

THE UNIVERSITY OF CALGARY

A Preemptive Scheduling

Heuristic

by

Gang Li

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS AND STATISTICS

CALGARY, ALBERTA

JULY, 1994

©Gang Li 1994



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-315-99407-X

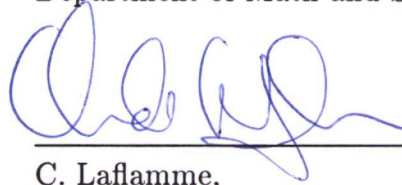
Canada

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "A Preemptive Scheduling Heuristic" submitted by Gang Li in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor, M. G. Stone,
Department of Math and Stats



C. Laflamme,
Department of Math and Stats



N. Bshoufy,
Department of Computing Science

16 Aug '94
Date

ABSTRACT

To introduce the main topic of this thesis, we present a collection of known results for preemptive and non-preemptive scheduling problems. Our goal is to explore the use of unit execution times for *critical path* elements (members of a chain of maximal length) as a scheduling heuristic. We observe that optimal schedules which are compatible with this heuristic are always possible for one and two machine problems. For three or more machines, we prove that tasks in critical paths in the original task set P cannot always be executed in unit time in an optimal schedule. We investigate the possibility of rescheduling all tasks in any maximum chain in the order $S(P)$ induced by an optimal schedule S , using unit execution times without lengthening the schedule. Finally, we identify a class of ordered sets called K-structures for which we design an algorithm based on this heuristic which appears to provide optimal preemptive schedules.

Preface

We examine optimal preemptive scheduling for a set of tasks with precedence constraints given by a partial order on this set, P . Our major focus will be on "preemptive scheduling" where work on individual tasks may be interrupted and resumed at a later time. We consider problems with a fixed but arbitrary number of identical processors, and tasks which each require unit execution time. First we summarize some of the known results and explore easy consequences of these for the questions of interest to us in the Thesis. For one and two machines most questions can be answered rather easily and completely. For three or more machines (the "multi-processor" case) these problems are harder. We comment on some of the questions about computational complexity, in particular, we discuss various approaches to the question of whether or not the multi-processor preemptive scheduling problem is NP-hard. Next, we pose a number of questions related to the central theme of this Thesis: Scheduling maximum chains (or *critical paths*). In 1989, Sauer and Stone proved that for every schedule S there is some maximum chain (in the order induced by S) which can be rescheduled in unit blocks (without preemption) in a schedule S' , of equal or shorter length, for the same poset P . It is natural to ask if this result can be improved to one which deals directly with maximal chains in P . The answer is "yes" for one or two machines. We show the answer to be "no" for three or more processors: There exist partial orders P for which every optimal schedule preempts each maximum chain in P . The smallest examples are surprisingly complex and rely on special properties of a poset introduced by Sauer and Stone in 1986. In the opposite direction, we prove for each three machine preemptive schedule S of P which

satisfies a certain chain condition there exists an identifiable maximum chain in the order extension $S(P)$ which can be rescheduled without preemption in a schedule S' , of equal or shorter length, for the same poset P . We further investigate the role of maximum chains from P in scheduling problems; in particular, we adopt the scheduling of maximum chains in unit blocks as a scheduling heuristic. We define an interesting class of posets(K-structures) which are amenable to scheduling by an algorithm based on this heuristic. We identify some interesting properties related to K-structures and the algorithm. The potential optimality of this algorithm provides a direction for further research in this area. The bibliography provides references to selected contemporary papers closely related to our research on multiprocessor preemptive scheduling problem. No attempt is made to provide a comprehensive bibliography for all of scheduling theory.

Acknowledgements

In the course of producing this thesis I have become indebted to many people. To begin with, I would like to express deepest appreciation to my supervisor Dr. M. G. Stone. He suggested to me to do research in scheduling theory, and provided me with every accessible reference. His broad knowledge in scheduling theory always guided me on the right track. And particularly, his creative ideas have made our research much more productive. In order to make the thesis more readable, he put much effort and patience in carefully verifying my results and checking every detail, including my English expression. I will say, without his intellectual, emotional and financial support, the thesis would not be possible.

My thanks to professor N.W. Sauer, C. Laflamme, P. Zvengrowski, and K. Varadarajan. The lectures and seminars they offered presented me a broad view of mathematics.

I am also grateful to Professors Stone and Zvengrowski for their financial support during my summer research. I would like also to take this chance to thank the Department of Mathematics and Statistics and the Faculty of Graduate Studies for their financial and emotional support which enabled me to complete my graduate study here.

Finally, a personal note of thanks to my wife for her patience and encouragement throughout my two years of graduate study, for her great help in taking care of my life.

CONTENTS

Approval page	ii
Abstract	iii
Preface	iv
Acknowledgements	vi
List of Figures	ix
Chapter 1. An Introduction to Scheduling Theory	1
1.1. Machine Scheduling Theory	1
1.2. Partially Ordered Sets	5
1.3. Preemptive Scheduling on Posets	8
1.4. Methodology	11
Chapter 2. Single-machine Scheduling Problem	14
2.1. Some Known Results	14
2.2. The Jump Number Problem	16
Chapter 3. Two-Machine Scheduling Problems	22
3.1. Some Known Results	22
3.2. A Proof for the General Two-machine Scheduling Problem	27
3.3 An Application of Transformation Methods	31
Chapter 4. Multi-Machine Scheduling Problems	36
4.1. Identical Processors and Independent Tasks	36
4.2. Identical Processors and Dependent Tasks	38
Chapter 5. The Complexity of Scheduling Problems	46

5.1. An Introduction to NP-Completeness Theory	46
5.2. Computational Complexity of Scheduling Problems	55
Chapter 6. Scheduling of Maximum Chains in Unit Blocks	57
6.1. Schematic Representations	57
6.2. A Counterexample	58
Chapter 7. Rescheduling Maximum Chains in $S(P)$	67
7.1. Observations	67
7.2. Rescheduling Maximum Chains in $S(P)$	70
Chapter 8. Multiprocessor Scheduling Problem on K-structures	78
8.1. K-Structure	78
8.2. Double-Labeling	80
8.3. An Algorithm for K-Structure	82
8.4. Directions for Further Research	86
Bibliography	87

List of Figures

Figure 1.2.1	6
Figure 1.2.2	7
Figure 1.3.1	10
Figure 1.3.2	10
Figure 2.2.1	18
Figure 2.2.2	20
Figure 2.2.3	20
Figure 3.1.1	23
Figure 3.1.2	24
Figure 3.2.1	28
Figure 3.2.2	28
Figure 3.2.3	29
Figure 3.3.1	32
Figure 4.1.1	38
Figure 4.2.1	39
Figure 5.1.1	49
Figure 5.1.2	50
Figure 6.1.1	58
Figure 6.1.2	58
Figure 6.2.1	59
Figure 6.2.2	60

Figure 6.2.3	60
Figure 6.2.4	61
Figure 6.2.5	62
Figure 6.2.6	64
Figure 6.2.7	65
Figure 6.2.8	65
Figure 6.2.9	66
Figure 7.1.1	69
Figure 7.2.1	72
Figure 7.2.2	73
Figure 7.2.3	73
Figure 7.2.4	75
Figure 7.2.5	76
Figure 7.2.6	76
Figure 8.1.1	78
Figure 8.2.1	81
Figure 8.2.2	81
Figure 8.2.3	82
Figure 8.3.1	84

CHAPTER ONE

An Introduction to Scheduling Theory

§1.1 Machine Scheduling Theory

In very general terms, machine scheduling theory studied in this thesis is concerned with the allocation over time of scarce resources in the form of machines or processors to activities known as jobs or tasks. Based on prespecified properties of and constraints on the tasks and processors, the problem is to find an efficient algorithm to sequence the tasks to optimize(or improve) some desired performance measure. The primary measure studied is the *makespan* of a schedule: the total time required for processors to complete all tasks under some constraints. As to the constraints, if the jobs can be performed in any order, they are said to be *independent*. But, this is often not the case; in many circumstances, of two given jobs, one must be performed before the other. Such *precedence constraints* of course impose a partial ordering on the task set.

The scheduling model used to treat subsequent problems is described here. We consider(in this order) the *resources*, *tasks*, *scheduling constraints*, and *performance measures* which will be used to describe subsequent problems.

- Resources:

In the majority of the models studied, the resources consist simply of a set $\mathcal{R} = \{P_1, \dots, P_m\}$ of processors. Depending on the specific problem, they are either totally identical in functional capability but different in speed, or different in both function and speed. In the most general model, there is also a set of additional resource type $\mathcal{R}^* = \{R_1, \dots, R_s\}$, some (possibly empty) subset of which is required during the

entire execution of a task on some processor. In computer applications, for example, such resources may represent primary or secondary storage, input/output devices, or subroutine libraries.

- The Tasks:

The tasks for a given scheduling problem can be described as a system $(\mathcal{T}, \prec, [t_{ij}], \{\mathcal{R}_j\}, \{w_j\})$ as follows:

1. $\mathcal{T} = \{T_1, \dots, T_n\}$ is a set of tasks to be executed.
2. \prec is an (irreflexive) partial order defined on \mathcal{T} which specifies operational precedence constraints; that is, $T_i \prec T_j$ signifies that T_i must be completed before T_j can begin.
3. $[t_{ij}]$ is an $m \times n$ matrix of execution times, where $t_{ij} > 0$ is the time required to execute T_j , $1 \leq j \leq n$, on processor P_i , $1 \leq i \leq m$. We suppose that $t_{ij} = \infty$ signifies that T_j can not be executed on P_i and that for each j there exists at least one i such that $t_{ij} < \infty$. When all processors are identical we let t_i denote the execution time of T_i common to each processor.
4. $\mathcal{R}_j = [R_1(T_j), \dots, R_s(T_j)]$, $1 \leq j \leq n$, specifies in the i 'th component, the amount of resource type R_i required throughout the execution of T_j . The amount of each resource R_i is assumed to be finite, say m_i , and we may assume $R_i(T_j) \leq m_i$, for all i and j .
5. The weights w_i , $1 \leq i \leq n$, are interpreted as deferral costs (or cost penalties), which in general may be arbitrary functions of schedule properties influencing T_i . However, the w_i will be taken to be zero in the models we analyze.

- Scheduling constraints:

By "constraint" we mean here a restriction of the scheduling algorithm to a specific type of schedule. Two main restrictions are considered:

1. Non-preemptive/preemptive Scheduling: *Non-preemptive scheduling* requires that no task be interrupted once the execution of this task has begun. By contrast, for *preemptive scheduling*, the execution of a task can be interrupted before its completion, and resumed later by the same or another machine for further processing. We will assume there is no loss of execution time due to preemptions.

2. Schedules may also be constrained to follow some particular strategy or priority, for example *list scheduling*. In this type of scheduling, an ordered list of the tasks in \mathcal{T} is assumed or constructed before hand. This list is often called the *priority list*. The sequence by which tasks are assigned to processors is then decided by a repeated scan of the list. Specifically, when a processor becomes free for assignment, the list is scanned until the first unexecuted task T is found which is ready to be executed. That is, we execute next the first task T in the list which can be executed on the given processor, provided all predecessors of T have been completed, and sufficient resources exist to satisfy $R_i(T)$ for each $1 \leq i \leq s$.

- Performance Measures:

The criteria to measure scheduling performance varies in different contexts. For a general problem with processor resources $\mathcal{R} = \{P_1, \dots, P_n\}$, task set $(\mathcal{T}, \prec, [\tau_{ij}], \{R_i\}, \{w_i\})$ where $\mathcal{T} = \{T_1, \dots, T_n\}$, with given scheduling constraints and an arbitrary schedule S , common measures are:

- 1) makespan(schedule-length required to complete all tasks)

$$\omega(S) = \max_{1 \leq i \leq n} \{\tau_S(T_i)\}$$

2) mean weighted finishing time

$$\omega(S) = \frac{1}{n} \sum_{i=1}^n w_i \tau_S(T_i)$$

where $\tau_S(T_i)$ denotes the time at which T_i is terminated (finish time) in S .

Using this general model, we may classify machine scheduling problems according to certain specific criteria:

- single-machine/multi-machine scheduling
- identical machine/non-uniform machine scheduling
- machines or processors in parallel/series
- machine scheduling with/without supporting resources (e.g. I/O, storage, etc)
- scheduling of independent tasks/precedence constrained tasks
- identical task size/non-uniform task size (which affects execution time)
- with/without release time and/or deadline restriction on the task set
- weighted/unweighted task system
- preemptive/non-preemptive scheduling
- list scheduling/scheduling without fixed priorities
- optimization of maximum finish time/mean weighted finish time

The scheduling we will mainly study is multi-machine preemptive scheduling with identical processors, and unit-execution-time precedence constrained tasks. We also assume there are no other resources required for performance of any task on any specific machine. Our object is to minimize the makespan (or maximum finish time) over all schedules on some fixed task set. The specific model for preemptive scheduling is this:

- resources: $\mathcal{R} = \{P_1, \dots, P_m\}$ set of machines/processors
- tasks: $(\mathcal{T}, \leq, [\tau_{ij}], \{R_i\}, \{w_i\})$ where $\mathcal{T} = \{T_1, \dots, T_n\}$ set of tasks/jobs

- scheduling constraint: preemptive
- performance measure: maximum finish time

Specifically, the preemptive scheduling we are mainly concerned has following properties:

- (1). $\tau_{ij} = \tau_{kl}$ for any $i, k \in \{1, 2, \dots, m\}$ and $j, l \in \{1, \dots, n\}$, i.e., all processors are identical, and execution time for each task is the same.
- (2). $\{R_i\} = \emptyset$: no additional resource is required throughout the execution of any task.
- (3). $w_j = 0$ for any $1 \leq j \leq n$; task set is unweighted, there are no deferral costs or other penalty costs weighted on any task. We only consider the execution time.
- (4). \leq is the precedence constraint on the task set \mathcal{T} , which is actually a partially ordered set.

§1.2 Partially Ordered Sets

A binary relation \leq on a set X is a *partial order* if it has the three properties:

Reflexivity: $\forall x \in X, x \leq x$

Antisymmetry: $\forall x, y \in X, (x \leq y) \wedge (y \leq x) \implies x = y$

Transitivity: $\forall x, y, z \in X, (x \leq y) \wedge (y \leq z) \implies x \leq z$

A set X , equipped with a partial order \leq , is called a *partially ordered set*, or briefly: a *poset*. In a partially ordered set (X, \leq) , there are some objects of special interest:

- (a) An element $m \in X$ is called a *maximal element* if each $x \in X$ related to m satisfies $x \leq m$. Equivalently, whenever $m \leq x$ and $x \in X$, then $m = x$.

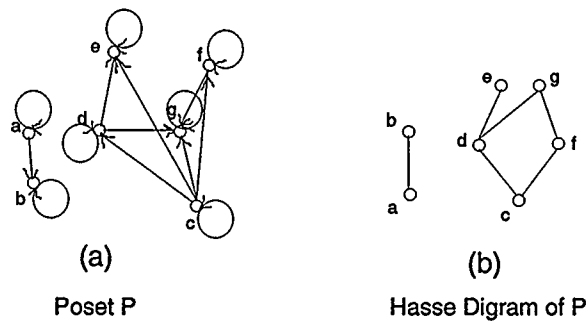
(b) An element $b \in X$ is called an *upper bound* for a subset $A \subset X$ if the relation $a \leq b$ holds for all $a \in A$.

(c) A subset $A \subset X$ is called a *chain* if all the elements of A are related, that is, for all $x, y \in A$, either $x \leq y$ or $y \leq x$.

Partially ordered sets can be visually displayed through *Hasse diagrams* which can be obtained by the following four steps:

- draw a digraph of the poset
- delete all cycles from the digraph
- eliminate all edges that are implied by transitive property
- redraw the digraph with all edges pointing "upward" and then delete arrows from edges

Figure 1.2.1 below depicts the Hasse Diagram for a typical poset P .



c	d	g	b
a	f	b	e

The Gantt Chart of a preemptive schedule of
above poset P

(c)

fig 1.2.1

Schedules for partially ordered sets may be represented by *Gantt Charts* which provide a visual presentation of the schedule. Such a chart displays the time intervals during which each machine is occupied with various tasks as a single row with a linear time scale so that length corresponds to scheduled time for tasks or partial tasks. The time intervals during which each machine is occupied with various tasks are easily read from the chart. Figure 1.2.1(c) displays such a Gantt Chart for the partial order whose Hasse diagram is given above in Figure 1.2.1(a). We will typically use Gantt Charts to visualize schedules and to discuss their properties. Indeed for practical purpose we may identify each schedule with its representation as a Gantt Chart.

A poset (X, \leq^*) is a *linear extension* of poset (X, \leq) if and only if:

- $\forall x, y \in X, x \leq y \implies x \leq^* y$
- $\forall x, y \in X, x \leq^* y$ or $y \leq^* x$

Actually, X is a chain under the relation \leq^* . Observe that if (X, \leq) is not itself a chain, then we can have many different linear extensions of (X, \leq) . For example, (A, \leq) shown in fig 1.2.2 has two linear extensions:

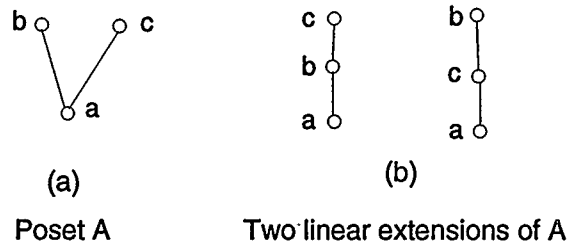


fig 1.2.2

A poset (P, \leq) is an *interval order* if and only if P is isomorphic to a set (P_R, \leq^*) of open intervals on real line \mathbf{R} where the ordering relation \leq^* on P_R is defined as:

$A <^* B$ for $A, B \in P_R$ iff a is less than b in \mathbf{R} for any $a \in A, b \in B$. Q is an *interval extension* of a poset P if and only if Q is an interval order which contains P .

There is a well known lemma which could be treated as an alternative definition for interval order:

Lemma 1.2.1 (P, \leq) is an interval ordered set if and only if for any A, A', B, B' in P with $A < B$ and $A' < B'$, we have either $A < B'$ or $A' < B$.

Linear extensions and interval extensions of posets receive a great deal of attention in scheduling theory. Such extensions are not just some ordering imposed on a task set, but are inherited from various schedules of the tasks. We observe that:

1) any single-machine non-preemptive schedule S of poset P gives a unique linear extension of P ; and any linear extension \mathcal{L} of a poset P gives a unique single-machine schedule S of P .

2) any m -machine schedule S of a poset P gives a unique interval extension \mathcal{I} of the poset P ; and any interval extension \mathcal{L} of width m of a poset P gives a m -machine schedule S of the poset P .

