schedule. Then*

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} \leq \frac{3}{2}.$$

Moreover, there are instances such that

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} = 1.308999.$$

Proof: S will be converted into a new schedule S', which has smaller mean flow time than S^* but may violate some precedence constraints. If one can show that

$$\frac{\sum C_j(S)}{\sum C_j(S')} \leq \frac{3}{2},$$

then one immediately obtains

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} \leq \frac{3}{2}.$$

S' is obtained from S as follows. Tasks in S are moved column by column, starting from the last column. Suppose the l^{th} column is being considered. If the previous column (i.e., the $(l-1)^{st}$ column) is full, then stop and S' is obtained already. Otherwise, for each processor that has less than t tasks moved before, move the task scheduled on the processor back to the first time instant s at which there is an idle processor. Iterate the above steps with the column previous to the l^{th} one; i.e., the $(l-1)^{th}$ column.

It is easy to see that S' has smaller mean flow time than S^*, but S' may violate some precedence constraints. It will now be shown that

$$\frac{\sum C_j(S)}{\sum C_j(S')} \leq \frac{3}{2}$$

.

For each $1 \leq j \leq m$, let n_j be the number of tasks scheduled on the j^{th} processor in S and let u_j be the number of tasks on the j^{th} processor that were moved in forming the schedule S'. It is clear that the new completion time of the task that was moved is greater than or equal to $t+1$. Let T_j be the total moving distance of the tasks on the j^{th} processor that were moved in forming S', and let $T = \sum_{j=1}^{m} T_j$. It is clear that $\sum C_j(S) - \sum C_j(S') = T$.

Consider the j^{th} processor. From S to S', the total moving distance of the tasks on this processor is

$$T_j \leq \sum_{i=1}^{u_j} (n_j - i + 1 - (t+1))$$

.

The total completion time of all the tasks on the j^{th} processor after the move is greater than or equal to

$$\sum_{i=1}^{n_j-u_j} i + u_j t \geq \sum_{i=n_j-2u_j+1}^{n_j-u_j} i + \sum_{i=n_j-3u_j+1}^{n_j-2u_j} i + u_j t$$

$$\geq \sum_{i=1}^{u_j} (n_j - i + 1 - u_j) + \sum_{i=1}^{u_j} (n_j - 2u_j - i + 1) + u_j t$$

$$\geq 2\sum_{i=1}^{u_j}(n_j - i + 1 - u_j) \geq 2T_j.$$

Thus, $\sum C_j(S') \geq 2T$. But $\sum C_j(S) - \sum C_j(S') = T$. Therefore,

$$\frac{\sum C_j(S)}{\sum C_j(S')} \leq \frac{3}{2}.$$

Instances for which the ratio can approach 1.308999 is now presented.

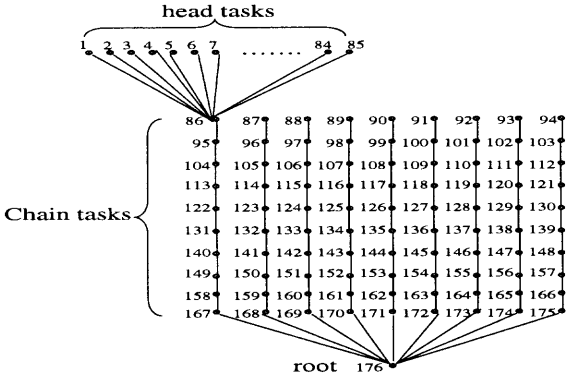


Figure 4.3 Example with $m = 17, a = 0$, and $k = 5$.

The example has $(k-a)m + (\frac{m+1}{2})2k + 1$ (m is an odd number) tasks. One task has level one and this task will be called the "root". There are $(\frac{m+1}{2})2k$ tasks divided into $\frac{m+1}{2}$ chains with $2k$ tasks in each chain and the level of these tasks range from 2 to $2k+1$. The root is the immediate successor of the tasks at level two in each chain. Call these $(\frac{m+1}{2})2k$ tasks the "chain tasks". Finally, there are $(k-a)m$ tasks at level $2k+2$ which is the highest level. These tasks are all immediate predecessor of the task at level $2k+1$ in the first chain. Call these tasks the "head tasks". Fig. 4.3 shows one such example with $m = 17, a = 0$, and $k = 5$.

The schedule produced by Hu's algorithm will have the "head tasks" scheduled in the first $k - a$ columns, the "chain tasks" scheduled on the first $\frac{m+1}{2}$ processors in the time

interval $[k-a, 3k-a]$, and the "root" scheduled on the first processor in the time interval $[3k-a, 3k-a+1]$. The schedule produced by Hu's algorithm for the example in Fig. 4.3 is shown in Fig. 4.4.

1	18	35	52	69	86	95	104	113	122	131	140	149	158	167	176				
2	19	36	53	70	87	96	105	114	123	132	141	150	159	168					
3	20	37	54	71	88	97	106	115	124	133	142	151	160	169					
4	21	38	55	72	89	98	107	116	125	134	143	152	161	170					
5	22	39	56	73	90	99	108	117	126	135	144	153	162	171					
6	23	40	57	74	91	100	109	118	127	136	145	154	163	172					
7	24	41	58	75	92	101	110	119	128	137	146	155	164	173					
8	25	42	59	76	93	102	111	120	129	138	147	156	165	174					
9	26	43	60	77	94	103	112	121	130	139	148	157	166	175					
10	27	44	61	78															
11	28	45	62	79															
12	29	46	63	80															
13	30	47	64	81															
14	31	48	65	82															
15	32	49	66	83															
16	33	50	67	84															
17	34	51	68	85															

Figure 4.4 Schedule by Hu's algorithm for the example in Fig. 4.3.

1	10	19	28	37	46	55	64	73	82	86	95	104	113	122	131	140	149	158	167	176
87	96	105	114	123	132	141	150	159	168										
88	97	106	115	124	133	142	151	160	169										
89	98	107	116	125	134	143	152	161	170										
90	99	108	117	126	135	144	153	162	171										
91	100	109	118	127	136	145	154	163	172										
92	101	110	119	128	137	146	155	164	173										
93	102	111	120	129	138	147	156	165	174										
94	103	112	121	130	139	148	157	166	175										
2	11	20	29	38	47	56	65	74	83										
3	12	21	30	39	48	57	66	75	84										
4	13	22	31	40	49	58	67	76	85										
5	14	23	32	41	50	59	68	77											
6	15	24	33	42	51	60	69	78											
7	16	25	34	43	52	61	70	79											
8	17	26	35	44	53	62	71	80											
9	18	27	36	45	54	63	72	81											

Figure 4.5 Optimal schedule for the example in Fig. 4.3.

An optimal schedule for this instance would be: Except the first chain, all other "chain tasks" are scheduled from the second processor to the $(\frac{m+1}{2})^{th}$ processor in the time interval $[0, 2k]$; the "head tasks" are scheduled on the other processors as fully as possible in the time interval $[0, 2(k-a)]$; the k "chain tasks" in the first chain are scheduled on the first processor in the time interval $[2(k-a), 2(k-a)+2k]$; the "root" is scheduled in the time interval $[2(k-a)+2k, 2(k-a)+2k+1]$. An optimal schedule for the example in Fig. 4.3 is shown in Fig. 4.5.

Let $C^*_{\max}$ denote the makespan of the optimal schedule. Comparing the schedule produced by Hu's algorithm and the optimal schedule, it is easy to see that when m is infinite, the ratio would approach 1.308999 when $a = 3820$ and $k = 10000$.

$$\frac{\sum C_j(S) - \sum C_j(S^*)}{\sum C_j(S^*)}$$

$$\leq \frac{(k+a)(k-a)\frac{m-1}{2} - (k-a)C^*_{\max} - \frac{(k-a)(k-a+1)}{2}}{\frac{m-1}{2}\sum_{i=1}^{2k} i + \frac{m-1}{2}\sum_{i=1}^{2(k-a)} i + \frac{C^*_{\max}(C^*_{\max}+1)}{2}}$$

$$\leq \frac{(m-1)\frac{(k^2-a^2)}{2} - (\frac{7}{2}k^2 + \frac{3}{2}k - 5ka + \frac{3a^2}{2} - \frac{3a}{2})}{\frac{m-1}{2}(4k^2 - 4ka + 2a^2 + 2k - a) + \frac{9k^2 - 6ka + a^2 + 9k - 3a + 2}{2}}$$

Let $M = \frac{m-1}{2}, \alpha 1 = k^2 - a^2 \geq 0, \alpha 2 = -(\frac{7}{2}k^2 + \frac{3}{2}k - 5ka + \frac{3a^2}{2} - \frac{3a}{2}) \leq 0,$

$\alpha 3 = 4k^2 - 4ka + 2a^2 + 2k - a \geq 0, \alpha 4 = \frac{9k^2 - 6ka + a^2 + 9k - 3a + 2}{2} \geq 0,$

then $\frac{(m-1)\frac{(k^2-a^2)}{2} - (\frac{7}{2}k^2 + \frac{3}{2}k - 5ka + \frac{3a^2}{2} - \frac{3a}{2})}{\frac{m-1}{2}(4k^2 - 4ka + 2a^2 + 2k - a) + \frac{9k^2 - 6ka + a^2 + 9k - 3a + 2}{2}}$

$= \frac{M\alpha 1 + \alpha 2}{M\alpha 3 + \alpha 4} = \frac{(M\alpha 3 + \alpha 4)\frac{\alpha 1}{\alpha 3} + \alpha 2 - \frac{\alpha 1 \alpha 4}{\alpha 3}}{M\alpha 3 + \alpha 4} = \frac{\alpha 1}{\alpha 3} + \frac{\alpha 2 - \frac{\alpha 1 \alpha 4}{\alpha 3}}{M\alpha 3 + \alpha 4} \leq \frac{\alpha 1}{\alpha 3}$

$= \frac{k^2 - a^2}{4k^2 - 4ka + 2a^2 + 2k - a}$

$= 0.308999.$

■

4.2 Arbitrary Precedence Constraints

In this section, the Coffman-Graham algorithm is used as an approximation algorithm for arbitrary precedence constraints. It will be shown that the Coffman-Graham algorithm obeys a worst-case bound of 2 and that the bound is tight.

For any instance of $P \mid p_j = 1, prec \mid \sum C_j$, let S denote the schedule produced by the Coffman-Graham algorithm and let S^* denote an optimal schedule. First, some simple characterizations of S will be given:

Property 4.2.1 *If there is no full column in S, then S must be optimal.*

Proof: It is easy to see that any task scheduled in the time slot $[t, t+1]$ $(t \geq 1)$ has a predecessor scheduled in $[t-1, t]$. So no task can be moved backward, and hence S is optimal. ■

Property 4.2.2 *For any task k scheduled in the time slot $[t, t+1]$, if there are f full columns and p partial columns before t, then task k can be moved backward by at most f time units.*

Proof: Clearly, $t = f + p$. For any task k scheduled in the time slot $[t, t+1]$, it must have a predecessor scheduled in the p^{th} partial column. For any task scheduled in the i^{th} $(2 \leq i \leq p)$ partial column, it must have a predecessor scheduled in the $(i-1)^{th}$ partial column. So, there must be a chain of length at least p before k. The tasks in the first partial column must be executed at or after time 0. So, task k must be executed at or after time $p+1$. Thus, task k can only be moved backward by at most $f + p + 1 - (p+1) = f$ time units. ■

The basic idea of proving the worst-case bound is identical to that in Section 4.1. The schedule S is converted to a new schedule S', which has smaller mean flow time than S^* but may violate precedence constraints and/or processor constraints. Then the improvement made to S' is bounded in terms of mean flow time. Finally, it will be shown that the mean flow time of S' must be greater than or equal to the improvement made to S'. Thus,

$$\sum C_j(S) - \sum C_j(S') \leq \sum C_j(S'),$$

and hence

$$\frac{\sum C_j(S)}{\sum C_j(S')} \leq 2.$$

Since the mean flow time of S' is smaller than that of S^*, one obtains

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} \leq 2.$$

The new schedule S' is obtained from S as follows. Starting from the first column in S, sequentially move the columns back some number of time units. Each column is moved back by one of the two rules described below. Suppose the l^{th} column is being considered.

- If the l^{th} column is a partial column and there are f full columns before it, from Property 4.2.2, the tasks in the l^{th} column can be moved backward by at most f time units. Then move these tasks backward by exactly f time units, regardless whether there are enough processors to execute these tasks.

- If the l^{th} column is a full column and there are f full columns before it, from Property 4.2.2, the tasks in the l^{th} column can be moved backward by at most f time units. Then move all the tasks in the l^{th} column backward by exactly f time units, regardless whether there are enough processors to execute these tasks. However, if another full column has been moved to the $(l-f)^{th}$ column before, then move forward from this point until a time unit is first encountered to which no previous full column had been moved. This will be the final destination of the l^{th} column. (Note that in this case the moving distance of the l^{th} column is less than f.)

It is easy to see that the mean flow time of S' must be less than the mean flow time of S^*. Moreover, S' may violate some precedence constraints and/or processor constraint. The next lemma gives a characterization of the moving distance of a full column.

Lemma 4.2.3 *When S' was obtained from S, if there are p partial columns before the $(f+1)^{th}$ full column, then the $(f+1)^{th}$ full column can be moved backward by at most $\min(f,p)$ time units.*

Proof: The lemma will be proved by induction on p. The basis case $p = 0$ is obvious. There is no partial column before the $(f+1)^{th}$ full column. It is easy to see that none of the full columns, up to and including the $(f+1)^{th}$ full column, will be moved backward at all. Now consider $p = 1$. There is one partial column before the $(f+1)^{th}$ full column. Assume that the partial column appears immediately before the $(f+1)^{th}$ full column. Since the full columns that appear before the partial column will not be able to move backward at all, the $(f+1)^{th}$ full column must be moved to where the partial column was; i.e. its moving distance is exactly one time unit. A full column immediately following the $(f+1)^{th}$ column can move to where the $(f+1)^{th}$ full column was; i.e., its moving distance is exactly one time unit as well.

Assume that the lemma is true for all $p < k$, it will be proved that the lemma is true for $p = k$. The $(f+1)^{th}$ full column has f full columns and p partial columns appearing before it. If $f \leq p$, then from Property 4.2.2, the $(k+1)^{th}$ full column can be moved backward by at most $f = \min(f,p)$ time units.

From the above, assume that $f > p$. Assume that the column before the $(f+1)^{th}$ full column is a partial column. Let there be p_1 partial columns before the f^{th} full column and p_2 partial columns between the f^{th} and the $(f+1)^{th}$ columns. Clearly, $p_1 < k$, $p_2 < k$, and $p = p_1 + p_2$. Since $f > p$, $f - 1 \geq p = p_1 + p_2$. Hence $f - 1 > p_1$. By the inductive hypothesis, the f^{th} full column can only be moved backward by at most $\min(f-1, p_1) = p_1$ time units. The only reason that it cannot be moved backward more (say by $f - 1$ time units) is that some full columns had been previously moved to those time units; i.e., they

have been occupied by some previous full columns. Therefore, the $(f+1)^{th}$ full column can only be moved to the time unit immediately after the time unit to which the f^{th} full column was moved. It is easy to see that the moving distance of the $(f+1)^{th}$ column is exactly $p_1 + p_2 = p$. If there were a full column immediately following the $(f+1)^{th}$ column, it will be moved to the time unit immediately following the one to which the f^{th} column was moved. Again, its moving distance is exactly p.

By mathematical induction, the lemma is true for all p. ■

Theorem 4.2.4 *For any instance of $P \mid p_j = 1, prec \mid \sum C_j$, let S denote the schedule produced by the Coffman-Graham algorithm and let S^* denote an optimal schedule. Then,*

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} \leq 2.$$

Moreover, there are instances of $P \mid p_j = 1, prec \mid \sum C_j$ for which the ratio approaches 2 arbitrarily closely.

Proof: The schedule S will be converted into a new schedule S' by the method as described above. If one can show that

$$\sum C_j(S) - \sum C_j(S') \leq \sum C_j(S'),$$

then the theorem is proved.

Let there be k full columns in S. Let t_j, $1 \leq j \leq k$, be the starting time of the j^{th} full column; i.e., the j^{th} full column is executed in the time interval $[t_j, t_j + 1]$. For each $1 \leq j \leq k-1$, define X_j to be the set of tasks executed in the time interval $[t_j + 1, t_{j+1}]$. Define X_k to be the set of tasks executed after $t_k + 1$. For each $1 \leq j \leq k$, let Y_j be the set of tasks executed in the j^{th} full column. For each task i, $1 \leq i \leq n$, define M_i to be the moving distance of task i from S to S'. There are two cases to consider.

Case 1: Any two full columns are separated by one or more partial column(s). That is, $t_j + 1 < t_{j+1}$ for any $1 \leq j \leq k-1$.

An example of this case is shown in Fig. 4.6. Arrow line from column i to j means that the tasks in column i will be moved to column j.

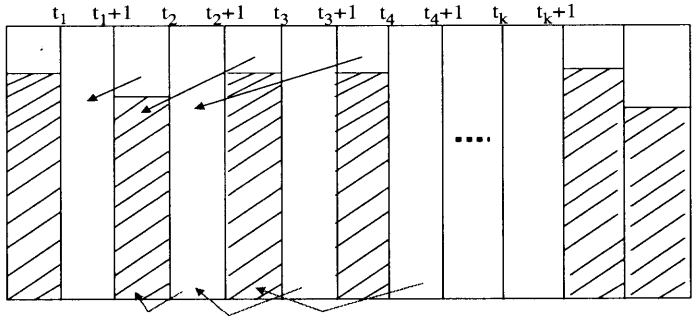


Figure 4.6 Example illustrating Case 1.

For any $1 \leq j \leq k-1$, the columns between $t_j + 1$ and t_{j+1} are all partial columns and the columns after $t_k + 1$ are all partial columns as well. Before $t_j + 1$ there are j full columns and at least $j-1$ partial columns. By Lemma 4.2.3, any task in the time interval $[t_j + 1, t_{j+1}]$ can be moved backward by at most j time units. After moving backward, their new completion times must be greater than or equal to j. On the other hand, any task in the j^{th} full column can be moved backward by at most $j-1$ time units, and their new completion times must be greater than or equal to $j-1$. Thus,

$$\sum_{i \in X_j} C_i(S') \geq \sum_{i \in X_j} M_i$$

and

$$\sum_{i \in Y_j} C_i(S') \geq \sum_{i \in Y_j} M_i.$$

Therefore,

$$\sum C_j(S) - \sum C_j(S') = \sum_{j=1}^{n} M_j \leq \sum_{j=1}^{n} C_j(S').$$

Case 2: There are full columns that appear contiguously.

For any group of full columns that appear contiguously, let the last full column of this group be the f^{th} full column in S and let there be p partial columns before this full column. There are two cases to consider.

Case 2(i): $f \leq p + 1$. (See Fig. 4.7)

The f^{th} full column lies in the time interval $[t_f, t_f + 1]$. The tasks in the time interval $[t_f + 1, t_{f+1}]$ can be moved backward by at most f time units and their new completion times must be greater than or equal to $p + 1$. The tasks in the f^{th} full column can be moved backward by at most $f - 1$ time units and their new completion times must be greater than or equal to $p + 1$. Thus, one can resort to the same argument as in Case 1.

