MINIMIZING SCHEDULE LENGTH ON IDENTICAL PARALLEL MACHINES: AN EXACT ALGORITHM

A THESIS

Submitted to the Department of Industrial Engineering
and the Institute of Engineering and Science
of Bilkent University
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

By
H. Cemal AKYEL
June, 1991

MINIMIZING SCHEDULE LENGTH ON IDENTICAL PARALLEL MACHINES: AN EXACT ALGORITHM

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSPHY

By
H. Cemal Akyel
June 1991

To my family...

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Ömer S. Benli

Assoc. Prof. Dr. Ömer S. Benli (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Dr. Halim Doğrusöz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Dr. Ali Doğramacı

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Dr. M. Akif Eyler

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Assoc. Prof. Dr. Nesim Erkip

Approved for the Institute of Engineering and Science:

M. Baray

Prof. Dr. Mehmet Baray,
Director of Institute of Engineering and Science

Abstract

MINIMIZING SCHEDULE LENGTH ON IDENTICAL PARALLEL MACHINES: AN EXACT ALGORITHM

H. Cemal Akyel
Ph. D. in Industrial Engineering
Supervisor: Assoc. Prof. Dr. Ömer S. Benli
June 1991

The primary concern of this study is to investigate the combinatorial aspects of the single-stage identical parallel machine scheduling problem and to develop a computationally feasible branch and bound algorithm for its exact solution. Although there is a substantial amount of literature on this problem, most of the work in this area is on the development and performance analysis of approximation algorithms. The few optimizing algorithms proposed in the literature have major drawbacks from the computer implementation point of view. Even though the single-stage scheduling problem is known to be unary $\mathcal{NP}$-hard, there is still a need to develop a computationally feasible optimizing algorithm that solves the problem in a reasonable time. Development of such an algorithm is necessary for solving the multi-stage parallel machine scheduling problems which are currently an almost untouched issue in the deterministic scheduling theory.

In this study, a branch and bound algorithm for the single-stage identical parallel machine scheduling problem is proposed. Promising results were obtained in the empirical analysis of the performance of this algorithm. Furthermore, the procedure that is developed to determine tight bounds at a node of the enumeration tree, is an approximation algorithm that solves a special class of identical parallel machine scheduling problems of practical interest. This algorithm delivers a solution that is arbitrarily close to 4/3 times the optimum. To our knowledge this is the best result obtained for this problem so far.

Keywords: Deterministic Machine Scheduling, Identical Parallel Machines, Minimizing Makespan, Computational Complexity Theory, Approximation Algorithms, Optimizing Algorithms, Performance Bounds.

Özet

ÖZDEŞ PARALEL MAKİNALARDA
ÇİZELGE UZUNLUĞUNUN ENAZLANMASI:
BİR KESİN ÇÖZÜM ALGORİTMASI

H. Cemal Akyel
Endüstri Mühendisliği Doktora
Tez Yöneticisi: Doç. Dr. Ömer S. Benli
Haziran 1991

Tek aşamalı özdeş paralel makinalı çizelgeleme problemlerinin kombinatoryel özelliklerinin incelenmesi ve hesap zamanı açısından uygulanabilir bir dal-sınır yönteminin geliştirilmesi bu çalışmanın ana içeriğini oluşturmaktadır. Çizelgeleme literatüründe bu problemle ilgili pek çok çalışma olmakla beraber, bu çalışmaların çoğu yaklaşık algoritmalar geliştirilmesi ve yaklaşık algoritmaların performans analizi ile ilgilidir. Literatürde önerilen kesin çözüm algoritmaları ise bilgisayar uygulamaları açısından bir takım problemleri içerir. Tek-aşamalı çizelgeleme problemleri için, $\mathcal{NP}$-zor olmalarına rağmen, eniyi çözüm veren, çalışma zamanı açısından uygulanabilir algoritmalara ihtiyaç vardır. Zira böyle bir algoritma çok-aşamalı paralel makinalı çizelgeleme problemlerinin çözümü için gereklidir ki bu son sınıftaki problemler çizelgeleme kuramında hemen hiç dokunulmamış bir alanı belirlerler.

Bu çalışmada, tek aşamalı özdeş paralel makinalı çizelgeleme problemleri için bir dal-sınır algoritması önerilmiştir. Algoritmanın deneysel performans analizinden elde edilen sonuçlar ümit vericidir. Buna ek olarak dal-sınır ağacındaki bir düğümdeki alt ve üst sınırları bulmak için geliştirilen algoritmanın kendi başına uygulanabileceği pratik durumlar da söz konusudur. Bu algoritma eniyi çözüm çarpı 4/3'e istenen ölçüde yakın sonuçlar verebilmektedir. Bu da adı geçen problem için bildiğimiz en iyi sınırdır.

Anahtar Sözcükler: Deterministik Makina Çizelgelemesi, Özdeş Paralel Makinalar, Çizelge Uzunluğunun Enazlanması, Hesap Karmaşıklığı Teorisi, Yaklaşık Algoritmalar, Kesin Çözüm Algoritmaları, Performans Sınırları.

Acknowledgement

I would like to express my gratitude to Professor Ömer S. Benli for his supervision to my graduate study. He introduced me the fascinating world of the parallel machine scheduling theory at the beginning of my Master's study eight years ago. Since then, I have had the utmost pleasure playing with machines, jobs and schedules. Professor Benli helped me through this playground in an enthusiastic and friendly way. His encouraging and patient guidance allowed me to complete this work in contentment. Moreover, I benefited a lot from his wisdom and invaluable advices, which I believe will be useful in the rest of my life.

I am grateful to Professor Nesim Erkip for his support and encouragement in all phases of this study. Chatting and discussing with him always gave me relief in times of discomfort.

I owe a lot to Ceyda "Countess" Oğuz with whom I had many fruitful discussions. She always gave me a hand in times of desperation. Especially, her efforts in typing the draft copy of this thesis rescued me from missing the deadline.

Special thanks are due to my comrade Dr. Erkan Tekman for his understanding and stimulating attention. We spent many sleepless nights discussing issues ranging from the War in the Gulf to problems in dynamic programming. It seems we should talk more on why our destinies are determined by stars.

I wish to thank to Dr. Ülkü Gürler for her help in the statistical analysis part of this study.

Last but not the least, my sincere thanks are due to my family for their continuous morale support.

Contents

List of Figures

List of Tables

List of Procedures

Chapter 1

Introduction

The *theory of deterministic scheduling* is concerned with the development and analysis of mathematical models which are useful in real life. In practice, scheduling problems may arise in a variety of situations. For example, consider a production system in which a set of jobs, each requiring a sequence of operations (*routing*) is to be performed by using a number of machining centers (*stages*). To perform a job, each of its operations must be processed in the order given by the routing. The processing of an operation requires the use of a particular stage for a given duration. Each machine of a stage can process only one operation at a time. Given a criterion to measure the quality of each possible (*feasible*) schedule, the problem is to find a processing order on each machine.

From the practical point of view, it can be argued that most deterministic scheduling models developed in the literature have certain restrictive assumptions. The crucial assumption that is usual in these models is related with the configuration of a production environment. In multi-stage scheduling problems availability of several parallel machines at each stage of production increases the routing flexibility and hence allows the greater possibility of generating "better" schedules. Unfortunately, scheduling models developed for these problems frequently assume the availability of a single machine at each stage of production.

Although the above assumption is unrealistic, it is enough to make most of the scheduling problems $\mathcal{NP}$-hard. A recent study due to [Lawler *et al.* 1989] showed that over 4,536 scheduling problems (*class of problems*) defined in the literature, only 416 were solvable in polynomial-time. 3,817 problems were shown to be $\mathcal{NP}$-hard (3,582 of them were unary $\mathcal{NP}$-hard). The status of 303 was unknown at the time the study was done. Perhaps these complexity results are the primary reason why more general models without this assumption are not well-studied in the literature. Nevertheless, these latter class of problems does exist and requires solutions.

The purpose of this study is to investigate the combinatorial aspects of a single-stage *identical parallel machine* scheduling problem, and develop a computationally feasible branch and bound algorithm for its exact solution. Undoubtedly, such an efficient algorithm is a basic requirement in solving the general class of multi-stage parallel machine scheduling problems.

The following section formally defines the class of problems that is the main concern of this study.

1.1 Problem Definition

Scheduling *independent* and *nonpreemptable* jobs on identical parallel machines so as to minimize schedule length (*makespan*) is one of the fundamental problems in deterministic scheduling theory. In an instance $\mathrm{I}_{\mathsf{pms}} = (\mathcal{J}, \mathcal{M})$ of this problem, we are given

- a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of independent jobs, each job J_j having a processing time $p_j \in \mathcal{Z}^+$, and
- a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of identical parallel machines.

In an instance $\mathrm{I}_{\mathsf{pms}}$, we assume that $m \geq 2$ and $m < n$ since otherwise the problem is trivial (if $m = 1$, the makespan is equal to $\sum_{J_j \in J} p_j$, for any sequence of n jobs and if $m \geq n$, the makespan is equal to the processing time of the longest job).

As described in [Coffman *et al.* 1978], a feasible (nonpreemptive) schedule is a *partition* $\mathcal{P} = \langle \mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m \rangle$ of $\mathcal{J}$ into m disjoint sets, one for each machine. The machine M_i ($i = 1, 2, \ldots, m$) processes jobs in $\mathcal{P}^i$. The total completion time of jobs in $\mathcal{P}^i$ ($i = 1, 2, \ldots, m$) is $l(\mathcal{P}^i)$ where $l(X) = \sum_{J_j \in X} p_j$ for any $X \subseteq \mathcal{P}$ (for an *empty set*, $l(\emptyset) = 0$). Such a completion time is possible since we assume that jobs are independent and thus a machine can process them consecutively without any idle time. The makespan for the schedule $\mathcal{P}$ is then given by

$$C_{max}(\mathcal{P}) = \max_{1 \leq i \leq m} l(\mathcal{P}^i)$$

This abstract problem can be used to model a variety of problems in the real world [Garey and Johnson 1981]. In a television station, machines could stand for commercial breaks and jobs could stand for commercials themselves. Given the duration of each commercial, the aim is to allocate the commercials into breaks such that the durations of breaks are as equal as possible. This objective can equivalently be stated as to minimize the maximum break-length that is, the schedule length. In a computer system, machines could be identical parallel processors and jobs could be independent tasks. Given the processing time of each task, the aim is to distribute the load among the processors as uniformly as possible. In the plumber's pipe-cutting problem, a plumber needs a collection of pipes of lengths $p_1, \ldots, p_n$, which can be obtained by cutting up purchased pipes with a standard length C. The plumber wishes to buy minimum number of these C-length pipes. Given an *upper bound*, ub, and a *lower bound*, lb, on the number of pipes that are to be purchased, the plumber can form a related single-stage identical parallel machine scheduling problem by treating pipes to be cut as jobs and the midpoint of lb and ub as the number of identical parallel machines. If the optimal makespan for the latter problem turns out to be greater than C, then the current number of purchased pipes is not enough to cut the needed pipes and hence lb is updated to the midpoint. If else ub is updated. This way, in a *binary search* the plumber determines the minimum number of purchased pipes. The same strategy can be used to solve several other problems. In a classroom scheduling problem, given the duration of each class and the total availability time, C of each room, the aim is to determine minimum number of rooms required to schedule all of the classes. In a truck loading problem, items with given weights have to be packed into minimum number of trucks each having a finite capacity C.

Although the above interpretations are possible, we will be using the production terminology throughout this study. The problem of determining an optimal m-machine schedule $\mathcal{P}_*$ with minimum $C^*_{max} = C_{max}(\mathcal{P}_*)$ is denoted by $1|P||C_{max}$, using a notation similar to the one in [Graham *et al.* 1979]. In this notation (see Appendix A for details) each scheduling problem is represented by a 4-tuple $\alpha|\beta|\gamma|\delta$, where

- α identifies the production environment, such as single stage (1),
- β identifies the machine environment at each stage of production, such as identical parallel machines (P),
- γ identifies further assumptions of the scheduling problem, such as "job preemption (*pmtn*) allowed", "each job must be completed by its *deadline* $\overline{d_j}$", etc.
- δ identifies the optimality criterion of the scheduling problems, such as makespan (C_{max}).

In Chapter 2, the characteristics of an optimum solution to $1|P||C_{max}$ are analyzed and the previous approaches to solve this problem are reviewed. A discussion on the major drawbacks of these approaches concludes that there is a lack of an exact algorithm which solves $1|P||C_{max}$ in a reasonable time. The main chapter, Chapter 3, presents a detailed development of the branch and bound algorithm, including the computational results. In Chapter 4, the significance and the importance of the results of this study and possible directions for future research are discussed. In Appendix A, the details of the classification scheme for deterministic scheduling problems are presented. Appendix B provides a glossary of basic complexity theoretic concepts used in the study. Finally, the detailed computational results are given in Appendix C in tabular format.