§1.3 Preemptive Scheduling on Partially Ordered Sets

Now, we will give some definitions and notations in preemptive scheduling problems:

Definition 1.3.1 For any poset P , we let

- (1) $H = H(P) = \text{height of poset } P = \text{the length of a maximum chain in } P$
- (2) $\mathcal{C}_a = \{C \text{ is a chain, and } c > a \text{ for any } c \in C\}$, and $l(a) = \text{level of } a \text{ in } P$

$$P = \max_{C \in \mathcal{C}_a} \{|C|\}.$$

(3) $\mathcal{C}^a = \{C \text{ is a chain, and } c < a \text{ for any } c \in C\}$, and $h(a) = \text{height of } a \text{ in } P = \max_{C \in \mathcal{C}^a} \{|C|\}$.

Definition 1.3.2 For any poset P , and a schedule S of P , we define:

- (1) $|S| = \text{makespan of the schedule } S$.
- (2) $\sigma_S(a) = \text{starting time in } S \text{ of job } a \in P$.
- (3) $\tau_S(a) = \text{finishing time in } S \text{ of job } a \in P$.
- (4) We denote by $(S(P), \leq^*)$ the order extension of (P, \leq) induced by the schedule S , that is, if $a < b$ then $a <^* b$, and if a precedes b in S , then $a <^* b$.
- (5) We denote by $S : Q$ the restriction of schedule S on subset $Q \subseteq P$.

If S is understood, we write simply $\sigma(a)$ for $\sigma_S(a)$, and $\tau(a)$ for $\tau_S(a)$. Note, for any job $a \in P$, if $|S| = t$, then $\sigma(a) \in [0, t)$, $\tau(a) \in (0, t]$

Definition 1.3.3 Given a poset P :

- (1) A *shift* in a schedule S of P is a maximal closed time interval $[t_1, t_2] \subseteq [0, t]$, where $t = |S|$, during which there is no machine/processor which changes its processing from one job to another.
- (2) A *waste interval* (or simply *waste*) in a schedule S of P is a time interval $[t_1, t_2] \subseteq [0, t]$ on some specific machine, during which this machine is idle .

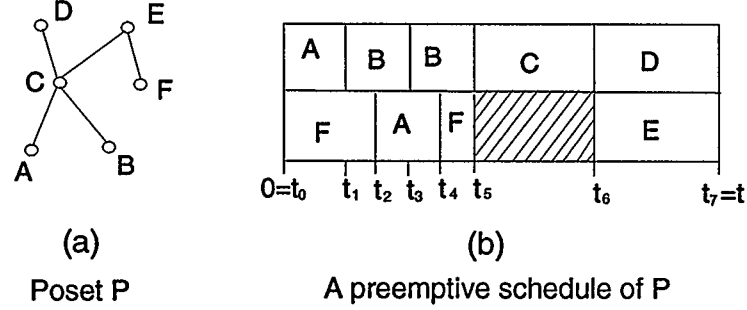
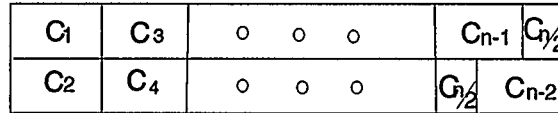


fig 1.3.1

As shown in fig 1.3.1, each $(t_i, t_{i+1}), 0 \leq i \leq 6$, is a shift in S , and $[t_5, t_6]$ is a waste interval on machine 2.

A useful technique concerning the preemptive scheduling of antichain is the *wrapping method*:

Let $C = \{C_1, C_2, \dots, C_n\}$ be an antichain, $|C|$ the number of nodes in C , and m the number of machines. If $m < |C|$, then we can wrap all nodes (or tasks) in the following way: we first schedule $|C| - (|C| \bmod m) - m$ nodes into $\lfloor \frac{|C|}{m} \rfloor - 1$ unit blocks, then "wrap" the remaining nodes into a block of length $1 + (\frac{|C| \bmod m}{m})$ as shown in figure 1.3.2 below for $m = 2$:



$$C = \{ C_1, C_2, \dots, C_n \}$$

fig 1.3.2

Lemma 1.3.1 The wrapping method produces an optimal schedule S for the antichain C .

Proof. It is trivial since S has no waste in it. \square

Actually, the wrapping lemma(Lamme 1.3.1) can be generalized into tasks of unequal size[26].

We say a schedule S of poset P is *optimal* if and only if the length of S is less than or equal to the length of any other schedule S' of P . We observe:

- (1). if $|S| = |C|$ where C is a maximum chain in P , then S is optimal, since for any other schedule S' , $|S'| \geq |C|$.
- (2). if S contains no waste, then S is optimal.

Let function $W(S)$ denote the sum of waste time in a schedule S of P . More generally, we have this lemma:

Lemma 1.3.2 Let S and S' be two schedules of P , if $W(S) \leq W(S')$ then $|S| \leq |S'|$.

Proof. It is trivial to see that $|S| = (|P| + W(S))/m$ where m is the number of machines used in this schedule. So $W(S) \leq W(S')$ implies $|S| \leq |S'|$. \square

Corollary 1.3.1 If $W(S^*)$ is minimum among all schedules S for P , then S^* is optimal for P .

§1.4. Methodology

Actually, a preemptive scheduling problem is a combinatorial optimization problem, where one attempts to find an optimal solution among numerous options.

Here is the general setting for a combinatorial optimization problem:

- (*) *Given a finite set S and a function $f : S \rightarrow \mathbf{R}$, find an $s^* \in S$ such that $f(s^*) \leq f(s)$ for all $s \in S$.*

Apart from "How does one find a good algorithm to produce s^* ", the other central question we want to answer is "How can one know that a certain $s^* \in S$ is optimal,

without examining all of S ?".

Generally, there are two standard ways to approach such problems, *min-max methods and transformation methods*

For any combinatorial optimization problem(as in (*)),suppose there is some other set T , derived from the structure underlying S , and a function $g : T \rightarrow \mathbf{R}$ such that $g(t) \leq f(s)$ for all $t \in T$ and $s \in S$. Hence, all the values $g(t)$ are a priori lower bounds for the optimal value of $f(s)$.

Theorem 1.4.1(Min-max) [22] For the situation above, if we have the equality $f(s^*) = g(t)$ for some $s^* \in S$ and $t \in T$, then s^* is optimal. And further we have $\min_{s \in S} f(s) = \max_{t \in T} g(t)$ and $f(s^*) = \min_{s \in S} f(s)$.

Proof. Suppose there exists $s' \in S$ such that $f(s') < f(s^*)$. But $f(s^*) = g(t)$, so $f(s') < g(t)$ which is contrary to our assumption. \square

Min-max results are very attractive goals in any mathematical endeavor dealing with optimization. One reason is that they always seem to go with good algorithms, and the second is they allow elegant proofs of optimality. We shall supply the details for one such optimization problem in Chapter Two.

The transformation methods we will discuss later involve the transformation of one solution of a combinatorial optimization problem to another solution for the same problem. Usually, this will be done by means of an algorithm. Such techniques can serve several purposes: they can be part of an algorithm to find an optimal solution; they can also be used to show that certain proposed algorithms do what they ought to; and finally, they can be used to prove a general statement on the

behavior of the objective function or the optimality of a solution.

The most interesting application of transformations would be an algorithm transforming any given $s \in S$ into an optimal $s^* \in S$. Suppose a subset $S' \subseteq S$ has the property that it is substantially smaller than S and is guaranteed to contain an optimal element. This reduces the number of cases to look at and can be very valuable if the problem is known to be very hard, for example NP-complete. In this case, the proof that S' contains an optimal element will typically use some transformation to show that "without loss of generality", it is enough to look at S' .

Another application of transformation methods arises when one wants to show that every element of S' is optimal. Typically, S' will consist of the elements of S that are outcomes of a proposed algorithm. This situation can be handled by devising an algorithm transforming any optimal element of S into any given element of S' without increasing the objective function.

For most problems, it would be too optimistic to hope to find powerful min-max results. Transformation techniques and new scheduling heuristics, which provide good schedules or improve existing schedules for a broad class of problems, seem to be needed for many practical scheduling problems.

CHAPTER TWO

Single-machine Scheduling Problems

In this chapter, we will discuss some of the known results for single-machine scheduling problems. In particular, we present a detailed application of the min-max method to the Jump Number Problem.

§2.1 Some Known Results

Theorem 2.1.1. The general one machine, n task scheduling problem to minimize makespan is polynomial.

Proof. It is trivial to see that any linear extension of the task set P provides an optimal schedule of P , and the following algorithm provides such an extension in polynomial time:

Algorithm 2.1.1. Let P be a poset, and $|P| = n$,

- 1) Choose any node $a_{i_1} \in P$, $i_j \in \{1, 2, \dots, n\}$, which has no successor,
 let $\alpha(a_{i_j}) = 1$ where α is a permutation function mapping nodes in P to a positive integer set $\{1, 2, \dots, n\}$.
- 2) Suppose we have chosen $k-1$ nodes $\{a_{i_1}, a_{i_2}, \dots, a_{i_{k-1}}\}$ in P by assigning $\alpha(a_{i_j}) = j, 1 \leq j \leq k-1$. Let function $S(a)$ denotes the set of successors of a in P .
 Let $Q = P - \cup_{1 \leq j \leq k-1} S(a_{i_j}) - \{a_{i_j} : 1 \leq j \leq k-1\}$. Take arbitrary nodes b with $S(b) = \emptyset$ in Q , and then let $\alpha(b) = k$.
- 3) Replace k by $k+1$, repeat step 2) until all nodes in Q have been chosen.

The algorithm above (Algorithm 2.1.1), which provides a linear extension, is

clearly polynomial. \square

In 1973, Lawler [15] has given a result based on the following situation for one-machine scheduling problems: Let $P = \{J_1, J_2, \dots, J_n\}$ be the set of tasks, and $c_i(t)$ = the cost of task J_i completed at time t . The problem is to minimize the total cost $c_{max} = \sum_{1 \leq i \leq n} c_i(t_i)$ where t_i is the completion time of task J_i in the respective schedule S of P . The following algorithm given by Lawler generates a permutation π^* such that $(J_{\pi^*(1)}, \dots, J_{\pi^*(n)})$ is an optimal solution to this problem.

Algorithm 2.1.2[15].

```

procedure n1cmax( $p, \pi^*$ );
  begin local  $S, k, i$ ;
     $S := \{1, \dots, n\}; k := n$ ;
    while  $S \neq \emptyset$  do
      begin  $i := \{j | j \in S, c_j(\sum_{g \in S} p_g) = \min_{h \in S} \{c_h(\sum_{g \in S} p_g)\}\}$ ;
         $\pi^*(k) := i$ ;
         $k := k - 1; S := S - \{i\}$ 
      end
    end n1cmax

```

Theorem 2.1.2 Algorithm 2.1.2 provides an optimal solution to the problem of Lawler above.

Proof. If J_f is chosen instead of J_i with $c_f(\sum_{g \in S} p_g) > \min_{h \in S} c_h(\sum_{g \in S} p_g)$, then interchanging J_i with the tasks following it, up to and including J_f , will not increase the total cost c_{max} , since $c_i(\sum_{g \in S} p_g) < c_f(\sum_{g \in S} p_g)$ and $c_f(\sum_{g \in S, g \neq i} p_g) \leq c_f(\sum_{g \in S} p_g)$.

□

An important criteria for practical scheduling problems is based on the concept of *due dates*. Suppose we have fixed a due date d_i for each task $J_i \in P$. Taking as the cost function $c_i(t) = t - d_i$, we define the *lateness* L_i by $L_i = C_i - d_i$ where C_i is the completion time a task J_i in a specified schedule S of P . Then we obtain the optimization criteria corresponding to the minimization of the *maximum lateness* L_{max} , the sum of *lateness* or *total lateness* $\sum L_i$.

Theorem 2.1.4 [18] The one-machine scheduling problem with due dates and optimization criteria L_{max} is polynomial.

The proof of this theorem is based on Smith's theorem on permutation function(see detailed proof in [18]).

§2.2 The Jump Number Problem

In this section, we present Duffus, Rival and Winkler's results [7] on the Jump Number Problem as an application of Min-max methods, and reproduce the proofs of their results. We consider a single machine which performs sequentially a set P of jobs, one at a time. P is an ordered set: any job can only be scheduled after all its predecessors in P have been scheduled. Any job, which is performed immediately after a job which is not constrained to precede it, requires some additional expense, called a "jump". The problem is to schedule the jobs to minimize the number of jumps.

Let L be a linear extension of P , that is , a total ordering of the underlying set

of P such that $a < b$ in L whenever $a < b$ in P .

Let $f(L) = \#$ of jumps in L , and $f(P) = \min\{f(L) | L \text{ is a linear extension of } P\}$. Let $w(P)$ be width of P , that is the size of a maximum antichain in P .

Lemma 2.2.1. [7] If $A \subseteq (P, \leq)$ is an antichain and (L, \leq^*) is a linear extension of (P, \leq) , then

$$f(L) \geq |A| - 1.$$

Proof. Let $|A| = n$. Suppose $A = \{a_1, a_2, \dots, a_n\}$ is this antichain of (P, \leq) , and suppose we have $a_1 \leq^* a_2 \leq^* \dots \leq^* a_n$ in some linear extension L of P .

Let C_i be a chain in P , such that a_i is an element of C_i , for any $i = 1, 2, \dots, n$. Let B_i be a chain in L so that $C_i <^* B_i <^* C_{i+1}$ for $i = 1, 2, \dots, n-1$.

Claim 1. If B_i is empty then there is a "jump" between C_i and C_{i+1} . Otherwise, $a_i < a_{i+1}$.

Claim 2: If B_i is not empty, then there is at least one "jump" in the chain, $C_i <^* B_i <^* C_{i+1}$. If there is no jump, then $C_i <^* B_i <^* C_{i+1}$ must be a chain in (P, \leq) , and hence produce the same contradiction, $a_i < a_{i+1}$.

So, the linear extension (L, \leq^*) contains at least $n - 1 = |A| - 1$ jumps. \square

Now, let us see how we can apply the min-max method: here we are given a set $S = \{L | L \text{ is a linear extension of } P\}$, and function $f(L) = \#$ of "jumps".

Let $T = \{A | A \subseteq P \text{ is antichain}\}$, and function $g(A) = |A| - 1$.

We know that $f(L) \geq g(A)$ for any $L \in S$, and $A \in T$ (by Lemma 1). In order to use the min-max method, if we can find a linear extension $L^* \in S$ such that $f(L^*) = g(A)$ for some $A \in T$, then L^* is an optimal linear extension according to

Theorem 1.4.1, and we have the equality:

$$\min(f(L)) = \max(g(A))$$

To carry out this plan, we next proceed to investigate a special class, *cycle-free*, of posets to which a polynomial algorithm has been found efficient to obtain such linear extension $L^* \in S$.

By cycle-free, here we mean that there exists no substructure in P which is isomorphic to any of the following structures(2n-cycles):

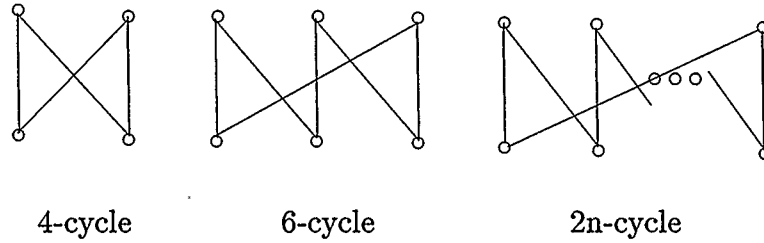


fig 2.2.1

We shall see that the optimal linear extension can be determined from the width of poset P , which can itself be computed by a polynomial algorithm. And the proof which follows illustrates the use of the min-max approach. We begin by giving the algorithm we shall need, and establishing some properties which assume that the algorithm can be executed as it is described.

Algorithm 2.2.1: [7]

- (1). find a maximal antichain a_1, a_2, \dots, a_m in P .
- (2). extend a_i into some maximal chain C_i in P such that $P = C_1 \cup C_2 \cup \dots \cup C_m$. (This can be done in polynomial time by Dilworth's Theorem, 1950 [6])
- (3). Let $P_i = C_i - \cup_{j \neq i} C_j$.

- (4). put a relation " \longrightarrow " on $\{C_i | i = 1, 2, \dots, m\}$ as follows: $C_i \longrightarrow C_j$ if there is an element $x \in P_i$ and $y \in C_j - C_i$ such that $x > y$ in (P, \leq) .
- (5). we claim there exists at least one index $i \in \{1, 2, \dots, m\}$ such that $C_i \not\rightarrow C_j$ for all $j \in \{1, 2, \dots, m\}$. We may suppose here $i = 1$, i.e., $C_1 \not\rightarrow C_j$ for all j . (See Lemma 2.2.2 for proof).
- (6). let $d_1 = \max(P_1 = C_1 - \cup_{i=2}^m C_i)$. $D_1 = \{x \in C_1 | x \leq d_1\}$ and $Q = P - D_1$.
- (7). suppose Q has an optimal linear extension L_Q (our induction hypothesis, since the case of $w(P) = 2$ is easy to solve), and $w(Q) = w(P) - 1$, (see Lemma 2.2.3 for proof).
- (8). we claim that $L^* = D_1 \oplus L_Q$. is an optimal linear extension of P . (See Lemma 2.2.4 for proof).
- (9). by inductively applying step 1 to step 6 to the poset Q , we obtain D_2, D_3, \dots, D_m . $L^* = D_1 \oplus D_2 \oplus \dots \oplus D_m$, where D_i is a chain in P and there is one jump between each pair (D_i, D_{i+1}) .

The assertions made in (5), (7) and (8) above which assume that the algorithm can be executed are established in the following three lemmas.

Lemma 2.2.2 The claim made in step (5) above is valid.

Proof. Suppose to the contrary that there exists no such $C_i \in \{C_1, C_2, \dots, C_{w(p)}\}$, such that $C_i \not\rightarrow C_j$ where $j \in \{1, 2, \dots, w(p)\}$. Then there exist some loops in $C_1, C_2, \dots, C_{w(p)}$. Suppose, after proper re-labelling, $1, 2, \dots, n$ is the smallest sequence so that $C_1 \longrightarrow C_2 \longrightarrow \dots \longrightarrow C_n \longrightarrow C_1$.

Choose $x_i \in P_i$ and $y_i \in C_i - C_{i-1}$ with $x_i > y_{i+1}$ for each $i = 1, 2, \dots, n$. By the minimality of n , $x_i > y_i$ for each i , $1 \leq i \leq n$. Clearly, $\{x_1, y_1, x_2, y_2, \dots, x_n, y_n\}$

forms a $2n$ -cycle. To prove that it is an embedding into P , we need to verify the relationships in claim 1 - 4 below.

Claim 1. $x_i \not\prec x_j$. If not, let $x > x_j$ in C_j and x is not comparable with x_i (since $x_i \in P_i$, hence $x_i \notin C_j$). And there exists $y < x_j$ in C_j which is not comparable with y_{j+1} , then $\{x_i, x, y_{j+1}, y\}$ is an embedded 4-cycle. See fig 2.2.2 below.