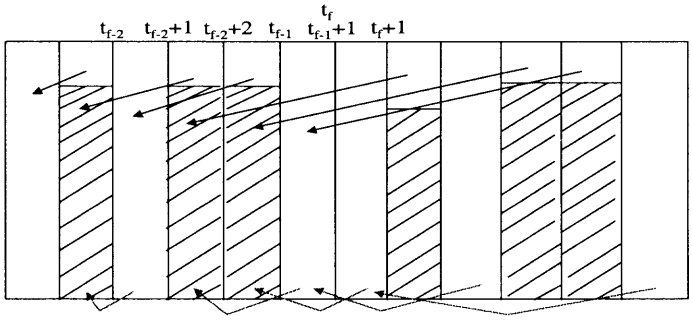


Figure 4.7 Example illustrating Case 2(i).

Case 2(ii): $f > p + 1$. (See Fig. 4.8)

Assume the f^{th} full column is the last full column in the first group of full columns that appear contiguously in S such that $f > p + 1$. Let the f^{th} full column completes at time t; i.e., the f^{th} full column is executed in the time interval $[t - 1, t]$. Let $f = p + q$. Then, $t = f + p = 2p + q$. From time t onward, locate the first time instant t^* such that

the number of full columns before t^* is exactly the number of partial columns before t^*. If t^* cannot be found before $C_{\max}$, add some empty columns after $C_{\max}$ as partial columns to get to t^*. Define P to be that part of the schedule S that includes all the tasks in the time interval $[t, t^*]$ as well as all the tasks in the p^{th}, $(p+1)^{th}$, ... , $(p+q)^{th}$ full columns. It will be shown that the total new completion times of the tasks in P is greater than or equal to the total moving distance of the tasks in P. Since the schedule after t^* in S resorts to Case 1, Case 2(i), or Case 2(ii), the theorem is proved.

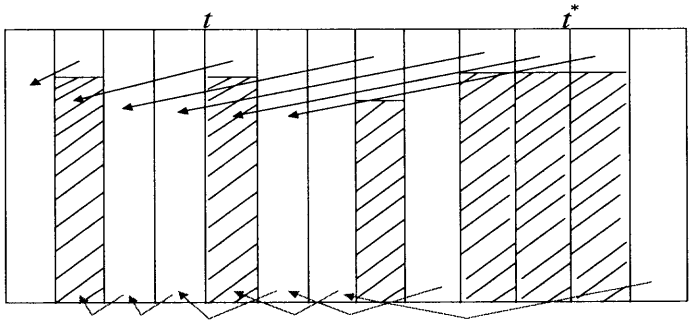


Figure 4.8 Example illustrating Case 2(ii).

The difference between the new completion time and the moving distance for partial columns will be computed separately from the full columns. For partial columns, the moving distance is larger than the new completion time, while the reverse is true for full columns. An upper bound for the sum of the differences between the moving distance and the new completion time for all the tasks in the partial columns in P will be computed, as well as a lower bound for the sum of the differences between the new completion time and the moving distance for all the tasks in the full columns in P. It will then be shown that the lower bound is greater than or equal to the upper bound. Thus,

$$\sum_{j \in P} C_j(S') \geq \sum_{j \in P} M_j = \sum_{j \in P} (C_j(S) - C_j(S')).$$

First, consider the partial columns. Assume there are x ($x \leq k - f$) full columns in the time interval $[t, t^*]$ (as shown in Fig. 4.9). Define U_i ($0 \leq i \leq x - 1$) to be the group of

partial columns between the $(f+i)^{th}$ full column and the $(f+i+1)^{th}$ full column, and U_x to be the group of partial columns after the $(f+x)^{th}$ full column in P. Let N_i $(0 \leq i \leq x)$ be the number of partial columns in U_i.

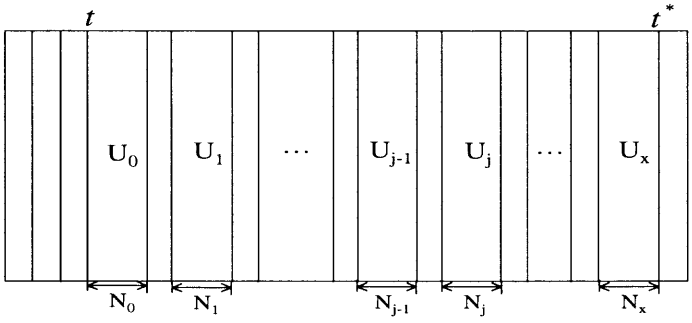


Figure 4.9 Illustrating the interval between t and t^*.

The computation of the difference between the moving distance and the new completion time for all the tasks in the partial columns in P is shown as follows:

U_0 is the group of partial columns in the time interval $[t, t+N_0]$. Any task in U_0 can be moved backward by f time units. The new completion time of any task scheduled in $[t, t+1]$ in S is $t+1-f$. So, the difference between the moving distance and the new completion time for these tasks is $f-(t+1-f) = 2f-(t+1) = f+p+q-(t+1) = q-1$. Similarly, the difference between the moving distance and the new completion time for any task in the interval $[t+N_0-1, t+N_0]$ is $2f-(t+N_0) = q-N_0$. Thus, the sum of the differences between the moving distance and the new completion time for all the tasks in the interval $[t, t+N_0]$ is less than

$$m(\sum_{u=1}^{N_0}(q-u)).$$

U_j is the group of partial columns in the time interval $[t+N_0+1+N_1+1+...+N_{j-1}+1, t+N_0+1+N_1+1+...+N_{j-1}+1+N_j]$. There are $f+j$ full columns before U_j. So, any task in U_j can be moved backward by $f+j$ time units. The new completion time

of any task in the first column of U_j is $t+N_0+1+N_1+1+...+N_{j-1}+2-(f+j)$. So, the difference between the moving distance and the new completion time for any task in the first column of U_j is $2(f+j)-(t+N_0+1+N_1+1+...+N_{j-1}+2)=q-N_0-N_1-...-N_{j-1}+j-1$. Similarly, the difference between the moving distance and the new completion time for any task in the last column of U_j is $2(f+j)-(t+N_0+1+N_1+1+...+N_{j-1}+1+N_j)=q-N_0-N_1-...-N_{j-1}-N_j+j$. Thus, the sum of the differences between the moving distance and the new completion time for all the tasks in U_j is less than

$$m(\sum_{u=N_0+...+N_{j-1}-(j-1)}^{N_0+...+N_j-j}(q-u)).$$

U_x is the group of partial columns in the time interval $[t+N_0+1+N_1+1+...+N_{x-1}+1, t+N_0+1+N_1+1+...+N_{x-1}+1+N_x]$. By similar computation, the sum of the differences between the moving distance and the new completion time for all the tasks in U_x is less than

$$m(\sum_{u=N_0+...+N_{x-1}-(x-1)}^{N_0+...+N_x-x}(q-u)).$$

Before t^*, the number of full columns is exactly the number of partial columns. Thus, $t^*=f+x+f+x=t+q+2x$. Since $t^*=t+N_0+1+N_1+1+...+N_{x-1}+1+N_x$, $N_0+...+N_x-x=q$. Thus, the sum of the differences between the moving distance and the new completion time for all the tasks in the partial columns of P is less than

$$m(\sum_{u=0}^{q-1}u+\sum_{j=0}^{x-1}(q-\sum_{i=0}^{j}N_i+j)).$$

Now consider the full columns. The computation of the difference between the new completion time and the moving distance for all the tasks in the full columns in P is shown as follows:

There are altogether p partial columns before t. The tasks in the $(p+i)^{th}$ $(1 \leq i \leq q)$ full column can be moved backward by at most p time units. The new completion time of the tasks in the $(p+i)^{th}$ full column is greater than or equal to $p+i$. The difference between the new completion time and the moving distance of the tasks in the $(p+i)^{th}$ full column is greater than or equal to i. Thus, the sum of the differences for the tasks in the $(p+1)^{th}$, $(p+2)^{th}$, ..., and $(p+q)^{th}$ full columns is greater than or equal to

$$m(\sum_{u=0}^{q-1} u).$$

The $(f+j)^{th}$ $(1 \leq j \leq x)$ full column is in the time interval $[t+N_0+1+\ldots+N_{j-1}, t+N_0+1+\ldots+N_{j-1}+1]$. There are $p+N_0+\ldots+N_{j-1}$ partial columns before this full column. So, the moving distance for the tasks in this column is $p+N_0+\ldots+N_{j-1}$. The new completion time for the tasks in this full column is $t+N_0+1+\ldots+N_{j-1}+1-(p+N_0+\ldots+N_{j-1}) = f+j$. Thus, the sum of the differences between the new completion time and the moving distance of the tasks in this full column is $m(f+j-(p+N_0+\ldots+N_{j-1})) = m(q-(N_0+\ldots+N_{j-1})+j)$. Thus, the sum of the differences between the new completion time and the moving distance of all the tasks in all the full columns in P is greater than or equal to

$$m(\sum_{u=0}^{q-1} u + \sum_{j=0}^{x-1}(q - \sum_{i=0}^{j} N_i + j)).$$

Comparing the upper bound for the partial columns and the lower bound for the full columns, one sees that the total new completion time for the tasks in P is greater than or equal to the total moving distance for the tasks in P.

Now, the bound will be shown to be tight. Lam and Sethi (Lam and Sethi [1977]) gave an example for $P \mid p_j = 1, prec \mid C_{\max}$, showing that the Coffman-Graham schedule has makespan approaching $2 - 2/m$ times the optimal makespan. The same example also shows that the Coffman-Graham schedule has mean flow time approaching two times the optimal mean flow time.

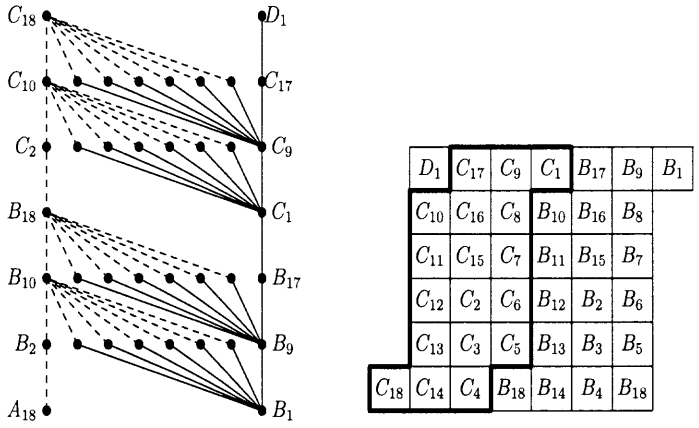


Figure 4.10 Worst case example of Coffman-Graham algorithm.

In their example, there are k groups of tasks. In each group there are $\frac{m^2}{2}$ tasks: one task at the lowest level, $m+2$ tasks at each of the next $\frac{m}{2}-1$ levels, and the remaining tasks at the highest level. Fig. 4.10 shows such an example for $m = 6$. Tasks $B_1, B_2, \ldots, B_{18}$ form one group. It is easy to verify that one possible labeling for the tasks in Fig. 4.10 would be to assign labels $1, 2, \ldots$ to $A_{18}, B_1, B_2, \ldots, B_{18}, C_1, \ldots, C_{18}, D_1, \ldots$, in this order. So, the Coffman-Graham schedule will be like: one time unit for each level with two tasks, and two time units for each level with $m + 2$ tasks. An optimal schedule is shown in Fig. 4.10 as well.

Now, compute the mean flow time for the Coffman-Graham schedule and the optimal schedule, respectively. Divide the Coffman-Graham schedule into k parts and each part has $m - 1$ columns. Define TC_i to be the mean flow time of the i^{th} part. The mean flow time of the first part is: $TC_1 = 2 + \frac{m^2}{2}(\frac{m}{2} - 1) + (m - 2) + m(\frac{m}{2} - 1)$. The mean flow time of

the schedule produced by Coffman-Graham algorithm is:

$$\sum C_j(S)$$

$$= TC_1 + TC_2 + ... + TC_k$$

$$= TC_1 + (m-1)\tfrac{m^2}{2} + TC_1 + 2(m-1)\tfrac{m^2}{2} + TC_1 + ...$$

$$+(k-1)(m-1)\tfrac{m^2}{2} + TC_1 + 2(k(m-1)+1)$$

$$= kTC_1 + \tfrac{k(k-1)}{2}(m-1)\tfrac{m^2}{2} + 2(k(m-1)+1)$$

$$= \tfrac{k^2m^3}{4} - \tfrac{k^2m^2}{4} + \tfrac{km^2}{4} + 2km - 2k + 2$$

The mean flow time of the optimal schedule is:

$$\sum C_j(S^*) = 1 + \sum_{i=2}^{\frac{km}{2}+1} mi + \frac{km}{2} + 2 = \frac{k^2m^3}{8} + \frac{3km^2}{4} + \frac{km}{2} + 3$$

Therefore,

$$\frac{\sum C_j(S)}{\sum C_j(S^*)} = \frac{\frac{k^2m^3}{4} - \frac{k^2m^2}{4} + \frac{km^2}{4} + 2km - 2k + 2}{\frac{k^2m^3}{8} + \frac{3km^2}{4} + \frac{km}{2} + 3} \leq 2,$$

and when $k \to \infty$, $m \to \infty$, "=" applies. ■

4.3 Concluding Remarks

In this chapter, Hu's algorithm has been proposed as an approximation algorithm for the problem $P \mid p_j = 1, intree \mid \sum C_j$ and the Coffman-Graham algorithm as an approximation algorithm for the problem $P \mid p_j = 1, prec \mid \sum C_j$. It has been shown that Hu's algorithm

obeys a worst-case bound no more than 1.5, and there are examples showing that the ratio can approach 1.308999. The Coffman-Graham algorithm is shown to have a worst-case bound of 2 and that the bound is tight.

The complexity of $P \mid p_j = 1, intree \mid \sum C_j$ has been open for a long time and it still remains open. For future research, it will be extremely rewarding to settle this issue. The bound given for Hu's algorithm is not tight. It will be desirable to tighten the bound. Ideal schedules are also interesting. Is it possible to characterize those instances that have ideal schedules?

CHAPTER 5

COMPLEXITY OF TWO DUAL CRITERIA SCHEDULING PROBLEMS

In this chapter, dual criteria scheduling problems with the following criteria are considered: the number of tardy jobs $\sum U_j$, the total completion time $\sum C_j$ and the total tardiness $\sum T_j$. In the notation introduced by Graham et al. [24], the problems considered in this chapter are $1 \mid\mid \sum C_j \mid \sum U_j$ and $1 \mid\mid \sum T_j \mid \sum U_j$.

In 1975, Emmons [20] studied the problem $1 \mid\mid \sum C_j \mid \sum U_j$. He proposed a branch-and-bound algorithm which in the worst case runs in exponential time. Complexity question was not addressed in [20]. Later, Chen and Bulfin [10] proved that the problem is NP-hard with respect to id-encoding. In id-encoding, jobs with the same characteristics are represented only once, and the number of jobs with the same characteristics is represented by a binary number. Notice that id-encoding scheme has the effect of significantly reducing the size of the input, making the problem harder to solve in polynomial time as a function of the size of the input. The complexity of the problem under standard encoding schemes remained open until now. In this chapter, the problem will be shown to be NP-Hard.

Vairaktarakis and Lee [59] studied the problem $1 \mid\mid \sum T_j \mid \sum U_j$. They gave a polynomial-time algorithm when the set of tardy jobs is specified. As well, a branch-and-bound algorithm was given for the general problem. Chen and Bulfin [10] mentioned that the complexity of this problem is open. In this chapter, it will be shown that this problem is also NP-Hard.

The NP-Hardness proofs are obtained by reductions from the Even-Odd Partition problem, which is known to be NP-complete (see Garey and Johnson [22] and Garey *et al.* [23]).

EVEN-ODD PARTITION Given $2n$ positive integers $a_1 < a_2 < \ldots < a_{2n}$ where $A = \frac{1}{2}\sum_{j=1}^{2n} a_j$, is there a partition of the integers into two sets, A_1 and A_2, such that

$$\sum_{j \in A_1} a_j = \sum_{j \in A_2} a_j = A$$

where A_1 and A_2 each contains exactly one element of each pair $\{a_{2i-1}, a_{2i}\}$, $i = 1, 2, \ldots, n$?

Notice that since each pair of integers, a_{2i-1} and a_{2i}, must be put into two different sets, one can add a constant c_i to each pair without changing the problem instance. By carefully choosing c_i, one may assume that the given instance of Even-Odd Partition satisfies the following properties:

$$a_1 > (2n+2) \cdot \max_{1 \le i \le n} (a_{2i} - a_{2i-1}) \tag{5.1}$$

$$a_{2i-1} > \sum_{j=1}^{2i-2} a_j = \sum_{j=1}^{i-1} a_{2j} + \sum_{j=1}^{i-1} a_{2j-1} \tag{5.2}$$

The organization of this article is as follows. In Sections 5.1 and 5.2, the NP-Hardness proofs for $1 \mid\mid \sum C_j \mid \sum U_j$ and $1 \mid\mid \sum T_j \mid \sum U_j$, respectively, will be be given. In the last section, some concluding remarks will be drawn.

5.1 Minimizing Total Completion Time Subject to Minimum $\sum U_j$

In this section, it will be shown that $1 \mid\mid \sum C_j \mid \sum U_j$ is NP-Hard. The decision version of this problem will be shown to be NP-Complete by reducing the Even-Odd Partition problem to it. Given an instance of the Even-Odd Partition problem, $a_1 < a_2 < \ldots < a_{2n}$, where $A = \frac{1}{2}\sum_{j=1}^{2n} a_j$, create an instance I of the scheduling problem as follows. There are $2n$ P-jobs each of which corresponds to an integer in the Even-Odd Partition instance, n *small* Q-jobs and a *large* R-job. The processing times and due dates of these jobs are shown in Table 5.1, where

$$x_i = a_{2i-1} - (2n - i + 2)(a_{2i} - a_{2i-1}) \tag{5.3}$$

for $1 \leq i \leq n$, and L is an integer greater than $2A$.

job	processing time	due date
P_{2i-1}	a_{2i-1}	$\sum_{k=1}^{i-1} a_{2k} + \sum_{k=1}^{i-1} x_k + a_{2i-1}$
P_{2i}	a_{2i}	$\sum_{k=1}^{i-1} a_{2k} + \sum_{k=1}^{i} x_k + a_{2i}$
Q_i	x_i	$\sum_{k=1}^{i-1} a_{2k} + \sum_{k=1}^{i} x_k + a_{2i-1}$
R	L	$\sum_{k=1}^{n} x_k + A + L$

Table 5.1 The Processing Times and Due Dates of the Jobs in Instance I

Let the threshold for the total completion time be B, where

$$B = \sum_{j=1}^{n} a_{2j} \cdot (3n + 2 - 2j) + \sum_{j=1}^{n} a_{2j-1} \cdot (n - j + 1) + \sum_{j=1}^{n} x_j \cdot (3n + 3 - 2j) + (n+1)\,L + \frac{1}{2}\sum_{j=1}^{n}(a_{2j} - a_{2j-1}) \ .$$

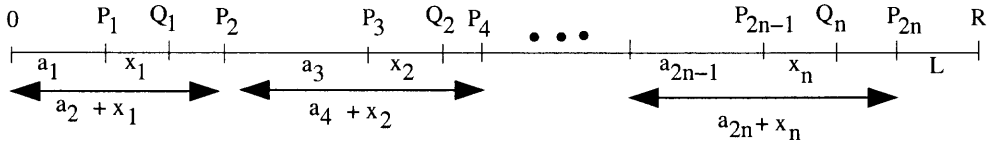


Figure 5.1 Illustration of the due dates of jobs in instance I.