Chapter 2

A Single-Stage Identical Parallel Machine Scheduling Problem

It is not difficult to be deceived by the apparent simplicity of the class of scheduling problems $1|P||C_{max}$, thus underestimating the complexity of the problem. The scheduling of n independent jobs on m identical parallel machines is among the hardest problems in the scheduling theory. A few number of optimizing efforts, each failing to solve the problem, reported in the literature can be considered as an indicator of this fact. However some of the characteristics of the problem makes the development of "good" algorithms possible. In Section 2.1, these characteristics will be discussed and the complexity class of the problem will be identified. Section 2.2 reviews the optimizing algorithms developed to solve $1|P||C_{max}$ and discusses their major drawbacks. In Section 2.3, we will briefly review one of the promising research areas of the deterministic scheduling theory, development of approximation algorithms for $1|P||C_{max}$ with "good" performance guarantees.

2.1 Problem Characteristics

An analysis of problem characteristics provides an insight which may be useful in developing efficient solution procedures. For $1|P||C_{max}$, even the trivial property stated in the following lemma leads to a substantial reduction in the computational burden of the proposed implicit enumeration algorithm (see Chapter 3).

Lemma 2.1 *There exists an optimal schedule for* $1|P||C_{max}$ *in which at least one job is assigned on machine* M_i $\forall\ i = 1, 2, \ldots, m$.

Proof: Suppose the contrary, that is, there is an optimal schedule $\mathcal{P}_*$ in which no job is assigned on machine M_{i_1} for some $i_1 = 1, 2, \ldots, m$. Without loss of generality, assume that $C^*_{max} = C_{max}(\mathcal{P}_*) = l(\mathcal{P}^{i_2}_*)$ for some $i_2 = 1, 2, \ldots, m$, is the makespan of this schedule. By scheduling job $J_j \in \mathcal{P}^{i_2}_*$ on machine M_{i_1} we obtain another schedule $\mathcal{P}_{**}$ in which $l(\mathcal{P}^{i_1}_{**}) \leq l(\mathcal{P}^{i_2}_*)$. Furthermore, $\forall\ \ell \neq i_1$, $l(\mathcal{P}^{\ell}_{**}) \leq l(\mathcal{P}^{i_2}_*) = C_{max}(\mathcal{P}_*)$. Thus we obtain a contradiction that $C_{max}(\mathcal{P}_{**}) \leq C^*_{max}$. ■

The following lemma provides a *necessary* condition that an optimal schedule for $1|P||C_{max}$ satisfies.

Lemma 2.2 *In an optimal schedule to* $1|P||C_{max}$*, the earliest start time of job* J_j *on machine* M_i, $EST_{ji} < \lambda = (1/m)l(\mathcal{J})$ $\forall\ j = 1, 2, \ldots, n$ *and* $i = 1, 2 \ldots, m$.

Proof: Suppose for a contradiction that there is an optimal schedule in which $EST_{j_1 i_1} \geq \lambda$ for some job J_{j_1} ($j_1 = 1, 2, \ldots, n$) and some machine M_{i_1} ($i_1 = 1, 2, \ldots, m$). Then there exists at least one other machine M_{i_2} ($i_2 \neq i_1$) with $EST_{j_1 i_2} < EST_{j_1 i_1}$ since $\sum_{\ell=1}^{m} EST_{j_1 \ell} < l(\mathcal{J}) = m\lambda$. Thus by scheduling job J_{j_1} on machine M_{i_2} we obtain a contradiction. ■

The well-known $\mathcal{NP}$-hardness result due to [Garey and Johnson 1979] rules out the possibility of finding necessary and *sufficient* conditions which can be used to determine an optimal schedule for $1|P||C_{max}$ in *polynomial-time* (since otherwise $\mathcal{P} = \mathcal{NP}$, which is extremely unlikely). In the remainder of this section, the above complexity result will be discussed in detail. In this discussion (and in the remainder of this study) we will use the relationship between the single-stage identical parallel machine scheduling problem and the *bin packing* problem.

In an instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{bp}} = (\mathcal{J}, m)$ of the bin packing problem, $\Pi^{\mathrm{d}}_{\mathrm{bp}}$, we are given

- a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of jobs (or, pieces) of size p_j for each $J_j \in \mathcal{J}$, and
- a set $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$ of bins with unit bin sizes.

The aim is to *decide* whether or not there exists a *feasible packing*. A feasible packing can be considered as a feasible schedule $\mathcal{P}$ which finishes at (or, before) time one (that is, the size of each bin). Such a packing may exist since we assume that in a nontrivial instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{bp}}$, $p_j \leq 1 \;\; \forall \;\; J_j \in \mathcal{J}$. Otherwise, the answer for the above decision problem is always "no feasible packing" since at least one job cannot fit into a bin.

$1|P||C_{max}$ can now be viewed as determining minimum bin size τ^*, for which a feasible packing exists. Suppose that we have a procedure *exactbinpack* which given an instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{bp}}$ either outputs "no feasible packing" or determines a feasible packing (that is, the procedure outputs "yes") for the decision problem $\Pi^{\mathrm{d}}_{\mathrm{bp}}$. Procedure 2.1 uses the procedure *exactbinpack* in a binary search to determine a minimum makespan schedule for the problem $1|P||C_{max}$. It is initialized with lower (*lb*) and upper (*ub*) bounds on the optimum schedule length, C^*_{max}. For the midpoint τ of the current range of possible optimum makespan values, related bin packing problem is exactly as defined before, except that the bin sizes $\tau \geq 1$. Therefore, the binary search procedure determines the scaled instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{bp}}/\tau$ after dividing job and bin sizes by τ. Clearly such a scaling does not affect the essential structure of the original instance. The procedure *exactbinpack* is called for the scaled instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{bp}}/\tau$; if a feasible packing (schedule) is determined, then the upper bound is updated to the midpoint, otherwise the lower bound is updated.

Procedure 2.1: An exact binary search procedure for $1|P||C_{max}$

arguments
$\mathrm{I}_{\mathsf{pms}}$: an instance of $1|P||C_{max}$
lb: a lower bound for $1|P||C_{max}$
ub: an upper bound for $1|P||C_{max}$

procedure called
exactbinpack: a hypothetical procedure that solves the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$

```
procedure  exactbinarysearch ($\mathrm{I}_{\mathsf{pms}}, lb, ub$)
begin
    while $ub - lb \geq 1$
        begin
            $\tau := (ub + lb)/2$
            if  exactbinpack ($\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}/\tau$) outputs "no feasible packing"
              $lb := \tau$
            else
              $ub := \tau$
        end
    output  $\lfloor ub \rfloor$ and the schedule related with $ub$
end
```

Thus at each packing attempt ("iteration") lb and ub are still lower and upper bounds on the problem, respectively. Moreover, $\lceil lb \rceil$ and $\lfloor ub \rfloor$ are also respective lower and upper bounds since all of the processing times are integer. Hence the search can be terminated whenever $ub - lb < 1$ with $C^*_{max} = \lfloor ub \rfloor$ since $\lfloor ub \rfloor \leq \lceil lb \rceil$ is a lower bound

This search interval is called *the interval of uncertainty.* After ℓ iterations of a binary search, the size of this interval is reduced by a factor of $2^{-\ell}$. Thus $\mathcal{O}$ (log $\lceil ub - lb \rceil$) iterations are required to satisfy the stopping condition of Procedure 2.1, $ub - lb < 1$. The initial lower and upper bounds are described in the following lemmas.

Lemma 2.3 [McNaughton 1959] *A lower bound on* $1|P||C_{max}$ *is*

$$\max\{l(\mathcal{J})/m, \max_{J_j \in \mathcal{J}} p_j\}$$

Proof: C^*_{max} is at least $\max_{J_j \in \mathcal{J}} p_j$ since each job has to be processed in any schedule for $1|P||C_{max}$. Furthermore the inequality $l(\mathcal{J}) \leq mC^*_{max}$ must hold since otherwise there is no way to finish all of the jobs by the time C^*_{max}. ■

The above lower bound is also an optimal makespan value for $1|P|pmtn|C_{max}$.

Lemma 2.4 [Graham 1966] *An upper bound on* $1|P||C_{max}$ *is* $2lb$.

Proof: Due to Lemma 2.2, in an optimal schedule, a job starts its processing on a machine before $(1/m)l(\mathcal{J}) \leq lb$. Hence $C^*_{max} - lb \leq \max_{J_j \in \mathcal{J}} p_j \leq lb$. Thus $C^*_{max} \leq 2lb$. ■

As it will be discussed in Section 2.3, it is possible to improve these bounds. Nevertheless, it is sufficient to show that Procedure 2.1 is polynomial in the binary encoding of the input provided that the procedure *exactbinpack* is polynomial. Unfortunately, the latter possibility is extremely unlikely since the bin packing problem $\Pi^{\mathrm{d}}_{\mathrm{bp}}$ is unary $\mathcal{NP}$-complete as shown in the following theorem.

Theorem 2.1 [Garey and Johnson 1979] *The bin packing problem* $\Pi^{\mathrm{d}}_{\mathrm{bp}}$ *is unary* $\mathcal{NP}$*-complete.*

Proof: Without loss of generality, we assume that in the bin packing problem the size of each bin is $\tau \geq 1$. Consider the following unary $\mathcal{NP}$-complete recognition problem [Garey and Johnson 1979]:

3-PARTITION:

INSTANCE: A finite set $\mathcal{J}$ of $3m$ elements, a bound $\tau \in \mathcal{Z}^+$, and a "size" $p_j \in \mathcal{Z}^+$ for each $J_j \in \mathcal{J}$, such that each p_j satisfies $\tau/4 < p_j < \tau/2$ and such that $\sum_{J_j \in \mathcal{J}} p_j = m\tau$.

QUESTION: Can $\mathcal{J}$ be partitioned into m disjoint sets $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$ such that, for $1 \leq i \leq m$, $l(\mathcal{P}^i) = \tau$?

An instance of the 3-PARTITION is a special case for the bin packing problem in which piece sizes and the number of pieces are so restricted that in a feasible packing each bin must contain exactly three pieces. Hence, we can directly use the procedure *exactbinpack* to solve an instance of 3-PARTITION (see Procedure 2.2). Clearly if *exactbinpack* were a (pseudo) polynomial-time procedure then 3-

Procedure 2.2: A solution procedure for 3-PARTITION

argument
I: an instance of 3-PARTITION

procedure called
exactbinpack: a hypothetical procedure that solves the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$

```
procedure  $3 - partition\ (I)$
begin
      call $exactbinpack\,(\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}/\tau)$
      if $l(\mathcal{P}^i) < \tau\ \ \forall\ i = 1, 2, \ldots, m$ or
        "no feasible packing" is output then
        output  "no"
      else
        output  "yes"
end
```

PARTITION would be solved in (pseudo) polynomial time which is not possible unless $\mathcal{P}$=$\mathcal{NP}$. Therefore the bin packing problem is unary $\mathcal{NP}$-complete. ■

Theorem 2.2 [Garey and Johnson 1979] *The general problem class* $1|P||C_{max}$, *is unary* $\mathcal{NP}$*-hard.*

Proof: Without loss of generality, we assume that in the bin packing problem the size of each bin is $\tau \geq 1$. Suppose that we have a hypothetical subroutine, 1*pcmax*, which solves $1|P||C_{max}$ to optimality in (pseudo) polynomial-time. Using 1*pcmax* we can solve the bin packing problem Π^{d}_{bp} as shown in Procedure 2.3.

Procedure 2.3: A solution procedure for the bin packing problem Π^{d}_{bp}

argument
I^{d}_{bp}: an instance of the bin packing problem, Π^{d}_{bp}

procedure called
1*pcmax*: a hypothetical subroutine that solves $1|P||C_{max}$

```
procedure  exactbinpack ($I^{d}_{bp}$)
begin
    call 1pcmax ($I_{pms}$)
    if $C_{max} > 1$ then
      output  "no feasible packing"
    else
      output  "yes" and the packing (schedule)
end
```

This shows bin packing problem is Turing-reducible to $1|P||C_{max}$. Hence the latter problem is unary $\mathcal{NP}$-hard since the former is unary $\mathcal{NP}$-hard due to Theorem 2.1. ■

Above complexity result provides a formal justification to use an implicit enumeration algorithm to determine an optimal solution for the problem.

2.2 Optimizing Algorithms

A quick review of the scheduling literature suggests that there is a lack of efficient optimizing algorithms for $1|P||C_{max}$ (See for example [Lawler *et al.* 1982; Lawler *et al.* 1989; Cheng and Sin 1990]). In the following sections we will analyze the algorithms based on the two approaches: Branch and Bound and Dynamic Programming.

2.2.1 A Branch and Bound Algorithm

In [Lawler *et al.* 1982] the only branch and bound algorithm for $1|P||C_{max}$ is reported as the algorithm due to [Bratley *et al.* 1975]. However this algorithm is developed to solve $1|P|r_j, \overline{d_j}|C_{max}$, a general case of $1|P||C_{max}$. In this section we argue that the above algorithm, when applied to the special case fails to provide an efficient solution procedure.