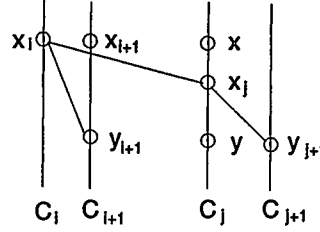


fig 2.2.2

Claim 2. $y_i \neq y_j$ if $i \neq j$. Otherwise, $C_{i-1} \rightarrow C_j$ contradicting the minimality of n .

Claim 3. $y_i \not\prec y_j$. If $y_i \notin C_j$, then there is $y < y_j$ in C_j incomparable with y_i , so $\{x_{j-1}, x_j, y_i, y\}$ is a 4-cycle. If $y_i \in C_j$ then $C_{i-1} \rightarrow C_j$, again contradicting the minimality of n . See fig 2.2.3 below.

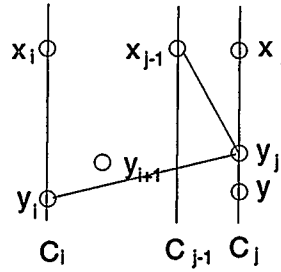


fig 2.2.3

Claim 4. $x_i \not\prec y_j$ where $i \neq j$ and $j \neq i+1$. Since y_j is incomparable with y_i , $y_j \notin C_i$, so $C_i \rightarrow C_j$ which is contrary to the minimality of n .

Claim 5. $x_i \not\prec y_j$. Otherwise $y_i < y_j$ which has been proved impossible in Claim 3.

These prove that C_1, C_2, \dots, C_n is an embedding of $2n$ -cycle into P . \square

Lemma 2.2.3. The claim made in step (7) above is valid.

Proof. Suppose there exists an antichain $a_1, a_2, \dots, a_{w(P)}$ of the length $w(P)$.

For the covering $C_1 - D_1 \cup C_2 \cup \dots \cup C_{w(P)}$ of P , if $a_i \in C_s$ (or $C_1 - D_1$) then $a_j \notin C_s$ (or $C_1 - D_1$), since a_i is incomparable with a_j .

Hence $a_1, a_2, \dots, a_{w(P)}$ belongs to chains $C_1 - D_1, C_2, \dots, C_{w(P)}$. Suppose $a_1 \in C_1 - D_1$, since $\{C_1 - D_1\} \cap P_1 = \emptyset$, so there exists some $j \in \{2, \dots, w(P)\}$ such that a_1 is comparable with a_j . \square

Lemma 2.2.4. The claim made in step (8) above is valid.

Proof. If there exist $x \in D_1, y \in L_Q$ (hence $y \in C_{j \neq 1}$) such that $x > y$, then $d_1 \geq x > y$ implies $C_1 \longrightarrow C_j$. \square

We are now in a position to observe that:

Theorem 2.2.1[7] If P is a cycle-free ordered set, then $f(P) = w(P) - 1$, i.e., the jump number for P is one less than the width of P .

The proof of the Theorem follows directly from lemmas 2.2.1 - 2.2.4. Recall, Lemma 2.2.4. tells that Algorithm 2.2.1 produces an optimal linear extension which means there is a minimal number of jumps in this extension.

CHAPTER THREE

Two-Machine Scheduling Problems

In this chapter, we will discuss some known results of interest for our research in several two-machine scheduling problems. Further, we provide an alternative proof for the general two-machine preemptive scheduling problem, suggested by an algorithm we designed. Finally, we explore the use of transformation methods in preemptive scheduling problems by drawing a brief plan for the proof of a lemma originally given by Muntz and Coffman [20], for which no complete proof has ever been published.

§3.1 Some Known Results

In 1969, Muntz and Coffman gave an algorithm which produces an optimal schedule for two-machine preemptive scheduling problems on unit-execution-time task sets. Later, in 1972, this result was extended to non-preemptive case by Coffman and Graham.[5]

§3.1.1. Preemptive Scheduling on Two-processor Systems

Our development of the algorithm for optimal preemptive scheduling with two processors proceeds by first defining a subclass of preemptive schedules which is shown to contain at least one minimal length schedule. We then show how to construct an optimal schedule for this subclass.

To describe this subclass of preemptive schedules, we must first introduce the notion of a *subset sequence*. A subset sequence for a graph G of a poset P is a

sequence of disjoint subsets of nodes of G , S_1, S_2, \dots, S_m such that:

- 1) if a is a node of G , then $a \in S_i$ for some i ,
- 2) if a, b are nodes of G , with $a \in S_i$, $b \in S_j$ and $a > b$, then $i < j$.

Thus, S_1, \dots, S_m forms a partition of G which respects the order of P .

The following figure gives a subset sequence of a poset P which produces an optimal schedule:

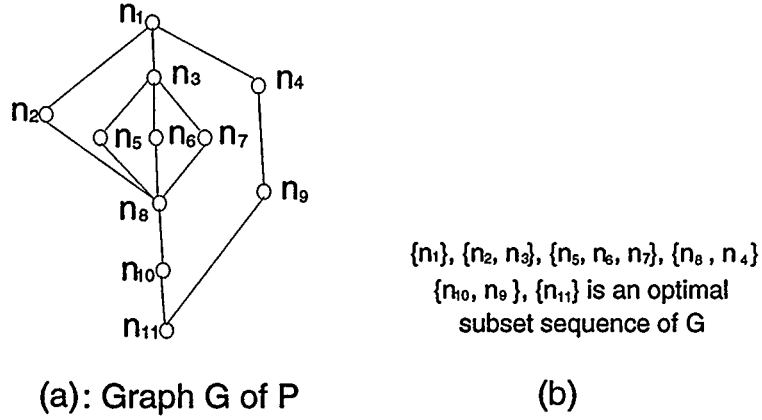


fig 3.1.1

The subclass of preemptive schedules we are going to construct is based on this notion of subset sequence. We call these schedules *subset schedules*.

Muntz and Coffman designed an algorithm to produce such an optimal subset schedule:

Algorithm 3.1.1: If P has height L ,

Step 1) Set index $j = 1$.

Step 2) Let A_j be the set of all nodes which have not yet been assigned to subsets and are at level $L - j + 1$. Assign all nodes in A_j to S_j . If $|A_j| = 1$ then go to

step 4).

Step 3) If $j = L$ then stop, otherwise, set $j = j + 1$ and go to step 2).

Step 4) Let $\omega = \{q_i\}$ be the set of all nodes which have not been assigned to subsets, but all of whose predecessors are contained in $S_1 \cup S_2 \cup \dots \cup S_{j-1}$. If $\omega = \emptyset$ then go to step 3). If $\omega \neq \emptyset$ then assign q_u to S_j where the level $l(q_u)$ satisfies $l(q_u) = \max_{q_i \in \omega} \{l(q_i)\}$, and go to step 3).

Theorem 3.1.1a [20] Each optimal preemptive subset schedule is an optimal schedule.

Their theorem tells us that the optimal solution in the subclass, subset schedules, is optimal in the whole class. That proof is based on the following lemma:

Lemma 3.1.1 Let G be the Hasse diagram of a poset P , and assume that all tasks have unit-execution-times. Then any two-machine preemptive schedule S for P can be transformed into a new schedule S' no longer than S , which has one of the three forms shown in fig 3.1.2. Here A, B, C are three tasks, and the shadowed block stands for waste.

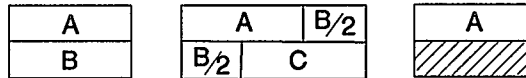


fig 3.1.2

Theorem 3.1.1b Algorithm 3.1.1 produces an optimal preemptive schedule. (For proof, c.f. [20]).

A detailed proof of Lemma 3.1.1 was not provided in Muntz and Coffman's paper. We will present a proof of that lemma in §3.3, as an application of transformation

methods.

§3.1.2. Non-preemptive Scheduling on Two-processor Systems

In 1969, the two machine non-preemptive scheduling problem for unit-execution-time task sets was proved to be polynomially solvable by Fujii, Kasami, and Ninomiya [10]:

Theorem 3.1.2(Fujii, Kasami, and Ninomiya) The general two-processor non-preemptive scheduling problem on a unit-execution-time task set is polynomial.

They showed that an optimal schedule can be constructed from a maximal matching for the incomparability graph of the given partial order. The best published algorithm for a graph on n vertices is of the order of n^4 [8].

In 1971, Coffman and Graham [5] solved the same problem by using the *list schedule strategy*: first produce a list of tasks in P , then assign the tasks by scanning for the first available task in this list which is ready to be executed. A polynomial algorithm was designed to produce the list. But before we introduce the algorithm, we need the following definition:

We linearly order decreasing sequences of positive integers as follows. If $N = (n_1, n_2, \dots, n_t)$ and $N' = (n'_1, n'_2, \dots, n'_{t'})$ are decreasing sequences of positive integers (where possibly $t=0$) we shall say that $N < N'$ if either

(i) for some $i \geq 1$, we have $n_j = n'_j$ for all j satisfying $1 \leq j \leq i-1$ and $n_i < n'_i$,

or

(ii) $t < t'$ and $n_j = n'_j$, $1 \leq j \leq t$.

Let r denote the number of tasks in P . Let $S(T)$ denote the set of successors

of each task T . Coffman and Graham's algorithm (Algorithm 3.1.2) produces a list L^* of the tasks in P by assigning an integer $\alpha(T) \in \{1, 2, \dots, r\}$ to each task T , as follows:

Algorithm 3.1.2

Step 1. Pick an arbitrary task T_0 with $S(T_0) = \emptyset$. Then let $\alpha(T_0) = 1$.

Step 2. Suppose for some $k \leq r$, the integers $1, 2, \dots, k-1$ have been assigned.

For each task T for which α has been defined on all elements of $S(T)$, let $N(T)$ denote the decreasing sequence of integers formed by ordering the set $\{\alpha(T') \mid T' \in S(T)\}$. At least one of these tasks T^ must satisfy $N(T^*) \leq N(T)$ for all such tasks T . Choose one such T^* and define $\alpha(T^*)$ to be k .*

Step 3. We repeat the assignment in Step 2 until all tasks of P have been assigned some integer in $\{1, 2, \dots, r\}$.

Finally, the list L^* is defined by Algorithm 3.1.2, that is $(T_0, T_1, \dots, T_{r-1})$, where $\alpha(T_i) = i + 1, 0 \leq i \leq r - 1$. Now the optimal schedule of P is the list schedule S based on L^* . Of course, precedence constraints on P are obeyed in the list schedule S .

Theorem 3.1.3[Coffman and Graham] Algorithm 3.1.2 produces an optimal two machine non-preemptive schedule S for any unit-execution-time task set P .

The proof of this theorem (see details in [5]) is based on a *tower structure* identified in the schedule S produced by the list L^* (defined by Algorithm 3.1.2). We shall use this notion of a tower structure to give a direct, and simpler proof of the

optimality of an algorithm for the general two-machine scheduling problem. This provides a complete alternative proof of the Coffman and Graham's result.

§3.2 A Proof for the General Two-Machine Preemptive Scheduling Problem

Let P be any given partially ordered set, and $a \in P$ be any node in P .

Recall the following definitions from Chapter One: $C^a = \{B | B \subseteq C, \text{ where } C \text{ is a maximal chain containing } a \text{ and for any } c \in C, c > a\}$, $C_a = \{B | B \subseteq C, \text{ where } C \text{ is a maximal chain and for any } c \in C, c < a\}$. Our schedule will make use of both the level and height:

< 1 >. $l(a) = \text{level of } a = \max(|C| : C \in C^a)$.

< 2 >. $H = \text{height of } P = |\text{maximal chain in } P|$.

< 3 >. $h(a) = \text{the height of } a = \max(|C| : C \in C_a)$.

We will also require the concept of a block throughout this Chapter: A *block* in a schedule S of poset P is a smallest closed time interval I such that, if there is a task T with $[\sigma(T), \tau(T)] \cap I \neq \emptyset$, then $[\sigma(T), \tau(T)] \subseteq I$.

Algorithm 3.2.1:

(1). *Label all nodes in P by level.*

(2). *Let $S = \{a \in P | l(a) = H - 1\}$, and assign the nodes in level $H - 1$ as follows:*

Case 1: $|S| \geq 2$, *wrap S in the following way, with the total length of completion time = $\frac{|S|}{2}$.*

T_1	T_3	o o o	T_{n-1}	$T_{n/2}$
T_2	T_4	o o o	$T_{n/2}$	T_{n-2}

$$S = \{ T_1, T_2, \dots, T_n \}$$

fig 3.2.1

Case 2: $|S| = 1$. Let $a \in S$,

$$l_{H_1} = \max(k : \exists c \in P - \{a\} \text{ such that } l(c) = k, k < H - 1 \text{ and } h(c) = 0)$$

$$S_{H_1} = \{a | l(a) = l_{H_1} \text{ and } h(a) = 0\},$$

pick $b \in S_{H_1}$, assign a and b into a unit block(see below). Let $P = P - \{a, b\}$.

o o o	a	o o o
	b	

fig 3.2.2

(3). Suppose all nodes in level k have been assigned.

$$\text{Let } S = \{a \in P | l(a) = k - 1\}.$$

case 1: $|S| \geq 2$, wrap them into blocks as in fig 3.2.1 above. The length of completion time is $\frac{|S|}{2}$. Replace P by $P - S$.

case 2: $|S| = 1$. Say $a_{k-1} \in S$.

$$l_{k-1} = \max(k^* : \exists a \in P - \{a_{k-1}\} \text{ s.t. } l(a) = k^*, k^* < k - 1 \text{ and } h(a) < h(a_{k-1})).$$

$$\text{Let } S_{k-1} = \{a | l(a) = l_{k-1} \text{ and } h(a) < h(a_{k-1})\},$$

pick $b \in S_{k-1}$, schedule a_{k-1}, b into a unit block as in fig 3.2.2.

(4). Replace k by $k - 1$. Go to (3) until all tasks have been scheduled.

Theorem 3.2.1. For any partially ordered set (P, \geq) , the algorithm above(Algorithm 3.2.1) produces an optimal preemptive scheduling of P on two parallel machines.

Proof. For the poset P , Algorithm 3.2.1 produces a schedule S of P .

Observe: (1) Any maximum chain can be scheduled into unit blocks according to Algorithm 3.2.1.

(2) Any block with waste in it must be a unit block, and there is exactly one unit of waste in it.

(3) All blocks occur in one of following forms whose Gantt Charts appear in fig 3.2.3.

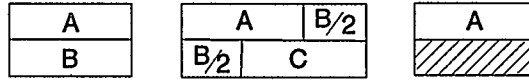


fig 3.2.3

Our task is only to prove that S is optimal.

Let $S(P) = (P^*, \geq^*)$ be the order extension of (P, \geq) induced by S . First look at the lowest (in terms of level) block B_{T_1} with waste in it. By observation (2), it must be a unit block.

Claim 1. Let T_1 be the job scheduled in this block B_1 , $P_1 = \{t \in P | t \geq^* T_1\}$, and $Q = \{t \in P | t \leq^* T_1\}$. So, $P = P_1 \cup Q$ and $P_1 \cap Q = \emptyset$. Then, for any $d \in Q$, $T_1 < d$.

This can be easily seen since Algorithm 3.2.1 uses the level labelling to assign jobs. The jobs on the lowest level will always be assigned first before the jobs on higher levels are dragged down to fill up some waste. Suppose there is some $d \in Q$ and $d < T_1$, thus $l(d) > l(T_1)$. By Algorithm 3.2.1, $d \in P_1$. But $P_1 \cap Q = \emptyset$, so $d \not\leq T_1$. Suppose there is some $D \subseteq Q$ and every member in D is incomparable with T_1 , then there is always some $d \in D$ which can be assigned to block B_1 together with T_1 . So there is no waste in block B_{T_1} .

Claim 2. For this $P_1 \subset P$, the induced schedule for P_1 given by the schedule S for P is optimal on P_1 .

Look at the schedule for (P_1, \geq^*) in S . Let $B = B_1 \oplus B_2 \oplus B_3 \oplus \dots \oplus B_m$ where $B_m = B_{T_1}$, and B_i is a block in (P_1, \geq^*) . We then define a *tower structure* as follows:

Suppose $\exists t_1 \in P_1$ with t_1 in block B_{t_1} , such that $\exists s_1 \in B_{t_1}$ with $l(s_1) \leq l(T_1)$. Choose such a t_1 with $l(t_1)$ maximal, and let $TW_1 = \{t \in (P, \geq) | t_1 \geq t\}$. Here we use TW to denote "tower".

Then let $P_1^1 = P_1 - \{B_i | B_i \leq^* B_{t_1}\}$. We repeat the construction of a tower on P_1^1 : we can find $t_2 \in P_1^1, s_2 \in P_1^1$ and define TW_2 as above, and so on. Since $|P_1| \leq \aleph_0$, the process will stop after finite many steps. Note, t_1 might be equal to T_1 , in this case $TW_1 = P_1$.

Then $P_1 = TW_1 \cup \{s_1\} \cup TW_2 \cup \{s_2\} \cup \dots \cup TW_m$. Observe TW_m contains T_1 , hence there exists no s_m . Moreover the restriction of S to P_1 and the towers satisfies $|S : P_1| = |S : TW_1| + |S : TW_2| + \dots + |S : TW_m|$.

We claim that $S : P_1$ is optimal for P_1 .

Firstly, for any task $a \in TW_i$ and $b \in TW_j$ with $i < j$, we have $a < b$. Suppose, to the contrary, there exists a job $a' \in TW_i$ and $b' \in TW_j$ with $i < j$, and $a' > b'$, then $l(a') < l(b')$. But this is contrary to Algorithm 3.2.1, since this algorithm is always schedule tasks with greater level first. Suppose there exist $a' \in TW_i$ and $b' \in TW_j$ so that a' and b' are incomparable. By Algorithm 3.2.1, $b' > T_i$. This implies a and T_i are incomparable and $l(a) > l(T_i)$. So, there exists some c such that $c > a$ and $l(c) = l(T_i)$. So c could be scheduled with T_i by Algorithm 3.2.1. But this is not the case since the task s_i which is scheduled with T_i in S has level far smaller than the level of T_i .

Secondly, s_1, s_2, \dots, s_m are all those which can be "dragged" down below T_1 (i.e. to levels greater than $l(T_1)$). And $|S : P_1| = |S : (P_1 - \{s_1, s_2, \dots, s_m\})|$. This is simply because the tower structure has a height which is independent of s_i .