Figure 5.1 shows the due date pattern of the jobs. Call a schedule *feasible* if it has the minimum number of tardy jobs. The decision problem asks: is there a feasible schedule with total completion time less than or equal to B?

The basic idea of the reduction is to create a P-job for each integer a_j, n small Q-jobs each of which has a due date between a pair of P-jobs, and a large R-job whose due date is the largest among all the jobs. By properly choosing the processing times and due dates of the jobs, one can show that in any feasible schedule: (a) Exactly one job from each pair

$\{P_{2i-1}, P_{2i}\}$ must be tardy; (b) The Q-jobs must be on time; (c) The R-job must be on time and is scheduled after all the other on-time jobs and before any tardy jobs; (d) The total processing time of the on-time P-jobs cannot exceed A. However, to minimize total completion time, one needs to have more even P-jobs to be on time. It can be shown that every time a pair of even and odd P-jobs is interchanged by making the even P-job tardy and the odd P-job on time, the total completion time is increased by a quantity equal to the difference between the processing times of the two jobs, which is exactly the quantity reduced in the total processing time of the on-time P-jobs. Thus, the optimal solution is obtained when the total processing time of the on-time P-jobs is exactly A. But this occurs only when there is a solution to the instance of the Even-Odd Partition problem. Notice that the first three terms in the formula for B represent the total completion time when all even P-jobs are on time (which does not yield a feasible schedule since the R-job will be tardy). The last term is the minimum increase in total completion time when the total processing time of the on-time P-jobs is reduced to A (which yields a feasible schedule since the R-job will then be on time).

The next three lemmas prove the assertions made above.

Lemma 5.1.1 *In any feasible schedule for the instance I, (a) there are exactly n tardy jobs, (b) the R-job is on time and is scheduled after all the other on-time jobs, and (c) at least one job from each pair $\{P_{2i-1}, P_{2i}\}$ must be on time.*

Proof: (a) is first proved, i.e., there must be n tardy jobs in any feasible schedule. As mentioned in the Introduction, the Hodgson-Moore algorithm yields a schedule with the minimum number of tardy jobs. Thus, it is sufficient to show that there are exactly n tardy jobs when the Hodgson-Moore algorithm is applied to the instance I.

Recall that Hodgson-Moore algorithm schedules jobs in increasing order of their due dates. In the course of scheduling, if a job misses its due date, then the job with the largest processing time among all jobs that are currently in the schedule (including the job that misses its due date), will be picked out as a tardy job and deleted from the schedule.

Continue to schedule the next job until all jobs have been processed. Finally, schedule all the tardy jobs (that were deleted from the schedule) at the end, in any order.

For the instance I, $d_{P_{2i-2}} < d_{P_{2i-1}} < d_{Q_i} < d_{P_{2i}}$ and $d_{P_{2n}} < d_R$. Therefore, the jobs would be scheduled in the order of P_1, Q_1,P_2, P_3, Q_2, P_4, ... ,P_{2n-1}, Q_n, P_{2n}, R by the Hodgson-Moore algorithm. It is easy to see that P_1 and Q_1 both meet their due dates, but P_2 will not. By (5.1) and (5.3), $x_1 < a_1 < a_2$. Hence, P_2 will be chosen as a tardy job. Suppose all the jobs P_{2j-1}, Q_j and P_{2j} have been scheduled, where $1 \leq j \leq i-1$, and all the even P-jobs, P_{2j}, $1 \leq j \leq i-1$, had been chosen as tardy jobs and discarded from the schedule. If P_{2i-1}, Q_i and P_{2i} are now scheduled sequentially, then the completion times will be

$$C_{P_{2i-1}} = \sum_{j=1}^{i-1} a_{2j-1} + \sum_{j=1}^{i-1} x_j + a_{2i-1} < \sum_{j=1}^{i-1} a_{2j} + \sum_{j=1}^{i-1} x_j + a_{2i-1} = d_{P_{2i-1}} \ ,$$

$$C_{Q_i} = \sum_{j=1}^{i-1} a_{2j-1} + \sum_{j=1}^{i} x_j + a_{2i-1} < \sum_{j=1}^{i-1} a_{2j} + \sum_{j=1}^{i} x_j + a_{2i-1} = d_{Q_i} \ ,$$

and

$$\begin{aligned} C_{P_{2i}} &= \sum_{j=1}^{i-1} a_{2j-1} + \sum_{j=1}^{i} x_j + a_{2i-1} + a_{2i} \\ &> \sum_{j=1}^{i} x_j + a_{2i-1} + a_{2i} \\ &> \sum_{j=1}^{i} x_j + \sum_{j=1}^{i-1} a_{2j} + a_{2i} \qquad \text{by (5.2)} \\ &= d_{P_{2i}} \ . \end{aligned}$$

Since P_{2i} misses its due date and it has the largest processing time among all the jobs currently in the schedule, P_{2i} will be chosen as a tardy job. Therefore, the Hodgson-Moore algorithm will pick all the even P-jobs as tardy jobs. For the R-job, the completion time

will be

$$\sum_{j=1}^{n} x_j + \sum_{j=1}^{n} a_{2j-1} + L < \sum_{j=1}^{n} x_j + A + L = d_R \ ,$$

so it is on time. Hence, the total number of tardy jobs is n. Thus, any feasible schedule for the instance I must have exactly n tardy jobs.

Since the R-job has a large processing time, a job scheduled after the R-job must miss its due date. Hence, all the other on-time jobs must be scheduled before the R-job. Thus, (b) also holds.

(c) can be proved by contradiction: at least one job from each pair $\{P_{2i-1}, P_{2i}\}$ is on time. Suppose $\{P_{2i-1}, P_{2i}\}$ is the first pair that are both tardy in a feasible schedule S. Consider now applying the Hodgson-Moore algorithm to the job set consisting of all Q-jobs, all P-jobs except $\{P_{2i-1}, P_{2i}\}$, and the R-job. As shown in the proof of (a), all the even P-jobs, P_{2j}, $j < i$, will be picked as tardy jobs by the Hodgson-Moore algorithm. If $\{P_{2i+1}, Q_{i+1}, P_{2i+2}\}$ is scheduled, then P_{2i+1} and Q_{i+1} will still be on time. However, the completion time of P_{2i+2} is

$$\sum_{j=1}^{i-1} a_{2j-1} + \sum_{j=1}^{i+1} x_j + a_{2i+1} + a_{2i+2} > \sum_{j=1}^{i+1} x_j + a_{2i+1} + a_{2i+2} \ .$$

By (5.2), $a_{2i+1} > \sum_{j=1}^{i} a_{2j}$. Thus, P_{2i+2} will miss its due date. Since P_{2i+2} has the largest processing time among all jobs currently in the schedule, it will be chosen as a tardy job. Using the same argument, one can show that all even P-jobs are tardy. By assumption, P_{2i-1} is also a tardy job. Thus, the total number of tardy jobs will be $n + 1$, contradicting the assumption that S is a feasible schedule. ■

Lemma 5.1.2 *In any optimal schedule, exactly one job from each pair* $\{P_{2i-1}, P_{2i}\}$, $1 \leq i \leq n$, *is tardy.*

Proof: It will be shown by contradiction that one of P_{2i-1} and P_{2i} must be tardy in any optimal schedule. Suppose both are on time in an optimal schedule S. By the proof

in Lemma 5.1.1, one job in $\{P_{2i-1}, Q_i, P_{2i}\}$ must be tardy. Since P_{2i-1} and P_{2i} are both on time, Q_i must be tardy. Consider now interchanging Q_i with P_{2i-1} (i.e., make P_{2i-1} tardy and Q_i on time) to get a new schedule S'. By (5.1) and (5.3), $x_i < a_{2i-1}$. On the other hand, $d_{Q_i} > d_{P_{2i-1}}$. So, Q_i meets its due date in S'. However, S' has a smaller total completion time than S, contradicting the assumption that S is optimal. ■

Lemma 5.1.3 *In any optimal schedule S, no tardy job can be scheduled before the R-job.*

Proof: By Lemma 5.1.1, the on-time P-jobs and Q-jobs are all scheduled before the R-job. The total processing time of these jobs is at least $\sum_{j=1}^{n} a_{2j-1} + \sum_{j=1}^{n} x_j$. Suppose there is a tardy job scheduled before the R-job. By Lemma 5.1.2, it must be a P-job. Suppose this job is P_m, where $1 \leq m \leq 2n$. Then the completion time of the R-job would be

$$
\begin{aligned}
& \sum_{j=1}^{n} a_{2j-1} + \sum_{j=1}^{n} x_j + L + a_m \\
> & \sum_{j=1}^{n} a_{2j-1} + \sum_{j=1}^{n} x_j + L + a_1 \\
> & \sum_{j=1}^{n} a_{2j-1} + \sum_{j=1}^{n} x_j + L + \sum_{j=1}^{n} (a_{2j} - a_{2j-1}) \qquad \text{by (5.1)} \\
= & \sum_{j=1}^{n} a_{2j} + \sum_{j=1}^{n} x_j + L \\
> & A + \sum_{j=1}^{n} x_j + L \ .
\end{aligned}
$$

Thus, the R-job will miss its due date. By Lemma 5.1.1, S can not be a feasible schedule. ■

According to Lemmas 5.1.1 and 5.1.2, in any optimal schedule, all the on-time jobs must be scheduled before any tardy jobs. One can easily show that the on-time jobs must be scheduled in increasing order of their due dates in order to be on time. For the tardy jobs,

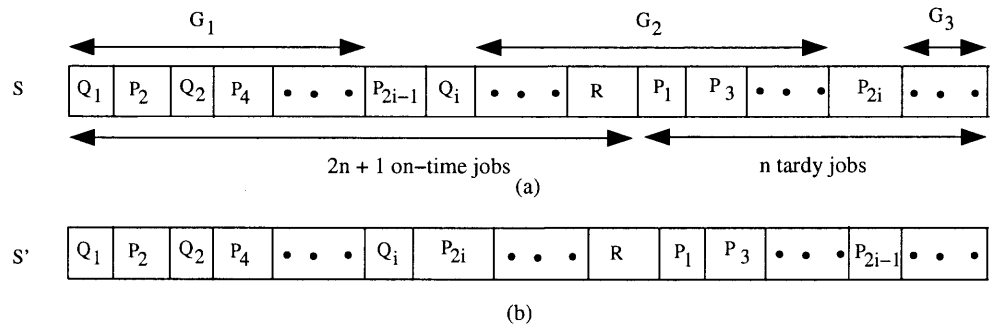


Figure 5.2 (a) A feasible schedule of jobs in instance I, (b) The schedule obtained from (a) by interchanging P_{2i} with P_{2i-1}.

it can be shown (by interchange argument) that in order to minimize the total completion time, they must be scheduled in increasing order of their processing times, which is the same order as the due dates of the P-jobs. There are only two possible configurations for each triplet $\{P_{2i-1}, Q_i, P_{2i}\}$, see Figure 5.2(a). Either P_{2i-1} and Q_i are on time and scheduled in this order, or Q_i and P_{2i} are on time and scheduled in this order.

It is now shown that in order to minimize the total completion time, it is always better to pick Q_i and P_{2i} to be on time. Suppose there is a feasible schedule in which P_{2i-1} and Q_i are on time and P_{2i} is tardy. It will be shown that by changing the configuration to Q_i and P_{2i} on time (see Figure 5.2(b)), the total completion time will be decreased by exactly $a_{2i} - a_{2i-1}$. Denote the original schedule as S, and the new schedule as S'. Use G_1 to denote the jobs scheduled before P_{2i-1} in S, G_2 to denote the jobs scheduled between Q_i and P_{2i}, and G_3 to denote the jobs scheduled after P_{2i} in S. It is easy to see that there are $2n - i$ jobs in G_2.

Note that for each job in G_1 and G_3, its completion time in S' remains the same as in S. For each job in G_2, the completion time will increase by $a_{2i} - a_{2i-1}$. So the total increase is $(2n - i)(a_{2i} - a_{2i-1})$. The completion time of Q_i decreases by a_{2i-1}. The completion time of P_{2i-1} in S' is the same as the completion time of P_{2i} in S. The completion time of P_{2i} in S' is larger than the completion time of P_{2i-1} by $x_i + (a_{2i} - a_{2i-1})$. Thus, the total

completion time of all jobs is decreased by:

$$\begin{aligned}
& a_{2i-1} - (x_i + (a_{2i} - a_{2i-1})) - (2n - i)(a_{2i} - a_{2i-1}) \\
&= a_{2i-1} - x_i - (2n - i + 1)(a_{2i} - a_{2i-1}) \\
&= (2n - i + 2)(a_{2i} - a_{2i-1}) - (2n - i + 1)(a_{2i} - a_{2i-1}) \qquad \text{by (5.3)} \\
&= (a_{2i} - a_{2i-1}) \ .
\end{aligned}$$

On the other hand, in order to ensure that the R-job is on time, one can not pick all the even P-jobs to be on time. Suppose all the even P-jobs are picked to be on time. Then the completion time of the R-job will be

$$\sum_{j=1}^{n} a_{2j} + \sum_{j=1}^{n} x_j + L > A + \sum_{j=1}^{n} x_j + L = d_R \ .$$

Therefore, the R-job will miss its due date. So, one must choose some even P-jobs as tardy jobs. Let B' be the total completion time when all the even P-jobs are on time. Then, $B = B' + \frac{1}{2}\sum_{j=1}^{n}(a_{2j} - a_{2j-1})$. As has been shown above, each time P_{2i} and P_{2i-1} are interchanged, the total completion time will increase by $a_{2i} - a_{2i-1}$. At the same time, the completion time of the R-job will decrease by exactly $a_{2i} - a_{2i-1}$. So, if one can schedule the jobs such that the R-job completes at exactly its due date, then the the total completion time has the minimum value B among all feasible schedules, and the total processing time of all the on-time P-jobs is exactly A. This means that there is a solution to the instance of the Even-Odd Partition problem if and only if there is a solution to the instance of the scheduling problem.

From the above discussions, the following theorem follows.

Theorem 5.1.4 *The problem* $1 \mid\mid \sum C_j \mid \sum U_j$ *is NP-Hard.*

5.2 Minimizing Total Tardiness Subject to Minimum $\sum U_j$

In this section, it will be shown that the Even-Odd Partition problem can be reduced to the decision version of the $1 \mid\mid \sum T_j \mid \sum U_j$ problem. Given an instance of the Even-Odd Partition problem, create an instance II of the scheduling problem as follows. There are $2n$ P-jobs and a R-job. The processing times and due dates of these jobs are shown in Table 5.2, where L is an integer greater than A.

job	processing time	due date
P_{2i-1}	a_{2i-1}	$\sum_{k=1}^{i} a_{2k} + i \cdot (a_{2i} - a_{2i-1})$
P_{2i}	a_{2i}	$\sum_{k=1}^{i} a_{2k}$
R	L	$A + L$

Table 5.2 The Processing Times and Due Dates of the Jobs in Instance II

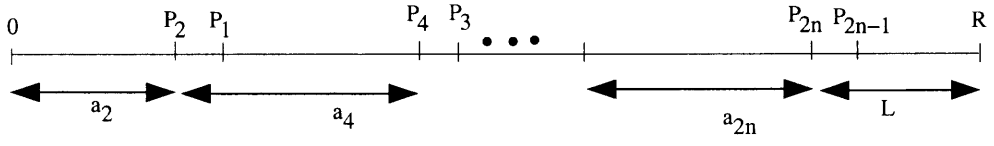


Figure 5.3 Illustration of the due dates of jobs in instance II.

The due date pattern of the jobs are shown in Figure 5.3. Let the threshold for the total tardiness be B, where

$$B = \sum_{i=1}^{n} \left(L + \sum_{k=i+1}^{n} a_{2k} + \sum_{k=1}^{i} a_{2k-1} - i \cdot (a_{2i} - a_{2i-1}) \right) + \frac{1}{2} \cdot \sum_{i=1}^{n} (a_{2i} - a_{2i-1}) \ .$$

The decision problem asks: is there a schedule with the minimum number of tardy jobs such that the total tardiness is less than or equal to B?

The basic idea of the reduction is similar to that in Section 5.1. For every pair of even and odd P-jobs, one of them must be on time and the other must be tardy. The R-job

must be on time. To minimize total tardiness, it will be more advantageous to schedule all even P-jobs to be on time and all odd P-jobs to be tardy. But if all even P-jobs are on time, then the R-job will miss its due date by A. To ensure that the R-job is on time, the total processing time of all the on-time P-jobs cannot exceed A. The total tardiness will attain its minimum, B, when the total processing time of all the on-time P-jobs is exactly A. Thus, there is a solution to the instance of the Even-Odd Partition problem if and only if there is a solution to the instance of the scheduling problem.

The next lemma proves the assertions made above.

Lemma 5.2.1 *In any feasible schedule for the scheduling problem instance II, (a) there are exactly n tardy jobs, (b) the R-job is on time and all the on-time P-jobs are scheduled before the R-job, (c) exactly one job from each pair $\{P_{2i-1}, P_{2i}\}$, $1 \leq i \leq n$, is tardy, (d) all tardy jobs are scheduled after the R-job.*

Proof: One proves (a) by showing that there are n tardy jobs when the Hodgson-Moore algorithm is applied to the instance II. For the instance II, $d_{P_{2i}} < d_{P_{2i-1}} < d_{P_{2i+2}}$ and $d_{P_{2n-1}} < d_R$. Therefore, the jobs would be scheduled by the Hodgson-Moore algorithm in the order of $P_2, P_1, P_4, P_3, \ldots, P_{2n}, P_{2n-1}, R$. It is easy to see that P_2 meets its due date, but P_1 will not. Since $a_1 < a_2$, P_2 will be chosen as a tardy job by the Hodgson-Moore algorithm. Suppose all the jobs P_{2j} and P_{2j-1} have been scheduled, where $1 \leq j \leq i-1$, and all the even P-jobs, P_{2j}, $1 \leq j \leq i-1$, were chosen as tardy jobs. Consider now the scheduling of P_{2i} and P_{2i-1}. The completion times will be

$$C_{P_{2i}} = \sum_{j=1}^{i-1} a_{2j-1} + a_{2i} < \sum_{j=1}^{i-1} a_{2j} + a_{2i} = d_{P_{2i}} \ ,$$

and

$$
\begin{aligned}
C_{P_{2i-1}} &= \sum_{j=1}^{i-1} a_{2j-1} + a_{2i-1} + a_{2i} \\
&> \sum_{j=1}^{i-1} a_{2j-1} + \left(\sum_{j=1}^{i-1} (a_{2j-1} + a_{2j}) \right) + a_{2i} \qquad \text{by (5.2)} \\
&= 2 \sum_{j=1}^{i-1} a_{2j-1} + \sum_{j=1}^{i} a_{2j} \\
&> 2(i-1) \cdot a_1 + \sum_{j=1}^{i} a_{2j} \\
&> n \cdot max_{1 \le j \le n}(a_{2j} - a_{2j-1}) + \sum_{j=1}^{i} a_{2j} \qquad \text{by (5.1)} \\
&> d_{P_{2i-1}} \ .
\end{aligned}
$$

Since P_{2i-1} misses its due date and since P_{2i} has the largest processing time among all jobs in the current schedule, P_{2i} will be chosen as a tardy job. Therefore, the Hodgson-Moore algorithm will pick all the even P-jobs as tardy jobs. For the R-job, the completion time will be

$$\sum_{j=1}^{n} a_{2j-1} + L < A + L = d_R \ ,$$

so it is on time. Hence the total number of tardy jobs is n, concluding the proof of (a).