Consider a related scheduling problem $1|P|r_j, \overline{d_j}|C_{max}$. In an instance of this problem we are given a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of independent jobs and a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of identical machines ($m < n$). Each job J_j ($j = 1, 2, \ldots, n$) becomes available for processing at a ready time $r_j \geq 0$; has a processing time $p_j \in \mathcal{Z}^+$; and must be completed by a deadline $\overline{d_j}$ (in a nontrivial instance $\overline{d_j} \geq r_j + p_j$). The aim is to determine a feasible schedule (if any) that minimizes makespan. In this problem, if we let $r_j = 0$ and $\overline{d_j} = ub \;\; \forall \;\; j = 1, 2, \ldots, n$, where ub is an upper bound on the optimal schedule length, the problem reduces to $1|P||C_{max}$. The implication of this observation is twofold. First, $1|P|r_j, \overline{d_j}|C_{max}$ is unary $\mathcal{NP}$-hard since its *restricted* version (i.e. $1|P||C_{max}$) is unary $\mathcal{NP}$-hard (see Theorem 2.2). Second, an optimizing algorithm for $1|P|r_j, \overline{d_j}|C_{max}$ can be used to solve $1|P||C_{max}$.

In $1|P|r_j, \overline{d_j}|C_{max}$, a feasible schedule can be considered as a partition $\mathcal{P} = \langle\, \mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m \,\rangle$ of $\mathcal{J}$ into m disjoint sets, one for each machine M_i ($i = 1, 2 \ldots, m$), such that the completion time of a job $J_j \in \mathcal{P}^i$, $C_j(\mathcal{P}^i) \leq \overline{d_j}$. In other words, a feasible schedule corresponds to an assignment of jobs to machines such that (i) no job is processed on more than one machine, and (ii) the jobs assigned on a machine must be completed before their deadlines. It is the second property that makes the sequencing of jobs on a machine necessary. For instance, consider jobs J_{j_1} and J_{j_2} that are somehow assigned on a machine M_i with $r_{j_1} = 2$, $r_{j_2} = 3$, $p_{j_1} = 1$, $p_{j_2} = 2$, $\overline{d_{j_1}} = 6$ and $\overline{d_{j_2}} = 5$. In a feasible schedule, the sequence of these jobs has to be $J_{j_2} - J_{j_1}$ ("J_{j_2} is processed before J_{j_1}"). The problem of determining a feasible sequence of jobs on a machine (i.e. a sequence that results in a feasible schedule) is denoted by $1||r_j, \overline{d_j}|C_{max}$, using the notation in Appendix A. In this feasibility problem, although there is no need to specify an objective, we arbitrarily choose C_{max} in order to be consistent with the overall objective in $1|P|r_j, \overline{d_j}|C_{max}$. The sequencing problem $1||r_j, \overline{d_j}|C_{max}$ is unary $\mathcal{NP}$-hard (complete) due to the following proposition.

Proposition 2.1 *The sequencing problem $1||r_j, \overline{d_j}|C_{max}$ is unary $\mathcal{NP}$-hard.*

Proof: Suppose for a contradiction that there exists an (optimizing) algorithm *1rjdjcmax* which solves $1||r_j, \overline{d_j}|C_{max}$ in polynomial-time. In an instance I of this problem we are given a set $\mathcal{J}$ of n jobs. Each $J_j \in \mathcal{J}$ becomes available for processing at a ready time $r_j \geq 0$; has a processing time $p_j \in \mathcal{Z}^+$; and must be completed by a deadline $\overline{d_j}$. Consider another sequencing problem $1||r_j, d_j|L_{max}$ which is known to be unary $\mathcal{NP}$-hard [Lenstra 1977; Rinnooy Kan 1976]. An instance I' of this latter problem is same as I, except that a job $J_j \in \mathcal{J}$ may not be completed by its *due date* d_j. The objective is to minimize the maximum of the differences between the job completion times and their due dates (i.e. L_{max}). Let lb and ub denote lower and upper bounds on the minimum L_{max}. Clearly $lb = -d_{max} = -\max_{J_j \in J} d_j$ (none of the jobs are scheduled) and $ub = r_{max} + \sum_{J_j \in \mathcal{J}} p_j = \max_{J_j \in \mathcal{J}} r_j + \sum_{J_j \in \mathcal{J}} p_j$ (all of the jobs are scheduled after maximum ready time r_{max} and $d_j = 0 \;\; \forall \; J_j \in \mathcal{J}$) are the simple bounds on this problem. Procedure 2.4 uses the procedure *1rjdjcmax* in a binary search to determine a schedule which minimizes L_{max} for the problem $1||r_j, d_j|L_{max}$.

Procedure 2.4: An exact binary search procedure for solving $1||r_j, d_j|L_{max}$

arguments
I': an instance of $1||r_j, d_j|L_{max}$
lb: a lower bound for $1||r_j, d_j|L_{max}$
ub: an upper bound for $1||r_j, d_j|L_{max}$

procedure called
1rjdjcmax: a hypothetical procedure that solves $1||r_j, \overline{d_j}|C_{max}$

```
procedure  $1rjdjlmax(I', lb, ub)$
begin
      while $ub - lb \geq 1$
            begin
                  $Lmax := (ub + lb)/2$
                  $\overline{d_j} := d_j + Lmax \ \forall \ J_j \in J$
                  if  $1rjdjcmax\ (I)$ outputs no feasible schedule
                     $lb := Lmax$
                  else
                     $ub := Lmax$
            end
      output  $\lfloor ub \rfloor$ and the schedule related with $ub$
end
```

Procedure 2.4 determines L^*_{max} in $\mathcal{O}\ (ub - lb) = \mathcal{O}\ (\sum_{J_j \in J} p_j + r_{max} + d_{max})$ calls of the procedure *1rjdjcmax*. Therefore, it would be polynomial in the binary encoding of the input provided that the procedure *1rjdjcmax* were a polynomial-time subroutine for $1||r_j, \overline{d_j}|C_{max}$. Thus $1||r_j, d_j|L_{max}$ is Turing reducible to $1||r_j, \overline{d_j}|C_{max}$. Thus the latter problem is unary $\mathcal{NP}$-hard since the former is known to be unary $\mathcal{NP}$-hard. ■

The above complexity result justifies the enumeration of all possible arrangements of n jobs on m machines. The number of arrangements of n jobs on exactly m

machines is

$$\binom{n-1}{m-1}\frac{n!}{m!} \tag{2.1}$$

This number can be verified as follows: suppose we have fixed a job on a machine (say J_1 on M_1). Then there are $\binom{n-1}{m-1}$ different ways to assign the remaining $n-1$ jobs (selected from a given list) to m machines. Since there are $(n-1)!$ different ways of forming a list of remaining jobs, we obtain $\binom{n-1}{m-1}(n-1)!$ distinct schedules provided that the job J_1 is fixed on machine M_1. Each time fixing one job J_j $(j = 2, \ldots, n)$ on machine M_1 we obtain $\binom{n-1}{m-1}(n-1)!$ distinct schedules provided that the job J_j is fixed on machine M_1. Since there are $m!$ different ways to arrange m identical machines, the number of distinct schedules that uses exactly m machines will be (2.1).

To the best of our knowledge the only (optimizing) algorithm for $1|P|r_j, \overline{d_j}|C_{max}$ is due to [Bratley *et al.* 1975]. In their branch and bound algorithm, leaf nodes of the enumeration tree correspond to all possible arrangements of n jobs on m or less machines. Using (2.1), it is easy to see that the total number of leaf nodes is

$$n!\sum_{i=1}^{m}\binom{n-1}{i-1}\Big/ i!$$

Table 2.1 shows how this number changes as n and m changes.

As opposed to the situation in $1|P|r_j, \overline{d_j}|C_{max}$, the ordering of the jobs that are assigned on the same machine is immaterial in $1|P||C_{max}$. Therefore, the set of schedules that the branch and bound algorithm of [Bratley *et al.* 1975] enumerates is much larger than what is required for $1|P||C_{max}$. Moreover, the computational results presented in [Bratley *et al.* 1975] suggest that the performance of the algorithm is poor even for $1|P|r_j, \overline{d_j}|C_{max}$. The problem instances that were generated in their empirical analysis are small and easy problems. In the largest problem instance generated, $n = 25$ and $m = 3$. When the processing time variability increases slightly, the branch and bound algorithm fails to solve 40% of the generated problems to optimality. This failure is partly due to the redundant

Table 2.1: Number of leaf nodes in the enumeration tree proposed by [Bratley *et al.* 1975]

		m			
		1	2	3	4
n	1	1	–	–	–
	2	2	3	–	–
	3	6	12	13	–
	4	24	60	72	73
	5	120	360	480	500
	6	720	2520	3720	4020
	7	5040	20160	32760	36960
	8	40320	181440	322560	381360
	9	362880	1814400	3507840	4354560
	10	3628800	19958400	41731200	54432000
	11	39916800	239500800	538876800	738460800
	12	479001600	3113510400	7504358400	10797494400
	13	6227020800	43589145600	112086374400	169167398400
	14	87178291200	653837184000	1787154969600	2826029606400
	15	1307674368000	10461394944000	30294456192000	50127517440000

		m			
		5	6	7	8
n	1	–	–	–	–
	2	–	–	–	–
	3	–	–	–	–
	4	–	–	–	–
	5	501	–	–	–
	6	4050	4051	–	–
	7	37590	37632	37633	–
	8	393120	394296	394352	394353
	9	4566240	4594464	4596480	4596552
	10	58242240	58877280	58937760	58941000
	11	808315200	822286080	823949280	824068080
	12	12114748800	12422108160	12466016640	12469937040
	13	194853859200	201703582080	202845202560	202967519040
	14	3345466924800	3501298120320	3530980252800	3534690519360
	15	61035701126400	64671762355200	65450918332800	65562226329600

enumeration of schedules in which no job is assigned on at least one machine. It is trivial to show that in $1|P|r_j, \overline{d_j}|C_{max}$, there exists an optimal schedule (if there exists a feasible solution) in which at least one job is scheduled on every machine M_i $(i = 1, 2, \ldots, m)$. Furthermore the branch and bound algorithm does not use an effective bounding scheme to cut the size of the tree.

2.2.2 Dynamic Programming Approaches

In the literature there are several dynamic programming formulations for the general problem $1|P||C_{max}$ or its restricted version in which the number of jobs with distinct processing times is fixed. Dynamic programming formulations that solve the general $1|P||C_{max}$ problem are due to [Rothkopf 1966; Lawler and Moore 1969; Blazewicz 1987]. First two of these formulations, [Rothkopf 1966; Lawler and Moore 1969], were not originally developed to solve $1|P||C_{max}$ but for more general class of scheduling problems. The last formulation, [Blazewicz 1987], was developed specifically for the $1|P||C_{max}$. In Section 2.2.2.1, these formulations will be discussed. The algorithm due to [Sahni 1976] solves $1|P||C_{max}$ by using the above formulations in a straightforward manner and will not be discussed in this section. In Section 2.2.2.2, we will present the work due to [Hochbaum and Shmoys 1988], a polynomial-time dynamic programming formulation for the special case of $1|P||C_{max}$. This formulation plays an important role in developing a polynomial ε-approximation scheme for the problem (see Section 2.3.2).

2.2.2.1 Dynamic Programming Formulations for the General Case

The formulations due to [Rothkopf 1966; Lawler and Moore 1969] are quite general providing a dynamic programming technique for a variety of scheduling problems including $1|P||C_{max}$ as a special case. As mentioned in [Rothkopf 1966; Lawler *et al.* 1989], the technique is applicable to any parallel machine scheduling problem (where the machines may not be identical and the objective may be of the general form $\sum f_j$ or f_{max}) if the following condition is satisfied: it is possible to index the jobs in such a way that the jobs assigned on a *given* machine can be assumed to be processed in the order of their indices. In $1|P||C_{max}$ this assumption is certainly valid for any indexing of jobs since order of the jobs scheduled on a machine is immaterial. However, the technique when applied for a particular problem (in our case $1|P||C_{max}$) has some computer implementation problems that makes it impractical (such as large space requirements, average run-time being close to the time-complexity of an algorithm). In what follows we first state the formulations developed for $1|P||C_{max}$ by using this technique.

Dynamic programming formulation for $1|P||C_{max}$ [Rothkopf 1966] Let $F_j(t_1, t_2, \ldots, t_m)$ denote the minimum makespan for the scheduling problem in which jobs $J_1, \ldots, J_j$ are to be scheduled nonpreemptively on m machines without an idle time such that none of these jobs start on a machine M_i $(i = 1, 2, \ldots, m)$ before the time t_i. Then the recursive equation becomes:

$$F_j(t_1, t_2, \ldots, t_m) = \min_{1 \leq i \leq m} \{\max\{t_i + p_j, F_{j-1}(t_1, \ldots, t_i + p_j, \ldots, t_m)\}\}$$

where $F_0(t_1, t_2, \ldots, t_m) = 0$.