Thirdly, $S : (P_1 - \{s_1, s_2, \dots, s_m\})$ is optimal on $P_1 - \{s_1, s_2, \dots, s_m\}$. Since each TW_i has to be performed after TW_{i-1} ($i > 1$) has been finished, optimal scheduling on the tower structure depends on minimization of completion time of each TW_i which is always greater than or equal to $\frac{|TW_i|+1}{2}$. So $\sum_{i=1}^m (\frac{|TW_i|+1}{2})$ would be the minimum completion time of $P_1 - \{s_1, s_2, \dots, s_m\}$. But we have

$$|S : P_1| = |S : (P_1 - \{s_1, s_2, \dots, s_m\})| = \sum_{i=1}^m (\frac{|TW_i|+1}{2})$$

by Algorithm 3.2.1. So $S : P_1$ is optimal.

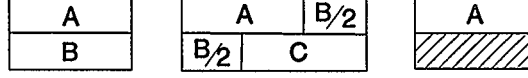
By claims 1 and 2, since Q has to be scheduled after T_1 has been finished, and all jobs $(s_1, s_2, \dots, s_{m-1})$ which are scheduled in $S : P_1$ but have levels less than $l(T_1)$ will not prolong the completion time of $P_1 - \{s_1, s_2, \dots, s_m\}$, so $S(P_1)$ is optimal. Inductively we can assume that $S(Q)$ is also optimal since height of Q is less than the height of P , and thus S is also optimal. \square

We observe that this algorithm demonstrates:

Proposition 3.2.1. Any maximum chain C in P can be scheduled into unit blocks in some optimal two machine preemptive schedule S of P .

§3.3 Application of Transformation Methods in the Two-machine Preemptive Scheduling Problem

This section gives an alternative proof of a lemma due originally to Muntz and Coffman [20] on two machine preemptive scheduling.



A, B, C are three different tasks and waste unit are shaded

fig 3.3.1

Lemma 3.3.1: If (P, \geq) is any poset, (S, \geq_s) is any two machine preemptive scheduling of P , then there exists another scheduling S^* such that S^* schedules all tasks in the forms in fig 3.3.1, and S^* can be obtained by using transformation methods on S with $|S| \geq |S^*|$.

This lemma can be easily proved by following the plan below:

- 1) Let $S(P)$ be the order extension of P induced by the given schedule S of P .
- 2) Apply Algorithm 3.2.1 on $S(P)$, we obtain a schedule S' of P which schedules all tasks in the forms shown in fig 3.3.1.
- 3) Theorem 3.2.1 proves that $|S'| \leq |S|$.

But this proof involves no transformation technique. The following algorithm provides a solution which is different from the proof above. Suppose $S(P) = (P^*, \geq^*) \subseteq (P, \geq)$ is the order extension of P induced from S .

Algorithm 3.3.1 Consider $P^* = S(P)$,

Step 1. Let a_1 be the first task completed in S . Let $\tau(a_1) = t_1$.

Step 2. We construct a storage pool G which is designed to contain those uncompleted tasks in the interim stage of this transformation, and a function $g : G \rightarrow (0, 1)$ which denotes the portion of a task remaining in G . At this time, let $G = \emptyset$.

Step 3. We define two pointers, p_1 and p_2 , which give the status of these two machines. We set $p_1, p_2 \in \mathcal{R}$ (real numbers), where $p_i = j$ means machine i is idle at time j . We first let $p_1 = p_2 = 0$.

Step 4. Look at the interval $[0, t_1]$.

Case 1. If there is only one task, say a_1 , finished in this interval, then we reschedule this interval as follows: schedule a_1 to block $[0, 1]$ on the first machine, then put all others into pool G , and assign a real number to each task in G by the function g which represent the completed proportion of each task. Let $p_1 = 1$. $p_2 = 1$.

Case 2. If there are exactly two tasks, say a_1, a_2 , finished in this block, then we schedule these two tasks into a unit block $[0, 1]$ on these two machines, and put all others into pool G , and reassign values, reflecting the completed portion, to each task by function g . Let $p_1 = 1$. p_2 remains the same.

Step 5. Suppose we have reached t_i , i.e., we have scheduled the i 'th completed task. Let t_{i+1} be the time when we find the next task completed.

Case 1. If there is only one task, say b , completed at t_{i+1} . First, we delete b from G .

If $p_1 > p_2$, then schedule b to the block $[p_2, p_2 + 1]$ on machine w , and move the pointer p_2 forward by replacing p_2 by $p_2 + 1$. Put other tasks in the interval $[t_i, t_{i+1}]$ to pool G , and reassign $g(a)$ for $a \in G$ if necessary. Go to Step 6.

If $p_1 = p_2$, look at the interval $[t_i, t_{i+1}]$:

If there are tasks scheduled in the interval $[t_i, t_{i+1}]$ other than b , then put these tasks into pool G , reassign $g(a)$ for $a \in G$ if necessary. Then schedule b into the interval $[p_1, p_1 + 1]$ on the first machine. Replace p_1 by $p_1 + 1$. Go to Step 6.

If there is no task other than b scheduled in the interval $[t_i, t_{i+1}]$, then look at two

tasks, say c, d , in the interval $[p_1 - 1, p_1]$:

If b is independent to both of these two tasks, c and d , in the interval $[p_1 - 1, p_1]$, then we reschedule b, c, d by wrapping them into block $[p_1 - 1, p_1 + 0.5]$ on both machines. Replace p_1 by $p_1 + 0.5$, p_2 by $p_2 + 0.5$. Go to Step 6.

If b is not independent to c or d in the interval $[p_1 - 1, p_1]$, then schedule b into the interval $[p_1, p_1 + 1]$. Replace p_1 by $p_1 + 1$. Go to Step 6.

Case 2. If there are exactly two tasks, say b, c , completed at t_{i+1} . First, we delete b, c from G . Put all other tasks in this interval $[t_i, t_{i+1}]$ into pool G . Reassign $g(a)$ for $a \in G$ if necessary. Then look at the two pointers p_1, p_2 :

If $p_1 > p_2$ and $p_1 - p_2 \geq 2$, then we schedule b, c into intervals $[p_2, p_2 + 1]$ and $[p_2 + 1, p_2 + 2]$ respectively. Replace p_2 by $p_2 + 2$. Go to Step 6.

If $p_1 > p_2$ and $p_1 - p_2 = 1$, then check the task, say d , scheduled in the interval $[p_1 - 1, p_1]$:

If b, c, d are independent, then wrap them into block $[p_1 - 1, p_1 + 0.5]$ on both machines. Replace p_1 by $p_1 + 0.5$, p_2 by $p_2 + 0.5$. Go to Step 6.

If both b, c are greater than d in P^* , then schedule b, c into block $[p_1, p_1 + 1]$, and replace p_1 by $p_1 + 1$, p_2 by $p_2 + 2$. Go to Step 6.

If only one of them, say b , is greater than d in P^* , then schedule c into block $[p_2, p_2 + 1]$ on machine 2. Schedule b into block $[p_1, p_1 + 1]$ on machine 1. Replace p_1 by $p_1 + 1$, p_2 by $p_2 + 1$. Go to Step 6.

If $p_1 = p_2$, then schedule b, c to the block $[p_1, p_1 + 1]$ on both machines, and replace p_1 by $p_1 + 1$, p_2 by $p_2 + 1$. Go to Step 6.

Step 6. Replace i by $i + 1$, go to Step 5 until all tasks have been rescheduled.

The algorithm above obviously produces the required schedule S' of P^* with all blocks in the forms presented in figure 3.1.2.

Proposition 3.3.1 For any two-machine preemptive schedule S of poset P , Algorithm 3.3.1 produces a two-machine preemptive schedule S' of P with $|S'| \leq |S|$.

Proof. If schedule S' contains no waste, obviously, S' is optimal by Corollary 1.3.1, hence $|S'| \leq |S|$.

Suppose the first waste occur in block B_1 in which task $a_1 \in P$ is scheduled.

Claim 1. For any task $b \in P$ with $\tau(a_1) \leq \sigma(b)$ and any task $c \in P$ with $\tau(c) \leq \tau(a_1)$, $c <^* b$ in order extension $S(P) = (P^*, \leq^*)$ of P induced from the given schedule S .

If $c >^* b$ then c should have been scheduled before b in S' by Algorithm 3.3.1. If c is incomparable with b , then c should have been scheduled with a_1 which is contrary to our assumption that there is a waste in block B_1 which contains a_1 .

Then we can find the next block B_2 with a unit waste in it, and a_2 scheduled in this block. Since P is finite, then we will construct a finite sequence B_1, B_2, \dots, B_m .

Let $T_i = \{a \in P \mid \tau(a) \leq \tau(a_i) \text{ and } \sigma(a) \geq \tau(a_{i-1})\}$, $1 \leq i \leq m$, and a_0 is the first task scheduled on machine 1. So, $\sigma(a_0) = 0$.

Claim 2. Sequence T_1, T_2, \dots, T_m forms a *tower structure*

The same argument applies here as in the proof of Theorem 3.2.1 in §3.2.

Trivially, by the properties of tower structure, $|S'| \leq |S|$. \square

Remark: The proof of Lemma 3.3.1 also shows that any maximum chain can be scheduled in unit blocks in some optimal schedules for every two-machine preemptive scheduling problem.

CHAPTER FOUR

Multi-Machine Scheduling Problems

The multi-processor scheduling problem is much more complicated than the corresponding one or two processor problems. Some problems have been found to be NP-hard (see Chapter Five), and deterministic algorithms have been developed for only a small class of posets for precedence constrained task scheduling. In this chapter, we discuss some of the known results for identical processor scheduling problems over independent as well as dependent task sets.

§4.1. Identical Processors and Independent Tasks

The specific model for this problem in terms of our general scheduling model is:

- Resource: m processors
- Tasks: n independent tasks with processing time $t_j, 1 \leq j \leq n$
- Scheduling Constraints: preemptive/non-preemptive
- Performance Measure: makespan

In the basic one-processor model, the makespan is easy to treat: it is always a constant with respect to a specific task set. In the multi-processor case, however, the makespan problem is no longer trivial.

We know that for any multi-processor problem, there is a lower bound for the minimum makespan, \mathcal{M} :

$$\mathcal{M} = \max\left\{\frac{1}{m} \sum_{j=1}^n t_j, \max_j[t_j]\right\}$$

A fundamental result for the makespan problem was presented by McNaughton [19] when the jobs are independent and preemption is allowed.

Here is the algorithm given by McNaughton:

Algorithm 4.1.1: [19]

- Step 1. Select some job to begin on machine 1 at time zero.*
- Step 2. Choose any unscheduled job and schedule it as early as possible on the same machine. Repeat this step until the machine is occupied beyond time \mathcal{M} (or until all jobs are scheduled).*
- Step 3. Reassign the processing scheduled beyond \mathcal{M} to the next machine instead, starting at time zero. Return to Step 2.*

It is quite easy to see that this algorithm produces an optimal schedule over the task set, since the resulting schedule has zero waste(see Corollary 1.3.1).

If job preemption is prohibited, the problem of minimizing makespan is somewhat more difficult. No efficient algorithm has been developed for calculating the optimal makespan or for constructing an optimal schedule in the general case for three or more processors. A simple yet efficient heuristic procedure for constructing a schedule involves the use of the longest processing time(LPT) for individual tasks as a dispatching mechanism.

Algorithm 4.1.2:

- Step 1. Construct a linear LPT ordering of the jobs, with the longest job first.*
- Step 2. Schedule the jobs in order, each time assigning a job to the machine with the least amount of processing already assigned.*

The LPT heuristic procedure does not guarantee an optimal makespan(See Example 4.1.1). Actually, this problem is NP-complete, since it is in fact a partition problem, and we discuss this problem further in Chapter Five.

Example 4.1.1 Given task set P is $\{T_1, \dots, T_{10}\}$ where the execution times of these tasks form the LPT sequence: 12, 11, 10, 9, 8, 6, 5, 4, 4, 1. Figure 4.1.1(a) gives the schedule S which is produced by Algorithm 4.1.2. Figure 4.1.1(b) is an optimal schedule S' of P . Easy to see S is not optimal, since $|S| = 36 > 35 = |S'|$.

12	9	8	4	1
11	10	6	5	4

The schedule produced by Algorithm 4.1.2

(a)

12	10	8	4	1
11	9	6	5	4

An optimal schedule of P

(b)

fig 4.1.1

§4.2. Identical Processors and Dependent Tasks

When the job set is dependent, the problem of minimizing makespan may be considerably more difficult. The fundamental results for such a situation are: Hu's algorithm on tree structures[13]; Muntz and Coffman's extension of Hu's results to the preemptive case[20]; and Sauer and Stone's algorithm for interval ordered job set[25].

§4.2.1. Multi-processor Scheduling on Forest

The general problem for n unit-execution-time jobs on m identical machines has been found to be NP-hard. But this problem can be solved in polynomial time [13] if the precedence constraints are in the form of a rooted tree or even forest.

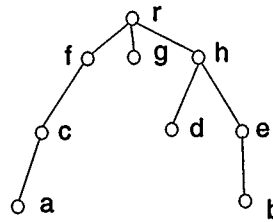
Hu's algorithm is actually a *level scheduling* using the following strategy:

Level strategy: whenever a processor becomes available, assign it an unexecuted available task at the greatest "level" (see Definition 1.3.1).

Algorithm 4.2.1(Hu)[13]:

- (1) label all tasks in P by level.
- (2) take a linear extension L of P according to descending level.
- (3) schedule next the first unexecuted available task in the list L .

Example 4.2.1 Using the level strategy on three processors, for the task system in Fig 4.2.1 (a), the schedule in Figure 4.2.1 (b).



Partially ordered task set P

(a)

a	c	f	r
b	e	h	
d	g		

The schedule S of P produced by Algorithm 4.2.1

(b)

fig 4.2.1

Theorem 4.2.1 If the unit-execution-time task set P is a dual tree, then Algorithm 4.2.1 produces an optimal non-preemptive schedule for P .

Proof: The proof of this theorem pursues the following plan:

1. Suppose the level strategy is not optimal. Then there is a smallest(in the number of tasks) task system P for which the schedule S produced by Algorithm 4.2.1 is not optimal.

2. Let S (the schedule provided by Algorithm 4.2.1) be of length ω . Since S is not optimal, there must be an idle processor during time $\omega - 2$ according to Algorithm 4.2.1

3. Consider the task set P' formed by deleting the root of dual tree P and all its immediate predecessors. A level schedule S' for P' can be formed from S by replacing the deleted tasks by idle periods.

4. The length of S' is $\omega - 2$.

5. Convert the dual forest P' to a dual tree P'' by adding a new task T_r that is an immediate successor of exactly the terminal points of P' . A level schedule S'' for P'' is of length $\omega - 1$.

6. Since P'' has been formed from P by deleting two levels and adding one, P'' must have at least one less task than P . But this level schedule S'' for P'' can not be optimal.

Since we have found a task system smaller than P for which a level schedule is not optimal, we have contradicted the minimality of $|P|$. The theorem follows. \square

Remark: Theorem 4.2.1 can be extended by duality to trees, as well as to dual forests and forests, and the same algorithm(Algorithm 4.2.1) will give an optimal schedule for these structures(here we define a forest as the union of trees).

§4.2.2. Preemptive Extension of Hu's Algorithm

Hu's results can be extended to the preemptive case by methods due to Muntz and Coffman [20]. They consider a unit-execution-time task set P which is a forest. We describe the algorithm which they show produces an optimal preemptive m -machine schedule for P .

Algorithm 4.2.2[20] *For a dual forest P :*

- (1) *label all tasks in P by level.*
- (2) *produce a list L according to the level: tasks with greater levels get arranged earlier in the list.*
- (3) *Use the following strategy to schedule list L : always assign tasks in L with the greatest level first. If there are more than m tasks with the same level, wrap them into a block[c.f. Lemma 1.3.1]. Also, of course, precedence constraints must be obeyed during such assignment.*

Theorem 4.2.2[20] If the unit-execution-time task set P is a dual forest, then Algorithm 4.2.2 produces an optimal preemptive schedule for P .

For a detailed proof for this theorem, see Muntz and Coffman's *Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems*[20].

§4.2.3. Multi-processor Scheduling for Interval Ordered Tasks

Recall that an interval ordered set (I, \prec) is a poset which is isomorphic to some finite set (I_R, \prec) of open intervals on a real line with the natural order: for intervals $A, B \in I_R$, if $A \cap B \neq \emptyset$, then A and B incomparable; otherwise, we say $A \prec B$ iff $a < b$ for all $a \in A$ and $b \in B$. In interval $[a, b]$, we call a the *left endpoint*, b the *right endpoint*.

Interval ordered sets play a very important role in scheduling theory for the following reasons:

- (1). Every schedule S for a poset P induces an order $S(P)$ which is an interval extension of P .
- (2). The m -machine unit-execution-time scheduling on arbitrary interval ordered tasks has a polynomial solution.
- (3). Every poset has an appropriate interval extension which may provide a heuristic approach to multi-machine scheduling problems.
- (4). An independent job set with release times and due times actually forms an interval ordered set with respect to these constraints.

The initial work on interval ordered job sets was done by Papadimitriou and Yannakakis [23] on multi-machine unit-execution-time scheduling for interval ordered job sets, for the non-preemptive case.

Theorem 4.2.3[Papadimitriou and Yannakakis] For any fixed integer m , there is a polynomial time algorithm which provides for each interval ordered set of unit-execution-time tasks P an optimal m -machine (non-preemptive) schedule.

We reproduce their algorithm below for m -machine optimal non-preemptive schedule for unit-execution-time task set P with arbitrary interval order constraints:

Algorithm 4.2.1

- (1) *Sort tasks in P in order of increasing right endpoints. This gives a list $L(P)$ of P .*
- (2) *We schedule tasks in P according to this list $L(P)$: schedule next the first available task in $L(P)$.*

Actually, the algorithm given above is a *list schedule*. But this algorithm works

only for non-preemptive scheduling constraints. For the preemptive case, Sauer and Stone proved the following result:

Theorem 4.2.2 For each fixed positive number m , there is a polynomial time algorithm which provides for each interval ordered set of task P an optimal m -machine preemptive schedule.

Before we introduce the details of the polynomial algorithm given by Sauer and Stone as they appear in [25], we need the following notations and lemma:

For any interval order P , and a fixed positive integer m , let

$$[P]^{\leq m} = \{B \subseteq P; |B| \leq m, B \text{ antichain}\}$$

We call this set an m -set. Now we define a binary relation $<$ on $[P]^{\leq m}$, by taking $B < B'$ for $B, B' \in [P]^{\leq m}$ iff there exists b in B and there exists b' in B' such that b is less than b' in P . We call the relation $<$ which is defined in this way that *induced by P on $[P]^{\leq m}$* .

Lemma 4.2.1 [25]. For any interval order P , the binary relation $<$ induced on $[P]^{\leq m}$ by P is a partial order.