Since the R-job has a large processing time, a job scheduled after the R-job must miss its due date. Hence, all the other on-time jobs must be scheduled before the R-job. This proves (b).

By (a) and (b), all tardy jobs are P-jobs. In order to prove (c), it is sufficient to prove that at least one job from each pair $\{P_{2i-1}, P_{2i}\}$ must be on time. This will be proved by contradiction. Suppose $\{P_{2i-1}, P_{2i}\}$ is the first pair such that both jobs are tardy in a feasible schedule S. Consider now applying the Hodgson-Moore algorithm to the job set consisting of all P-jobs except $\{P_{2i-1}, P_{2i}\}$, and the R-job. It is easy to see that all jobs

P_{2j}, $j < i$, are picked as tardy jobs. If $\{P_{2i+2}, P_{2i+1}\}$ is now scheduled, then P_{2i+2} will still be on time. However, the completion time of P_{2i+1} is

$$\sum_{j=1}^{i-1} a_{2j-1} + a_{2i+1} + a_{2i+2}.$$

By plugging in (5.2) and (5.1), one can easily show that P_{2i+1} will miss its due date. Since P_{2i+2} has the largest processing time among all jobs in the current schedule, it will be chosen as a tardy job by the Hodgson-Moore algorithm. Using the same argument, it can be shown that all even P-jobs are tardy. By assumption, P_{2i-1} is also a tardy job. Hence, the total number of tardy jobs will be $n+1$, contradicting the assumption that S is a feasible schedule.

(d) will also be proved by contradiction. As has been shown, all the on-time P-jobs must be scheduled before the R-job. The total processing time of these jobs is at least $\sum_{j=1}^{n} a_{2j-1}$. Suppose P_m is a tardy job scheduled before the R-job, where $1 \leq m \leq 2n$. Then the completion time of the R-job would be

$$\begin{aligned}
&\sum_{j=1}^{n} a_{2j-1} + L + a_m \\
&= \left(A - \frac{1}{2} \sum_{j=1}^{n} (a_{2j} - a_{2j-1}) \right) + L + a_m \\
&= A + L + \left(a_m - \frac{1}{2} \sum_{j=1}^{n} (a_{2j} - a_{2j-1}) \right) \\
&> A + L \ .
\end{aligned}$$

Thus, the R-job will miss its due date. By Lemma 5.2.1, S can not be a feasible schedule. ■

By Lemma 5.2.1, in any feasible schedule, all the on-time P-jobs are scheduled before the R-job and all the tardy P-jobs are scheduled after the R-job. It is easy to see that all the on-time P-jobs must be scheduled in increasing order of their due dates; otherwise,

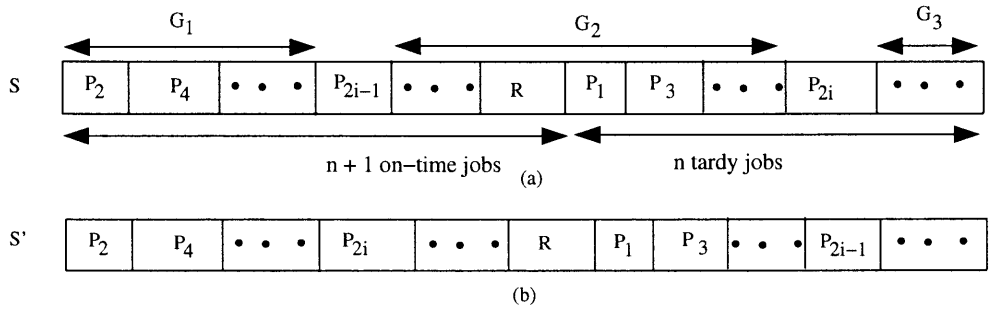


Figure 5.4 (a) A feasible schedule of the jobs in instance II, (b) The schedule obtained from (a) by interchanging P_{2i} with P_{2i-1}.

some of them will miss their due dates. For the tardy jobs, one can show (by interchange argument) that the total tardiness is minimized by scheduling them in increasing order of their due dates, which is also the same order as their processing times. By Lemma 5.2.1, there are only two possible choices for each pair $\{P_{2i-1}, P_{2i}\}$, either P_{2i-1} is on time and P_{2i} is tardy, or P_{2i-1} is tardy and P_{2i} is on time.

It is now shown that in order to minimize the total tardiness, it is always better to pick P_{2i-1} as a tardy job. Suppose a feasible schedule picks P_{2i} as a tardy job. It will be proved that by interchanging P_{2i} with P_{2i-1}, the total tardiness will be decreased by $a_{2i} - a_{2i-1}$. Denote the original schedule as S, and the new schedule as S'. Use G_1 to denote the jobs scheduled before P_{2i-1} in S, G_2 to denote the jobs scheduled between P_{2i-1} and P_{2i}, and G_3 to denote the jobs scheduled after P_{2i} in S; see Figure 5.4. Since all jobs scheduled before the R-job are on time, there are exactly $(i-1)$ tardy jobs in G_2.

Note that this interchange will not change the completion time and consequently the tardiness of the jobs in G_1 and G_3. For each tardy job in G_2, the tardiness will be increased by $a_{2i} - a_{2i-1}$. So, the total increase in tardiness of jobs in G_2 is $(i-1)(a_{2i} - a_{2i-1})$. The completion time of P_{2i-1} in S' is the same as the completion time of P_{2i} in S. However, since $d_{P_{2i-1}} = d_{P_{2i}} + i \cdot (a_{2i} - a_{2i-1})$, the tardiness of P_{2i-1} in S' is $i \cdot (a_{2i} - a_{2i-1})$ less than that of P_{2i} in S. In total, the total tardiness is decreased by $(a_{2i} - a_{2i-1})$.

On the other hand, in order to ensure that the R-job is on time, one can not pick all the even P-jobs as on-time jobs. Suppose one picks all the even P-jobs. Then the completion

time of the R-job will be

$$\sum_{j=1}^{n} a_{2j} + L = d_R + \frac{1}{2}\sum_{j=1}^{n}(a_{2j} - a_{2j-1}) \ ,$$

and hence the R-job will miss its due date. So, one must choose some odd P-jobs as tardy jobs. Let B' be the total tardiness when all odd P-jobs are picked as tardy jobs. Then,

$$\begin{aligned} B' &= \sum_{i=1}^{n} T_{P_{2i-1}} \\ &= \sum_{i=1}^{n}\left(L + \sum_{k=1}^{n} a_{2k} + \sum_{k=1}^{i} a_{2k-1} - \left(\sum_{k=1}^{i} a_{2k} + i\cdot(a_{2i} - a_{2i-1}\right)\right) \\ &= \sum_{i=1}^{n}\left(L + \sum_{k=i+1}^{n} a_{2k} + \sum_{k=1}^{i} a_{2k-1} - i\cdot(a_{2i} - a_{2i-1})\right) \ . \end{aligned}$$

Thus, $B = B' + \frac{1}{2}\sum_{j=1}^{n}(a_{2j} - a_{2j-1})$. As has been shown above, each time P_{2i} and P_{2i-1} are interchanged, the total tardiness will increase by $a_{2i} - a_{2i-1}$. At the same time, the completion time of the R-job will decrease by exactly $a_{2i} - a_{2i-1}$. So, if one can choose the on-time jobs such that the R-job completes at exactly its due date, then the total tardiness has the minimum value B among all feasible schedules, and the total processing time of all the on-time P-jobs is exactly A. This means that there is a solution to the instance of the Even-Odd Partition problem if and only if there is a solution to the instance of the scheduling problem.

From the above discussions, the following theorem follows.

Theorem 5.2.2 *The problem* $1 \mid\mid \sum T_j \mid \sum U_j$ *is NP-Hard.*

5.3 Conclusion

In this article $1 \mid\mid C_j \mid \sum U_j$ and $1 \mid\mid T_j \mid \sum U_j$ have both been shown to be NP-Hard. It is not known whether they are unary NP-Hard, or that they can be solved in pseudo-polynomial time. These issues represent major challenges for future research.

CHAPTER 6

BI-CRITERIA SCHEDULING PROBLEMS: NUMBER OF TARDY JOBS AND MAXIMUM WEIGHTED TARDINESS

In this chapter dual criteria scheduling problems with the following criteria will be considered: the number of tardy jobs $\sum U_j$, the maximum tardiness $T_{\max}$ and the maximum weighted tardiness $\max\{w_jT_j\}$. In the notation introduced by Graham et al. [24], the problems considered in this chapter are $1 \mid\mid T_{\max} \mid \sum U_j$, $1 \mid\mid \sum U_j \mid T_{\max}$,$1 \mid\mid \max\{w_jT_j\} \mid \sum U_j$, $1 \mid\mid \sum U_j \mid \max\{w_jT_j\}$.

There are a lot of applications related to these four scheduling problems in the industrial areas. Woolsey [60] described a problem faced by the scheduler at a southwestern company that needs to satisfy simultaneously the salespeople and the customers. In this company, when the salespeople take customer orders, they promise the job will be ready on a specific date. Salespeople are paid commissions based on the tardiness of the order; full commissions are paid for on-time orders, but the commission decreases to a certain minimum value as the tardiness increases. Clearly, the scheduler at this company is faced with unhappy customers and salespeople if not all jobs can be on time. From the perspective of the salespeople, minimizing maximum tardiness will be the fairest measure since the person penalized the most is hurt as little as possible. However, such a schedule could have many tardy jobs, which is not good from the customer's point of view. In this situation there are two criteria in play: number of tardy jobs and maximum tardiness. There may be several schedules which minimize maximum tardiness, so it seems reasonable to choose the one which has the fewest number of tardy jobs. Such a schedule is fair to the sales force, while keeping as many customers as possible happy. Since orders are released to the plant one-at-a-time, the scheduler is faced with a single machine scheduling problem with minimizing maximum

tardiness as the primary objective, and minimizing the number of tardy jobs as a secondary objective.

Following the applications, a lot of research have been done on these problems. The problem $1 \mid\mid T_{\max} \mid \sum U_j$ has been studied by Shanthikumar [56], who gave a branch-and-bound algorithm for the general problem. As well, a polynomial-time algorithm was given when the set of tardy jobs is specified. Chen and Bulfin [11] studied the problem $1 \mid\mid \sum U_j \mid T_{\max}$ and gave a branch-and-bound algorithm for the general problem. Further results about primary and secondary criteria scheduling problems can be found in Chen and Bulfin [10], Dileepan and Sen [15] and Lee and Vairaktarakis [44].

Lee and Vairaktarakis [44] further differentiate bi-criteria scheduling problems between hierarchical problems and dual criteria problems. In a dual criteria problem, one merely requires the primary criterion to satisfy the constraint that $\gamma_1 \leq \alpha$, where α is an input parameter. A hierarchical problem is a special case of dual criteria problem where α is stipulated to be the minimum value of γ_1 (and hence is not an input parameter). The problems mentioned up to now are all hierarchical problems.

In this chapter both dual criteria and hierarchical problems are considered. A *feasible schedule* for a problem $1 \mid\mid \gamma_2 \mid \gamma_1$ (be it a dual criteria problem or a hierarchical problem) is a schedule in which the primary criterion is satisfied. An *optimal schedule* is a feasible schedule that minimizes the secondary criterion. For a given set of jobs, let k^* be the minimum number of tardy jobs, T^* be the minimum $T_{\max}$ and T_w^* be the minimum value of $\max\{w_j T_j\}$. In this chapter the following problems are considered:

P1: $1 \mid\mid \sum U_j \mid T_{\max} \leq T$, where $T \geq T^*$

P2: $1 \mid\mid T_{\max} \mid \sum U_j \leq k$, where $k \geq k^*$

P3: $1 \mid\mid \sum U_j \mid T_{\max} = T^*$, or $1 \mid\mid \sum U_j \mid T_{\max}$ for simplicity

P4: $1 \mid\mid T_{\max} \mid \sum U_j = k^*$, or $1 \mid\mid T_{\max} \mid \sum U_j$ for simplicity

P5: $1 \mid\mid \sum U_j \mid \max\{w_j T_j\} \leq T_w$, where $T_w \geq T_w^*$

P6: $1 \mid\mid \max\{w_jT_j\} \mid \sum U_j \leq k$, where $k \geq k^*$

P7: $1 \mid\mid \sum U_j \mid \max\{w_jT_j\} = T_w^*$, or $1 \mid\mid \sum U_j \mid \max\{w_jT_j\}$ for simplicity

P8: $1 \mid\mid \max\{w_jT_j\} \mid \sum U_j = k^*$, or $1 \mid\mid \max\{w_jT_j\} \mid \sum U_j$ for simplicity

Note that P1, P2, P5 and P6 are dual criteria problems, while P3, P4, P7 and P8 are hierarchical problems.

Problems P1, P3, P5 and P7 are related to the problem $1 \mid \bar{d}_j \mid \sum U_j$. In the problem $1 \mid \bar{d}_j \mid \sum U_j$, each job j is given a deadline $\bar{d}_j$ which must be met, and the goal is to minimize the number of tardy jobs. For P1 and P3, one can view each job j as having a deadline $\bar{d}_j = d_j + T$ and $\bar{d}_j = d_j + T^*$, respectively. For P5 and P7, each job j has a deadline $\bar{d}_j = d_j + T_w/w_j$ and $\bar{d}_j = d_j + T_w^*/w_j$, respectively. In later sections of this chapter, the problem $1 \mid \bar{d}_j \mid \sum U_j$ will be used as a substitute for problems P1, P3, P5 and P7, where the deadline of each job is set appropriately.

In this chapter the complexity relationships between the above eight problems will first be considered. Then problems P7 and P8 are shown to be NP-Hard, which implies that problems (3) and (4) in the list of open problems of Lee and Vairaktarakis [44] are also NP-Hard. These results are given in Section 2. Optimal algorithms for three special cases of problems P1 to P8 will then be developed, which will be presented in Section 3. Finally, several heuristics for the general problem will be proposed and their effectiveness are studied empirically. The experiment indicates that one heuristic performs extremely well compared to optimal solutions. These results will be described in Section 4. The last section concludes.

6.1 Complexity Results

The complexity relationships between the eight problems will first be studied. Given two problems Π_1 and Π_2, $\Pi_1 \Rightarrow \Pi_2$ denotes that a polynomial-time algorithm for Π_1 implies a polynomial-time algorithm for Π_2. $\Pi_1 \equiv \Pi_2$ denotes that $\Pi_1 \Rightarrow \Pi_2$ and $\Pi_2 \Rightarrow \Pi_1$.

Lemma 6.1.1 *$P1 \Rightarrow P3$, $P2 \Rightarrow P4$, $P5 \Rightarrow P7$ and $P6 \Rightarrow P8$.*

Proof: It follows from the fact that hierarchical problems are special cases of dual criteria problems. ■

Lemma 6.1.2 *$P1 \equiv P3$ and $P5 \equiv P7$.*

Proof: By Lemma 6.1.1, $P1 \Rightarrow P3$ and $P5 \Rightarrow P7$. Thus, all is left to show is that $P3 \Rightarrow P1$ and $P7 \Rightarrow P5$. $P3 \Rightarrow P1$ will be shown first. Suppose an instance I of $P1$ consisting of a set of n jobs and a parameter T is given, where $T \geq T^*$. If $T = T^*$, then this instance is also an instance of $P3$, so the polynomial-time algorithm for $P3$ can be used to solve I. Otherwise, I will be solved as follows.

Without loss of generality, one may assume that the jobs in I are indexed in ascending order of due dates; i.e., $d_1 \leq d_2 \leq ... \leq d_n$. Construct an instance II of $P3$ as follows: II consists of all the jobs in I plus an additional job $n+1$. Job $n+1$ has processing time $p_{n+1} = T + (1 + d_n - \sum_{i=1}^{n} p_i)$ and due date $d_{n+1} = 1 + d_n$. Using the algorithm for $P3$, one can find an optimal schedule S_{II} for II in polynomial time. Let S_I be the schedule obtained from S_{II} by deleting job $n+1$ from S_{II} and compacting the schedule if possible. Clearly, S_I can be obtained in polynomial time. S_I is now shown to be an optimal schedule for I.

Let $T^*(II)$ be the minimum $T_{\max}$ for the jobs in II. First, $T^*(II) = T$ is shown. As mentioned in Section 1, the schedule obtained by the EDD rule minimizes the maximum tardiness. Since job $n+1$ has the largest due date in II, it must be scheduled as the last job in the EDD schedule. Thus, $T_{n+1} = \max(0, \sum_{j=1}^{n+1} p_j - d_{n+1}) = T$. If job $n+1$ is discarded from this EDD schedule, the resulting schedule will still be a EDD schedule for the jobs in I. By assumption, the maximum tardiness of the jobs in I is at most T. Therefore, the maximum tardiness obtained in this EDD schedule is at most T. Thus, $T^*(II) = T_{n+1} = T$.

Since $T^*(II) = T$, in any feasible schedule for II, the completion time of any job i, $1 \leq i \leq n$, cannot be greater than $d_i + T$ which is strictly less than $d_{n+1} + T = \sum_{j=1}^{n+1} p_j$. Thus, the only job that can be scheduled last is job $n+1$. In other words, all the jobs in I must be scheduled before job $n+1$ and have tardiness at most T. So S_I is also a feasible schedule for the instance I of problem $P1$. Thus, there is a one-to-one correspondence between the feasible schedules for I and the feasible schedules for II. The number of tardy jobs in a feasible schedule for I is exactly one less than that for II (since job $n+1$ is always the last one to complete and is always tardy). Therefore, the optimal schedule S_{II} for II must correspond to the optimal schedule S_I for I.

Now, $P7 \Rightarrow P5$ will be shown. Suppose I is an instance of $P5$ consisting of a set of n jobs and a parameter T_w, where $T_w \geq T_w^*$. If $T_w = T_w^*$, then this instance is also an instance of $P7$, and hence I can be solved by the polynomial-time algorithm for $P7$. Otherwise, I will be solved as follows.

Let the jobs in I be indexed in ascending order of due dates (i.e., $d_1 \leq d_2 \leq ... \leq d_n$) and let w^* be the smallest weight among the n jobs in I. Construct an instance II of $P7$ as follows: II consists of all the jobs in I plus an additional job $n+1$, which has processing time $p_{n+1} = \frac{T_w}{w^*} + (1 + d_n - \sum_{i=1}^{n} p_i)$, due date $d_{n+1} = 1 + d_n$ and weight $w_{n+1} = w^*$. Using the algorithm for $P7$, one can find an optimal schedule S_{II} for II in polynomial time. Let S_I be the schedule obtained from S_{II} by deleting job $n+1$ from S_{II} and compacting the schedule if possible. S_I will be shown to be an optimal schedule for I.

Let $T_w^*(II)$ be the minimum value of $\max\{w_j T_j\}$ for the jobs in II. First $T_w^*(II) = T_w$ will be shown. If job $n+1$ is scheduled last in S_{II}, then $w_{n+1} T_{n+1} = T_w$. On the other hand, if any job i, $1 \leq i \leq n$, is scheduled last in S_{II}, then $w_i T_i > T_w$. Thus, job $n+1$ must be the last job scheduled in S_{II}. By assumption, the minimum value of $\max\{w_j T_j\}$ for the jobs in I is at most T_w. Thus, $T_w^*(II) = T_w$. So S_I is also a feasible schedule for the instance I of problem $P5$. Consequently, there is a one-to-one correspondence between the

feasible schedules for I and the feasible schedules for II, and hence the optimal schedule S_{II} for II must correspond to the optimal schedule S_I for I. ■

Lemma 6.1.3 *$P1 \equiv P2$ and $P5 \equiv P6$.*

Proof: $P1 \Rightarrow P2$ will be proved first. Suppose there is an algorithm A that takes a parameter T and a set of n jobs, and outputs a schedule that minimizes $\sum U_j$ such that the schedule has maximum tardiness at most T, where $T \geq T^*$. It will now be shown that binary search can be used to find the minimum $T_{\max}$ among all schedules such that $\sum U_j \leq k$ for a given parameter $k \geq k^*$.