$F_j(t_1, t_2, \ldots, t_m)$ is computed for $j = 0, 1, \ldots, n$; $t_i = 0, 1, \ldots, U$; $i = 1, 2, \ldots, m$, where U is an upper bound on the minimum makespan. The problem is solved by the calculation of $F_n(0, 0, \ldots, 0)$.

Dynamic programming formulation for $1|P||C_{max}$ [Lawler and Moore 1969] Let $F_j(t_1, t_2, \ldots, t_m)$ denote the minimum makespan for the scheduling problem in which jobs $J_1, \ldots, J_j$ are to be scheduled nonpreemptively on m machines without an idle time such that no job is completed later than time t_i on machine M_i $(i = 1, 2, \ldots, m)$. Then the recursive equation becomes:

$$F_j(t_1, t_2, \ldots, t_m) = \min_{1 \leq i \leq m} \{\max\{t_i, F_{j-1}(t_1, \ldots, t_i - p_j, \ldots, t_m)\}\}$$

where

$$F_0(t_1, t_2, \ldots, t_m) = \begin{cases} 0 & \text{if } t_i = 0 \text{ for } i = 1, 2, \ldots, m \\ +\infty & \text{otherwise} \end{cases}$$

$F_j(t_1, t_2, \ldots, t_m)$ is computed for $j = 0, 1, \ldots, n$; $t_i = 0, 1, \ldots, U$; $i = 1, 2, \ldots, m$. The problem is solved by the calculation of $F_n(U, U, \ldots, U)$.

Although there is a slight difference between the above formulations, the time required to solve the related recursive equations is the same. In each of these formulations, we have $\mathcal{O}\ (U^m)$ different states in a stage (simply each t_i for some M_i may take a value between 0 and U). For each state we spend $\mathcal{O}\ (m)$ time to compute the outer minimization in $F_j(t_1, t_2, \ldots, t_m)$. Then for n stages the total effort is bounded by $\mathcal{O}\ (mnU^m)$.

Dynamic programming formulation for $1|P||C_{max}$ [Blazewicz 1987] In this formulation the computational effort is reduced. Let

$$F_j(t_1, t_2, \ldots, t_m) = \begin{cases} \mathit{true} & \text{if jobs } J_1, \ldots, J_j \text{ can be scheduled} \\ & \text{on machines } M_1, M_2, \ldots, M_m \\ & \text{in such a way that } M_i \text{ is busy} \\ & \text{in time interval } [0, t_i]\ i = 1, 2, \ldots, m \\ \mathit{false} & \text{otherwise} \end{cases}$$

where

$$F_0(t_1, t_2, \ldots, t_m) = \begin{cases} \mathit{true} & \text{if } t_i = 0 \text{ for } i = 1, 2, \ldots, m \\ \mathit{false} & \text{otherwise} \end{cases}$$

Then the recursive relation is

$$F_j(t_1, t_2, \ldots, t_m) = \bigvee_{i=1}^{m} F_{j-1}(t_1, \ldots, t_i - p_j, \ldots, t_m)$$

After $F_j(t_1, t_2, \ldots, t_m)$ is computed for $j = 0, 1, \ldots, n$; $t_i = 0, 1, \ldots, U$; $i = 1, 2, \ldots, m$, the minimum makespan is determined as

$$C^*_{max} = \min\{\max\{t_1, t_2, \ldots, t_m\} \mid F_n(t_1, t_2, \ldots, t_m) = \mathit{true}\}$$

Time complexity of the above procedure is $\mathcal{O}\ (nU^m)$ (effort spent in determining the outer minimum of the recursive equations in the previous formulations is eliminated since the value of $F_j(t_1, t_2, \ldots, t_m)$ can immediately be determined as *true* whenever $F_{j-1}(t_1, t_2, \ldots, t_i - p_j, \ldots, t_m)$ turns out to be *true* at the previous stage).

Observing that only $m - 1$ of the values $t_1, t_2, \ldots, t_m$ in the equations $F_j(t_1, t_2, \ldots, t_m)$ of the above formulations are independent (i.e. once a schedule is determined for $m-1$ machines, the schedule on the m-th machine can be determined by simply scheduling the remaining jobs on this machine), we improve the time bound to $\mathcal{O}\ (mnU^{m-1})$ (or to $\mathcal{O}\ (nU^{m-1})$ in the formulation of [Blazewicz 1987]). This bound can further be reduced to $\mathcal{O}\ (mn2^nU)$. In $1|P||C_{max}$, schedules $\mathcal{P}' = \langle\ \mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^t, \ldots, \mathcal{P}^\ell, \ldots, \mathcal{P}^m\ \rangle$ and $\mathcal{P}'' = \langle\ \mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^\ell, \ldots, \mathcal{P}^t, \ldots, \mathcal{P}^m\ \rangle$ $(t \neq \ell; t = 1, 2, \ldots, m; \ell = 1, 2, \ldots, m)$ have the same makespan value since the machines are identical. This type of schedules are referred as *symmetric* schedules. Consider the dynamic programming formulation of [Lawler and Moore 1969]. At any stage j of the dynamic programming, the (redundant) enumeration of symmetric schedules can be avoided in $\mathcal{O}\ (Um2^n)$ time as follows: (i) select a subset of the set $\{J_1, \ldots, J_j\}$ (a subset has a length between 0 and U), (ii) select m subsets one for each machine (the outer minimum can be updated during this selection) and (iii) check whether or not the selected subsets are distinct by searching $\mathcal{O}\ (2^n)$ sets in the set of all subsets. Since the above steps are repeated for each stage, the total effort becomes $\mathcal{O}\ (mn2^nU)$. Such a procedure has a storage requirement of at least $\mathcal{O}\ (2^n)$ (to store all subsets of the set $\{J_1, \ldots, J_j\}$. This number excludes the storage requirement for intermediate solutions at a stage). Clearly such a procedure is impractical considering realistic problem instances. Furthermore any attempt to reduce the storage requirement increases the computation time.

2.2.2.2 A Dynamic Programming Formulation for the Special Case

In this section we present a dynamic programming formulation for the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, in which there is a fixed number $k > 0$ of piece sizes. An algorithm that uses this formulation is polynomial if k is fixed. As we discussed in Section 2.1, an optimizing algorithm for this problem can be called polynomial number of times to determine an optimal solution for $1|P||C_{max}$ (see Procedure 2.1).

Consider an instance of the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$ in which we are given n_j jobs having processing time p_j for all $j = 1, 2, \ldots, k$ and for some $k > 0$. For such an instance, a packing $\mathcal{P}^i$ of a bin B_i $(i = 1, 2, \ldots, m)$ can be uniquely described by an array of the distribution of piece sizes that are packed in that bin. Such an array is called a *configuration* [Hochbaum and Shmoys 1987; Hochbaum and Shmoys 1988]. In a configuration $(x_1, x_2, \ldots, x_k)$, an entry x_j $(j = 1, 2, \ldots, k)$ shows the number of pieces with size p_j that are packed in a bin. A configuration is called *feasible* if each $x_j \geq 0$ and $\sum_{j=1}^{k} x_j p_j \leq 1$ (the bin size).

Let the state vector $F_i = (u_{i1}, u_{i2}, \ldots, u_{ik})$ be the distribution of the unpacked pieces of each size before we pack the bin B_i. Then u_{ij} $(i = 1, 2, \ldots, m-1;$ $j = 1, 2, \ldots, k)$ is the number of unpacked pieces of size p_j before we pack the bin B_i. Since each entry can take a value between 0 and n, total number of possible state vectors when packing a bin is n^k. Consider a directed layered graph where the nodes correspond to state vectors. Let $V_0, \ldots, V_m$ be the nodes in the 0-th through m-th layers, respectively. V_i $(i = 1, 2, \ldots, m-1)$ contains a vertex F_i for each possible state vector. V_0 and V_m are the dummy nodes; the former corresponding to the initial distribution of piece sizes and the latter corresponding to the "success" node where all the pieces are packed feasibly (that is, $F_m = (0, 0, \ldots, 0)$). From each node F_i, there is an arc directed towards the node F_{i+1} if and only if there is a feasible configuration $(x_1, x_2, \ldots, x_k)$ such that

$$(u_{(i+1)1}, u_{(i+1)2}, \ldots, u_{(i+1)k}) = (u_{i1}, u_{i2}, \ldots, u_{i\ell} - x_\ell, \ldots, u_{ik}) \quad \forall \; \ell = 1, 2, \ldots, k$$

In the directed layered graph, there exists a path from V_0 to V_m if and only if there exists a feasible packing. The complexity of the algorithm is determined as follows: at a layer of the graph we have $\mathcal{O}$ (n^k) nodes for which $\mathcal{O}$ (n^k) configurations (arcs) have to be constructed. Since the feasibility of each node is checked in $\mathcal{O}$ (n) time, time bound of an algorithm is $\mathcal{O}$ (n^{2k+1}) for each layer. Thus for $\mathcal{O}$ (m) layers, the complexity of the algorithm is $\mathcal{O}$ (mn^{2k+1}) which is polynomial when k is fixed.

As far as the real life scheduling problems are concerned, there may be cases where the assumptions on the problem instance become realistic. Consider a shop for example, where items are similar to each other and require almost the same amount of processing. In this case the above formulation has practical importance. Moreover, as it will be presented in next section, it can be used to develop approximation algorithms with as small performance guarantees as we wish to solve the general problem.

2.3 Approximation Algorithms

In the deterministic scheduling literature the most studied scheduling problem from the viewpoint of approximation algorithms is $1|P||C_{max}$ [Lawler *et al.* 1989]. The focus of the research is on the development of an approximation algorithm and/or on the performance analysis of an approximation algorithm. In general these algorithms can be classified as

(i). *list scheduling algorithms*, and

(ii). *bin packing based algorithms*, the algorithms that are using the relation between $1|P||C_{max}$ and the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$(see Section 2.1).

In this section, we present the approximation algorithms which play a pioneering role by giving rise to the development of several other algorithms in the same category. Although we state the performance guarantee of each algorithm, the details of their derivation are not presented. See [Coffman *et al.* 1988; Fisher 1982; Friesen 1978; Garey *et al.* 1978] for a review of the related research.

2.3.1 List Scheduling Algorithms

The motivation behind these algorithms is the observation that it is always possible to find a (sorted) *list* L of jobs using which we can determine an optimal schedule as in Procedure 2.5 in $\mathcal{O}(n)$ time.

Procedure 2.5: A list scheduling heuristic for $1|P||C_{max}$

arguments
$\mathsf{I_{pms}}$: an instance of $1|P||C_{max}$
L: a list of jobs

procedure called
listschedule: a recursive call

```
procedure  listschedule ($\mathsf{I_{pms}}, L$)
begin
      if $L \neq \emptyset$
        begin
              pick job $J_j$ from the top of the list $L$
              assign $J_j$ to the first available machine $M_i$
              $L := L \setminus \{J_j\}$
              call  listschedule ($\mathsf{I_{pms}}, L$)
        end
      else
        output makespan and the related schedule
end
```

Consider the following 7-job $1|P3||C_{max}$ instance

J_j	1	2	3	4	5	6	7
p_j	1	1	1	1	1	1	3

Calling the procedure *listschedule* (L) for $L = (J_1, J_2, \ldots, J_7)$ we obtain the schedule depicted in Figure 2.1. An optimal schedule is as shown in Figure 2.2. It can easily be verified that interchanging the places of the jobs J_7 and J_1 in the above list and calling the procedure *listschedule* (L) for this list we obtain the schedule depicted in the Figure 2.2. If we let $C_{max}(LS)$ denote the makespan value that can be determined by the procedure *listschedule* (L), then for the above problem $C_{max}(LS)/C^*_{max} = 5/3 = 2 - (1/3)$. [Graham 1966] showed that the