Proof. It suffices to check that $<$ is transitive and irreflexive. Assume $P = \{I_1, \dots, I_n\}$ is a set of open interval representation for P on the real line, where $I_k < I_j$ means: $\forall x \in I_k, \forall y \in I_j, x < y$. Then for $B, B' \in [P]^{\leq m}$, observe $B < B' \iff \cap B < \cap B'$. Thus, from $B < B' < B''$ we have $\cap B < \cap B' < \cap B''$, hence $\cap B < \cap B''$ and $B < B''$. Thus $<$ is transitive. Observe that $B < B$ is impossible since B is itself an antichain in P . Thus, $<$ is a partial order on $[P]^{\leq m}$. \square

Next let $<^*$ be any linear extension of the order $<$ induced on $[P]^{\leq m}$. Now, look at the following linear programming problem:

The Variables α_i are indexed by the m-set $i \in I = [P]^{\leq m}$. The objective function to be minimized is $\sum \alpha_i$ subject to the boundary condition:

$$\forall p \in P, \sum_{p \in i \in I} \alpha_i = 1, \alpha_i \geq 0 \text{ for all } i \in I$$

This is a linear programming problem in standard form. Let S be an optimal preemptive m-machine schedule for P which schedules P over intervals I_1, \dots, I_k where the tasks assigned to each machine do not vary within each I_j . By taking $\alpha_j =$ the width of the interval I_j , we have a feasible solution to the problem, and more recently Khachian and others have given polynomial time algorithms for finding such solutions. Employing a polynomial solution for the linear programming problem, we then have the following polynomial solution to the preemptive scheduling problem for an interval order P with fixed number of machines m :

Algorithm 4.2.2.(Sauer & Stone)

- (1) *construct the partial order on $[P]^{\leq m}$ by P .*
- (2) *Take a linear extension $<^*$ of the order induced on $[P]^{\leq m}$ by P .*
- (3) *Solve the linear programming problem using a polynomial algorithm: $\min \sum_i \alpha_i$ subject to $\sum_{p \in i \in I} \alpha_i = 1$ with $\alpha_i \geq 0$ for all $i \in [P]^{\leq m}$.*
- (4) *Schedule P with each shift $i \in [P]^{\leq m}$ scheduled for duration α_i , with the shifts ordered as in $<^*$.*
- (5) *The result is an optimal schedule for P .*

Remark: Sauer and Stone's result can also be extended to the variable completion time case: we just modify the linear programming problem in the Algorithm 4.2.2 by requiring for all $p \in P$ that $\sum_{p \in i} \alpha_i = t_p$ where t_p is the time required for each of

the machines to complete job p .

The general multi-processor scheduling problem has been proved to be NP-hard (see [4] and [28]). Several special classes of posets are shown above to have a polynomial time solution. Most other multi-processor scheduling results focus on a heuristic approach. We will, in the following chapters, investigate a heuristic method for multi-processor scheduling based upon an effort to schedule maximum chains in unit blocks, as a first step in achieving an optimal schedule. We will explore both the limitations and the potential for this heuristic in providing optimal scheduling for certain classes of precedence constrained tasks.

CHAPTER FIVE

The Complexity of Scheduling Problems

In the field of scheduling theory, as in many other areas, we come across numerous problems that can be solved relatively easily, while other, similar problems appear quite hard. For example, the optimal scheduling on two processors of n unit-execution-time tasks can be solved in $O(n^2)$ steps (i.e. the required number of steps in the solution is bounded by a polynomial of degree 2). However, the same problem with three processors appears to require time that is exponential in n .

There is no known way to prove that exponential time is required for the three processor unit-execution-time problem, or for many other basic scheduling problems, for that matter. What we can do is to show that there is a large class of problems, called "NP-complete" problems, for which either all or none of them have polynomial-time solutions. The NP-complete problems include many well-known problems, such as the traveling salesman problem or the general m -processor, n -task scheduling problem.

As a consequence, it becomes important to determine whether or not a given problem is NP-complete; for if so, we very likely can obtain results which provide at best only a heuristic which gives useful approximations to the optimal solution. None of the results we present here is new, however we provide our own proof for NP-completeness of the two machine non-preemptive scheduling problem on independent task set. These results provide a general overview of computational complexity for scheduling problems.

§5.1 An Introduction to NP-completeness Theory

As a matter of convenience, the theory of NP-completeness is most often described in the literature with reference to *decision problems*. Such problems have only two possible solutions, either the answer "yes" or the answer "no". So we can simply describe a decision problem as a finite set of "instances" plus a yes-no question.

Example 5.1.1:

Instance: A finite set of n numbers $T = \{k_1, k_2, \dots, k_n\}$.

Question: Does there exist a subset T' of T so that the sum of the numbers in T' is equal to a half of the sum of the numbers in T .

Example 5.1.2:

Instance: A finite poset P of tasks, and a three machine schedule S of P .

Question: Is S optimal? (i.e., does there exist another three-machine schedule S' of P such that the makespan of S' is less than S ?).

The reason that the theory of NP-completeness is usually restricted to decision problems is that they have a very general and simple pattern, which is very suitable to study in mathematical "language". This *language* is defined in the following way:

For any finite set Σ of symbols, we denote by Σ^* the set of all finite strings of symbols from Σ . If x is a string, we denote by $|x|$ the length of x . For example, if $\Sigma = \{0, 1\}$ then Σ^* consists of the empty string ϵ , the string 0, 1, 00, 111, 011, ... and all finite strings of 0's and 1's. If \mathcal{L} is a subset of Σ^* , we say that \mathcal{L} is a *language* over the alphabet Σ . Thus $\{01, 001, 111, 1010\}$ is a language over $\{0, 1\}$.

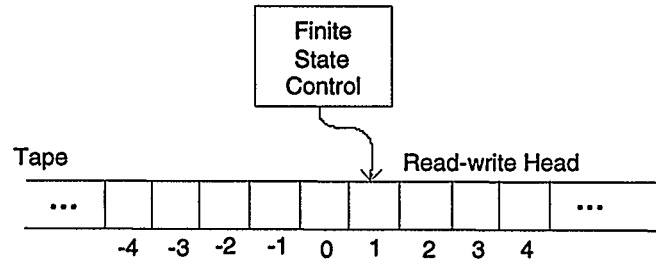
The correspondence between decision problems and languages is brought about

by the *encoding schemes* we use for specifying problem instances whenever we intend to compute with them. An encoding scheme \mathcal{E} for a problem \mathcal{P} provides a way of describing each instance of \mathcal{P} by an appropriate string of symbols over some fixed alphabet Σ . Thus the problem \mathcal{P} and encoding scheme \mathcal{E} for \mathcal{P} partition Σ^* into three classes: (1) those that are not encoding of instances of \mathcal{P} ; (2) those that encode instances of \mathcal{P} for which the answer is "no" (3) those that encode instances of \mathcal{P} for which the answer is "yes".

Obviously, language consists of encoded instances, and the size of the instance is just the length of the string which is the encoding code of this instance under some encoding scheme. Hence different encoding schemes will produce different languages even for the same problem. For convenience of study, we generally use some "widely accepted" encoding scheme for each specific problem.

§5.1.1 Deterministic Turing Machine and Class P

In order to formalize the notion of an algorithm which is used to solve a problem, we will need to fix a particular model for computation which will provides a standard measure to determine the complexity of the algorithm, and tells exactly how many steps it will take to solve a problem. The model we choose is the *deterministic one-tape Turing machine* (briefly DTM), which is pictured in figure 5.1.1. It consists of a *finite state control*, a *read-write head*, and a *tape* made up of a two-way infinite sequence of tape squares, labeled $\dots, -2, -1, 0, 1, 2, 3, \dots$



Schematic representation of a DTM

fig 5.1.1

A *program* for a DTM specifies the following information:

- (1) A finite set Γ of tape symbols, including a subset $\Sigma \subset \Gamma$ of "input symbols" and a distinguished "blank symbol" $b \in \Gamma - \Sigma$.
- (2) A finite set Q of states, including a distinguished "start-state" q_0 and two distinguished "halt-state" q_Y and q_N .
- (3) A transition function $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, +1\}$.

The operation of such a program is straightforward. The input to the DTM is a string $x \in \Sigma^*$. The string x is placed on the tape in positions 1 through $|x|$, one symbol per square. The program starts its operation in state q_0 , with the read-write head scanning tape square 1. Then the processing continues according to the transition function δ . If the current state q is either q_Y or q_N then the computation has ended. Otherwise, the machine will continue to run.

Time complexity of a DTM program M on an input x is measured by the number of steps occurring in that computation until a halt state is entered.

Let x be an input of a program M on a DTM, where $|x| = n$. Suppose computation of M on input x takes time m , then the time complexity function: $T_M : Z^+ \longrightarrow Z^+$ is given by:

$$T_M(n) = \max\{m : m \text{ is computation time of } M \text{ on } x \text{ with } |x| = n\}$$

Such a program M is called a *polynomial time DTM program* if there exists a polynomial p such that, for all $n \in \mathbb{Z}^+$, $T_M(n) \leq p(n)$.

Now we can define an important class of problems (or languages), the *Class P*:

$$P = \{\mathcal{L} : \text{there is a polynomial time DTM program } M \text{ for which } \mathcal{L} = \mathcal{L}_M\}$$

where \mathcal{L}_M is the language for program M on DTM using some encoding scheme, i.e., a set of strings for which the halt state of M is q_Y .

We say an algorithm for a given problem is a *polynomial time algorithm* if the algorithm produces a DTM program, under some encoding scheme, which solves the problem in polynomial time.

§5.1.2 Non-deterministic Computation and the Class NP

A Non-Deterministic one-tape Turing Machine (NDTM) has exactly the same structure as a DTM, except that it is augmented with a guessing module having its own write-only head, as illustrated in figure 5.1.2:

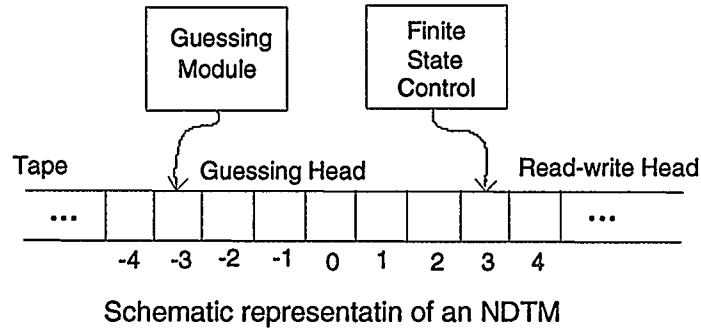


fig 5.1.2

An NDTM program is specified in exactly the same way as a DTM program, including the tape alphabet Γ , input alphabet Σ , blank symbol b , state set Q , initial

state q_0 , halt states q_Y and q_N and a transition function. The computation procedure of an NDTM program is different from that for a DTM program. It takes place in two distinct stages: the first stage is the *guessing stage*, the second is the *checking stage*.

Example 5.2.1(The Traveling Salesman Problem)

Instance: A finite set $C = \{c_1, c_2, \dots, c_m\}$ of "cities", a "distance" $d(c_i, c_j) \in \mathbb{Z}^+$ for each pair of cities $c_i, c_j \in C$, and a bound $B \in \mathbb{Z}^+$ (where \mathbb{Z}^+ denotes the positive integers).

Question: Is there a "tour" of all the cities in C having total length no more than B , that is, an ordering $\langle c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(m)} \rangle$ of C such that

$$\{\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\} + d(c_{\pi(m)}, c_{\pi(1)}) \leq B ?$$

A non-deterministic algorithm for the Traveling Salesman Problem could be constructed using a guessing stage that simply guesses an arbitrary sequence of the given cities and a checking stage that provides a polynomial "proof verifier" to check that the sequence of cities is presenting a valid tour, and to determine whether the tour has length less than or equal to B .

Note that the time of a NDTM program used to "solve" a problem is just the number of steps which leads the program halting at state q_Y . We do not count the steps which leads to state q_N . A NDTM program is *polynomial time program* if the program can verify a instance to be true in polynomial time.

Finally, the class NP is formally defined as:

$$\text{NP} = \{\mathcal{L} : \text{there is a polynomial time NDTM program } M \text{ for which } \mathcal{L} = \mathcal{L}_M\}$$

Informally, we say a problem belongs to the class NP if there exists a non-

deterministic checking algorithm which can verify any specific instance of the problem in polynomial time. A non-deterministic algorithm is the counterpart of an NDTM program, which also has two stages: guessing and checking.

Based on the above discussions, we observe that $P \subset NP$. Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time non-deterministic algorithm. Since any deterministic algorithm can be used as the checking stage of a non-deterministic algorithm, where the guessing stage is ignored in this circumstance. It is a well known, long outstanding and very difficult problem to determine whether $P = NP$ or not.

§5.1.3 NP-completeness

If P differs from NP , then the distinction between P and $NP-P$ is meaningful and important. All problems in P can be solved by some polynomial time algorithms, whereas all problems in $NP-P$ are said to be *intractable* because a polynomial solution to any problem in $NP-P$ would necessarily show $P = NP$. Thus, for a given problem, it is important to distinguish which of these two possibilities holds for the problem, and to seek primarily heuristic approaches only for problems known to be NP -complete.

Thus unless we can prove that $P \neq NP$, there is no hope of showing that a specific problem Π belongs to $NP-P$. For this reason, NP -completeness theory focuses on proving weaker results of the form: "if $P \neq NP$, then $\Pi \in NP-P$ ", e.g., "If $P \neq NP$ then "The Partition Problem" belongs to $NP-P$.

To approach such conditional results might appear as difficult as the corresponding unconditional results. A key technique, *polynomial transformation*, was devel-

oped to make it more straightforward.

A *polynomial transformation* from a language $\mathcal{L}_1 \subset \Sigma_1^*$ to a language $\mathcal{L}_2 \subset \Sigma_2^*$ is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ that satisfies the following two conditions: (1) there is a polynomial time DTM program that computes function f ; (2) for all $x \in \Sigma_1^*, x \in \mathcal{L}_1$ if and only if $f(x) \in \mathcal{L}_2$.

If there is a polynomial transformation from \mathcal{L}_1 to \mathcal{L}_2 , we write $\mathcal{L}_1 \propto \mathcal{L}_2$, read as " \mathcal{L}_1 transforms to \mathcal{L}_2 " (we can omit "polynomial" here which is to be understood.)

The significance of polynomial transformations comes from the following results:

Theorem 5.1.1 If $\mathcal{L}_1 \propto \mathcal{L}_2$, then $\mathcal{L}_2 \in P$ implies $\mathcal{L}_1 \in P$ (and, equivalently, $\mathcal{L}_1 \notin P$ implies $\mathcal{L}_2 \notin P$)

Theorem 5.1.2 If $\mathcal{L}_1 \propto \mathcal{L}_2$ and $\mathcal{L}_2 \propto \mathcal{L}_3$, then $\mathcal{L}_1 \propto \mathcal{L}_3$.

Proofs of these two theorems are trivial ,but can be found for example in [11].

We say that two languages \mathcal{L}_1 and \mathcal{L}_2 (or two decision problems Π_1 and Π_2) are *polynomially equivalent* whenever both $\mathcal{L}_1 \propto \mathcal{L}_2$ and $\mathcal{L}_2 \propto \mathcal{L}_1$ (or, respectively, both $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$).

The *NP-completeness class* is defined as follows: a language \mathcal{L} is NP-complete if $\mathcal{L} \in \text{NP}$ and, for all other languages $\mathcal{L}' \in \text{NP}$, $\mathcal{L}' \propto \mathcal{L}$. Informally, a decision problem Π is NP-complete if $\Pi \in \text{NP}$ and, for all other decision problems $\Pi' \in \text{NP}$, $\Pi' \propto \Pi$.

Theorem 5.1.3 If \mathcal{L}_1 and \mathcal{L}_2 belong to NP, \mathcal{L}_1 is NP-complete, and $\mathcal{L}_1 \propto \mathcal{L}_2$, then \mathcal{L}_2 is NP-complete.

Proof: Since $\mathcal{L}_2 \in \text{NP}$, all we need to do is show that, for every $\mathcal{L}' \in \text{NP}$, $\mathcal{L}' \propto \mathcal{L}_2$.

Consider any $\mathcal{L}' \in NP$. Since \mathcal{L}_1 is NP-complete, it must be the case that $\mathcal{L}' \propto \mathcal{L}_1$. The transitivity of \propto and the fact that $\mathcal{L}_1 \propto \mathcal{L}_2$ then imply that $\mathcal{L}' \propto \mathcal{L}_2$. \square

This theorem provides us a way to prove that a certain problem is NP-complete, once we have at least one known NP-complete problem. Such a problem is provided by Cook's fundamental theorem [3].

The so-called "first" NP-complete problem originates in a decision problem from Boolean logic, which is usually referred to as the *SATISFIABILITY* problem. The terms we shall use to describe it are defined below.

Let $U = \{u_1, u_2, \dots, u_m\}$ be a set of Boolean *variables*. A *truth assignment* for U is a function $t : U \longrightarrow \{T, F\}$. If $t(u) = T$ we say that u is "true" under t ; if $t(u) = F$ we say that u is "false". If u is a variable in U , then u and \bar{u} are *literals* over U . The literal u is true under t if and only if the variable u is true under t ; the literal \bar{u} is true if and only if the variable u is false.

A *clause* over U is a set of literals over U , such as $\{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. The clause above will be satisfied by t unless $t(u_1) = F, t(u_3) = T$, and $t(u_8) = F$. A collection C of clauses over U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C . Such a truth assignment is called a *satisfying truth assignment* for C . The *SATISFIABILITY* problem is specified as follows:

SATISFIABILITY:

Instance: A set U of variables and a collection C of clauses over U .

Question: Is there a satisfying truth assignment for C ?

Theorem 3.1.4(Cook's Theorem): SATISFIABILITY problem is NP-complete.

Proof: A complete proof is given in [3].

Cook's theorem provides for us a way to show a new problem is NP-complete by proving that it is polynomially transformable from a known NP-complete problem, such as the SATISFIABILITY problem. There are some other well-known NP-complete problems such as the PARTITION problem, and the TRAVELLING SALESMAN problem.

§5.2 Computational Complexity of Scheduling Problems

Throughout Chapter One to Chapter Four, we have encountered many scheduling problems that are quite hard. Complexity analysis may suggest that we avoid attempts to solve some intractable scheduling problem in polynomial time, and put more energy toward developing heuristic approaches to optimal or suboptimal solutions of some tough problems. Some of the most well known NP-complete scheduling problems are described below.

Theorem 5.2.1 The general scheduling problem is NP-complete.

Proof. The lengthy proof can be found in [14] and [4].

More specifically, we have following small classes of scheduling problems which have been found NP-complete:

Theorem 5.2.2 The general two machine scheduling problem on an independent

job set with arbitrary execution times is NP-complete.