Since $k \geq k^*$, there is always a feasible schedule (and hence an optimal schedule) for an instance of $P2$. The maximum tardiness of any feasible schedule must lie between T^* and $\sum_{j=1}^{n} p_j$, and hence the optimal $T_{\max}$ must also lie in this range. Given $k_1 \geq k_2 \geq k^*$, let T_1 and T_2 be the optimal $T_{\max}$ to the problems $1 \mid\mid T_{\max} \mid \sum U_j \leq k_1$ and $1 \mid\mid T_{\max} \mid \sum U_j \leq k_2$, respectively. Then one must have $T_1 \leq T_2$, since any feasible schedule for the latter problem is also a feasible schedule for the former problem, Thus, as T increases, the number of tardy jobs decreases, and conversely. Therefore, there must be a smallest integer T_O, $T^* \leq T_O \leq \sum_{j=1}^{n} p_j$, such that the number of tardy jobs in an optimal solution to the problem $1 \mid\mid \sum U_j \mid T_{\max} \leq T_O$ is at most k. One knows that T^* can be obtained by the EDD schedule in $O(n \log n)$ time. For each value of T obtained in the binary search, algorithm A will be called to find the optimal number of tardy jobs. One then searches the upper half or the lower half of the range, depending on whether the number of tardy jobs returned by algorithm A is larger than or at most k. If binary search is used to find T_O, algorithm A need to be called at most $\lceil \log(\sum_{j=1}^{n} p_j) \rceil$ times. So the overall running time is still polynomial if A is a polynomial-time algorithm.

$P2 \Rightarrow P1$ will now be shown. Suppose there is an algorithm B that takes a parameter k and a set of n jobs, and outputs a schedule that minimizes $T_{\max}$ such that the schedule has at most k tardy jobs. It will be shown that binary search can be used to find the minimum

number of tardy jobs such that the schedule has maximum tardiness at most T for a given $T \geq T^*$.

Since $T \geq T^*$, there is always a feasible schedule (and hence an optimal schedule) for an instance of $P1$. The number of tardy jobs in any feasible schedule must lie between k^* and n, and hence the optimal number of tardy jobs must also lie in this range. Given $T_1 \geq T_2 \geq T^*$, let k_1 and k_2 be the optimal number of tardy jobs to the problems $1 \mid\mid \sum U_j \mid T_{\max} \leq T_1$ and $1 \mid\mid \sum U_j \mid T_{\max} \leq T_2$, respectively. Then $k^* \leq k_1 \leq k_2 \leq n$, since a feasible schedule for the latter problem is also a feasible schedule for the former. Thus, as the number of tardy jobs increases, T decreases, and conversely. Therefore, there must be a smallest integer k_O, $k^* \leq k_O \leq n$, such that the maximum tardiness in an optimal solution to the problem $1 \mid\mid T_{max} \mid \sum U_j \leq k_O$ is at most T. k^* can be obtained by the Hodgson-Moore algorithm in $O(n \log n)$ time. For each value of k obtained in the binary search, algorithm B is called to find the optimal $T_{\max}$. Then the upper half or the lower half of the range will be searched, depending on whether the $T_{\max}$ returned by algorithm B is larger than or at most T. If binary search is used to find k_O, algorithm B is called at most $\lceil \log n \rceil$ times. So the overall running time is still polynomial if B is a polynomial-time algorithm.

The proof of $P5 \equiv P6$ follows the same arguments as above. ■

From Lemmas 6.1.1, 6.1.2 and 6.1.3, the following theorem follows.

Theorem 6.1.4 *$P3 \equiv P1 \equiv P2 \Rightarrow P4$ and $P7 \equiv P5 \equiv P6 \Rightarrow P8$.*

Next, the complexity of problems $P7$ and $P8$ will be considered.

Theorem 6.1.5 *Problems $P7$ and $P8$ are both NP hard.*

Proof: Lawler [42] has shown that the problem $1 \mid \bar{d}_j \mid \sum U_j$ is NP hard by a reduction from the partition problem; see also [1]. The proof is based on his reduction. For completeness, his reduction will be sketched first.

An instance of the partition problem has n integers $a_1, a_2, \ldots, a_n$ with $\sum a_i = 2A$. The problem is to decide whether there is a subset S of the index set $\{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = A$. From an instance of the partition problem, construct an instance of $1 \mid \bar{d}_j \mid \sum U_j$ as follows. There will be $4n$ jobs, with each job i having a processing time p_i, a due date d_i and a deadline $\bar{d}_i$, see Table 6.1[1], where $x_{i-1} = 3A \cdot 2^{i-1}$ and $P_{i-1} = \sum_{k=1}^{n}(2^n + 1)x_{k-1} + \sum_{k=1}^{i-1}(2^n + 1)x_{k-1}$.The problem is to decide whether there is a feasible schedule with at most $2n$ tardy jobs.

The idea of the reduction is that, for each $i = 1, \ldots, n$, the jobs $\{i, 3n + i\}$ and $\{n + i, 2n + i\}$ form two pairs, one of which goes into the on-time set and the other one goes into the tardy set. One can show that there is a schedule with exactly $2n$ tardy jobs such that every job meets its deadline if and only if there is a solution to the partition instance.

The decision version of the problem $P7$ is first shown to be NP complete. Almost the same reduction as in Lawler's proof will be used, but instead of having a deadline, each job now has a weight. An additional job 0 will be introduced. Specifically, let $T_w = \max\{\bar{d}_j - d_j\}$, $1 \leq j \leq 4n$, where $\bar{d}_j$ and d_j have the same values as above. Create $4n + 1$ jobs, $0, 1', \ldots, 4n'$. Let $p_0 = T_w = \max\{\bar{d}_j - d_j\}$, $d_0 = 0$ and $w_0 = 1$. For $1 \leq j' \leq 4n$, let $p'_j = p_j$, $w'_j = T_w/(\bar{d}_j - d_j)$ and $d'_j = p_0 + d_j$, where p_j and d_j have the same values as above. The problem is to decide whether there is a feasible schedule with at most $2n + 1$ tardy jobs such that $\max\{w_j T_j\}$ is at most T_w?

Because job 0 has due date 0, its weighted tardiness has to be at least $w_0 p_0 = T_w$ in any schedule. Thus, $\max\{w_j T_j\}$ is at least T_w. On the other hand, if job 0 is scheduled first and then the remaining jobs are scheduled in an order corresponding to a feasible schedule for the instance of the $1 \mid \bar{d}_j \mid \sum U_j$ problem, then one can easily show that every job has a weighted tardiness at most T_w. Thus, T_w^* is exactly T_w. Hence, in an optimal schedule, job 0 has to be scheduled before any other job. For all other tardy jobs, $w'_i T'_i = w'_i(C'_i - d'_i) \leq T_w$,

[1]There is a typo in [1]. The deadline of job i, $1 \leq i \leq n$, should be $P_{i-1} + x_{i-1} + A$ instead of $P_{i-1} + x_{i-1} - A$

where C'_i is the completion time of job i' in the optimal schedule. Plugging in the values of w'_i and d'_i, $C'_i \leq (\bar{d}_i + p_0)$. This means that for each job i', $w'_i T'_i \leq T_w$ if and only if i' completes before $\bar{d}_i + p_0$. Using almost the same argument, one can show that there is a schedule with at most $2n+1$ tardy jobs such that $C'_i \leq \bar{d}_i + p_0$ for every $1 \leq i' \leq 4n$ if and only if there is a solution to the partition instance.

Thus, problem $P7$ is NP-Hard. On the other hand, using the Hodgson-Moore algorithm, one can show that the minimum number of tardy jobs for these $4n+1$ jobs is exactly $2n+1$. So the same argument shows that problem $P8$ is also NP-Hard. ■

Corollary 6.1.6 *The problems* $1 \mid\mid \sum U_j \mid f_{\max}$ *and* $1 \mid\mid f_{\max} \mid \sum U_j$ *are both NP-Hard.*

Corollary 6.1.7 *The problems* $P5$ *to* $P8$ *are all NP-Hard.*

6.2 Optimal Algorithms for Special Cases

In this section polynomial-time algorithms to solve three special cases of problems $P1$ to $P8$ will be presented. Only $P1$ and $P5$ will be studied. By Theorem 1, a polynomial-time algorithm for $P1$ yields a polynomial-time algorithm for $P2$, $P3$ and $P4$, whereas a polynomial-time algorithm for $P5$ yields a polynomial-time algorithm for $P6$, $P7$ and $P8$. Thus, it will be sufficient to consider these two problems only. $1 \mid \bar{d}_j \mid \sum U_j$ will be studied as a substitute for $P1$ and $P5$, by setting the deadline of each job appropriately; i.e., $\bar{d}_j = d_j + T$ for $P1$ and $\bar{d}_j = d_j + \frac{T_w}{w_j}$ for $P5$.

In this section jobs are assumed to be indexed in ascending order of their due dates; i.e., $d_1 \leq d_2 \leq \ldots \leq d_n$. The constraints of the three special cases are given as follows.

Case 1: $d_i \leq d_j$ implies $\bar{d}_i \leq \bar{d}_j$ and $p_i \leq p_j$.

Case 2: $p_i \geq p_j$ implies $sl_i \leq sl_j$, where $sl_k = d_k - p_k$ is the slack of job k.

Case 3: There is an integer m, $1 < m < n$, such that (1) for $1 \leq i < j \leq m$, $d_i \leq d_j$ implies $\bar{d}_i \leq \bar{d}_j$ and $p_i \leq p_j$, (2) $\max_{j=1}^{m}\{\bar{d}_j\} \leq \min_{j=m+1}^{n}\{\bar{d}_j\}$, and (3) for $m < i, j \leq n$, $p_i \geq p_j$ implies $sl_i \leq sl_j$.

Case 1 requires that deadlines are nondecreasing as a function of the due dates. This condition always holds for $P1$, and for $P5$ it holds when $d_i \leq d_j$ implies $d_i + \frac{T_w}{w_i} \leq d_j + \frac{T_w}{w_j}$. Case 1 also requires that processing times are nondecreasing as a function of due dates. For Case 2 the condition "$p_i \geq p_j$ implies $sl_i \leq sl_j$" can be satisfied by the condition "$d_i \leq d_j$ implies $p_i \geq p_j$". This special case corresponds to the situation where processing times are nonincreasing as a function of due dates. Case 3 is a combination of Cases 1 and 2; i.e., there is an integer m such that the first m jobs satisfy the condition of Case 1, the last $n-m$ jobs satisfy the condition of Case 2, and the largest deadline in the first m jobs is less than or equal to the smallest deadline in the last $n-m$ jobs..

In each case the set of jobs is assumed to have a feasible schedule; i.e., there is a schedule such that all jobs can meet their deadlines. A polynomial-time algorithm for each case will be given and the algorithm is proved to be optimal. The three cases are given in the next three subsections.

6.2.1 Case 1

In this subsection the condition that $d_i \leq d_j$ implies $\bar{d}_i \leq \bar{d}_j$ and $p_i \leq p_j$ is assumed. Let TS be a subset of JS, where JS is a set of n jobs, $1, \ldots, n$. A *tight schedule* for JS with respect to TS is a schedule such that: (1) the jobs in $JS \setminus TS$ are scheduled in nondecreasing order of their due dates, (2) all jobs in TS are scheduled in nondecreasing order of their deadlines, and (3) for each job $i \in TS$ scheduled immediately before a job $j \in JS \setminus TS$, $C_i \leq \bar{d}_i$ and $C_i + p_j > \bar{d}_i$, where C_i is the completion time of job i. In other words, jobs in TS are scheduled as late as possible without missing their deadlines.

A tight schedule with respect to TS can be obtained as follows. Jobs in JS are scheduled backwards, starting at time $t = \sum p_j$. If the job in TS with the largest deadline can be scheduled to complete at time t (i.e., without missing its deadline), then schedule it to complete at time t. Otherwise, schedule the job in $JS \setminus TS$ with the largest due date to

complete at time t. Iterate this process with the remaining jobs starting at time t', where t' is the starting time of the previously scheduled job.

It is easy to see that for any optimal schedule, there is another optimal schedule that is tight. Thus, it is sufficient to concentrate on tight schedules only. The algorithm given below solves Case 1.

Algorithm 1

Input: A set of n jobs

Output: A schedule S that minimizes $\sum U_j$ subject to the condition that every job meets its deadline

1. Schedule the jobs in EDD order. If there is a tie, schedule the one with the smallest processing time first. Let the schedule be S
2. Let $TS = \emptyset$ be the initial tardy set
3. Repeat until every tardy job in S is contained in TS:
   (a) let $i \notin TS$ be the first tardy job in S
   (b) $TS = TS \cup \{i\}$
   (c) reschedule the n jobs such that S is a tight schedule with respect to TS

Theorem 6.2.1 *The schedule produced by Algorithm 1 is optimal for the problem* $1 \mid \bar{d}_j \mid \sum U_j$ *under the condition that* $d_i \leq d_j$ *implies* $\bar{d}_i \leq \bar{d}_j$ *and* $p_i \leq p_j$.

Proof: Assume that $d_i \leq d_{i+1}$ for $1 \leq i \leq n-1$. Let $TS = \{i_1, \ldots, i_k\}$, $i_1 < i_2 < \ldots < i_k$, be the tardy set obtained by Algorithm 1. The theorem is proved by showing that any feasible and tight schedule must have at least k tardy jobs. It is sufficient to prove that (1) there must be at least m tardy jobs from the job set $\{1, 2, 3, \ldots, i_m\}$, $1 \leq m \leq k$, in any feasible schedule, and (2) if a feasible schedule chooses $\{j_1, j_2, \ldots, j_m\}$ to be the tardy jobs, then $d_{j_k} \leq d_{i_k}$, $1 \leq k \leq m$. This is proved by induction on m.

By the algorithm, i_1 is the first tardy job in the EDD schedule. Hence, there must be at least one tardy job from the jobs $1, \ldots, i_1$ in any feasible schedule. By assumption, i_1 has the largest due date. Thus, (1) and (2) are true for $m = 1$.

Suppose it holds for $m-1$. Let S' be any feasible and tight schedule. By the induction hypothesis, there are at least $m-1$ tardy jobs from the jobs $1, \ldots, i_{m-1}$. Let these tardy jobs be $TS' = \{j_1, j_2, \ldots, j_{m-1}\}$. It will be shown, by contradiction, that there is at least one more tardy job from $1, \ldots, i_m$ in S'. Suppose all jobs x, $1 \leq x \leq i_m$ and $x \notin TS'$, are on time in S'. Since i_m has the largest due date and since S' is a tight schedule, all other on-time jobs $x < i_m$ may be assumed to be scheduled before i_m. Let TS'_1 be the subset of TS' consisting of all jobs scheduled after i_m in S'. Since i_m is on time,

$$C'_{i_m} = \sum_{j=1}^{i_m} p_j - \sum_{j \in TS'_1} p_j \leq d_{i_m}$$

where C'_{i_m} is the completion time of job i_m in S'. Consider the schedule S at the beginning of the m-th iteration of Algorithm 1. For each job $j_x \in TS'$, by the induction hypothesis, there must be a tardy job $i_x \in TS$ such that $d_{i_x} \geq d_{j_x}$ which implies $\bar{d}_{i_x} \geq \bar{d}_{j_x}$ and $p_{i_x} \geq p_{j_x}$. Thus, if job j_x can be scheduled after i_m in S', job i_x can also be scheduled after i_m in S without missing its deadline. Let TS_1 be the set of tardy jobs scheduled after i_m in S, then $\sum_{j \in TS_1} p_j \geq \sum_{j \in TS'_1} p_j$. Thus,

$$C_{i_m} = \sum_{j=1}^{i_m} p_j - \sum_{j \in TS_1} p_j \leq \sum_{j=1}^{i_m} p_j - \sum_{j \in TS'_1} p_j \leq d_{i_m}$$

where C_{i_m} is the completion time of job i_m in S. This is a contradiction, since i_m is a tardy job at the beginning of the m-th iteration. Thus, S' also has m tardy jobs. Since i_m has the largest due date among the jobs $1, 2, 3, \ldots, i_m$, (2) follows immediately. ■

Corollary 6.2.2 *Under the condition stated in Theorem 6.2.1, problems $P1$ to $P8$ can be solved in polynomial time.*

6.2.2 Case 2

In this subsection the condition that $p_i \geq p_j$ implies $sl_i \leq sl_j$ is assumed, where $sl_k = d_k - p_k$ is the slack of job k. As noted above, this case includes the case where $d_i \leq d_j$ implies $p_i \geq p_j$. The algorithm to solve Case 2 is given below.

Algorithm 2

Input: A set of n jobs

Output: A schedule with the minimum $\sum U_j$ such that every job meets its deadline

1. $t \leftarrow \sum_{j=1}^{n} p_j$
2. Repeat until all jobs have been scheduled:

   let i be the unscheduled job with the largest due date

   If $d_i \geq t$

   schedule job i in the time interval $(t - p_i, t]$

   $t \leftarrow t - p_i$

   Else

   let j be the unscheduled job with the largest processing time such that $\bar{d}_j \geq t$. In case of a tie, choose the one with the smallest due date

   schedule job j in the time interval $(t - p_j, t]$

   $t \leftarrow t - p_j$

Theorem 6.2.3 *The schedule produced by Algorithm 2 is optimal for the problem* $1 \mid \bar{d}_j \mid \sum U_j$ *under the condition that* $p_i \geq p_j$ *implies* $sl_i \leq sl_j$.

Proof: Given two jobs i and j such that $p_i \geq p_j$, job i is said to be *dominate* job j if $sl_i \leq sl_j$. Suppose there are two jobs i and j such that job i dominates job j. If both jobs become tardy when scheduled to complete at time t and one of them has to be scheduled to complete at time t, then in order to minimize the number of tardy jobs, it is sufficient

to schedule job i to complete at time t (see [11], [20]). In other words, the non-dominated job should be chosen. Since at each time t, the algorithm either schedules an on-time job, or a tardy job that dominates all other jobs whose deadlines are at least t, the optimality of Algorithm 2 follows immediately. ■

Corollary 6.2.4 *Under the condition stated in Theorem 6.2.3, problems $P1$ to $P8$ can be solved in polynomial time.*

6.2.3 Case 3

Case 3 is a combination of Case 1 and Case 2; i.e., there is an integer m, $1 < m < n$, such that the first m jobs satisfy the condition of Case 1, the last $n - m$ jobs satisfy the condition of Case 2, and the largest deadline in the first m jobs is less than or equal to the smallest deadline in the last $n - m$ jobs.

Algorithm 3

Input: A set of n jobs satisfying the condition of Case 3.