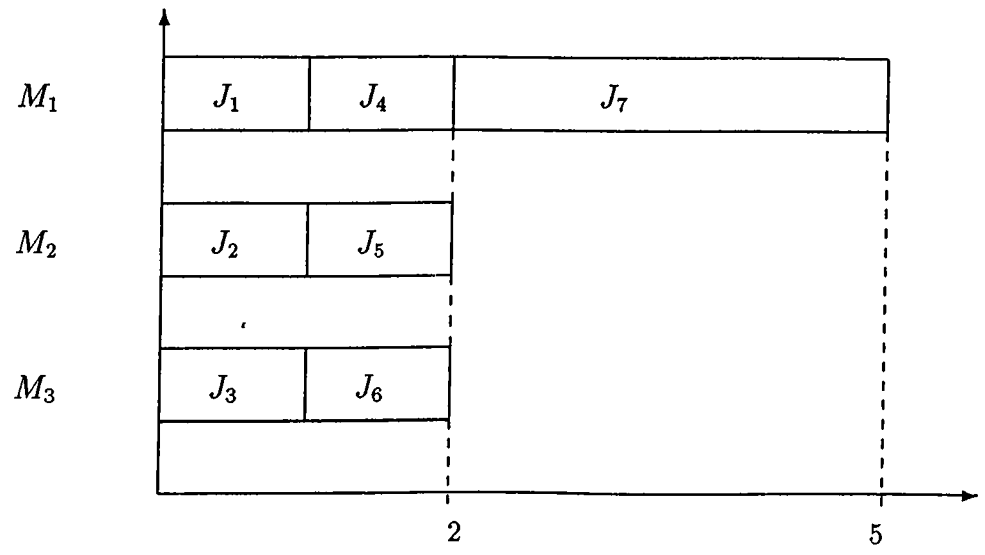


Figure 2.1: A list schedule for the 7-job $1|P3||C_{max}$ instance for $L = (J_1, J_2, \ldots, J_7)$

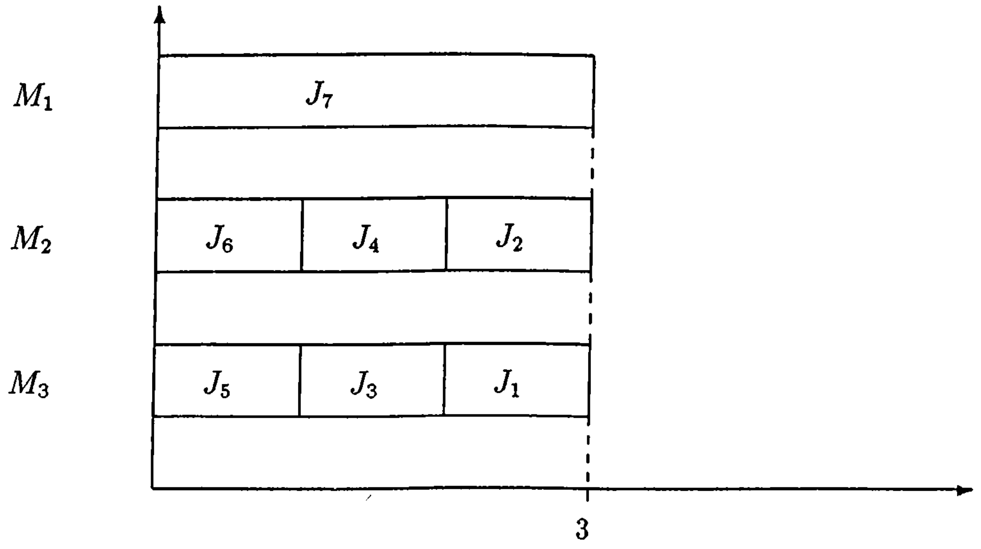


Figure 2.2: An optimal schedule for the 7-job $1|P3||C_{max}$ instance for $L = (J_1, J_2, \ldots, J_7)$

above result can be generalized as a list scheduling algorithm that uses arbitrary list of jobs has the performance guarantee $C_{max}(LS)/C^*_{max} \leq 2 - (1/m)$. As it is shown in the above example this bound is tight. This work was the first worst-case analysis of a heuristic. Later [Graham 1969] showed that if the list is constructed according to descending order of processing times, the list scheduling heuristic, known as the *Longest Processing Time* (*LPT*) heuristic, has the performance guarantee $C_{max}(LPT)/C^*_{max} \leq \frac{4}{3} - \frac{1}{3m}$. The computational effort required by the *LPT* heuristic is dominated by the effort required to form a list (sorting) which is $\mathcal{O}$ $(n \ log \ n)$.

2.3.2 Bin Packing Based Algorithms

The relation between the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$and $1|P||C_{max}$ allows us to solve the latter by using a procedure for the former in a binary search (see Procedure 2.1). This principle can be used to determine an approximate solution for $1|P||C_{max}$ if we call an approximation algorithm for $\Pi^{\mathsf{d}}_{\mathsf{bp}}$at each iteration of the binary search. In this section we will present two approaches developed to solve $1|P||C_{max}$ based on this scheme.

2.3.2.1 Primal Algorithms

[Coffman *et al.* 1978] were the first who have used the above principle. Their *MULTIFIT* algorithm is given in Procedure 2.6. At an iteration of the binary search, the procedure *FFD* (*First Fit Decreasing Algorithm*) is called by *MULTIFIT* to pack a bin (if it can). For a sorted list L of piece sizes (in nonincreasing order of sizes), *FFD* works as shown in Procedure 2.7.

At each packing attempt if the procedure *FFD* outputs a feasible packing, then the current bin size, τ, becomes an upper bound for the minimum makespan problem. Thus each time ub is updated, the new ub is still an upper bound on the problem. On the other hand, if the procedure outputs "no feasible packing", then for a particular bin size, τ, a feasible packing may exist. Therefore lb may not be a lower bound on the problem. But upon a termination of the binary search procedure *MULTIFIT*, we have a valid upper bound on the minimum makespan for $1|P||C_{max}$. The time complexity of *MULTIFIT* is $\mathcal{O}$ $(n \ \log \ n + nk \ \log \ m)$

Procedure 2.6: A primal bin packing based approximation algorithm for $1|P||C_{max}$

arguments
$\mathrm{I}_{\mathsf{pms}}$: an instance of $1|P||C_{max}$
lb: a lower bound for $1|P||C_{max}$
ub: an upper bound for $1|P||C_{max}$.

procedure called
FFD: First Fit Decreasing heuristic for the bin packing problem (see Procedure 2.7).

```
procedure  MULTIFIT ($\mathrm{I}_{\mathsf{pms}}, ub, lb$)
begin
    while $ub - lb \geq 1$
        begin
            $\tau := (ub + lb)/2$
            if  FFD ($\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}/\tau, L$) outputs "no feasible packing"
              $lb := \tau$
            else
              $ub := \tau$
        end
    output  $\lfloor ub \rfloor$ and the schedule related with $ub$
end
```

time for k iterations of the binary search. With the development of this algorithm the question of "What is the worst-case performance bound of the *MULTIFIT*?" motivated many researchers. Although [Coffman *et al.* 1978] were able to show that the bound is at most $1.22C^*_{max}$, they could not show that this is the tightest bound. After a decade and several attempts, [Minyi 1989] have proved (in thirty seven pages) that the bound is exactly 13/11.

Procedure 2.7: *FFD* heuristic for the bin packing problem, Π^{d}_{bp}

arguments
I^{d}_{bp}: an instance of the bin packing problem, Π^{d}_{bp}
L: list of pieces sorted in nonincreasing order of their sizes

procedure called
FFD: a recursive call

```
procedure  FFD ($I^{d}_{bp}$, $L$)
begin
    if $L \neq \emptyset$
     begin
         pick job $J_j$ from the top of the list $L$
         find the first bin $B_i$ that $J_j$ fits (assume bins are indexed)
         if none
          begin
              $L := \emptyset$
              output "no feasible packing"
          end
         else
          begin
              assign $J_j$ to $B_i$
              $L := L \setminus \{J_j\}$
              call  FFD ($I^{d}_{bp}$, $L$)
          end
     end
    else
     output the packing
end
```

2.3.2.2 Dual Algorithms

[Hochbaum and Shmoys 1987] have used the same principle of solving the bin packing problem in a binary search to solve $1|P||C_{max}$. However their approach differs from that of [Coffman *et al.* 1978] in the procedure they have used to solve the underlying bin packing problem. An ε-relaxed decision procedure, as termed in [Hochbaum and Shmoys 1988], is a polynomial time procedure which, given an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}} = (\mathcal{J}, \mathcal{B})$ of the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, either

(i). produces an ε-relaxed packing, that is a feasible packing for the original instance in which size of each bin B_i, $i = 1, 2, \ldots, m$ is extended to $1 + \varepsilon$; or

(ii). outputs "no feasible packing" indicating that there is no feasible packing.

Consider Procedure 2.8. At a packing attempt if the procedure $\varepsilon - \mathit{relaxed}$ outputs "no feasible packing" then due to the definition of an ε-relaxed decision procedure there will be no feasible packing. Hence each time lb is updated, the new lb is still a lower bound on the problem. On the other hand, if the procedure $\varepsilon - \mathit{relaxed}$ does not output "no feasible packing", then given a particular bin size τ a feasible packing may not exist. In such a case the only thing that an ε-relaxed decision procedure provides is an ε-relaxed packing. Therefore, after an update of ub, newly found ub may not be an upper bound on the problem. But given a ub, the procedure $\varepsilon - \mathit{relaxed}$ computes an upper bound.

Since all of the processing times are integer, it is clear that $\lceil lb \rceil$ is a lower bound. Hence the search can be terminated whenever $ub - lb < 1$. However, the procedure $\varepsilon - \mathit{relaxed}$ should be called once more, with bin sizes scaled by $\tau = \lceil lb \rceil$. If the output is "no feasible packing" then $\lceil lb \rceil + 1$ will be a lower bound on the optimum makespan and the schedule produced by $ub < \lceil lb \rceil + 1$ can be used to determine an upper bound on the minimum makespan. Otherwise $\lceil lb \rceil$ will be a lower bound and the schedule produced by $\lceil lb \rceil$ can be used to determine an upper bound on the minimum makespan. Thus at the termination of the binary search we obtain a schedule whose length $(1 + \varepsilon)$ times the lower bound. Therefore the algorithm has a worst case bound of $1 + \varepsilon$.

As mentioned before, the stopping condition of the binary search is satisfied in $\mathcal{O}\,(\log \lceil ub - lb \rceil)$ time. If we use the lower and upper bounds given in Lemmas

Procedure 2.8: A dual bin packing based approximation algorithm for $1|P||C_{max}$

arguments
$\mathrm{I_{pms}}$: an instance of $1|P||C_{max}$
lb: a lower bound for $1|P||C_{max}$
ub: an upper bound for $1|P||C_{max}$

procedure called
$\varepsilon - relaxed$: a hypothetical ε-relaxed decision procedure for the bin packing problem, $\Pi^{\mathrm{d}}_{\mathrm{bp}}$

```
procedure  $\varepsilon - makespan\ (\mathrm{I_{pms}}, lb, ub)$
begin
    while $ub - lb \geq 1$
        begin
            $\tau := (ub + lb)/2$
            if  $\varepsilon - relaxed\ (\mathrm{I^d_{bp}}/\tau)$ outputs "no feasible packing"
              $lb := \tau$
            else
              $ub := \tau$
        end
    if  $\varepsilon - relaxed\ (\mathrm{I^d_{bp}}/\lceil lb \rceil)$ outputs "no feasible packing"
      output  $ub$ and related schedule
    else
      output  $\lceil lb \rceil$ and related schedule
end
```

2.3 and 2.4, respectively, then we will have a polynomial-time procedure which produces schedules with lengths at most $(1+\varepsilon)C^*_{max}$.

[Hochbaum and Shmoys 1987] have provided a family of approximation algorithms D_ε for $\varepsilon > 0$, such that for a fixed ε, D_ε is an ε-relaxed decision procedure (hence a polynomial-time procedure) that runs in $\mathcal{O}\left((m/\varepsilon)(n/\varepsilon)^{1/\varepsilon^2}\right)$. Such a procedure can directly be obtained from the dynamic programming formulation given in Section 2.2.2.2.

Suppose that the bin packing problem is scaled such that all bins have sizes one and the piece sizes are less than one. Obviously, for such an instance if $\sum_{J_j \in \mathcal{J}} p_j > m$, then there is no feasible packing. We exclude this trivial case by assuming $\sum_{J_j \in \mathcal{J}} p_j \leq m$.

Let $\mathcal{J}_{large}$ denote the set of pieces for which $p_j > \varepsilon$. Any piece $J_j \in \mathcal{J}_{large}$ is called as a *large* piece. Then the set $\mathcal{J}_{small} = \mathcal{J} \setminus \mathcal{J}_{large}$ denotes the set of *small* pieces (that is, pieces with size $p_j \leq \varepsilon$). Consider only the pieces with $p_j > \varepsilon$ ($j = 1, 2, \ldots, n$) for the time being. Partition the interval of large piece sizes $(\varepsilon, 1]$ into $\lceil (1-\varepsilon)/\varepsilon^2 \rceil$ equal length subintervals. Thus the size of each interval is at most ε^2. Round the pieces sizes in an interval to the lower end of that interval. As a result we obtain an instance with at most $k = \lceil (1-\varepsilon)/\varepsilon^2 \rceil$ distinct piece sizes. This problem can be solved by using the dynamic programming formulation given in Section 2.2.2.2 in $\mathcal{O}\left((m/\varepsilon)(n/\varepsilon)^{1/\varepsilon^2}\right)$ time. There is a clear reduction in the computation time as compared with the dynamic programming formulation given in Section 2.2.2.2. This is due to the fact that in a feasible configuration each entry can take a value between 0 and $\lfloor 1/\varepsilon \rfloor$ whereas in the previous case this value can be n. If the large pieces with rounded sizes cannot be packed by the dynamic programming algorithm, then there is "no feasible packing" for the original instance where the processing times are larger and there are additional small pieces. If on the other hand they are packed then there exists a path from "initial" to "success" referring to a feasible packing for the restricted problem. In a feasible packing there are at most $\lfloor 1/\varepsilon \rfloor$ large pieces with rounded piece sizes. Therefore, if we restore the sizes of large pieces to their original sizes, in a bin total piece size cannot exceed $1 + \varepsilon$.