Proof. This problem, denoted in the literature as $P2||C_{max}$, can be easily transformed from the partition problem which is a well-known NP-complete problem. Here is the general language of the Partition Problem:

Instance: *Finite set A and a size $s(a_i) \in \mathbb{N}$ for each $a_i \in A$ where \mathbb{N} is the set of positive integers.*

Question: *Is there a subset $A' \subset A$ such that $\sum_{a_i \in A'} s(a_i) = \sum_{a_i \in A-A'} s(a_i)$?*

Given any instance of Partition Problem defined by the set of positive integers $\{s(a_i) : a_i \in A\}$, we define a corresponding instance of the decision counterpart of $P2||C_{max}$ as follows: assume job set $P = \{b_1, b_2, \dots, b_n\}$ where $n = |A|$, the execution time p_j of job $b_j = s(a_j)$ for $j = 1, 2, \dots, n$, and the threshold value of schedule length $y = (1/2) \sum_{a_i \in A} s(a_i)$. It is obvious that there exists a subset A' with the desired property for the instance of Partition Problem if, for the corresponding instance of $P2||C_{max}$, there exists a schedule with $C_{max} \leq y$, and the theorem follows. \square

Theorem 5.2.3 The general scheduling problem of n unit-execution-time tasks on m processors is NP-complete.

For detailed proof, see [29] or [4].

CHAPTER SIX

Scheduling Maximum Chains in Unit Blocks

In Chapter Five (Complexity of Scheduling Problems), we have seen that many general scheduling problems are NP-hard. Only heuristic approaches therefore provide practical approaches to work on for such problems. Here in this chapter and the following chapters, we will explore a new heuristic method, scheduling maximum chains in unit blocks, to approach the multi-machine preemptive scheduling problem. We examine the conjecture that each task in a maximum chain can be scheduled in unit time in some optimal preemptive schedule.

§6.1 Schematic Representations

In order to simplify the Hasse Diagram of a poset, we introduce the concept of *schematic structure of a poset* which will be adopted in the following chapters.

(1) A binary equivalence relation \sim on a poset P is defined for $a, b \in P$ by $a \sim b$ iff $c < a \iff c < b$ and $a < d \iff b < d$, where $c, d \in P$.

(2) A *schematic structure* of a poset P is P / \sim . In this structure, we partition the poset P into equivalence classes P_1, \dots, P_m . A box with only a positive integer $n = |P_i|$ in it stands for the block P_i (it is actually an antichain of size n). Two boxes with one above another denotes a "bipartite" graph in which all nodes in the upper poset are above all nodes in the lower poset. For example:

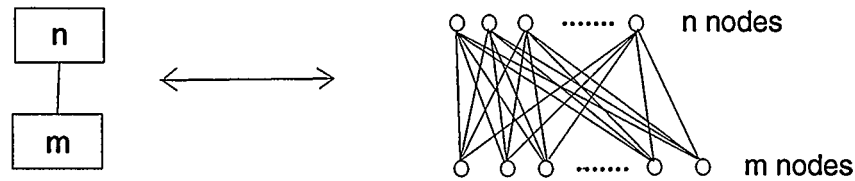


fig 6.1.1

(3) A *revised schematic structure* of a poset P is a simplified schematic structure of P by using a box with an uppercase word in it which stands for a special structure we have specified before. For example, we will frequently use the 3-S/S structure in fig 6.1.2 (a). It can also be represented schematically in fig 6.1.2 (b), and as an element in other revised schematic structures as shown in fig 6.1.2(c).

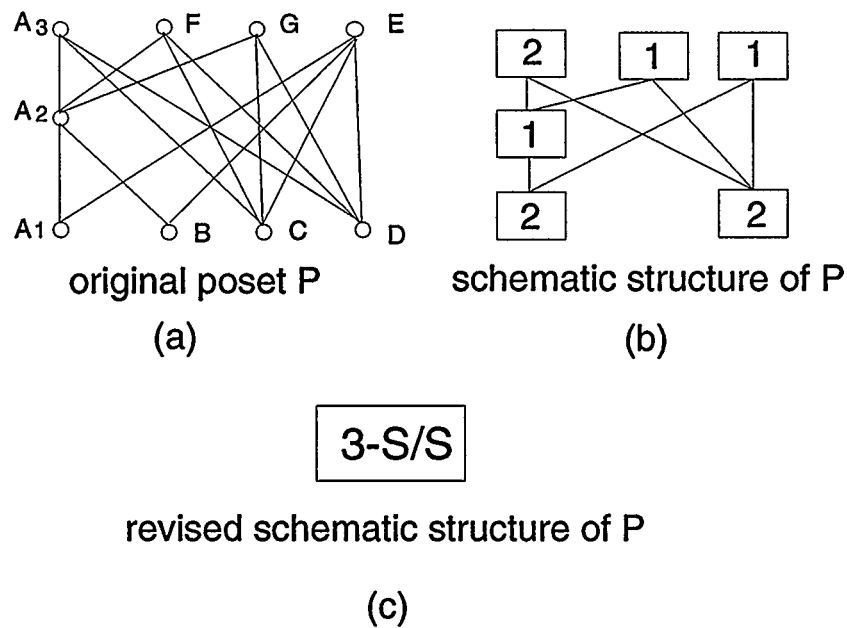


fig 6.1.2

§6.2 A Counterexample

Part of the early motivation for this thesis was to investigate and prove or disprove the following conjecture: For any poset P , and any maximum chain $C_1 < \dots < C_n$ in P , there exists a optimal scheduling S , such that $\tau(C_i) - \sigma(C_i) = 1$, for $1 \leq i \leq n$.

In our early efforts to prove this conjecture, we attempted to use transformation methods. The question is "Can we obtain such scheduling S by locally rearranging some other optimal scheduling S' ?" By "local rearrangement", we mean a rearrangement of the block which contains all tasks scheduled simultaneously with some C_i in C . We found the answer is no, and an example is given in fig 6.2.1:

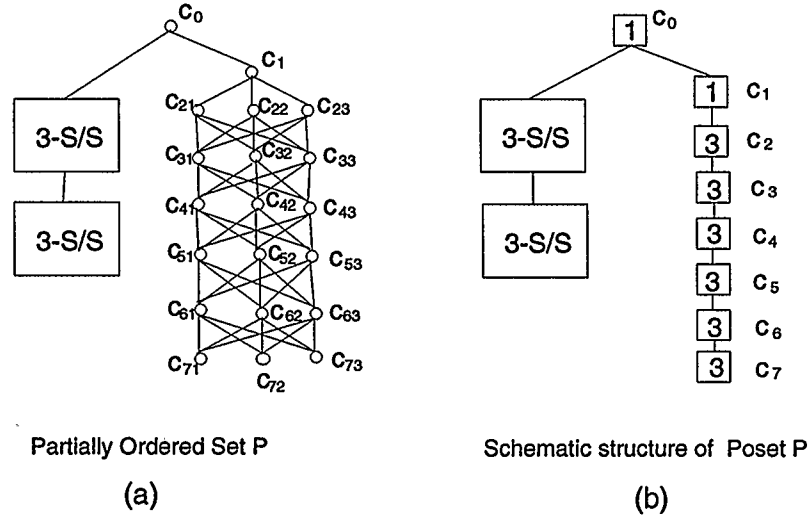


fig 6.2.1

Indeed, the conjecture itself is generally false. Not only is it impossible to rearrange some optimal schedules locally, but there are actually posets P in which every maximum chain C has to be split for some member $C_i \in C$ in order for a schedule to be optimal. An example of such a poset is shown below in fig 6.2.4.

It is known (see [24]) that the optimal schedule of 3-S/S in fig 6.1.2 is (up to the order isomorphism on $S(P)$) unique:

A ₁		A ₂		A ₃	F/6
B		C/	E/2	5/6 F	G/3
C/	D/	D/		2/3 G	E/2

The Optimal Schedule for 3-S/S

fig 6.2.2

An optimal schedule S^* of P in fig 6.2.1 is shown below in fig 6.2.3:

C ₀	3-S/S	3-S/S	C ₂₁	C ₃₁	C ₄₁	C ₅₁	C ₆₁	C ₇₁
			C ₂₂	C ₃₂	C ₄₂	C ₅₂	C ₆₂	C ₇₂
			C ₂₃	C ₃₃	C ₄₃	C ₅₃	C ₆₃	C ₇₃

Schedule for P in fig 6.2.1

fig 6.2.3

In the above scheduling S^* of the poset P , C_1 is scheduled together with two identical structures, $3 - S/S$. It splits into two halves to fix two intervals of waste found in the schedules for these two copies of $3 - S/S$. One proves easily that node C_1 on the maximum chain C of P cannot be locally rearranged so that optimality can be maintained and C_1 scheduled into a unit block.

The above example shows that for some specific schedule S^* of P , some tasks in a maximum chain will be required to be scheduled in pieces. However for the example in fig 6.2.1, there do exist some other optimal schedules of P which schedule C_1 in a unit block(as well as all other tasks in the maximum chain C).

The following example shows that for some posets P , certain tasks in some or even all maximum chains have to be scheduled in non-consecutive pieces in order to

get an optimal scheduling. This is one of the major results of our thesis. It disproves a natural conjecture, and shows that any successful heuristic based on scheduling maximum chains in unit blocks will necessarily apply only to a restricted class of posets.

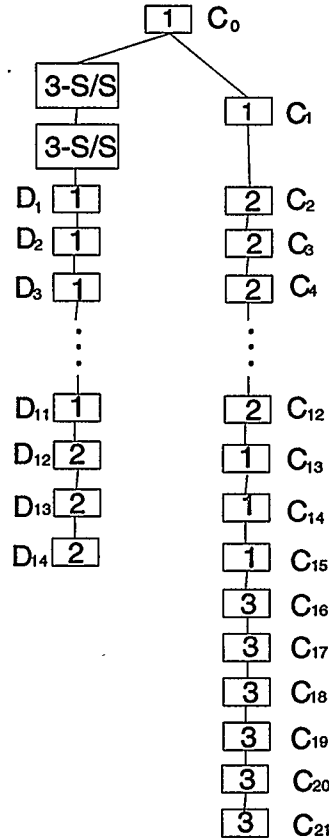


fig 6.2.4

Proposition 6.1. Every optimal 3-machine preemptive schedule S for the poset shown in fig 6.2.4 schedules C_1 in non-consecutive pieces, with $\tau(C_1) - \sigma(C_1) > 1$, and every maximum chain in P contains C_1 .

Proof: First, let us investigate some of the obvious properties of the poset P . From the schematic structure of P , we can see, for three-machine preemptive scheduling:

- 1). C_1 is on any maximum chain of P .
- 2). there are at least two units waste for any scheduling of P . Two of them occur during the processing of C_0 .
- 3). there is a pattern shown in the structure of P : $|D_1| + |C_2| = |D_2| + |C_3| = \dots = |D_{14}| + |C_{15}| = 3$. These fourteen levels can be easily scheduled in fourteen unit blocks with no waste. The tasks after the last level ($D_{14} + C_{15}$) can also be scheduled with no waste by three parallel machines.
- 4). the substructure $3 - S/S$ in fig 6.2.4 has been discussed in [24] ; the waste has to appear in the middle of the structure if optimally scheduled by three machines.

To schedule P , from our observations 1) to 4) above, if C_1 can fix the waste which may occur when we schedule two copies of $3 - S/S$ structure, then we are done. We have found an optimal scheduling, by Corollary 1.2.2, since two units of waste is best possible.



C_0			C_2	C_3	C_4	C_5	...	D_{21}
	$3-S/S$	$3-S/S$	C_2	C_3	C_4	C_5	...	D_{21}
	$C_{1/2}$	$C_{1/2}$	D_1	D_2	D_3	D_4	...	D_{21}

fig 6.2.5

Let us consider all the possible situations where C_1 might be scheduled into a unit block in some scheduling. We shall see no such schedule is optimal. It then follows that every optimal schedule S for P splits the task C_1 (which occurs in every maximum chain) into non-consecutive pieces. For convenience, we denote the copy of the structure $3 - S/S$ near C_0 A , the other copy B .

- 1). If we schedule C_1 ahead of A and B , then from Observation 4), there will

be some idle time period on some machines when we schedule A and B into three machines, which will make this scheduling not optimal.

So, C_1 has to be scheduled with the $3 - S/S$ near C_0 . It may or may not be scheduled with the other copy of $3 - S/S$, since we can schedule C_2 with this copy, or even C_3, C_4, \dots

2). If we schedule part of C_2 with $A \oplus B$, and the remainder of C_2 with D_1 . To avoid extra waste with D_1 , we need to pull part of C_3 back to schedule with D_1 , and part of C_4 back with D_2 , and so on. Finally, when we're trying to pull C_{13} back to schedule with D_{11} , we find we can't avoid occurrence of additional waste.

So, either C_2 has to be scheduled ahead of B , or completely with $A \oplus B$.

2.1) If C_2 is scheduled completely with $A \oplus B$, then we have to move C_3 up at least with D_1 , and C_4 at least to D_2 , and finally, C_6 at least to D_4 . Then we may create waste during the schedule of $C_{13} \oplus C_{14} \oplus C_{15}$.

2.2). There are only two ways to fix waste created by scheduling of $C_{13} \oplus C_{14} \oplus C_{15}$: one is to use $D_{12} \oplus D_{13} \oplus D_{14}$; another is to use $A \oplus B$. If we use the first option, it will force C_1 to be split. If we use the second, we will find it is impossible for $A \oplus B$ to fix all waste created by $C_1 \oplus C_2 \oplus \dots \oplus C_{15}$. There are in total 19 units of waste when we schedule $C_1 \oplus \dots \oplus C_{15}$ into three machines. But these two copies of the 3-S/S structure have only 18 tasks.

3). If we schedule D_{14} after part of C_{15} , then we create waste. So, D_{14} has to be scheduled with or before C_{15} .

4). If we schedule part of D_{14} before C_{15} , and the remainder of D_{14} with C_{15} , then we need to pull D_{13} back to schedule with C_{15} , and then D_2 back with C_{14} , and finally we find waste cannot be avoid when we are trying to pull D_{11} back to

schedule with C_{13} .

So, D_{14} has to be scheduled with C_{15} . This completes the proof of Proposition 6.1.

Extension to the n -machine case, $n > 3$

In order to construct a counterexample for the n -machine preemptive scheduling case, first, we introduce a poset n -S/S similar to the 3-S/S structure in fig 6.1.2:

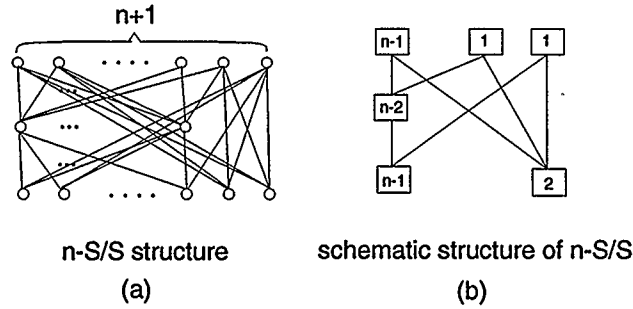
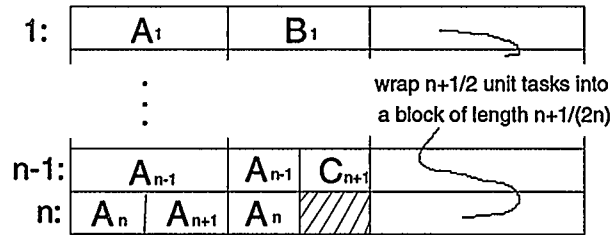


fig 6.2.6

Any optimal preemptive schedule of the above poset will force a half unit waste in the middle of the Gantt Chart (see fig 6.2.7 below). The argument for this is very similar to that in [24].



An optimal preemptive schedule
of n -S/S structure

fig 6.2.7

Next, we construct a counterexample for the n -machine case similar to that for 3-machine case given in fig 6.2.4:

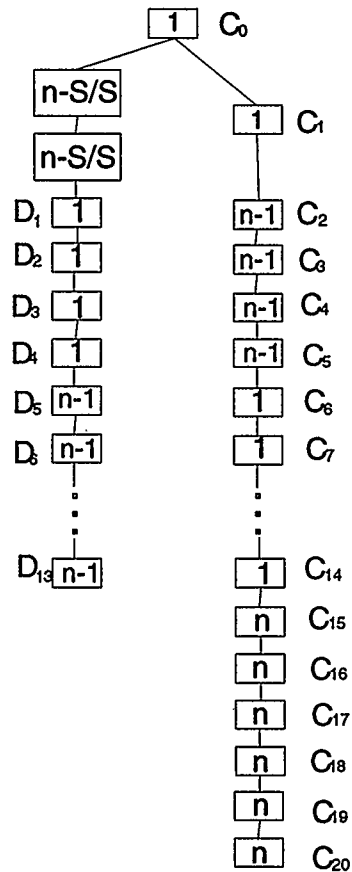
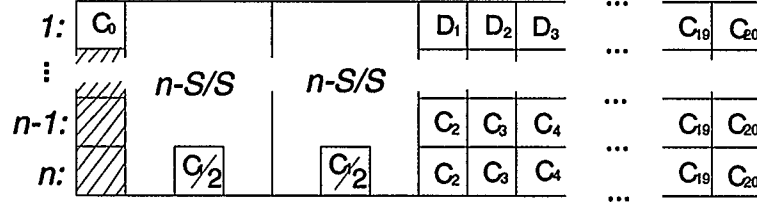


fig 6.2.8

For the dual counterpart of the poset shown in fig 6.2.8, each optimal preemptive schedule S must split the task C_1 into two separate shifts which cannot be scheduled consecutively. The only optimal schedules for this poset are essentially those shown in fig 6.2.9 below:



An optimal n -machine preemptive schedule of poset in fig 6.10

fig 6.2.9

Proposition 6.2. Every optimal n -machine preemptive schedule S for the poset shown in fig 6.2.8 schedules C_1 in non-consecutive pieces, with $\tau(C_1) - \sigma(C_1) > 1$, and every maximum chain in P contains C_1 .

The proof of this proposition is quite similar to that of 3-machine case.

Remark: For the example shown in fig 6.2.4 and fig 6.2.8, we observe for the order extension $S(P)$ of P induced by a schedule S :

$S(P) = S'(P)$ where S, S' are any two optimal schedules of the given poset.

This shows that any optimal preemptive schedule S of poset P in fig 6.2.8, for example, will produce the same order extension $S(P)$ with same maximum chains in it. And further, the maximum chains in this order extension $S(P)$ can always be rescheduled in unit blocks, as we shall see. Thus, although maximum chains in P cannot be scheduled in unit blocks in any optimal schedule for this poset P , maximum chains in the order extension $S(P)$ for any optimal schedule S , can be so scheduled. We look further into the problem of scheduling maximum chains of $S(P)$ in unit blocks in the next chapter.

CHAPTER SEVEN

Rescheduling Maximum Chains in $S(P)$

In Chapter Six, we have shown that maximum chains in a poset P can not necessarily be scheduled in unit blocks in any optimal schedule. Moreover, even for a particular task, local transformation will fail, in certain schedules, to provide us with a way to locally adjust the given schedule so that this "fragmented" task can be rescheduled locally into a unit block. We now investigate problems encountered in rearranging the schedule itself, with respect to scheduling tasks in unit blocks.