Output: A schedule S with minimum $\sum U_j$ such that every job meets its deadline

1. $JS \leftarrow \{1, \ldots, n\}$, $JS_1 \leftarrow \{1, \ldots, m\}$ and $JS_2 \leftarrow \{m+1, \ldots, n\}$
2. Apply Algorithm 1 to JS_1 to obtain a schedule S_1 with tardy set TS_1
3. Let S be the tight schedule of JS with respect to TS_1
4. $TS \leftarrow TS_1$ and $t \leftarrow \sum_{j=1}^{n} p_j$
5. Repeat until each job in S is either on time or a tardy job in TS
   (a) scan S backwards from t. let i be the first tardy job in S such that $i \notin TS$
   (b) let t_0 be the completion time of job i in S
   (c) pick a job x from those jobs scheduled in the interval $(0, t_0]$ as follows:
   let $x_1 \notin TS$ be the job in JS_1 with the largest processing time. if there is a tie, choose the one with the largest due date

let x_2 be the job in JS_2 with the largest processing time such that $\bar{d}_{x_2} \geq t_0$. if there is a tie, choose the one with the smallest due date

If $\bar{d}_{x_1} < t_0$

$x \leftarrow x_2$

Else If $p_{x_1} \geq p_{x_2}$

$x \leftarrow x_1$

Else

let $u \in JS_2$ be the first job scheduled after x_1 in S such that $p_u > p_{x_1}$

If there is at least one tardy job $z \in JS_2$ scheduled between x_1 and u in S

$x \leftarrow x_1$

Else

$x \leftarrow x_2$

(d) $TS \leftarrow TS \cup \{x\}$

(e) $t \leftarrow t_0 - p_x$

(f) obtain a tight schedule S in the time interval $(0, t]$ with respect to the tardy jobs in TS that are scheduled before t

6. Output S

Theorem 6.2.5 *The schedule S produced by Algorithm 3 is optimal for the problem* $1 \mid \bar{d}_i \mid \sum U_j$ *if there is an integer* m, $1 < m < n$, *such that (1) for* $1 \leq i < j \leq m$, $d_i \leq d_j$ *implies* $\bar{d}_i \leq \bar{d}_j$ *and* $p_i \leq p_j$, *(2)* $\max_{j=1}^{m}\{\bar{d}_j\} \leq \min_{j=m+1}^{n}\{\bar{d}_j\}$, *and (3) for* $m < i, j \leq n$, $p_i \geq p_j$ *implies* $sl_i \leq sl_j$.

Proof: Let S be the schedule produced by Algorithm 3. S is proved to be optimal by showing that there is an optimal schedule S^* such that (1) $TS_1 \subseteq TS^*$, where TS^* is the set of tardy jobs in S^* and TS_1 is the tardy set obtained at step 2 of the algorithm; (2) In the time interval $(t, \sum_{j=1}^{n} p_j]$, where t is the time instant obtained at step (5e) in the last

iteration of the repeat loop in the algorithm, S has the same set of jobs scheduled in the same order as S^*; i.e., S^* and S are identical in this time interval.

If (1) and (2) could be proved, then S and S^* have the same set of jobs scheduled in the time interval $(0, t]$. All the tardy jobs in this interval in S are contained in TS_1. By (1), they are also tardy jobs in S^*. Thus, S must be optimal for the set of jobs scheduled in the time interval $(0, t]$. Combining with (2), S is an optimal schedule for the n jobs.

First, (1) is proved. Let $TS_1 = \{i_1, i_2, \ldots, i_k\}$ and $i_j < i_{j+1}$ for $1 \leq j \leq k-1$. Suppose that $i_1, \ldots, i_{j-1}$ are all in TS^* but $i_j \notin TS^*$. It will be shown that another optimal schedule S' can be obtained from S^* such that i_j is a tardy job.

As noted in subsection 3.1, S^* may be assumed to be a tight schedule. By the proof of Theorem 6.2.1, there are at least j tardy jobs from the jobs $1, \ldots, i_j$ in S^*. Since i_j is on time in S^*, there must be another tardy job that is not in TS_1, has due date at most d_{i_j} (and hence processing time at most p_{i_j}), and is scheduled after i_j in S^*. Let j' be such a tardy job with the smallest due date. Let IS_1 be the set of jobs scheduled before i_j in S^*, IS_2 be the set of jobs scheduled between i_j and j', and IS_3 be the set of jobs scheduled after j'; see Figure 6.1(a). Let the start time of i_j in S^* be t_1 and the completion time of j' be t_2; see Figure 6.1(a).

A new schedule S' is obtained from S^* as follows. Let TS_1^* be the tardy jobs of S^* scheduled in the time interval $(0, t_1]$. In S', the jobs in $IS_1 \cup \{j'\}$ are scheduled first so that it is tight with respect to TS_1^*, then the jobs in IS_2 in the same order as S^*, then the job i_j, and finally the jobs in IS_3 in the same order as S^*; see Figure 6.1(a). It is easy to see that the jobs in IS_2 complete in S' no later than in S^* and the jobs in IS_3 complete in S' at the same time as in S^*. Since $d_{j'} \leq d_{i_j}$, $\bar{d}_{j'} \leq \bar{d}_{i_j}$. Since job j' can meet its deadline in S^*, job i_j can also meet its deadline in S'. To show that S' is optimal, it is sufficient to show that all jobs in $IS_1 \cup \{j'\}$, except those in TS_1^*, are on time.

By Theorem 6.2.1, the tight schedule $S^*_{i_j-1}$ of the jobs $1, 2, \ldots, i_j - 1$ with respect to the tardy set $\{i_1, i_2, \ldots, i_{j-1}\}$ has all jobs on time, except i_k, $1 \leq k \leq j-1$. On

the other hand, by assumption, every job i_k, $1 \leq k \leq j-1$, is a tardy job in S^*. Since $j' \in \{1, 2, \ldots, i_j - 1\}$, j' must be on time in $S^*_{i_j-1}$ and hence in S'. For any job $x \in IS_1$, if it is on time in S^*, it must still be on time in S'.

In summary, S' has the same number of tardy jobs as S^*, but S' has i_j as a tardy job. By induction, one can show that there is an optimal schedule that contains all tardy jobs in TS_1.

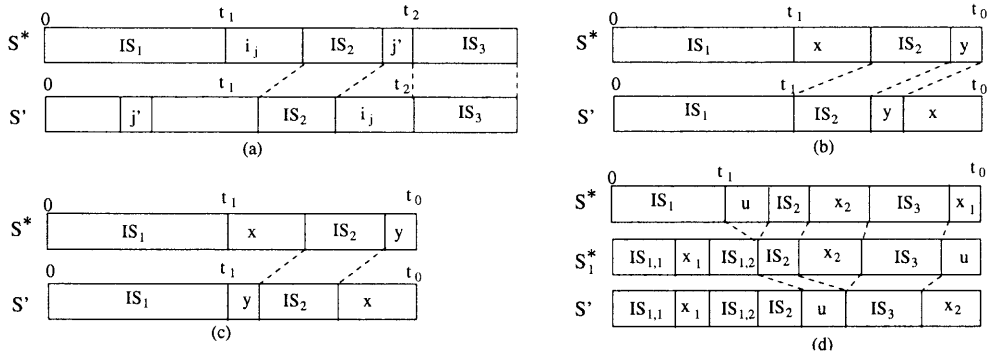


Figure 6.1 A new schedule S' obtained from the optimal schedule S^*.

Now, (2) is proved: S is identical to S^* in the time interval $(t, \sum_{j=1}^{n} p_j]$, where t is the time instant obtained at step (5e) in the last iteration of the repeat loop in the algorithm. This will be proved backwards. Suppose that S is identical to S^* in the time interval $(t_0, \sum_{j=1}^{n} p_j]$, $t < t_0 \leq \sum_{j=1}^{n} p_j$, and the job that completes at t_0 in S is job x. Suppose $y \neq x$ is the job that completes at t_0 in S^*. It will be shown that another optimal schedule S' can be obtained from S^* such that job x completes at t_0 in S'. Thus, S is identical to S' in the time interval $(t_0 - p_x, \sum_{j=1}^{n} p_j]$, and hence (2) holds in the time interval $(t_0 - p_x, \sum_{j=1}^{n} p_j]$ as well. There are three cases to consider.

Case I: x is on time in S.

In this case $d_x \geq t_0$. One can take x out of S^*, compact the schedule, and insert x after y; see Figure6.1(b). In the new schedule S', x is still on time. Furthermore, all other jobs complete in S' no later than in S^*. Thus, S' is also optimal.

Case II: x is tardy in S and $x \in TS^*$.

In this case $\bar{d}_x \geq t_0$. S' is formed exactly as in Case I; see Figure6.1(b). In S' all jobs, except x, complete no later than in S^*. Thus, S' has the same number of tardy jobs as in S^*, and hence S' is also optimal.

Case III: x is tardy in S and $x \notin TS^*$.

In this case job x is tardy in S but on time in S^*. Let S" be the schedule at the beginning of the repeat loop in the algorithm when x is picked as a tardy job, and let TS" be the tardy set at that time. Note that $TS" \subseteq TS^*$, and S" and S^* are both tight schedules in the time interval $(0, t_0]$. Job y may be assumed to be tardy in S^*. For if y were on time in S^*, then both x and y could have to be on time in S^*. Since S" and S^* are both tight schedules in the time interval $(0, t_0]$, S" would have y scheduled to complete at time t_0 as well.

The proof in this case is quite involved. In the following three observations will be given that will be used later in the proof. Let x_1 and x_2 be as defined in the algorithm.

Observation 1: Since x_1 has a smaller due date than any job in JS_2, x_1 dominates any job $k \in JS_2$ if $p_k \leq p_{x_1}$. In particular, x_1 dominates z, where z is as defined in the algorithm.

Observation 2: Let $k \notin TS_1$ be a job in JS_1 such that $d_k \leq d_{x_1}$. If in an optimal schedule x_1 is on time while k completes later than d_{x_1} (i.e., k is tardy), then there is another optimal schedule in which x_1 is tardy while k is on time.

The correctness of Observation 2 follows from the fact that TS_1 is an optimal tardy set of JS_1 which is also a subset of TS^*. Let IS be the set of jobs scheduled before k in the optimal schedule. A new schedule can be obtained that starts with a tight schedule of the jobs in the set $IS \setminus \{x_1\} \cup \{k\}$, then the job x_1, and finally the remaining jobs in the same order as in the optimal schedule. Using similar arguments as (1) is proved, it will be shown that k becomes on time in the new schedule and if any other job is on time in the optimal schedule, it will still be on time in the new schedule. Thus, the new schedule is also optimal.

Observation 3: Since x_2 has the largest processing time among those jobs of JS_2 that are scheduled before t_0 and have deadlines at least t_0, by assumption, x_2 dominates all these jobs.

It is now shown how to obtain another optimal schedule S' from S^* such that x completes at t_0 in S'. Consider two cases of y: $y \in JS_1$ and $y \in JS_2$. If $y \in JS_1$, then since $p_{x_1} \geq p_y$ and larger processing times imply larger due dates for jobs in JS_1, $d_y \leq d_{x_1}$. Also x_1 is on time in S^*; otherwise, x_1 and y are both tardy in S^* and x_1 would have been scheduled to complete at time t_0, rather than y (since S^* is a tight schedule in the time interval $(0, t_0]$). Applying Observation 2 where y plays the role of k, $y = x_1$ may be assumed. If $y \in JS_2$, by Observation 3, $y = x_2$ may be assumed. By the algorithm, either $x = x_1$ or $x = x_2$. Thus, there are two subcases to consider: (i) $x = x_1$ and $y = x_2$; (ii) $x = x_2$ and $y = x_1$.

Subcase (i): $x = x_1$ and $y = x_2$.

In this case job x_1 is tardy in S but on time in S^*. Also, x_2 is tardy in S^*. If $p_{x_1} \geq p_{x_2}$, then x_1 dominates x_2, by Observation 1. Thus, another optimal schedule S' can be obtained such that x_1 completes at time t_0 and x_2 becomes on time in S'.

From the above discussion, $p_{x_1} < p_{x_2}$ may be assumed. According to the algorithm, there is a tardy job $z \in JS_2$ scheduled between x_1 and u, where u is as defined in the algorithm. Note that $z \notin TS"$. Since $p_{x_1} \geq p_z$, x_1 dominates z. Now, if z is a tardy job in S^*, then another optimal schedule S' can be obtained with x_1 being tardy and completing at time t_0 and z being on time. On the other hand, if z is on time in S^*, then there must be a tardy job $x_0 \notin TS"$ such that x_0 is scheduled after z but before t_0 in S^*. This is because S and S^* are identical after t_0, $TS" \subseteq TS^*$, and $S"$ has z as a tardy job but S^* has z as an on-time job. Moreover, $d_{x_0} < d_z$, since jobs with due date larger than z cannot help make z on time in S^*. There are two cases to consider. If $x_0 \in JS_1$, then $p_{x_0} \leq p_{x_1}$ (and hence $d_{x_0} \leq d_{x_1}$), since x_1 has the largest processing time by the algorithm. Applying Observation 2 where x_0 plays the role of k, another optimal schedule S' can be obtained

such that x_1 becomes tardy and completes at time t_0 and x_0 becomes on time. On the other hand, if $x_0 \in JS_2$, then x_0 must appear before z in S", since $d_{x_0} \leq d_z$. Moreover, x_0 cannot appear before x_1 in S", since x_1 has smaller due date than x_0. Thus, x_0 must appear after x_1 but before z in S". By the algorithm, $p_{x_1} \geq p_{x_0}$, and hence x_1 dominates x_0, by Observation 1. Thus, another optimal schedule S' can be obtained such that x_1 becomes tardy and completes at time t_0 and x_0 becomes on time in S'.

Subcase (ii): $x = x_2$ and $y = x_1$.

In this case job x_2 is tardy in S but on time in S^*. Also, job x_1 is tardy in S^*. Let u be as defined in the algorithm. Since x_1 can complete at time t_0 and since x_1 has deadline no larger than that of u, u must also be able to complete at time t_0. By the algorithm, x_2 dominates u. If u is a tardy job in S^*, another optimal schedule S' can be obtained from S^* such that x_2 is tardy and u is on time in S'. Thus, both u and x_2 may be assumed to be on time in S^*. In this case S' is obtained from S^* in two steps. Let IS_1 be the set of jobs scheduled before u, IS_2 be the set of jobs scheduled between u and x_2, and IS_3 be the set of jobs scheduled between x_2 and x_1, see Figure 6.1(d).

In the first step, a new schedule S_1^* is obtained in the time interval $(0, t_0]$ that starts with a tight schedule of the jobs in $IS_1 \cup \{x_1\}$, followed by the jobs in IS_2, followed by the job x_2, followed by the jobs in IS_3, and finally followed by the job u. Since $p_u > p_{x_1}$, this change can only affect the jobs in IS_1 (in terms of tardy job or on-time job). Let $IS_{1,1}$ be the set of jobs scheduled before x_1 and $IS_{1,2}$ be the set of jobs scheduled after x_1 in S_1^*. It can be shown that x_1 is on time in $S1^*$. Thus, only those jobs in $IS_{1,2}$ need to be considered that were previously on time in S^*. Note that these jobs must belong to JS_2, since x_1 has the largest due date among the jobs in JS_1. By the algorithm, all the jobs between x_1 and u that are in JS_2 are on time in S". Since $TS" \subseteq TS^*$, these jobs must also be on time in S_1^*.

In the second step, S' can be obtained from S_1^* with x_2 being tardy and u being on time, since x_2 dominates u; see Figure 6.1(d).

In all three cases it has been shown that there is an optimal schedule with the same job x completing at time t_0. Therefore, (2) holds in the time interval $(t_0 - p_x, \sum_{j=1}^{n} p_j]$ as well. ■

Corollary 6.2.6 *Under the conditions stated in Theorem 6.2.5, problems $P1$ to $P8$ can be solved in polynomial time.*

6.3 Heuristics and Experimental Results

Since problems $P5$ and $P7$ are NP-Hard, fast heuristics will be proposed for them. Again, the problem $1 \mid \bar{d}_j \mid \sum U_j$ will be used as a substitute. Note that the heuristics are also applicable to problems $P1$ and $P3$. Although problems $P6$ and $P8$ are also NP-Hard, no good heuristics have been devised for them.

The heuristics fall into three categories. Heuristics that belong to the first category schedule jobs backwards, starting at time $t = \sum p_j$. The heuristics determine by some rules a job to complete at time t. Then, t is decremented by the processing time of the chosen job and the process is iterated to schedule the remaining jobs.

Heuristics that belong to the second category first construct an EDD schedule S, and initialize the tardy set TS to be the empty set. It then repeats the following until every tardy job in S is already in TS: (1) Locate the first tardy job i in S that is not in TS; (2) From those jobs scheduled before and including i, pick a job according to some rule and put it into TS; (3) Obtain a tight schedule S with respect to TS; (4) If a tardy job becomes on time in S, delete the job from TS.

The third type of heuristics, called the *hybrid-scheduling* heuristic, schedules jobs in the same manner as the second type, except that at each iteration the tardy set is updated with respect to the jobs scheduled up to and including job i. This update process is done by a backward scheduling algorithm.

In the next subsection the heuristics will be described in detail. In subsection 4.2, the worst-case ratios of these heuristics will be discussed. All of these heuristics, except the

hybrid-scheduling heuristic, have unbounded worst-case ratios. While a constant bound cannot be proved for the hybrid-scheduling heuristic, it is not possible to come up with an example with a worst-case ratio larger than two. Finally, in subsection 4.3, an empirical study will be reported and the effectiveness between the various heuristics as well as relative to optimal solutions will be compared. According to the result, the hybrid-scheduling heuristic is the best among all heuristics and its average performance is within 1% more than the optimal value. If all heuristics are run and the best solution is chosen, then the composite heuristic has average performance within 0.7% more than the optimal value. The result shows that extremely good solutions can be obtained within a reasonable amount of time.

6.3.1 Heuristics

The first type of heuristics, *backward-scheduling heuristic*, schedules jobs backwards. Depending on the implementation of step 3(a), there are two different heuristics. The first heuristic, denoted by LPT-B (Largest Processing Time Backward), picks the job i with the largest processing time. In case of a tie, choose the one with the smallest due date. The rationale is that there may be many jobs scheduled after i (in the EDD schedule) that have tardiness smaller than p_i. By scheduling i to complete at time t, these jobs will become on time.

The second heuristic, denoted by LS-B (Largest Score Backward), computes a score for each job and picks the one with the highest score. The *score* of each job i reflects the number of tardy jobs that can be made on time if i were scheduled to complete at time t, and it is computed as follows. Let S' be the EDD schedule of all unscheduled jobs, starting at time 0. If job i is tardy in S', then the score of i is defined to be the number of tardy jobs j scheduled after i in S' such that $T_j \leq p_i$; otherwise, its score is this number less 1. Note that the score of job i is the net decrease of tardy jobs if job i were scheduled to complete

at time t. The rationale is that by scheduling the job with the highest score, more jobs can be made on time.

There are two ways to break ties in the LS-B heuristic; i.e., when several jobs have the same (highest) score. One way is to choose the one with the largest processing time, denoted by LS-P. Another way is to choose the one with the smallest due date, denoted by LS-D. Both heuristics are implemented in the experiment.