Suppose no "no feasible packing" message is output when packing large pieces. In such a packing let $S^t = m - \sum_{J_j \in \mathcal{J}_{large}} p_j$ denote the total slack. Due to our assumption that in a nontrivial instance of the bin packing problem $\sum_{J_j \in \mathcal{J}} p_j \leq m$, $S^t \geq \sum_{J_j \in \mathcal{J}_{small}} p_j$. Moreover, in the packing of large pieces $\sum_{i=1}^{m} \max\{0, 1 - \sum_{J_j \in \mathcal{P}^i} p_j\} \geq S^t$ where $\mathcal{P}^i$ denotes the set of jobs packed on bin B_i (since after restoring the large piece sizes to their original sizes, some of the bins may be filled over the capacity). Hence the remaining slack $\sum_{i=1}^{m} \max\{0, 1 - \sum_{J_j \in \mathcal{P}^i} p_j\} \geq \sum_{J_j \in \mathcal{J}_{small}} p_j$. Therefore, it is sufficient to check whether or not $\sum_{J_j \in \mathcal{J}} p_j \leq m$. If it is, then small pieces can be packed within bins with at most $1 + \varepsilon$ times their true capacity. The procedure is simple, choose a small piece and pack it into any bin with positive remaining slack. Clearly the complexity of the overall algorithm is due the first phase, large-pack, since the second phase takes $\mathcal{O}\ (n)$ time.

Although this result is theoretically appealing, it has less practical importance considering its space requirement and time complexity. More practical algorithms were developed for $\varepsilon = 1/5$ and $\varepsilon = 1/6$ which run in $\mathcal{O}\ (n(k + \log\ n))$ and $\mathcal{O}\ (n(km^4 + \log\ n))$ times for the $\mathcal{O}\ (k)$ iterations of the binary search, respectively.

In [Hochbaum and Shmoys, 1988] the same idea was applied to solve a *uniform parallel machine* scheduling problem, $1|Q||C_{max}$. In this problem, a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of independent jobs with processing times $p_1, p_2, \ldots, p_n$ are to be scheduled nonpreemptively on a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of non-identical machines. These machines run at different speeds $s_1, s_2, \ldots, s_m$. Hence, if job J_j is processed on machine M_i, it takes p_j/s_i time units to be completed. Consider the related question of deciding whether there exists a schedule in which all of the jobs are completed by time τ. More precisely, in this decision problem we are looking for a schedule $\mathcal{P}$ in which $\sum_{J_j \in \mathcal{P}^i} p_j/s_i \leq \tau$ for $i = 1, 2, \ldots, m$ or equivalently $\sum_{J_j \in \mathcal{P}^i} p_j \leq \tau s_i$ for $i = 1, 2, \ldots, m$. Rescaling both the processing requirements and the speeds by a factor $1/\tau$, we obtain *the bin packing problem with variable bin sizes*, $s_1, s_2, \ldots, s_m$. $\mathcal{NP}$-completeness result of Theorem 2.3 associated with the bin packing problem (with unit bin sizes) proves that the above decision problem is unary $\mathcal{NP}$-complete since the bin packing problem with variable bin sizes is an obvious generalization of the problem with unit sizes. This suggests that the existence of an efficient optimizing algorithm for this problem is highly unlikely. [Hochbaum and Shmoys 1988] have generalized the idea of ε-relaxed decision procedure to obtain good approximation algorithms. Suppose that for this problem we have a polynomial-time algorithm which either

(i) produces an ε-relaxed packing, a feasible packing in which the size of each bin is extended to $(1+\varepsilon)s_i$, $i = 1, 2, \ldots, m$; or

(ii) outputs "no feasible packing" indicating that there is no feasible packing.

Although such an algorithm is not the same as the previous one, they refer to it as an ε-relaxed decision procedure to emphasize the similarity. As before such an algorithm can be used in a binary search to solve $1|Q||C_{max}$. [Hochbaum and Shmoys 1988] have provided a family of such ε-relaxed decision procedures with time complexity $\mathcal{O}\ (mn^{(10/\varepsilon^2)+3})$. For $\varepsilon = 1/2$ they provided an algorithm to solve the related bin packing problem with variable bin sizes which runs in $\mathcal{O}\ (n)$ (if the piece and bin sizes are sorted in advance). An algorithm along the same lines will be proposed in Section 3.2.1.

Chapter 3

A Branch and Bound Algorithm

In Section 2.2.1, we have presented the branch and bound algorithm of [Bratley *et al.* 1975] which was developed to solve $1|P|r_j, \overline{d_j}|C_{max}$. As mentioned before it can be used to solve a special case of the original problem, $1|P||C_{max}$. A general algorithm when applied to a special case, may not utilize some of the characteristics of the solution space. As a result the set of solutions that need to be enumerated to determine optimal solution and hence the size of the enumeration tree becomes unnecessarily large.

Let Σ be the set of schedules satisfying Lemma 2.1. Any schedule $\mathcal{P} \in \Sigma$ can be represented as an *onto* function $f : \mathcal{J} \mapsto \mathcal{M}$ since no machine remains idle. This is illustrated in Figure 3.1 for a 4-job $1|P3||C_{max}$.

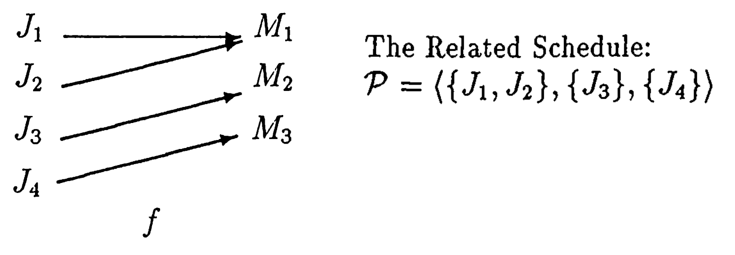

J_1
J_2
J_3
J_4
M_1
M_2
M_3
f
The Related Schedule:
$\mathcal{P} = \langle\{J_1, J_2\}, \{J_3\}, \{J_4\}\rangle$

Figure 3.1: A schedule represented as an onto function

It can be shown that for $n \geq m$ the number of onto functions is

$$s(n,m) = \sum_{t=0}^{m}(-1)^t \begin{pmatrix} m \\ m-t \end{pmatrix} (m-t)^n \tag{3.1}$$

Consider the example in Figure 3.1. In this example there are 3^4 different functions from $\mathcal{J}$ to $\mathcal{M}$. Considering subsets of $\mathcal{M}$ of size 2, there are 2^4 functions from $\mathcal{J}$ to $\{M_1, M_2\}$, 2^4 functions from $\mathcal{J}$ to $\{M_2, M_3\}$, and 2^4 functions from $\mathcal{J}$ to $\{M_1, M_3\}$. So we have $(3)2^4 = (\,{}^3_2\,)2^4$ functions from $\mathcal{J}$ to $\mathcal{M}$ that are definitely not *onto*. However, it should be realized that not all of these $(\,{}^3_2\,)2^4$ functions are distinct. For when we consider all the functions from $\mathcal{J}$ to $\{M_1, M_2\}$, we are removing, among these, the function $\{(J_1, M_2), (J_2, M_2), (J_3, M_2), (J_4, M_2)\}$. Then considering the functions from $\mathcal{J}$ to $\{M_2, M_3\}$, we remove the same function: $\{(J_1, M_2), (J_2, M_2), (J_3, M_2), (J_4, M_2)\}$. Consequently, in the result $3^4 - (\,{}^3_2\,)2^4$, we have twice removed each of the constant functions $f : \mathcal{J} \mapsto \mathcal{M}$, where $f(\mathcal{J})$ is one of the sets $\{M_1\}, \{M_2\}$, or $\{M_3\}$. Hence for the above example the number of onto functions from $\mathcal{J}$ to $\mathcal{M}$ is $3^4 - (\,{}^3_2\,)2^4 + 3 = (\,{}^3_3\,)3^4 - (\,{}^3_2\,)2^4 + (\,{}^3_1\,)1^4$. This intuitive explanation can be extended for a general case as shown in Equation 3.1.

Furthermore the cardinality of Σ is much less than $s(n,m)$. Schedules represented by distinct onto functions may be same since the machines are identical. For example, consider two schedules $\mathcal{P}_1 = \langle\{J_1, J_2\}, \{J_3\}, \{J_4\}\rangle$ and $\mathcal{P}_2 = \langle\{J_4\}, \{J_1, J_2\}, \{J_3\}\rangle$ for a 4-job $1|P3||C_{max}$. [Lawler and Moore 1969] have referred to this type of schedules as symmetric schedules. Although these schedules are represented by two distinct onto functions, they refer to the same schedule, $\mathcal{P} = \langle\{J_1, J_2\}, \{J_3\}, \{J_4\}\rangle$. Thus the number of distinct schedules in Σ is $S(n,m) = s(n,m)/m!$ since there are $m!$ different ways to arrange m identical machines. This latter number is known as the *Stirling number of the second kind.* Table 3.1 shows the behavior of this number for some m and n values. An analysis of Tables 2.1 and 3.1 shows that the number of feasible schedules that need to be enumerated is substantially less as compared with the number

Table 3.1: The Stirling number of the second kind

		m							
		1	2	3	4	5	6	7	8
	1	1	–	–	–	–	–	–	–
	2	1	1	–	–	–	–	–	–
	3	1	3	1	–	–	–	–	–
	4	1	7	6	1	–	–	–	–
	5	1	15	25	10	1	–	–	–
	6	1	31	90	65	15	1	–	–
n	7	1	63	301	350	140	21	1	–
	8	1	127	966	1701	1050	266	28	1
	9	1	255	3025	7770	6951	2646	462	36
	10	1	511	9330	34105	42525	22827	5880	750
	11	1	1023	28501	145750	246730	179487	63987	11880
	12	1	2047	86526	611501	1379400	1323652	627396	159027
	13	1	4095	261625	2532530	7508501	9321312	5715424	1899612
	14	1	8191	788970	10391745	40075035	63436373	49329280	20912320
	15	1	16383	2375101	42355950	210766920	420693273	408741333	216627840

that is enumerated by the branch and bound algorithm of [Bratley *et al.* 1975]. The cardinality of Σ may further be reduced due to Lemma 2.2. The number of feasible schedules satisfying both Lemma 2.1 and Lemma 2.2 is at most $S(n,m)$ since Lemma 2.2 may reduce the size of the set defined by Lemma 2.1 depending on the processing time data. Hereon we will refer to this set as the set of all solutions that are candidate for being optimal and we will denote it by Σ^*.

The proposed branch and bound algorithm implicitly enumerates the feasible schedules in Σ^*. In Section 3.1 we will explain the branching scheme and the enumeration tree generated by this scheme. Section 3.2 explains the bounding scheme we used. The strategy for searching the tree during enumeration process will be explained in Section 3.3. Finally results of our computer implementation will be presented in Section 3.4.

3.1 Branching Scheme

At a node N_k of the enumeration tree we are given a subproblem which is a partially solved version of the original $1|P||C_{max}$. With this node we associate

- a *partial schedule* $\mathcal{P}_k = \langle\, \mathcal{P}_k^1, \mathcal{P}_k^2, \ldots, \mathcal{P}_k^m \,\rangle$ where $\mathcal{P}_k^i$ is the set of jobs scheduled on machine M_i such that $\bigcap_{i=1}^m \mathcal{P}_k^i = \emptyset$ and, $\mathcal{P}_0 = \emptyset$ at the root node N_0.

- a set $\overline{\mathcal{P}}_k$ of *unscheduled jobs* where $\overline{\mathcal{P}}_0 = \mathcal{J}$ at the root node N_0.

- a set of *discarded machines* $\delta_k = \{M_i \mid l(\mathcal{P}_k^i) \geq \lambda \quad i = 1, 2, \ldots, m\}$, where $\lambda = (1/m)l(\mathcal{J})$. For the ease of notation we assume that the last $|\delta_k|$ machines are in the set of discarded machines. If this is not the case however, machines can be reindexed such that $M_i \in \delta_k \ \ \forall\, i = m - |\delta_k| + 1, \ldots, |\delta_k|$ since they are identical. We let $\overline{\delta}_k$ denote the complement set of δ_k so that $\overline{\delta}_k \cup \delta_k = \mathcal{M}$.