§7.1 Observations

We will investigate several cases as follows:

(1). One-machine preemptive scheduling.

It is trivial to see that any linear extension of a poset P forms a (preemptive) optimal schedule of P in which each task, in particular any maximum chain in P , is easily scheduled into unit blocks.

(2). Two-machine preemptive scheduling.

For the two machine case, Algorithm 3.1.1 and Algorithm 3.2.1 in Chapter Three have given two-machine preemptive optimal schedules of P which schedule any chosen maximum chain into unit blocks.

(3). Multi-machine preemptive scheduling.

In Chapter Six, we have given a counterexample which shows that, for some posets, there exists no multi-machine optimal schedule which assigns any maximum chain in unit blocks.

The counterexamples given in Chapter Six suggest that we explore another problem: whether or not maximum chains in $S(P)$ can be rescheduled into unit blocks, without lengthening the schedule S .

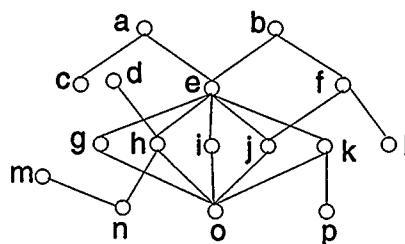
In 1990, Sauer and Stone showed in a paper presented at the Second International Congress on Scheduling at Compiègne, France [23], that every optimal schedule S contains a maximum chain in the induced order $S(P)$ which can be rescheduled in unit blocks, without increasing the makespan. But the remaining question is how to identify this maximum chain, a priori, and whether each maximum chain in $S(P)$ can be rescheduled into unit blocks.

Example 7.1.1 We have shown in Chapter Six that the poset P shown in fig 6.6 has the following properties:

(1) any maximum chain in P can not be scheduled into unit blocks in order for the schedule to be optimal; C_1 must be split as shown in fig 6.7.

(2) any chosen maximum chain in $S(P)$ can be rescheduled into unit blocks in some optimal schedule. Without loss of generality, we may choose for example the maximum chain $C_0 > A_3 > A_2 > A_1 > C_2 > \dots > C_{14}$. It is scheduled into unit blocks in the schedule S shown in fig 6.7.

Example 7.1.2 We present still another example below in fig 7.1.1. The poset P given in fig 7.1.1 has an optimal schedule S , shown in (b). Then, based on the schedule S , we have an order extension $S(P)$ of P , shown in (c). Now, look at the maximum chain $p < j < f < c < a$ in $S(P)$. It is not scheduled into unit blocks in (b), but is rescheduled into unit blocks in (d).



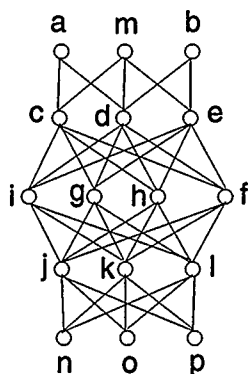
Partially ordered task set P

(a)

n	j	i	f	c	a
o	k	f	g	d	b
p	l	g	h	e	m

An optimal preemptive schedule S of P

(b)

The order extension $S(P)$ of P induced by the schedule S

(c)

p	j	f	i	c	a
o	k	i	g	d	b
n	l	g	h	e	m

A rescheduling of $S(P)$

(d)

fig 7.1.1

§7.2 Rescheduling Maximum Chains in $S(P)$

In order to approach this problem, we provide some necessary definitions below:

Definition 7.2.1 For any specific preemptive schedule S of some poset P ,

(1) A *shift* in S of P is a closed time interval in which there is no change from one task to another on any processor, and not contained in any larger such interval.

(2) $shift(S)$ = set of all shifts in the schedule S of P . For instance, if P is a singleton with one unit-execution-time task a , then for any schedule S of P , $shift(S) = \{[0, 1]\} = \{[\sigma(a), \tau(a)]\}$.

(3) If T is a task in P , $H(T) = \{T^i \text{ is a part of } T : \exists I_j \in shift(S) \text{ such that } [\sigma(T^i), \tau(T^i)] = I_j\}$. For example, let $A \in P$ be a task which appears in n shifts of a schedule S of P . For this scheduling S of P , we have $H(A) = \{A^1, A^2, \dots, A^n\}$ where $[\sigma(A^i), \tau(A^i)] \in shift(S)$.

(4) Let $I = [a, b]$ be a interval in the schedule S of P , then we denote by T_I all tasks which are performed (partially or totally) in the interval $I \subseteq [0, |S|]$. And further, we use function $g_I(a) \in (0, 1]$ to represent the portion of the task $a \in T_I$ completed in the interval I , when the interval I is clearly understood.

We now identify a certain maximum chain, called "critical sequence" in $S(P)$ for some schedule S of poset P :

Algorithm 7.2.1: Let $H(P^*) = m$. Let $P^* = S(P)$ be the interval extension of poset P induced from the schedule S of P . We represent (P^*, \leq^*) by a set of intervals: $\{[\sigma(a), \tau(a)] : a \in P\}$

Step 1. First we label all nodes in P^ by levels and heights (see double-labeling above).*

Step 2. Let $A_1 = \{a \in P : l(a) = m - 1, h(a) = 0\}$

Step 3. Let $c_1 \in A_1$ be the first nodes in the "critical sequence" C_L satisfying $\tau(c_1) \leq \tau(b)$ for all $b \in A_1$.

Step 4. Suppose we have constructed c_1, c_2, \dots, c_k . Let $A_{k+1} = \{a \in P : l(a) = m - (k + 1), h(a) = k, \sigma(a) \geq \tau(a_k)\}$. Choose c_{k+1} from A_{k+1} such that $\tau c_{k+1} \leq \tau b$ for all $b \in A_{k+1}$.

Step 5. Replace k by $k + 1$. Go to Step 4 until we find the last node for the "critical sequence" in P^ .*

Lemma 7.2.1. The "critical sequence" c_1, c_2, \dots, c_m produced by Algorithm 7.2.1 is a maximum chain in P^* .

Proof. First the length of the sequence is the same as the length of any maximum chain in P^* . Second, the sequence is a chain in P^* since $\tau(c_i) \leq \sigma(c_{i+1})$ for $1 \leq i < m$.

□

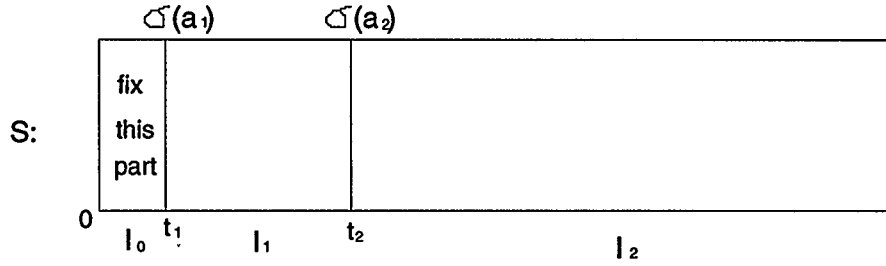
Lemma 7.2.2[26](Extended Wrapping Lemma) Let A be a set of independent tasks of different length, and $|a|$ denotes the length of a task a in A . If $\sum_{a \in A} |a| \geq (\max_{a \in A} (|a|) * m)$, then we can always always schedule all these tasks in A in a block of length $\frac{\sum_{a \in A} |a|}{m}$ by "wrapping" any linear extension of A into this block.

We say that a task $a \in P$ is *complex* with respect to a direct successor a' of a in $S(P)$ for a schedule S of P if for any $b \in P$, $\tau_S(b) \leq \sigma_S(a')$ implies $\sigma_S(b) \geq \sigma_S(a)$.

Lemma 7.2.3[Stone] For any poset P and any preemptive 3-machine schedule S of P , if a_1, a_2, \dots, a_n ($n > 1$), is a maximum chain in $S(P)$ and a_1 is complex respect to a_2 , then there exists a schedule S' such that $|S'| \leq |S|$ and these two schedules

agree on intervals $[0, \sigma_S(a_1)]$ and $[\sigma_S(a_2), |S|]$.

Proof. Let $I_0 = [0, \sigma_S(a_1)]$, $I_1 = [\sigma_S(a_1), \sigma_S(a_2)]$, and $I_2 = [\sigma_S(a_2), |S|]$. Let $W = \{W_i | W_i \text{ is a continuous waste interval } [\sigma(W_i), \tau(W_i)] \subseteq I_1 \text{ on some machine } \}$. Let T_{I_1} denote the tasks performed in I_1 , and recall that function $g_{I_1}(a) \in (0, 1]$ stands for the portion of task a performed in the interval I_1 .



The preemptive schedule S of poset P

fig 7.2.1.

Let $Q = \{a \in T_{I_1} : \tau_S(a) \leq \sigma_S(a_2)\}$, $R = \{a \in T_{I_1} : \tau_S(a) > \sigma_S(a_2)\}$, and $D = \{a \in R : \exists b \in Q \text{ such that } b < a \text{ in } S(P)\}$.

Note that, for any $a \in Q$, $[\sigma_S(a), \tau_S(a)] \subseteq I_1$, i.e. $g_{I_1}(a) = 1$ for all $a \in P$. Since a_1 is complex respect to a_2 , we have either $\sigma_S(a) > \tau_S(a_1)$ or $\tau_S(a) < \sigma_S(a_1)$. But this is contrary to the maximality of chain a_1, \dots, a_n in $S(P)$.

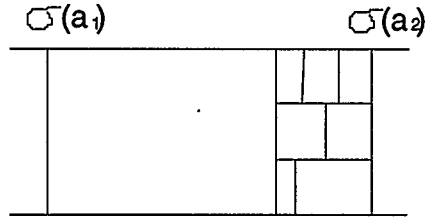
The plan is: first, fix the interval I_0 , and then re-arrange the interval I_1 so that a_1 will be scheduled in unit time without preemption.

We suppose $D \neq \emptyset$. Otherwise $Q \cup R \cup W$ is an antichain, we can easily use the Extended Wrapping Lemma to get S rescheduled so that a_1 is scheduled in unit time. Let $d_1 \in D$ be a task such that $g_{I_1}(d_1) \geq g_{I_1}(d)$ for any $d \in D$.

We first empty the interval I_1 , then refill the interval according to the following plan:

1) pack tasks in D to the very end of the interval which produces a partially completed schedule S' of P :

Case 1. If $\frac{\sum_{d \in D} g_{I_1}(d)}{3} \geq g_{I_1}(d_1)$ then then we wrap all tasks in D into block $[\sigma_S(a_2) - \frac{\sum_{d \in D} g_{I_1}(d)}{3}, \sigma_S(a_2)]$ as in figure 7.2.2. below:



Wrap all tasks in D to the end of the interval I_1

fig 7.2.2

Case 2. If $\frac{\sum_{d \in D} g_{I_1}(d)}{3} < g_{I_1}(d_1)$, then wrap all tasks in D in one of the following forms as shown in figure 7.2.3 below:

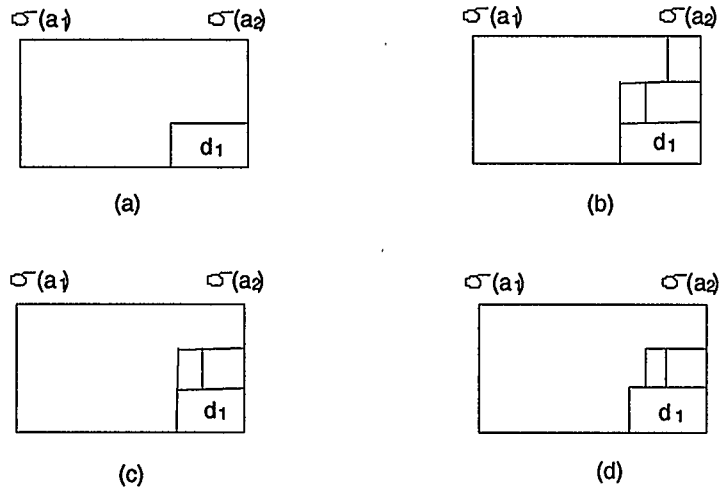


fig 7.2.3

2). Wrap all tasks in $Q \cup (R - D) \cup W$ into the remaining blank space in the interval I_1 . This will produce a complete rescheduling S'' of P .

Claim 1. Tasks in D can always be rescheduled as stated in 1) case 1 and 2.

Since D is an antichain, case 1 can be easily seen by applying the Extended Wrapping Lemma (Lemma 7.2.2); for case 2, we attempt to arrange d_1 into the third machine and fill the second machine as far as we can to make an even edge with the third machine.

Claim 2. Let $t = \sigma_{S'}(b)$ so that $\sigma_{S'}(b) \leq \sigma_{S'}(b')$ for any $b' \in D$. Then $t - \sigma_S(a_1) \geq 1$.

Suppose the form of $S' : D$ is as in figure 7.2.2 and $t - \sigma_S(a_1) < 1$. Let $e \in P$ be such that $\tau_S(e) < \sigma_S(d_1)$, then there is at least one $d \in D$ such that $\sigma_S(d) < t$. This is impossible since there exists some $a \in Q$ such that $\sigma_S(a) > \sigma_S(a_1)$ and $\tau_S(a) < \sigma_S(d)$ which implies $\tau_S(a) - \sigma_S(a) < 1$.

Suppose the form of $S' : D$ is one of the forms shown in fig 7.2.3. Then it is easy again to see that $t - \sigma_S(a_1) \geq 1$. Otherwise the tasks preceding d_1 would not be completed in the interval I_1 .

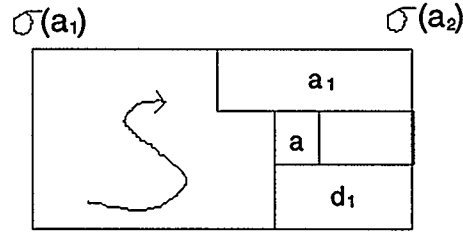
Claim 3. The tasks in $Q \cup (R - D) \cup W$ can be always wrapped into the remaining blank space in the interval I_1 .

Suppose $S' : D$ is as in fig 7.2.2. Since $Q \cup (R_D) \cup W$ is an antichain, by the Extended Wrapping Lemma, they can be wrapped perfectly into the block $[\sigma_S(a_1), t]$ and a_1 can be scheduled in unit time.

Suppose $S' : D$ is as in fig 7.2.3(a) or (b) or (c), then we can also wrap all tasks in $Q \cup (R_D) \cup W$ into the blank space while a_1 is scheduled in unit time by this process by the first machine.

Suppose $S' : D$ is as in fig 7.2.3(d). First pick a task $a \in Q \cup (R_D) \cup W$ such that $a \neq a_1$ and a is long enough to make the right edge of the blank space in the

second and third machine even by rescheduling enough portion of a to the second machine. (If there exists no task longer enough, we can use several tasks to achieve this.) Then we start wrapping from the task a , end with task a_1 (as shown in the fig 7.2.4 below).



Wrapping tasks in $Q \cup (R-D) \cup W$

fig 7.2.4

Now we see $|S''| = |S|$ and a_1 is rescheduled into a unit block in S'' , and S'' , S agree on the interval I_0 . Note, in the wrapping process, we regard each waste interval W_i as a partial task. Actually, a proper wrapping with decreased amount of waste may decrease the makespan of S'' with respect to S . \square

We say that a critical sequence, c_1, \dots, c_n , is a *complex critical sequence* if c_i is complex respect to c_{i+1} for $1 \leq i < n$.

Theorem 7.2.1[Stone & Li]. For any poset P and any scheduling S of P , if $c_1 < c_2 < \dots < c_m$ is a complex critical sequence in $S(P)$, the order extension of P induced by S , then there exists a schedule S^* of P with $|S^*| \leq |S|$ and c_1, c_2, \dots, c_m each scheduled by S^* in unit blocks. Moreover, S^* and S agree on the time interval $[0, \sigma_S(c_1)]$.

Proof. By induction on m .

The basic case. If $m = 1$, by Lemma 7.2.1, c_1 is a maximum chain in $S(P)$. So,

$S(P)$ is an antichain. Fix the scheduling before $\sigma(c_1)$, then reassign the unprocessed tasks in the following way: schedule c_1 on the first machine after $\sigma_S(c_1)$, then wrap the other tasks (together with waste as in Lemma 7.2.3) around by using the Extended Wrapping Lemma (Lemma 7.2.2).

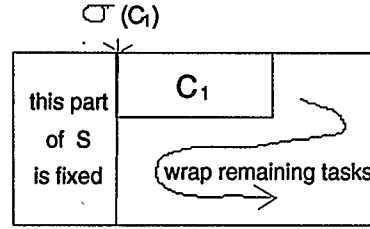


fig 7.2.5

Next, for the inductive step: assume for all $m \leq k$ for some fixed $k \geq 1$, that the statement made by the Theorem is true. We wish to establish the statement for the case $m = K + 1$.

Consider a poset P , a scheduling S of P , and a complex critical sequence $c_1, c_2, \dots, c_k, c_{k+1}$ in $S(P)$.

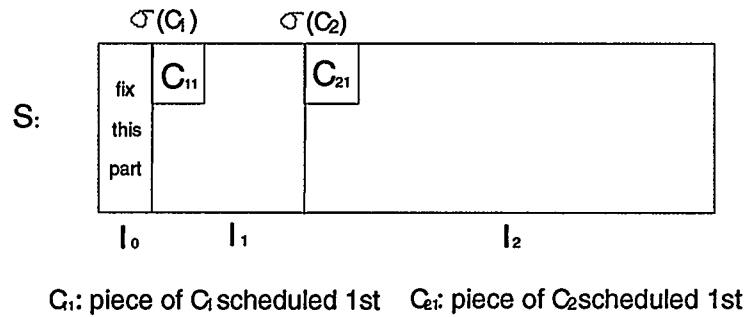


fig 7.2.6

In the Gantt Chart above, let $t_1 = \sigma(c_1)$, $t_2 = \sigma(c_2)$. Consider intervals $I = [0, |S|]$, $I_0 = [0, t_1]$, $I_1 = [t_1, t_2]$, and $I_2 = I - I_0 - I_1$.

First fix the interval I_0 , then reschedule block I_1 so that c_1 is scheduled in a unit time in a new schedule S' of P and $|S'| \leq |S|$ by Lemma 7.2.3. This can be accomplished, since c_1, \dots, c_{k+1} is a maximum chain in $S(P)$ by Lemma 7.2.1, and c_1 is complex respect to c_2 .

Next we apply our inductive assumption to the interval $I_2 = [\sigma_S(c_2), |S|]$. Then T_{I_2} is a poset of height k , and c_2, \dots, c_{k+1} is a complex critical sequence and maximum chain in T_{I_2} . $S' : T_{I_2}$ is a schedule for T_{I_2} . By assumption, we can reschedule $[\sigma_{S'}(c_2), |S'|]$ so that c_2, \dots, c_{k+1} is scheduled into unit blocks. This gives a new schedule S'' on T_{I_2} .