Backward-Scheduling Heuristic

Input: A set of n jobs

Output: A feasible schedule if one exists

1. $t \leftarrow \sum_{j=1}^{n} p_j$
2. $JS \leftarrow \{1, 2, ..., n\}$
3. Repeat until all jobs are scheduled or no job can be scheduled at t:

    If there is a job i such that $d_i \geq t$

        schedule i in $(t - p_i, t]$

        delete i from JS

        $t \leftarrow t - p_i$

    Else

        If every job in JS has deadline at least t

            schedule all jobs in JS using the Hodgson-Moore algorithm

        Else

        (a) from among those jobs whose deadline is at least t, choose a non-dominated job i according to some rule. schedule i in the time interval $(t - p_i, t]$

        (b) delete i from JS

        (c) $t \leftarrow t - p_i$

The second type of heuristics, *forward-scheduling heuristic*, schedules jobs forward. Depending on the implementation of step (3c), there are again two different heuristics. The first heuristic, denoted by LPT-F (Largest Processing Time Forward), picks the job with the largest processing time that can be scheduled after the current job i. In case of a tie, pick the one with the largest deadline. The second heuristic, denoted by LDL-F (Largest Deadline Forward), picks the job with the largest deadline that can be scheduled after the current job i. In case of a tie, pick the one with the largest processing time.

Forward-Scheduling Heuristic

Input: A set of n jobs

Output: A feasible schedule if one exists

1. Let S initially be the EDD schedule
2. Let tardy set TS initially be empty
3. Repeat until every tardy job in S is in TS or no feasible schedule can be found:
   (a) let i be the first tardy job in S such that $i \notin TS$
   (b) let job i completes at time t
   (c) from among those jobs scheduled before and including i, pick a non-dominated job j with deadline at least t according to some rules
   (d) $TS = TS \cup \{j\}$
   (e) obtain a tight schedule S with respect to TS
   (f) if a job $j \in TS$ becomes on time in S, delete j from TS

The third type of heuristics, called *hybrid-scheduling heuristic*, schedules jobs forward, but at each iteration the tardy set is updated by a backward scheduling algorithm.

Let JS be a set of n jobs, $1, \ldots, n$. Let TS be a tardy set of JS such that if a tight schedule of JS is formed with respect to TS, then the remaining jobs in JS will be on time. Define an operation, *update tardy set TS for job set JS*, as shown below. The update

operation has the effect of reducing the number of tardy jobs in TS without reducing the possible number of on-time jobs in JS.

Update(TS,JS)

Input: A job set JS and a tardy set $TS \subseteq JS$

Output: A tardy set $\hat{TS} \subseteq TS$

1. $t \leftarrow \sum_{i \in JS} p_i$
2. Repeat until every job in JS is scheduled:
   (a) let i be the job in JS with the largest due date
   (b) If $d_i \geq t$
   schedule i to complete at time t
   Else
   let $CS \subseteq TS$ be the set of unscheduled jobs whose deadline is at least t
   If $\mid CS \mid = 1$
   schedule the job in CS to complete at time t
   Else
   let TS' be the unscheduled jobs of TS and $TS" = TS' \setminus CS$
   let JS' be the unscheduled jobs (including those in TS')
   obtain a tight schedule S' of JS' with respect to $TS"$
   let x be the first tardy job in S'
   pick the job $y \in CS$ scheduled before and including x with the largest processing time
   schedule y to complete at time t
   (c) if there is a job in TS that becomes on time, delete the job from TS
   (d) decrement t by the processing time of the job that was chosen to complete at time t
3. $\hat{TS} \leftarrow TS$

4. return $\hat{TS}$

Note that y must exist, since x is either a job in TS or an on-time job when all jobs in TS are tardied. The hybrid-scheduling heuristic is given as follows.

Hybrid-Scheduling Heuristic

Input: A set of n jobs

Output: A feasible schedule S if one exists

1. Let JS be the set of n jobs
2. Let S initially be the EDD schedule
3. Let tardy set TS initially be empty
4. Repeat until every tardy job in S is in TS or no feasible schedule can be found:
   (a) let i be the first tardy job in S such that $i \notin TS$
   (b) from among those jobs scheduled before and including i, pick a job j such that $\bar{d}_j \geq t$ and such that it has the largest processing time
   (c) $TS \leftarrow TS \cup \{j\}$
   (d) $TS \leftarrow$ Update(TS, $\{1, 2, ..., i\}$)
   (e) obtain a tight schedule S of JS with respect to TS

One can easily show that at step (4b), if a job $j \neq i$ is picked, then i must become on time in the schedule obtained in step (4e).

6.3.2 Worst-Case Bounds

Let k_O be the number of tardy jobs in an optimal schedule. A trivial upper bound for the performance ratio of any heuristic would be n/k_O. The worst-case bounds of the performance ratios of the above heuristics are sought. Unfortunately, all of the heuristics, except the hybrid-scheduling heuristic, have performance ratios asymptotically not much better than $O(n/k_O)$, even if $\bar{d}_j = d_j + C$, where C is a constant. This is shown by

giving instances such that when certain heuristics are applied to them, a performance ratio of $O(n/k_O)$ is obtained. All of these instances can be generalized to arbitrarily large n. For the hybrid-scheduling heuristic, the largest ratio that has been found so far is 2. It is conjectured that this is a tight bound.

Table 6.2 gives an instance where the performance ratios of LDL-F and LPT-F are unbounded. Table 6.3 gives an instance for LPT-B, while Table 6.4 gives an instance for LS-B. Both tables show that the performance ratios are unbounded. Note that Table 6.4 is applicable to both LS-P and LS-D heuristics, since there are no ties when the LS-B heuristic is applied to the instance. Table 6.5 gives an instance for the hybrid-scheduling heuristic with a performance ratio of 2.

6.3.3 Experimental Results

An empirical study of the heuristics discussed in Section 6.3.1 is performed for the $1 \mid \bar{d}_j \mid \sum U_j$ problem. While these heuristics have large or unbounded worst-case ratios, they perform very well in practice, as shall be seen later.

Data Generation Instances are characterized by three parameters: number of jobs n, due date range factor R and tardiness factor τ. The factor R controls the range of the due date distribution, while τ provides an indication of the average tightness of the due dates, see also [4], [36], [52]. In the experiment, let $n \in \{10, 20, 50, 100, 150, 200\}$, $R \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ and $\tau \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$.

Two cases are investigated, depending on the difference between the due date and the deadline: (1) the difference between $\bar{d}_j$ and d_j is a constant, which corresponds to the $T_{\max}$ criterion, and (2) the difference between $\bar{d}_j$ and d_j is a function of w_j, which corresponds to the $\max\{w_j T_j\}$ criterion. Instead of generating deadlines randomly, a weight for each job is generated. In the first case every job has a weight 1. In the second case a weight w_j for job j is generated from the uniform distribution [1, 10]. Then Lawler's algorithm [41] is used to compute $T_w^* = \max\{w_j T_j\}$. Finally, the deadline is computed as $\bar{d}_j = d_j + T_w^*/w_j$.

The effect of processing times on the heuristics is also investigated. Two types of instances are generated, one in which the processing time is drawn uniformly from $[1, 10]$ (which corresponds to the case where the largest and the smallest processing times do not differ by much) and the other in which the processing time is drawn uniformly from $[1, 50]$ (which corresponds to the case where the largest and the smallest processing times differ a lot).

Five instances are generated for each given n, R, τ, processing time range and weight range. For each instance, the processing time of each job is first generated from the uniform distribution of the given processing time range; i.e., either [1, 10] or [1,50]. Then an integer due date d_i for each job i is randomly generated from the uniform distribution $\left[\sum_{j=1}^{n} p_j(1 - \tau - R/2), \sum_{j=1}^{n} p_j(1 - \tau + R/2)\right]$. Since there are six values of n, five values of R, five values of τ, two processing time ranges and two weight ranges, a total of 3,000 instances were generated.

Empirical Results and Analysis All of the algorithms are implemented in C++. The running environment is based on the RedHat Linux 7.0 operating system. The PC used is a Pentium II 400Mhz with 128MB RAM. To test the performance of the heuristics relative to the optimal solution, an enumerative algorithm was developed to find the optimal value. No attempt is made to optimize the running time of the enumerative algorithm, since the objective is only to compare the performance of the heuristics with the optimal solution. The time limit of running the enumerative algorithm is set to seven days. If the algorithm does not terminate in seven days, the enumerative algorithm is deemed as not being able to find an optimal solution. In this case the instance will be discarded and it will not be included in the statistics. Out of the 3,000 instances generated, the enumerative algorithm fails to find an optimal solution in only 38 instances.

The heuristics run very fast, in matters of seconds and minutes. The enumerative algorithm takes hours and days to run in some instances. The heuristics are fast enough

to be able to meet real-time environment, while the enumerative algorithm most likely can not.

For each instance, the enumerative algorithm and all six heuristics are applied. The results[2] are summarized in Tables 6.6-6.9. Tables 6.6 and 6.7 give the statistics for the unweighted case with processing time ranges $[1, 10]$ and $[1, 50]$, respectively. Tables 6.8 and 6.9 give the statistics for the weighted case with processing time ranges $[1, 10]$ and $[1, 50]$, respectively. In each of these tables, instances with n = 10, 20 and 50 are grouped as small instances and instances with n = 100, 150 and 200 as large instances, and there are 375 instances in each group. Statistics are generated for small instances separate from large instances. Statistics for all instances are also generated.

In each of these tables, the first column "Opt" refers to the enumerative algorithm, the next six columns refer to the six heuristics, and the last column "Comp" refers to the composite algorithm of running all six heuristics and outputs the best solution. The row "# of opt" gives the number of instances in which each algorithm generates an optimal solution. The row "#/xxx" gives the fraction of instances in which each algorithm generates an optimal solution. Note that "Opt" always gives a fraction of 1, since instances that take longer than seven days to run are discarded. The row "avg ratio" gives the average performance ratios of the heuristics versus the optimal value, while the row "worst ratio" gives the worst-case ratios.

From the tables the following conclusions can be drawn:

- The composite algorithm has the best performance; its worst-case ratio is never more than 1.25 and its average ratio is never more than 1.007. Since all six heuristics run very fast, it is indeed viable to use the composite algorithm in practice.
- For a single heuristic, the hybrid-scheduling heuristic outperforms all other heuristics. Its worst-case ratio is never more than 1.639 and its average ratio is never more than

[2]The raw data and results are available at "web.njit.edu/~leung/dual-criteria"

1.01. The hybrid-scheduling heuristic outperforms just about every heuristic in every category: number of optimal solution found, average ratio and worst-case ratio.

- While most of the heuristics have unbounded worst-case performance ratios, their performance (both average ratio and worst-case ratio) are much better in practice.
- All heuristics perform better with small instances than large instances.
- All heuristics perform better with the unweighted case than the weighted case.
- Processing time range does not play an important role in the performance of the heuristics.
- Geneally speaking, the backward-scheduling heuristics (LS-P, LS-D and LPT-B) are more effective than the forward-scheduling heuristics (LPT-F and LDL-F).
- Between the two forward-scheduling heuristics, the LPT-F heuristic performs better than the LDL-F heuristic for the unweighted case, while the opposite is true for the weighted case.
- Among the three backward-scheduling heuristics (LS-P, LS-D and LPT-B), both LS-P and LPT-B outperforms LS-D most of the times, while LS-P and LPT-B are comparable with each other.

6.4 Conclusions

In this chapter single-machine scheduling problems with two criteria are studied. The focus is on the number of tardy jobs $\sum U_j$, the maximum tardiness $T_{\max}$ and the maximum weighted tardiness $\max\{w_jT_j\}$. Both dual criteria and hierarchical problems are studied. Altogether eight problems, $P1$ to $P8$ have been considered. If the primary criterion is $T_{\max}$ or $\max\{w_jT_j\}$ and the secondary criterion is $\sum U_j$, the problems can be viewed as special cases of $1 \mid \bar{d}_j \mid \sum U_j$.

The complexity relationships between these eight problems was first established. Then $1 \mid\mid \sum U_j \mid \max\{w_jT_j\}$ and $1 \mid\mid \max\{w_jT_j\} \mid \sum U_j$ are shown to be both NP-Hard. These two results answer the open questions posed by Lee and Vairaktarakis [44]: What is

the complexity of $1 \mid\mid \sum U_j \mid \max\{f_j(C_j)\}$ and $1 \mid\mid \max\{f_j(C_j)\} \mid \sum U_j$? Despite much efforts spent on $1 \mid\mid \sum U_j \mid T_{\max}$ and $1 \mid\mid T_{\max} \mid \sum U_j$, their complexity remain open. It will be worthwhile to settle this issue in the future.

Polynomial-time algorithms are given for three special cases of $1 \mid \bar{d}_j \mid \sum U_j$, which yield polynomial-time algorithms for problems $P1$ to $P8$. For Case 1, if all but one job satisfy the condition of Case 1, the problem can still be solved in polynomial time. This is because the optimal solution either contains the special job or it doesn't. Both solutions can be kept, one containing the special job and the other doesn't, and then the better of the two solutions will be chosen. This idea can be generalized to any fixed number of special jobs. Similar remarks can be made about Case 2. For future research, it will be interesting to identify other special cases that can be solved in polynomial time.

Several fast heuristics have been proposed for the general problem. Among all the heuristics proposed, all except one have unbounded performance ratio in the worst case. The exception is the hybrid-scheduling heuristic for which only a ratio of 2 is found. It has been conjectured that it has a tight worst-case bound of 2. It will be interesting to prove (or disprove) the conjecture.

Empirical study was performed to get a feel for the effectiveness of the heuristics. According to the result, the hybrid-scheduling heuristic gives extremely good solutions within a reasonable amount of time. The average ratio of the hybrid-scheduling heuristic is never more than 1.01, while the worst-case ratio is no more than 1.639. If time permits, all six heuristics should be run and the best solution is chosen. The composite algorithm has even better statistics: average ratio no more than 1.007 and worst-case ratio no more than 1.25.

Table 6.1 The Jobs in the Reduction

j	p_j	d_j	$\bar{d}_j$
$i\ (1 \leq i \leq n)$	$x_{i-1} + a_i$	$\left(\sum_{k=1}^{i} x_{k-1}\right) + A$	$P_{i-1} + x_{i-1} + A$
$n + i\ (1 \leq i \leq n)$	x_{i-1}	$\left(\sum_{k=1}^{i} x_{k-1}\right) + A$	$P_{i-1} + x_{i-1} + 2^n \cdot x_{i-1} - A$
$2n + i\ (1 \leq i < n)$	$2^n \cdot x_{i-1}$	$\left(\sum_{k=1}^{n} x_{k-1} + \sum_{k=1}^{i} 2^n \cdot x_{k-1}\right) + A$	$P_{i-1} + 2^n \cdot x_{i-1} - A$
$3n$	$2^n \cdot x_{n-1}$	$\left(\sum_{k=1}^{n} x_{k-1} + \sum_{k=1}^{n} 2^n \cdot x_{k-1}\right) - A$	$P_{n-1} + 2^n \cdot x_{n-1} - A$
$3n + i\ (1 \leq i < n)$	$2^n \cdot x_{i-1} - 2a_i$	$\left(\sum_{k=1}^{n} x_{k-1} + \sum_{k=1}^{i} 2^n \cdot x_{k-1}\right) + A$	$P_{i-1} + x_{i-1} + 2^n \cdot x_{i-1} - A$
$4n$	$2^n \cdot x_{n-1} - 2a_n$	$\left(\sum_{k=1}^{n} x_{k-1} + \sum_{k=1}^{n} 2^n \cdot x_{k-1}\right) - A$	$P_{n-1} + x_{n-1} + 2^n \cdot x_{n-1} - A$

Table 6.2 An Example with Unbounded Performance Ratio for Both LDL-F and LPT-F, where $\bar{d}_j = d_j + 29$. An Optimal Schedule has 2 Tardy Jobs $\{J_4, J_7\}$, while the LPT-F and LDL-L Schedules have $2n/3$ Tardy Jobs $\{J_1, J_2, J_3, J_4, J_5, J_6\}$

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
p_i	12	11	10	9	8	7	6	6	6
d_i	13	23	33	35	43	50	57	63	69
$\bar{d}_i$	42	52	62	64	72	79	86	92	98

Table 6.3 An Example with Unbounded Performance Ratio for LPT-B, where $\bar{d}_j = d_j + 35$. An Optimal Schedule has 2 Tardy Jobs $\{J_1, J_2\}$, while the LPT-B Schedule has All n Jobs Tardy

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
p_i	35	1	2	3	4	5	6	7	8
d_i	0	35	37	40	44	49	55	62	70
$\bar{d}_i$	35	70	72	75	79	84	90	97	105

Table 6.4 An Example with Unbounded Performance Ratio for LS-B, where $\bar{d}_j = d_j + 62$. This Example will Work For Both LS-P and LS-D, since There is no Tie when the LS-B Heuristic is applied to the instance. An Optimal Schedule has 2 Tardy Jobs $\{J_1, J_8\}$, while the LS-B Schedule has $n/2$ Tardy Jobs $\{J_1, J_4, J_5, J_6\}$

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8
p_i	30	8	8	11	10	10	10	29
d_i	30	57	57	57	57	57	57	77
$\bar{d}_i$	92	119	119	119	119	119	119	139

Table 6.5 An Example with Performance Ratio of 2 for the Hybrid-Scheduling Heuristic, where $\bar{d}_i = d_i + 42$. An Optimal Schedule has 3 Tardy Jobs $\{J_8, J_9, J_{10}\}$, while the Hybrid-Scheduling Heuristic has 6 Tardy Jobs $\{J_1, J_2, J_3, J_{11}, J_{12}, J_{13}\}$

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}	J_{11}	J_{12}	J_{13}
p_i	9	8	7	6	6	6	6	6	6	6	7	8	9
d_i	12	17	24	30	36	42	48	48	48	48	70	73	73
$\bar{d}_i$	54	59	66	72	78	84	90	90	90	90	112	115	117

Table 6.6 Empirical Results for Instances with Processing Time Range $[1, 10]$ and Weight 1

		Opt	Hybrid	LS-P	LS-D	LPT-B	LPT-F	LDL-F	Comp.
small instances	# of opt	375	364	349	296	349	292	269	367
	#/375	1.0	0.971	0.931	0.789	0.931	0.779	0.717	0.979
	avg ratio	-	1.003	1.005	1.024	1.005	1.042	1.059	1.001
	worst ratio	-	1.2	1.2	1.375	1.2	4.0	4.0	1.167
large instances	# of opt	374	327	315	142	315	203	165	344
	#/374	1.0	0.874	0.842	0.380	0.842	0.543	0.441	0.92
	avg ratio	-	1.004	1.005	1.06	1.005	1.031	1.07	1.002
	worst ratio	-	1.154	1.167	1.306	1.167	1.667	1.667	1.091
all instances	# of opt	749	691	664	438	664	495	434	711
	#/749	1.0	0.923	0.887	0.585	0.887	0.661	0.580	0.949
	avg ratio	-	1.003	1.005	1.042	1.005	1.037	1.065	1.002
	worst ratio	-	1.2	1.2	1.375	1.2	4.0	4.0	1.167

Table 6.7 Empirical Results for Instances with Processing Time Range $[1, 50]$ and Weight 1

		Opt	Hybrid	LS-P	LS-D	LPT-B	LPT-F	LDL-F	Comp.
small instances	# of opt	375	360	343	300	343	322	315	365
	#/375	1.0	0.96	0.915	0.8	0.915	0.859	0.84	0.973
	avg ratio	-	1.004	1.008	1.022	1.008	1.025	1.029	1.002
	worst ratio	-	1.333	1.333	1.333	1.333	1.75	1.75	1.25
large instances	# of opt	359	318	290	132	290	222	210	329
	#/359	1.0	0.847	0.774	0.368	0.774	0.602	0.571	0.872
	avg ratio	-	1.004	1.008	1.057	1.008	1.034	1.042	1.003
	worst ratio	-	1.111	1.143	1.327	1.143	1.407	1.519	1.083
all instances	# of opt	734	664	621	432	621	538	520	678
	#/734	1.0	0.905	0.846	0.589	0.846	0.733	0.708	0.924
	avg ratio	-	1.004	1.008	1.039	1.008	1.03	1.035	1.002
	worst ratio	-	1.333	1.333	1.333	1.333	1.75	1.75	1.25