- a set of *captured machines* $\mu_k = \{M_1\} \cup \{M_i \mid l(\mathcal{P}_k^i) \neq l(\mathcal{P}_k^{i-1}) \quad i = 2, \ldots, |\overline{\delta}_k|\}$. It can be observed that $M_i \notin \delta_k \ \ \forall\, i = 1, \ldots, |\overline{\delta}_k|$ due to the definition of the set δ_k. Clearly $|\mu_k| \leq |\overline{\delta}_k|$ since there may be several machines with the same completion time. It is convenient to assume that $M_i \in \mu_k \ \ \forall\, i = 1, \ldots, |\mu_k|$ and $l(\mathcal{P}_k^1) > \ldots > l(\mathcal{P}_k^{|\mu_k|})$. If the situation is different, machines can be reindexed accordingly.

Using the above notation, the subproblem at node N_k is defined as to schedule the set $\overline{\mathcal{P}}_k$ of jobs on the set $\mathcal{M}$ of machines such that the schedule length is minimized and each machine M_i processes the already allocated load $l(\mathcal{P}_k^i)$ nonpreemptively. Clearly an optimal schedule for this problem has the characteristics described by Lemmas 2.1 and 2.2. Therefore only the machines $M_i \in \overline{\delta}_k$ need to be considered while solving the problem since in an optimal solution none of the jobs in $\overline{\mathcal{P}}_k$ will be scheduled on the remaining machines. When we determine the optimal makespan $C^*_{max}(\mathcal{P}_k, \overline{\delta}_k)$ for this restricted problem, we can determine the optimal makespan for the original subproblem as $C^*_{max}(\mathcal{P}_k) = \max\{\max_{M_i \in \delta_k} l(\mathcal{P}_k^i),\ C^*_{max}(\mathcal{P}_k, \overline{\delta}_k)\}$. Nevertheless even the restricted problem is still unary $\mathcal{NP}$-complete (see Section 3.2). This result provides a formal justification to continue enumeration.

The branching mechanism decomposes the scheduling problem at node N_k into subproblems (descendants) with the property that solving all of them solves the original. However the mechanism may decide to stop generating new nodes. If none of the feasible schedules that can be obtained from the current partial schedule can have a makespan better than the best known so far, then there is no need to enumerate them. Similarly, if an optimal solution for the problem at node N_k is determined, related descendants will not be generated. We will approach the presentation of the branching scheme by explaining these fathoming conditions in detail.

We let $LB(\mathcal{P}_k, \bar{\delta}_k)$ and $UB(\mathcal{P}_k, \bar{\delta}_k)$ denote the lower and upper bounds on the restricted problem at node N_k, respectively. As we mentioned above, in this problem machines $M_i \in \delta_k$ are not considered when scheduling. An algorithm to determine these bounds and other related issues will be explained in Section 3.2. Once these bounds are determined the bounds on the original problem can easily be determined as $LB(\mathcal{P}_k) = \max\{\max_{M_i \in \delta_k} l(\mathcal{P}_i),\ LB(\mathcal{P}_k, \bar{\delta}_k)\}$ and $UB(\mathcal{P}_k) = \max\{\max_{M_i \in \delta_k} l(\mathcal{P}_i),\ UB(\mathcal{P}_k, \bar{\delta}_k)\}$. The lower bound $LB(\mathcal{P}_k)$, refers to the makespan of the best (not necessarily feasible) solution and the upper bound $UB(\mathcal{P}_k)$ is the makespan of *a* feasible schedule in the descendants of node N_k. For the time being suppose node N_k is generated and the bounds on the scheduling problem are determined in the way we mentioned briefly. Let Z be the incumbent value (i.e. best makespan value determined among the enumerated feasible schedules) as before. Upon the generation of node N_k, this value is updated as $Z := \min\{UB(\mathcal{P}_k), Z\}$. Below propositions describe the circumstances under which node N_k is fathomed.

Proposition 3.1 *If the condition $LB(\mathcal{P}_k) \geq Z$ is satisfied for some node N_k then the descendants of this node are not generated.*

Proof: None of the feasible schedules encountered among the descendants of the current node can have a makespan better than the best known. ■

Proposition 3.2 *At a node N_k, if the condition $UB(\mathcal{P}_k, \bar{\delta}_k) \leq l(\mathcal{P}_k^{i_1})$ is satisfied for some $M_{i_1} \in \delta_k$ with $l(\mathcal{P}_k^{i_1}) = \max_{M_i \in \delta_k} l(\mathcal{P}_k^i)$, then the descendants of this* *node are not generated.*

Proof: None of the descendants of the current node can have a makespan better than $l(\mathcal{P}_k^{i_1})$. ■

Proposition 3.3 *At a node N_k, if the number of unscheduled jobs is two, that is $|\overline{\mathcal{P}}_k| = 2$, then the descendants of this node are not generated.*

Proof: In this case two jobs remain as unscheduled. Let J_{max} be the one with the maximum processing time and J_{min} be the other one. In an optimal schedule to the problem at node N_k, either both of them are scheduled on the same machine or on different machines. In both of these cases, the earliest start time of J_{max} will be less than or equal to that of J_{min}. When the jobs are scheduled on the same machine, the order is immaterial, thus without loss of generality assume that J_{max} precedes J_{min}. In the second case, if the earliest start time of J_{max} is greater than that of J_{min}, then simply interchanging the jobs decreases the makespan. Thus, the LPT heuristic (see Section 2.3.1) determines the optimal schedule for the problem at node N_k. Therefore after determining the makespan, Z, the incumbent value, is updated if necessary, and the current node is fathomed. ■

Proposition 3.4 *At a node N_k, if the number of idle machines is equal to one less than the number of unscheduled jobs, that is $|\overline{\mathcal{P}}_k| - 1$, then the descendants of this node are not generated.*

Proof: Suppose at node N_k, the number of idle machines is equal to $|\overline{\mathcal{P}}_k| - 1$. In an optimal solution to the scheduling problem at this node $|\overline{\mathcal{P}}_k|$ largest jobs in $\overline{\mathcal{P}}_k$ are scheduled on idle machines and the remaining job is scheduled on the machine that becomes idle first (i.e. use LPT heuristic). This property is proved by interchange arguments as follows: suppose for a contradiction that there was an optimal schedule $\mathcal{P}_*$ for the problem at node N_k in which one of the first $|\overline{\mathcal{P}}_k|$ largest jobs, J_{j_1}, is not scheduled on one of the idle machines but on some other machine M_t. Then some other job $J_{j_2} \in \overline{\mathcal{P}}_k$ with $p_{j_2} \leq p_{j_1}$ will be scheduled on one of these idle machines M_i, since in an optimal schedule no machine remains idle. By interchanging these two jobs we obtain another schedule $\mathcal{P}_{**}$ in which $l(\mathcal{P}^i_{**}) \leq l(\mathcal{P}^t_{**}) \leq l(\mathcal{P}^t_*)$. Thus we obtain the contradiction that $C_{max}(\mathcal{P}_{**}) \leq C^*_{max}$ since $l(\mathcal{P}^t_*) \leq C^*_{max}$ and none of the completion times of the machines other than M_t and M_i have changed. Hence the LPT heuristic determines optimal schedule for the problem at node N_k. After determining the makespan, Z is updated if necessary, and the current node is fathomed. ■

Proposition 3.5 *At a node N_k, if all but one of the machines are discarded, that is $|\delta_k| = m - 1$, then the descendants of this node are not generated.*

Proof: In this case due to Lemma 2.2, unscheduled jobs in $\overline{\mathcal{P}}_k$ are scheduled on the remaining machine which is not discarded yet. After determining the related makespan, the incumbent value , Z, is updated if necessary, and the current node is fathomed. ■

Suppose that none of the conditions in Propositions 3.1 through 3.5 are satisfied at the node N_k. In this case the node N_k is placed in the *active nodes list* since it is not yet decided whether the optimal solution for $1|P||C_{max}$ corresponds to one of its descendants or not. Suppose that after generating several other nodes of the enumeration tree, we select[‡] the node N_k for further enumeration and a job $J_c \in \overline{\mathcal{P}}_k$ to be scheduled (fixed). For this node Proposition 3.1 has to be applied once more since the incumbent value Z may be reduced when generating other nodes. Either of the following circumstances may be encountered if node N_k is not fathomed:

(i) The set of jobs scheduled on a machine M_i, $\mathcal{P}^i_k$, is not empty $\forall\ i = 1, 2, \ldots, m$. In this case, we have $|\mu_k|$ alternative machines on which J_c might be scheduled in an optimal solution. At each descendant node

[‡]Selection rules are discussed in Section 3.3

N_{k_d} $(d = 1, \ldots, |\mu_k|)$ of node N_k, let $\mathcal{P}^d_{k_d} = \mathcal{P}^d_k \cup \{J_c\}$, and $\mathcal{P}^i_{k_d} = \mathcal{P}^i_k$ $\forall\, i \neq d$. Clearly $|\mu_k| \leq m$ since there may be several machines with the same completion time. In such a case, since these machines are identical, it is sufficient to consider only one of them while generating descendant nodes. After generating all of its descendants, node N_k is removed from the active nodes list.

(ii) The set of jobs scheduled on a machine M_i, $\mathcal{P}^i_k$, is empty for some $i = t, t+1, \ldots, m$. Such a t is determined by reindexing machines in a way that the last $m - t + 1$ machines have no jobs yet assigned. Number of branches will be equal to the number of captured machines, $|\mu_k|$ which $\leq t$. A descendant node N_{k_d} $(d = 1, \ldots, t)$ of the node N_k is determined as, $\mathcal{P}^d_{k_d} = \mathcal{P}^d_k \cup \{J_c\}$ and $\mathcal{P}^i_{k_d} = \mathcal{P}^i_k$ $\forall\, i \neq d$. Number of branches $|\mu_k|$, may be less than t since several machines may have the same completion time. After generating all of the descendants node N_k is removed from the active nodes list.

Theorem 3.1 *The above branching scheme enumerates all feasible schedules that are candidate for being optimal without any repetition.*

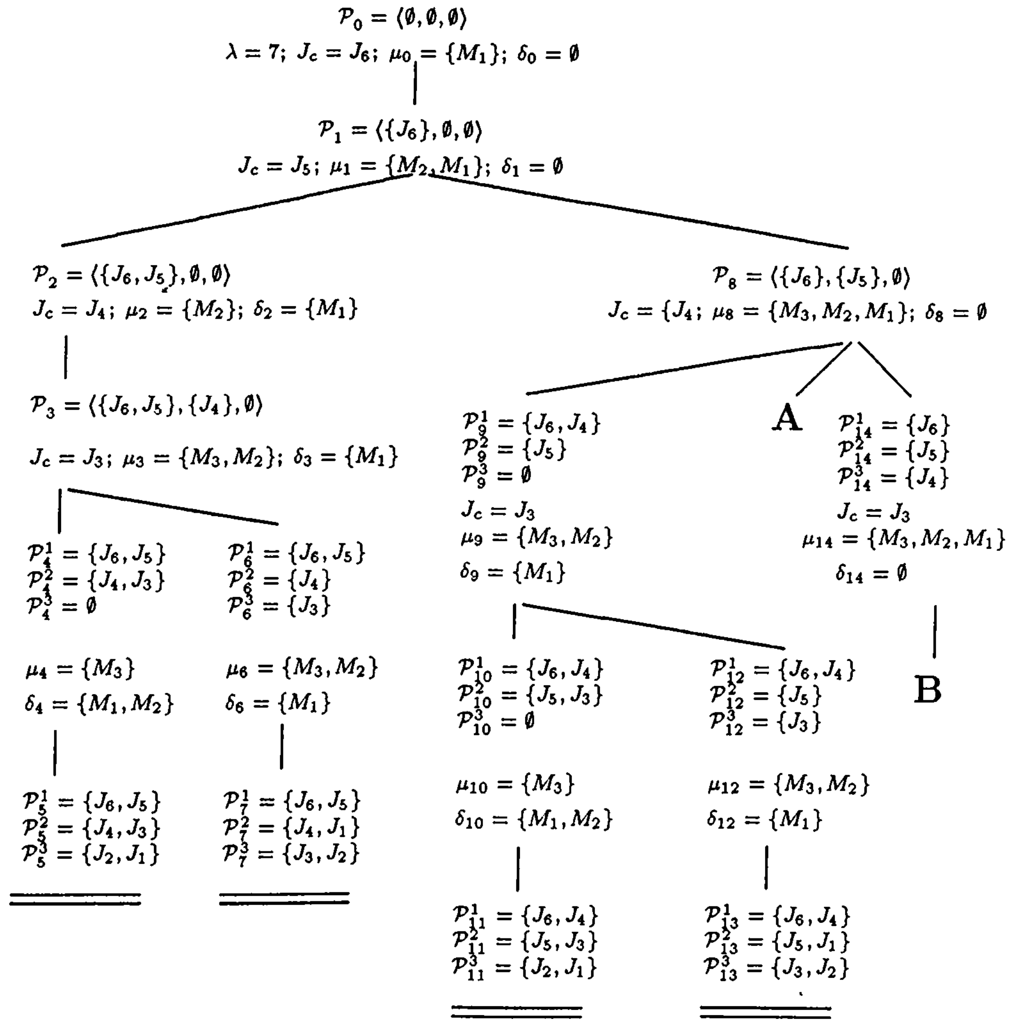
Proof: Suppose that the enumeration tree is generated as described above without making use of the fathoming conditions due to Propositions 3.1 through 3.5. Let $Y(\mathcal{P}_{k_d})$ $(d = 1, \ldots, |\mu_k|)$ denote the set of feasible schedules (i.e. leaf nodes) that can be generated from a descendant node N_{k_d} of N_k. At a node N_k of the enumeration tree, $Y(\mathcal{P}_{k_1}) \cap Y(\mathcal{P}_{k_2}) = \emptyset$ for all k_1 and k_2 such that $k_1 \neq k_2$ since otherwise there exists two schedules $\mathcal{P}_a$ and $\mathcal{P}_b$ such that $\mathcal{P}^i_a = \mathcal{P}^i_b$ for all $i = 1, 2, \ldots, m$ (choose any two descendant nodes N_{k_1} and N_{k_2}; due to the way we are branching, in the related partial schedules there exists a machine M_i such that $\mathcal{P}^i_{k_1} \neq \mathcal{P}^i_{k_2}$). Hence no partial schedule is generated twice in the enumeration tree. Moreover a node is removed from the active nodes list once its descendants are generated. Thereby the branching scheme enumerates the feasible schedules without any repetition. Furthermore using Propositions 3.3