Now, let $S^* = (S : I_0) \oplus (S' : I_1) \oplus (S'' : I_2)$. S^* agrees on I_0 with S and $|S^*| \leq |S|$, and the complex critical sequence c_1, \dots, c_{k+1} is scheduled in unit blocks in S^* . \square

CHAPTER EIGHT

K-Structure Task Sets

In this chapter, we first identify a new interesting poset called K-structure. We then characterize some special properties of this structure and design an algorithm for three machine preemptive scheduling for arbitrary K-structures based on the heuristic of scheduling a critical path in unit blocks. We conjecture that this algorithm produces an optimal schedule for any K-structure. The establishment of the optimality of this algorithm is one direction for further research in this area.

§8.1 K-Structures

Definition 8.1.1 A poset P is *loop-free* if and only if for all sequences $a < b < c$ and $a < d < c$ while a, b, c, d in P , we have either $b > d$ or $b < d$.

A poset P is a *K-structure* if and only if P is connected, loop-free and there exists maximum chain (we call it the *backbone*) $C_P \subseteq P$ such that for any $a \in P$ and any maximal chain C containing a , $C \cap C_P \neq \emptyset$.

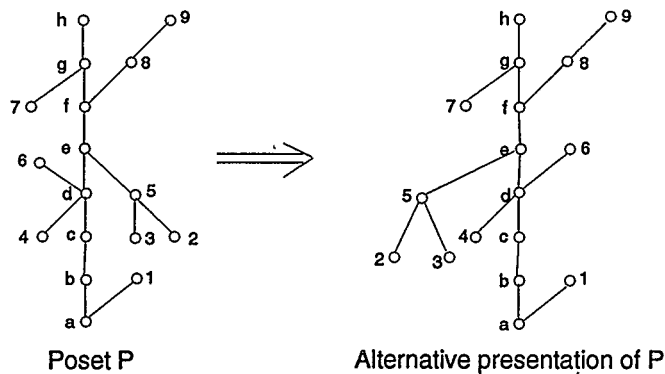


fig 8.1.1

The above graph gives an example of what K-structure posets look like. The maximum chain $a < b < c < \dots < h$ is the "backbone". We can interpret this graph as an industry process control stream in which some small workshop chains provide supplies or parts for main stream production (for example, node "h" in fig 8.1.1 is the end-product of the main stream production), and in some stages, there are by-products (such as node 8 in fig 8.1.1) produced which may be further used to make much more sophisticated by-products (such as node "9" in fig 8.1.1). For this purpose, we can divide all branches in P into two groups: supply-provider (shown as the left-wing in the second graph in fig 8.1.1: 2,3,4,5,7) and by-product-producer (shown as the right-wing in the second graph in fig 8.1: 1, 6, 8, 9), in addition to the main-stream which produces an end-product h.

In the above example (fig 8.1.1), the dominant role of the "backbone", C_P , in the K-structure P is apparent. Some observations below provide a much clearer picture of the nature of K-structures.

Lemma 8.1.1 Let P be a K-Structure. For each $a \in P - C_P$, there exists a partition $P_1 \cup P_2 = C_P$, where P_i is not empty for $i = 1, 2$, so that a is incomparable with P_1 but either less than or greater than all members in P_2 .

Proof: For any $a \in P$, there always exists a partition $P_1 \cup P_2 = C_P$ of C_P with the properties stated in the lemma. Suppose P_1 is empty. This means a is either greater than or less than every element of P_2 . This is contrary to the maximality of C_P . Also, P_2 is always nonempty, since P is connected, and there is always some chain, which contains a , intercepting C_P . \square

Definition 8.1.2 Let P be a poset, $a \in P$. Then $P_a = \{b \in P | b \geq a\}$ is the *order*

ideal above a , and $P^a = \{b \in P \mid b \leq a\}$ is the *order ideal* below a .

Lemma 8.1.2. If P is a K-structure, then for any $a \in P$, P_a and P^a are trees (or dual trees).

Proof. Due to the duality of K-structures, we shall only prove that P_a is a tree. We know that any order ideal P_a with no loop is a tree with root a : Suppose $b \in P_a (b \neq a)$ has two predecessors, say d_1, d_2 . Since all task in P_a is comparable (greater than) a , there exist at least two different paths, say C_1 and C_2 , in P_a , starting from a to b . This is impossible since P_a is loop-free \square

Based on these two observations, we can see that although a K-structure is not a tree or a dual tree, it does have some relation with trees: every tree is a K-structure; and every K-structure is amalgamated from a set of trees or dual trees. By amalgamation, we mean to the union of two or more graphs obtained by identifying certain common subgraphs which are imbedded into each of these graphs.

§8.2 Double-Labeling

To approach the scheduling for K-structures, let us recall Hu's results on scheduling trees or dual trees[13]:

Given m machines, if P is a tree, then non-preemptive level scheduling S of P is optimal.

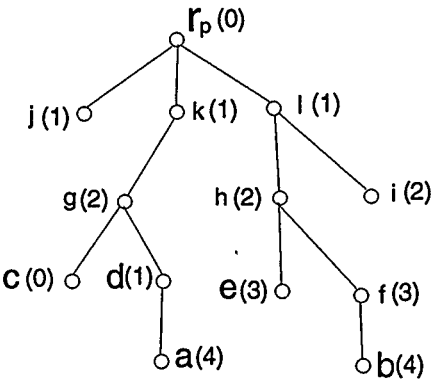
From Chapter One, we recall that level scheduling works like this: first we labeling all tasks in P (suppose P is a dual tree) by *levels* (as defined in Chapter One); then we schedule the dual tree by levels assigned to each task in the following way:

- 1) always schedule tasks with greatest level first

2) if a set of tasks have the same level, and we only need some of them for the scheduling, then we arbitrarily pick as many as we need for the current unit time interval.

Here is an example showing how level scheduling works:

For the dual tree P in fig 8.2.1, we first label the nodes in P by levels (shown in fig 8.2.1 as coordinates of each node).



Single-labeling of the dual tree P

fig 8.2.1

Fig 8.2.2 gives the level scheduling of the poset P in fig 8.2.1:

a	d	g	j	r_p
b	f	h	k	
c	e	i	l	

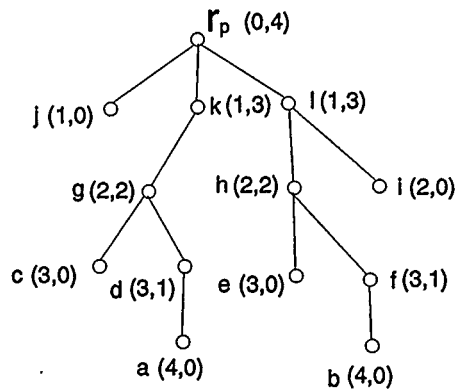
level scheduling of dual tree P

fig 8.2.2

Hu's Theorem establishes that level scheduling will produce optimal schedules for any tree or dual tree. We have shown in Lemma 8.1.1 and 8.1.2 that K-structures have some common properties with trees. We will present an algorithm which appears to produce an optimal preemptive schedule for any K-structure on three machines.

Before we introduce the algorithm, we need some notations and concepts. *Double-labeling* means to assign two-dimensional coordinates to each task in P , where the first coordinate denotes the level of the task, and the second the height.

Fig 8.2.3 double-labels the poset P in fig 8.2.1:



Double labeling of poset P in fig 8.2

fig 8.2.3

§8.3 An Algorithm for K-Structures

The following is a deterministic algorithm which produces a three-machine preemptive schedule for any K-structure. This algorithm will schedule tasks from both ends of the graph P working toward the middle, using double-labeling.

Algorithm 8.3.1

(1). First associate all nodes $a \in P$ with coordinates $(l(a), h(a))$. (double-labelling all nodes by levels and heights).

(2). Let $T_1 = \{a \in P | l(a) = 0, h(a) = H - 1\}$, $B_1 = \{a \in P | h(a) = 0, l(a) = H - 1\}$. Then T_1 and B_1 are both antichains.

(3). Let $T = T_1$.

(4). Scheduling of T :

Case 1. If $|T| \geq 3$, then wrap all tasks in T into a block of length $|T|/3$ with no waste.

Case 2. If $|T| < 3$ then we look at the set $Q = \{a \in P | l(a) = 0\} - T$. If $|Q| \leq 3 - |T|$, then schedule $T \cup Q$ into a unit block. If $|Q| > 3 - |T|$, then we choose a subset $Q' \subset Q$ from Q with $|Q'| = 3 - |T|$ and $h(a) \geq h(b)$ for any $a \in Q', b \in Q - Q'$, i.e., we choose tasks with greatest height first. Then schedule $T \cup Q'$ into a unit block.

(5). Let $B = B_1$.

(6). Scheduling of B .

Case 1. If $|B| \geq 3$ then wrap all tasks in B into a block without waste as in (4) case 1.

Case 2. If $|B| < 3$, then look at the set $Q = \{a \in P | h(a) = 0\} - B$. If $|Q| \leq 3 - |B|$, then schedule $B \cup Q$ into a unit block. If $|Q| > 3 - |B|$, then we choose subset $Q' \subset Q$ from Q with $|Q'| = 3 - |B|$ and $l(a) \geq l(b)$ for any $a \in Q', b \in Q - Q'$, i.e., we choose tasks with greater level first. Then we schedule $B \cup Q'$ into first unit block.

(7) Now delete all tasks from P which have been assigned in stage (4) and (6), then relabel P by levels and heights.

(8) Suppose we have scheduled T_m, B_m and $2m < H$.

Let $T_{m+1} = \{a \in P \mid l(a) = 0, h(a) = H - 2m - 1\}$.

Let $T = T_{m+1}$, do (4).

If all tasks in P have been scheduled, go to (11).

(9). Let $B_{m+1} = \{a \in P \mid h(a) = 0, l(a) = H - 2m - 1\}$.

Let $B = B_{m+1}$, do (6).

If all tasks in P have been scheduled, go to (11).

(10). Replace m by $m + 1$, go to (7).

(11). Terminate the program. Now the schedule is constructed as $B_1 \oplus B_2 \oplus \dots \oplus T_2 \oplus T_1$.

For the K-structure given in figure 8.1.1, the schedule given by Algorithm 8.3.1 appears in fig 8.3.1 below:

a	b	c	d	e	f	g	h
2	4	/	/	/	7	8	9
3	5	/	/	/	/	1	6

fig 8.3.1

Properties of Algorithm 8.3.1:

From the definition of the K-structure, we know that if a poset P is a K-structure then there exists a maximum chain C_P in P such that for any $a \in P$ and any maximal chain C containing a , $C \cap C_P \neq \emptyset$.

Let S be the schedule of P produced by Algorithm 8.3.1. For any $a \in P$, recall that $\sigma(a)$ is used to denote the starting time of a in a schedule S of P ; $\tau(a)$ is the

completion time of a in S . So for any $a \in P$, $\tau(a) - \sigma(a) \geq 1$. Let $|S| = t$.

Observation 1. The backbone C_P is scheduled into unit blocks in S produced by Algorithm 8.3.1, i.e. $\tau(c) - \sigma(c) = 1$ for each $a \in C_P$.

Observation 2. If there exists an idle interval (i.e. waste) $[i, j] \subset [0, t]$ in S , then there must be a unique $c \in C_P$ such that $[i, j] \subseteq [\sigma(c), \tau(c)]$ and $\tau(c) - \sigma(c) = 1$.

Algorithm 8.3.1 appears to produce an optimal schedule S of the K-structure P .

We provide a brief discussion regarding the nature of schedules produced by Algorithm 8.3.1 as follows.

Suppose S is the schedule produced by Algorithm 8.3.1 for K-structure P , where the height of P is n . If there are no waste intervals in S , by Corollary 1.3.1, S is optimal. Suppose B_1 is the first block which contains a waste interval. By Observation 2 above, B_1 must be a unit block in which a task, say c_i , in C_P is scheduled. Suppose B_1 contains the only waste interval in S . Let $c_1 < c_2 < \dots < c_{i-1} < c_i < \dots < c_n$ be the "backbone" (i.e. C_P) of the K-structure P . Let $S(P) = (P^*, \leq^*)$ be the order extension of P induced from schedule S . Let $A = \{a \in P^* | a < c_i\}$ and $B = \{a \in P^* | a > c_i\}$. Obviously, $S : A$ and $S : B$ are both optimal by Corollary 1.3.1., and $\{c_1, \dots, c_{i-1}\} \subset A$ and $\{c_{i+1}, \dots, c_n\} \subset B$. Suppose S' is an optimal schedule of P . Let $\sigma(c_i) = s$ and $\tau(c_i) = t$. If $|S'| < |S|$, then S' contains less waste than S . The only choice is to schedule some portions of tasks in $A \cup B$ together with c_i , and at the same time shorten the intervals $[0, \sigma(c_i)]$ and $[\tau(c_i), |S|]$. But $\sigma(c_i) \geq i-1$ and $|S| - \tau(c_i) \geq n-i$. The only chance to shorten $[0, \sigma(c_i)]$ in S occurs when we wrap more than three tasks into a block, for example a, b, c, d and $c \in C_P$. And at this time any tasks, say a , in this wrapped block will have the same level or

height with some task c in C_P . And by Lemma 8.1.2, P_a and P^a are each a tree or a dual tree. So any shifting of these tasks(if we try to schedule some tasks with c_i) in $[0, \sigma(c_i)]$ may cause the height of the order in $P : [\tau(c_i), |S|]$ of P^* become bigger. Similarly, any shift of these tasks(trying to schedule some tasks with c_i) may cause the height of poset on $[0, \sigma(c_i)]$ prolonged. Hence the length of S' may be prolonged.

§8.4 Directions for Further Research

Each K-structure is a loop-free poset with a special maximum chain(backbone) C_P . We have investigated the relation between K-structures and trees. This relation provides an idea for an algorithm to schedule the K-structures. There is also a common feather imbeded in both loop-free structures and K-structures, that is: given any task $c \in P$ where (P, \leq) is a loop-free or K-structure poset, then $Q = \{a \in P | a \leq c\}$ is a tree(dual tree), and $Q' = \{a \in P | a \geq c\}$ is also a tree(dual tree). This commonality between K-structures and loop-free posets suggests a possible extension of the algorithm to m-machine preemptive scheduling for loop-free posets. The optimality for Algorithm 8.3.1 for K-structures, and an investigation of a similar approach to loop-free posets are directions for further research.

Bibliography

- [1] K. R. Baker(1974), *Introduction to Sequencing and Scheduling*, John Wiley & Sons, Inc.
- [2] Jacek Blazewicz(1987), *Selected Topics in Scheduling Theory*, Annals of Discrete Math. 31 p1-60.
- [3] E.G. Coffman, Jr(1976), *Computer and Job-shop Scheduling Theory*, John Wiley & Sons, Inc.
- [4] E.G. Coffman Jr. and R.L. Graham(1972), *Optimal Scheduling for Two Processor Systems*, Acta Informatica, 1(3), p200-213.
- [5] S.A. Cook(1971), *The Complexity of Theorem-proving Procedures*, Proc. 3rd Annual ACM Symp., Theory of Computing, p151-158.
- [6] R. P. Dilworth(1950), *A Decomposition Theorem for Partially ordered Sets*, Ann. of Math. 51, p161-166.
- [7] D. Duffus, I. Rival and P. Winkler(1982) *Minimizing Setups for Cycle-free Ordered Sets*, Proc. Amer. Math. Soc. 85, p509-513.
- [8] J. Edmond(1965), *Paths, Trees, and Flowers*, Canad. J. Math. 17, p449-467.
- [9] M.L. Fisher(1982), *Worst-case Analysis of Heuristic Algorithms for Scheduling and Packing*, M.A.H. Dempster et al(eds.) *Deterministic and Stochastic Scheduling*, p15-34. D. Reidel Publishing Co.

- [10] M. Fujii, T. Kasami, and K. Ninomiya(1969) *Optimal Sequencing of Two Equivalent Processors*, SIAM Journal on Appl. Math. 17(3), p784-789
- [11] M.R. Garey and D.S. Johnson(1979), **Computers and Intractability: a Guide to the Theory of NP-completeness**, Freeman, San Francisco.
- [12] L.G. Hacıjan(1979), *A polynomial Algorithm in Linear Programming*, Soviet Math. Dokl. 20(1), p191-194.
- [13] T.C. Hu(1961), *Paralle Sequencing and Assembly Line Problems*, Operations Research 9, p841-848.
- [14] J.R. Jackson(1955), *Scheduling a Production Line to Minimize Maximum Tardiness*, Research Report No. 43, Management Sciences Reserach Project, U.C.L.A.
- [15] R.M. Karp(1972), *Reducibility Among Combinatorial Problems*, **Complexity of Computer Computation**, R.E. Miller and J.W. Thatcher(eds.), p85-104. Plenum Press.
- [16] E.L. Lawler(1982), *Preemptive Scheduling of Precedence Constrained Jobs on Parallel Machines*, M.A.H. Dempster et al. (eds.), **Deterministic and Stochastic Scheduling**, p101-123. D. Reidel Publishing Co.
- [17] E.L. Lawler(1973), *Optimal Sequencing of a Single Machine Subject to Precedence Constraints*, Management Science, 19, p544-546.
- [18] E.L. Lawler and J.K. Lenstra(1982), *Machine Scheduling with Precedence Constraints*, I. Rival(ed.), **Ordered Sets**, p655-675.

- [19] R. McNaughton(1959), *Scheduling with Deadlines and Loss Functions*, Management Science, 6(1).
- [20] R.R. Muntz and E.G. Coffman, Jr(1969), *Preemptive Scheduling of Real-time Tasks on Multiprocessor Systems*, Journal of the Association for Computing Machinery, 17(2) , p324-338.
- [21] C.H. Papadimitriou and M. Yannakakis(1979), *Scheduling Interval-ordered Tasks*, SIAM J. Computing, 8(3), p405-409.
- [22] W. Poguntke(1986), *Order Theoretic Aspects of Scheduling*, AMS Contemp. Math. 57, p1-32.
- [23] N.W. Sauer and M.G. Stone(1990), *A Normal Form for Preemptive Three Machine Scheduling*, Proceedings, 2nd International Workshop on Proj. Management and Scheduling, p92-94.
- [24] N.W. Sauer and M.G. Stone(1989) *Preemptive Scheduling*, I. Rival(ed.), *Algorithms and Order*, p307-323, by Kluwer Academic Pub.
- [25] N.W. Sauer and M.G. Stone(1989), *Preemptive Scheduling of Interval Orders is Polynomial*, Order 5, p345-348.
- [26] N.W. Sauer and M.G. Stone(1987), *Rational Preemptive Scheduling*, Order 4, p195-206.
- [27] J.D. Ullman(1975), *NP-complete Scheduling Problems*, J. Comput. Syst. Sci. 10, p384-393.

- [28] J.D. Ullman(1973), *Polynomial Complete Scheduling Problems*, Operating Systems Review, 7(4), p96-101.