Table 6.8 Empirical Results for Instances with Processing Time Range $[1, 10]$ and Weight Range $[1, 10]$

		Opt	Hybrid	LS-P	LS-D	LPT-B	LPT-F	LDL-F	Comp.
small instances	# of opt	375	352	281	289	281	188	200	361
	#/375	1.0	0.939	0.749	0.771	0.749	0.501	0.533	0.963
	avg ratio	-	1.004	1.023	1.016	1.023	1.098	1.095	1.002
	worst ratio	-	1.25	1.5	1.25	1.5	2.0	2.0	1.08
large instances	# of opt	367	246	163	121	163	83	84	261
	#/367	1.0	0.670	0.444	0.330	0.444	0.226	0.229	0.711
	avg ratio	-	1.01	1.034	1.033	1.034	1.16	1.143	1.007
	worst ratio	-	1.286	1.333	1.182	1.333	2.0	2.0	1.083
all instances	# of opt	742	598	444	410	444	271	284	622
	#/742	1.0	0.806	0.598	0.553	0.598	0.365	0.383	0.829
	avg ratio	-	1.007	1.028	1.025	1.028	1.129	1.119	1.004
	worst ratio	-	1.286	1.5	1.25	1.5	2.0	2.0	1.083

Table 6.9 Empirical Results for Instances with Processing Time Range $[1, 50]$ and Weight Range $[1, 10]$

		Opt	Hybrid	LS-P	LS-D	LPT-B	LPT-F	LDL-F	Comp.
small instances	# of opt	375	353	266	289	266	182	205	362
	#/375	1.0	0.941	0.709	0.771	0.709	0.485	0.547	0.965
	avg ratio	-	1.006	1.029	1.017	1.029	1.108	1.074	1.002
	worst ratio	-	1.273	1.333	1.5	1.333	1.75	1.667	1.143
large instances	# of opt	362	265	159	114	159	83	82	282
	#/362	1.0	0.699	0.436	0.312	0.436	0.229	0.227	0.743
	avg ratio	-	1.01	1.036	1.041	1.034	1.187	1.126	1.007
	worst ratio	-	1.639	1.639	1.639	1.32	1.966	1.724	1.167
all instances	# of opt	737	606	424	402	424	265	287	631
	#/737	1.0	0.822	0.575	0.545	0.575	0.360	0.389	0.856
	avg ratio	-	1.008	1.032	1.029	1.031	1.148	1.1	1.004
	worst raio	-	1.639	1.639	1.639	1.333	1.966	1.724	1.167

CHAPTER 7

CONCLUSIONS

This chapter concludes the dissertation by summarizing its contributions and discussing some possible avenues for future work.

7.1 Summary of Contributions

Online Scheduling. This dissertation considers the situation where tasks, along with their precedence constraints, are released at different times, and the scheduler has to make scheduling decisions without knowledge of future releases. Both preemptive and nonpreemptive schedules are considered. This dissertation shows that *optimal* online algorithms exist for some cases, while for others it is impossible to have one. The results give a sharp boundary delineating the possible and the impossible cases [30].

Fast Implementation of Algorithm. This dissertation considers the problem of scheduling a set of n unit-processing-time tasks, with release time and outtree precedence constraints, on $m \geq 1$ identical and parallel processors so as to minimize the total completion time. This dissertation shows an $O(n \log n)$-time implementation [31] for the algorithm given by Brucker, Hurink and Knust.

Approximation Algorithms. This dissertation considers the problem of scheduling a set of n tasks, with precedence constraints, on $m \geq 1$ identical and parallel processors so as to minimize the mean flow time. Approximation algorithms are presented for intree and arbitrary precedence constraints, respectively [32].

Dual Criteria Scheduling Problems.

(1) This dissertation first considers dual criteria scheduling problems with the following criteria: the number of tardy jobs $\sum U_j$, the total completion time $\sum C_j$ and the total tardiness $\sum T_j$. The complexity question of $1 \mid\mid \sum C_j \mid \sum U_j$ and $1 \mid\mid \sum T_j \mid \sum U_j$ have

been open for a long time. Both problems are shown to be NP-Hard in this dissertation [33].

(2) This dissertation then considers dual criteria scheduling problems with the following criteria: the number of tardy jobs $\sum U_j$, the maximum tardiness $T_{\max}$ and the maximum weighted tardiness $\max\{w_jT_j\}$. Both $1 \mid\mid \max\{w_jT_j\} \mid \sum U_j$ and $1 \mid\mid \sum U_j \mid \max\{w_jT_j\}$ are shown to be NP-Hard even when the penalty function f_j for each job j is simply the weighted tardiness of job j. This dissertation also considers $1 \mid\mid T_{\max} \mid \sum U_j$ and $1 \mid\mid \sum U_j \mid T_{\max}$. Although the complexity result for these two problems are still open, this dissertation shows the complexity relationships between these four dual criteria scheduling problems and gives polynomial algorithms for several special cases of these four problems. For the general case, this dissertation proposes several heuristics, gives the worst case bound for each heuristic, and at last gives a heuristic which shows better performance than others by experiment results [34].

7.2 Future Work

Scheduling issues are fundamental in many diverse applications and novel problems, variants and models will continue to come up. A good understanding of the complexity and algorithms of basic scheduling problems are both necessary and useful for future applications. For example, the online algorithms for some problems in Chapter 2 and the approximation algorithms for the problems in Chapter 4 are based on earlier ideas of Hu's algorithm and Coffman-Graham algorithm. Scheduling theory has been an active area of research for the last four decades and impressive progress has been made on several fundamental problems despite the fact that many open problems and challenges remain. At the end of each chapter specific open problems related to the topics addressed in that chapter have been pointed out. Here are some broader directions for future research.

Dissertation Expansions. There are two interesting research topics left from this dissertation. (1) **Online Scheduling problems.** For those scheduling problems for which

it is impossible to find optimal online algorithms, it is necessary to study approximation algorithms for them. The known result is that for any scheduling problems, if there is an off-line ρ-approximation algorithm, a 2ρ-competitive on-line algorithm can be obtained [55]. The goal is to find better approximation algorithms and get better competitive ratios.
(2)**Multi-Criteria Scheduling Problems.** Left from the dissertation, the complexity question of $1 \mid\mid T_{\max} \mid \sum U_j$ and $1 \mid\mid \sum U_j \mid T_{\max}$ are still open. Except considering single machine dual criteria scheduling problems, many more multiple machine dual criteria scheduling problems are still open and need to be studied. Moreover, a lot of multi-criteria scheduling problems which have a lot of applications in the industrial ares should also be considered.

Scheduling Problems with Processors Subject to Breakdown and Repair. A broad class of challenging scheduling problems whose complexity is not well understood is that of scheduling jobs on the processors which are identical and subject to breakdown and repair. Working with processors that are subject to breakdowns is an important issue since a breakdown has a direct impact on the number of available processors. Assume at time t the number of processors equals to $m(t)$, preemptions are allowed, and job j have a due date d_j. The complexity of $Pm(t) \mid prmt \mid \sum C_j$ remains open and the worst-case bound of the Preemptive SPT rule for this problem is still not known. The Preemptive SPT rule is an online algorithm. What is the competitive ratio of the Preemptive SPT rule compared with an optimal offline algorithm? What is the complexity of $Pm(t) \mid prmt, intree \mid C_{max}$ and of $Pm(t) \mid prmt, outtree \mid C_{max}$? While the intree and outtree have the same complexity when the number of processors is fixed, this may not be the case when the number of available processors varies over time. Thus an important direction for future work is to find answers for these questions.

Open Scheduling Problems. In their website, Dr. Peter Brucker and Dr. Sigrid Knust[8] listed the complexity results for all classes. These results include the maximal polynomially solvable problems, maximal pseudopolynomially solvable problems, minimal

NP-hard problems, minimal open problems and maximal open problems. It will be interesting to solve some of them.

New Topics in Algorithm and Computational Complexity. Many new topics have emerged in algorithm and computational complexity area, for example, quantum computing, the complexity of real number computations, zero-knowledge proof systems, average complexity theory, trade-offs between computational resources, computational biology etc. Not much results have been obtained in these new topics, so there are still lots of work that can be done.

REFERENCES

[1] AKKER, M. VAN DEN AND HOOGEVEEN, H., *Minimizing the number of tardy jobs*, In J.Y-T. Leung (ed.), Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman and Hall/CRC, Boca Raton, FL, USA, (2004).

[2] AHO, A. V. , HOPCROFT, J. E. AND ULLMAN, J. D., *The Design and Analysis of Computer Algorithms,* Addison Wesley, Reading, Massachusetts, (1974).

[3] ALBERS, S. AND LEONARDI, S., *On-line algorithms*, ACM Computing Surveys (CSUR), 31 (3es) (1999).

[4] BAKER, K.R. AND MARTIN, J.B., *An experimental comparison of solution algorithms for the single machine tardiness problem*, Naval Research Logistics Quarterly, 21 (1974), pp. 187–200.

[5] BAPTISTE, PH., BRUCKER, P., KNUST, S. AND TIMKOVSKY, V. G., *Fourteen notes on equal-processing-time scheduling*, OSM Reihe P, Heft 246 (2002).

[6] BORODIN, A., LINIAL, N. AND SAKS, M., *An optimal online algorithm for metrical task systems.*, Journal of the ACM, 39 (1992), pp. 745–763.

[7] BRUCKER, P., *Scheduling Algorithms*, Springer-Verlag, New York, (2001).

[8] BRUCKER, PETER AND KNUST, SIGRID http://www.mathematik.uni-osnabrueck.de/research/OR/class/

[9] BRUCKER, P., HURINK, J. AND KNUST, S. , *A polynomial algorithm for* $P \mid p_j = 1, r_j, outtree \mid \sum C_j$, Mathematical Methods of Operations Research, 56 (2003), pp. 407–412.

[10] CHEN, C.L. AND BULFIN, R.L., *Complexity of single machine multi-criteria scheduling problems.*, European Journal of Operational Research, 70 (1993), pp. 115–125.

[11] CHEN, C.L. AND BULFIN, R.L., *Scheduling a single machine to minimize two criteria: maximum tardiness and number of tardy jobs.*, IIE Transactions, 26 (1994), pp. 76–84.

[12] CHRETIENNE, P. ET AL.(EDITORS), Scheduling Theory and Its Applications, John Wiley & Sons, (1995).

[13] COFFMAN, E. G. AND GRAHAM, R. L., *Optimal scheduling for two-processor systems*, Acta Informatica, 1 (1972), pp. 200–213.

[14] COFFMAN, E. G., SETHURAMAN, J. AND TIMKOVSKY, V. G., *Ideal preemptive schedules on two processors*, Acta Informatica, 39 (2003), pp. 597–612.

[15] DILEEPAN, P. AND SEN, T., *Bicriterion static scheduling research for a single machine*, OMEGA, 16 (1988), pp. 53–59.

[16] DU, J. AND LEUNG, J. Y-T., *Minimizing mean flow time with release time and deadline constraints*, J. of Algorithms, 14 (1993), pp. 45–68.

[17] DU, J. AND LEUNG, J. Y-T., *Minimizing total tardiness on one machine is NP-hard*, Mathematics of Operations Research 15 (1990), pp. 483–495.

[18] DU, J., LEUNG, J. Y-T. AND YOUNG, G. H., *Minimizing mean flow time with release time constraints*, Theoret Comput Sci 75 (1990), pp. 347–355.

[19] DU, J., LEUNG, J. Y-T. AND YOUNG, G. H., *Scheduling chain-structured tasks to minimize makespan and mean flow time*, Information and Computation, 92 (1991), pp. 219–236.

[20] EMMONS, H., *One machine sequencing to minimize mean flow time with minimum number tardy*, Naval Research Logistics Quarterly, 22 (1975a), pp. 585–592.

[21] EMMONS, H., *A note on a scheduling problem with dual criteria*, Naval Research Logistics Quarterly, 22 (1975b), pp. 615–616.

[22] GAREY, M. R. AND JOHNSON, D. S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, San Francisco, (1979).

[23] GAREY, M.R., TARJAN, R.E., AND WILFONG, G.T., *One-processor scheduling with symmetric earliness and tardiness*, Mathematics of Operations Research, 13 (1988), pp. 330–348.

[24] GRAHAM, R. L., LAWLER, E. L., LENSTRA, J. K. AND RINNOOY KAN, A. H. G., *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Annals of Discrete Math., 5 (1979), pp. 287–326.

[25] GRAHAM, R. L., *Bounds for certain multiprocessing anomalies*, Bell systems Technical Journal, 45 (1966), pp. 1563-1581.

[26] HECK, H. AND ROBERTS, S., *A note on the extension of a result on scheduling with a secondary criteria*, Naval Research Logistics Quarterly, 19 (1972), pp. 403–405.

[27] HONG, K. S. AND LEUNG, J. Y-T., *On-line scheduling of real-time tasks*, IEEE Transactions on Computers, C41 (1992), pp. 1326–1331.

[28] HORN, W. A., *Single-machine job sequencing with treelike precedence ordering and linear delay penalties*, SIAM Journal on Applied Mathematics, 23 (1972), pp. 189–202.

[29] HU, T. C., *Parallel sequencing and assembly line problems*, Operations Research, 9 (1961), pp. 841–848.

[30] HUO, Y. AND LEUNG, J.Y-T., *Online Scheduling of Precedence Constrained Tasks*, SIAM J. on Computing, accepted for publication.

[31] HUO, Y. AND LEUNG, J.Y-T., *Minimizing Total Completion Time for UET Tasks with Release Time and Outtree Precedence Constraints*, Mathematical Methods of Operations Research, accepted for publication.

[32] HUO, Y. AND LEUNG, J.Y-T., *Minimizing mean flow time for UET tasks*, ACM Transactions on Algorithms, accepted for publication.

[33] HUO, Y., LEUNG, J.Y-T., AND ZHAO, H., *Complexity of two-dual criteria scheduling problems*, Submitted. (2004).

[34] HUO, Y., LEUNG, J.Y-T., AND ZHAO, H., *Bi-criteria Scheduling Problems: Number of Tardy Jobs and Maximum Weighted Tardiness*, Submitted. (2004).

[35] JOHNSON, D.S., *Approximation algorithm for combinatorial problems*, Journal of Computer and System Sciences, 9 (1974), pp. 256–278.

[36] KAN, A.H.G. RINNOOY, LAGEWEG, B.J. AND LENSTRA, J.K., *Minimizing total costs in one machine scheduling*, Operations Research, 23 (1975), pp. 908–927.

[37] KARLIN, A., MANASSE, M., RUDOLPH, L. AND SLEATOR, D.D., *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 79–119.

[38] KARP, RICHARD M., *Reducibility among combinatorial problems*, Complexity of Computer Computations (Proceedings of the Symposium on the Complexity of Computer Computations,March,(1972), Yorktown Heights, NY), Plenum Press, New York, (1972), pp. 85–103.

[39] LAM, S. AND SETHI, R., *Worst case analysis of two scheduling algorithms*, SIAM Journal on Computing, 6(3) (1977), pp. 518–536.

[40] LAWLER, E. L., *Sequencing jobs to minimize total weighted completion time subject to precedence constraints*, Annals of Discrete Mathematics, 2 (1978), pp. 75–90.

[41] LAWLER, E. L., *Optimal sequencing of a single machine subject to precedence constraints*, Management Science, 19 (1973), pp. 544–546.

[42] LAWLER, E. L., *Scheduling a single machine to minimize the number of late jobs*, Unpublished manuscript.

[43] LAWLER, E. L., LENSTRA, J.K., RINNOOY KAN, A.H.G. AND SHMOYS, D.B., Sequencing and Scheduling: Algorithms and Complexity, in Handbooks in Operations Research and Management Science, Vol 4: Logistics of Production and Invertory, (1990).

[44] LEE, C.-Y. AND VAIRAKTARAKIS, G.L., *Complexity of single machine hierarchical scheduling: A survey*, In Panos M. Pardalos (ed.), Complexity in Numerical Optimization, (1993), pp. 269–298, World Scientific Publishing Co., New Jersey, U.S.A.

[45] LEUNG, J. Y-T. AND PINEDO, M. L., *A note on the scheduling of parallel machines subject to breakdown and repair*, Naval Research Logistics, 51 (2004), pp. 60–72.

[46] MANASSE, M.S., MCGEOCH, L.A. AND SLEATOR, D.D., *Competitive algorithms for online problems*, In Proc. 20th Annual ACM Symp. on Theory of Computing, (1988), pp. 322–333.

[47] MCNAUGHTON, R., *Scheduling with deadlines and loss functions*, Management Science, 6 (1959), pp. 1–12.

[48] MOORE, J.M., *An n job, one machine sequencing algorithm for minimizing the number of late jobs*, Management Science, 15 (1968), pp. 102–109.

[49] MOTWANI, R., *Lecture notes on approximation algorithms*, Technical Report STAN-CS-92-1435, Department of Computer Science, Stanford University, (1992).

[50] MUNTZ, R. R. AND COFFMAN, E. G., *Optimal preemptive scheduling on two-processor systems*, IEEE Transactions on Computers, C-18 (1969), pp. 1014–1020.

[51] MUNTZ, R. R. AND COFFMAN, E. G., *Preemptive scheduling of real-time tasks on multiprocessor systems*, J. of the Association for Computing Machinery, 17 (1970), pp. 324–338.

[52] OW, P.S. AND MORTON, T.E., *The single machine early/tardy problem*, Management Science, 35 (2) (1992), pp. 177–191.

[53] PANWALKAR, S.S., DUDEK, R.K., AND SMITH, M.L.,*Sequencing research and the industrial scheduling problem*, Symposium on the Theory of Scheduling and Its Application, S.E. Elmaghraby (ed.), Springer-Verlag, New York, pp. 29–38.

[54] PINEDO, M., *Scheduling: Theory, Algorithms and Systems*, Prentice-Hall, Englewood Cliffs, NJ, (2002).

[55] SGALL, J., *On-line scheduling - A survey*, In A. Fiat and G. Woeginger, editors, On-line algorithms. Springer-Verlag, Berlin, (1997).

[56] SHANTHIKUMAR, J.G., *Scheduling n jobs on one machine to minimize the maximum tardiness with minimum number tardy*, Computers Operations Research, 10 (1983), pp. 255–266.

[57] SLEATOR, D.D. AND TARJAN, R.E., *Amortized efficiency of list update and paging rules*, Communications of the ACM, 28 (1985), pp. 202–208.

[58] SMITH, W.E., *Various optimizers for single stage production*, Naval Research Logistics Quarterly, 3 (1956), pp. 59–66.

[59] VAIRAKTARAKIS, G.L. AND LEE, C-Y., *The single-machine scheduling problem to minimize total tardiness subject to minimum number of tardy jobs*, IIE Transactions, 27 (1995), pp. 250–256.

[60] WOOLSEY, R.E.D., *Survival scheduling with Hodgson's rule or see how those salesmen love one another*, INTERFACES, 22 (1992), pp. 81–84.