through 3.5, the branching scheme generates the set of feasible schedules that are candidate for being optimal because only the set of captured machines is considered while branching (due to Lemmas 2.1 and 2.2) and the propositions do not allow a machine to be idle at the leaf node. Furthermore Propositions 3.1 and 3.2 if applied, will reduce the size of the enumeration tree without destroying this property. ■

Figure 3.2 depicts the enumeration tree for a 6-job $1|P3||C_{max}$ problem with processing time of each job being equal to its job index and with job selections at each node as indicated. As shown in Table 3.1 the number of schedules that need to be enumerated is 90. In the branching scheme used, however, this number is reduced to 9 with the use of Propositions 3.3 through 3.5. Since the bounding scheme is not yet discussed, Propositions 3.1 and 3.2 are not applied in this example.

The size of an enumeration tree directly affects the performance of a branch and bound algorithm. In order to achieve a reduction in size we make use of the fathoming rules described above. These fathoming rules test the condition at the currently generated descendant node and decide whether or not to eliminate this node. On the other hand, it is possible to develop fathoming conditions that, given the situation at a generated node N_{k_t} (a descendant of N_k), eliminates rest of the descendants of N_k. One such condition will be presented in Section 3.3 since it uses the bounding scheme (Section 3.2).

The bounding scheme and the search strategy used also affect the size of the enumeration tree. If a bounding scheme generates tight bounds, the size of the tree is likely to be reduced further due to the fathoming rules of Propositions 3.1 and 3.2. The bounding scheme will be discussed in Section 3.2. In order to illustrate how a search strategy affects the size of the enumeration tree consider the simple example in Figure 3.2. In this figure the optimal schedule is encountered at node N_{20}. The earlier this node is selected, the sooner the enumeration terminates since rest of the nodes would then be fathomed with the use of Proposition 3.1. In this example it can also be seen that if we had used



$\mathcal{P}_0 = (\emptyset,\emptyset,\emptyset)$
$\lambda = 7;\ J_c = J_6;\ \mu_0 = \{M_1\};\ \delta_0 = \emptyset$
$\mathcal{P}_1 = (\{J_6\},\emptyset,\emptyset)$
$J_c = J_5;\ \mu_1 = \{M_2,M_1\};\ \delta_1 = \emptyset$
$\mathcal{P}_2 = (\{J_6,J_5\},\emptyset,\emptyset)$
$J_c = J_4;\ \mu_2 = \{M_2\};\ \delta_2 = \{M_1\}$
$\mathcal{P}_8 = (\{J_6\},\{J_5\},\emptyset)$
$J_c = \{J_4;\ \mu_8 = \{M_3,M_2,M_1\};\ \delta_8 = \emptyset$
$\mathcal{P}_3 = (\{J_6,J_5\},\{J_4\},\emptyset)$
$J_c = J_3;\ \mu_3 = \{M_3,M_2\};\ \delta_3 = \{M_1\}$
A
B

Figure 3.2: The enumeration tree for a 6-job $1|P3||C_{max}$ problem

different job selections at a node, we would have ended up with a larger tree. These issues will be discussed in Section 3.3.

The branch and bound procedure is summarized in Procedure 3.1. An analysis of this algorithm suggests that the computational burden involved at a node N_k of the tree is due to the procedures *bounds*, *selectanode* and *selectajob*. The remaining operations can be done in linear time. At a node N_k, these remaining

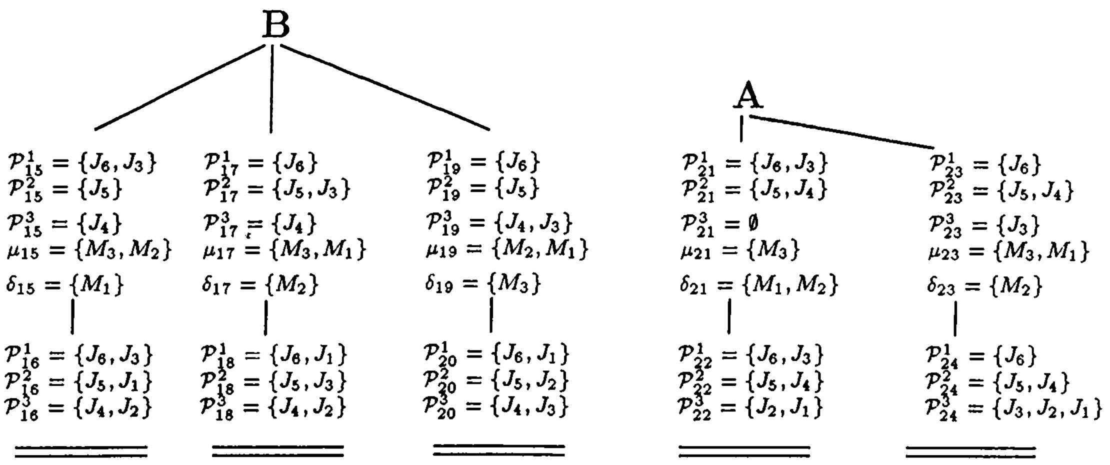

B
A
$\mathcal{P}_{15}^1 = \{J_6, J_3\}$
$\mathcal{P}_{15}^2 = \{J_5\}$
$\mathcal{P}_{15}^3 = \{J_4\}$
$\mu_{15} = \{M_3, M_2\}$
$\delta_{15} = \{M_1\}$
$\mathcal{P}_{17}^1 = \{J_6\}$
$\mathcal{P}_{17}^2 = \{J_5, J_3\}$
$\mathcal{P}_{17}^3 = \{J_4\}$
$\mu_{17} = \{M_3, M_1\}$
$\delta_{17} = \{M_2\}$
$\mathcal{P}_{19}^1 = \{J_6\}$
$\mathcal{P}_{19}^2 = \{J_5\}$
$\mathcal{P}_{19}^3 = \{J_4, J_3\}$
$\mu_{19} = \{M_2, M_1\}$
$\delta_{19} = \{M_3\}$
$\mathcal{P}_{21}^1 = \{J_6, J_3\}$
$\mathcal{P}_{21}^2 = \{J_5, J_4\}$
$\mathcal{P}_{21}^3 = \emptyset$
$\mu_{21} = \{M_3\}$
$\delta_{21} = \{M_1, M_2\}$
$\mathcal{P}_{23}^1 = \{J_6\}$
$\mathcal{P}_{23}^2 = \{J_5, J_4\}$
$\mathcal{P}_{23}^3 = \{J_3\}$
$\mu_{23} = \{M_3, M_1\}$
$\delta_{23} = \{M_2\}$
$\mathcal{P}_{16}^1 = \{J_6, J_3\}$
$\mathcal{P}_{16}^2 = \{J_5, J_1\}$
$\mathcal{P}_{16}^3 = \{J_4, J_2\}$
$\mathcal{P}_{18}^1 = \{J_6, J_1\}$
$\mathcal{P}_{18}^2 = \{J_5, J_3\}$
$\mathcal{P}_{18}^3 = \{J_4, J_2\}$
$\mathcal{P}_{20}^1 = \{J_6, J_1\}$
$\mathcal{P}_{20}^2 = \{J_5, J_2\}$
$\mathcal{P}_{20}^3 = \{J_4, J_3\}$
$\mathcal{P}_{22}^1 = \{J_6, J_3\}$
$\mathcal{P}_{22}^2 = \{J_5, J_4\}$
$\mathcal{P}_{22}^3 = \{J_2, J_1\}$
$\mathcal{P}_{24}^1 = \{J_6\}$
$\mathcal{P}_{24}^2 = \{J_5, J_4\}$
$\mathcal{P}_{24}^3 = \{J_3, J_2, J_1\}$

Figure 3.2: The enumeration tree for a 4-job $1|P3||C_{max}$ problem (continued)

operations are

(i) generation of the descendants can be done in $\mathcal{O}\ (m)$ time since the number of the descendants is $|\mu_k| \leq m$.

(ii) determination of the set μ_{k_d} for each descendant can be done in $\mathcal{O}\ (m)$ time. We have defined the set of captured machines, μ_k, in such a way that $M_i \in \mu_k \quad \forall\ i = 1, \ldots, |\mu_k|$ and $l(\mathcal{P}_k^1) > \ldots > l(\mathcal{P}_k^{|\mu_k|})$. We generate descendants by fixing a selected job J_c on machines in the set μ_k in the order given above. At a descendant N_{k_d}, we define a pointer, $i_\rightarrow$ to the the smallest machine index i such that $l(\mathcal{P}_k^i) > l(\mathcal{P}_k^d) + p_c$. Consider the first descendant at which job J_c is fixed on machine M_1. Without loss of generality we assume that at this descendant $i_\rightarrow = 0$ and $l(\mathcal{P}_k^0) = \infty > l(\mathcal{P}_k^1) + p_c$. Hence the set μ_{k_1} is same as the set μ_k assuming that $l(\mathcal{P}_k^1) + p_c < \lambda$. Suppose at descendants $N_{k_2}, \ldots, N_{k_{t-1}}$ for some $t \geq 2$, respective sets $\mu_{k_2}, \ldots, \mu_{k_{t-1}}$ remain same as the set μ_k (i.e. $l(\mathcal{P}_k^{d-1}) > l(\mathcal{P}_k^d) + p_c$ and $l(\mathcal{P}_k^d) + p_c < \lambda$ for $d = 2, \ldots, t-1$ and for some $t \geq 2$). While generating the t-th descendant, the set μ_{k_t} may change since the position of $l(\mathcal{P}_k^t) + p_c$ in the original list

Procedure 3.1: The proposed branch and bound procedure for $1|P||C_{max}$

argument
Ipms: an instance of $1|P||C_{max}$

procedures called
bounds: given $\mathcal{P}_k$, computes $LB(\mathcal{P}_k)$ and $UB(\mathcal{P}_k)$ (see Procedure 3.6)
selectanode: selects a node N_k among the ones in the active nodes list (see Section 3.3)
selectajob: selects a jobs $J_c \in \overline{\mathcal{P}}_k$ at a selected node N_k (see Section 3.3)

```
procedure 1pcmax (Ipms)
begin
  Initialize node $N_0$
  call bounds ($\mathcal{P}_0, LB(\mathcal{P}_0), UB(\mathcal{P}_0)$)
  if $LB(\mathcal{P}_0) = UB(\mathcal{P}_0)$ then
    stop with $C^*_{max} := LB(\mathcal{P}_0)$ and $\mathcal{P}_* := \mathcal{P}_{k_d}$
  activenodeslist := $\{N_0\}$
  $Z := UB(\mathcal{P}_0)$
  while activenodeslist $\neq \emptyset$
    begin
      call selectanode (activenodeslist, $N_k$)
      if $LB(\mathcal{P}_k) < Z$ then
        begin
          call selectajob ($\overline{\mathcal{P}}_k, J_c$)
          for each descendant node $N_{k_d}$ $(d = 1, \ldots, |\mu_k|)$
            begin
              call bounds ($\mathcal{P}_{k_d}, LB(\mathcal{P}_{k_d}), UB(\mathcal{P}_{k_d})$)
              if one of Propositions 3.1–3.5 is satisfied
                fathom the node $N_{k_d}$
              else
                begin
                  if $UB(\mathcal{P}_{k_d}) = LB(\mathcal{P}_0)$ then
                    stop with $C^*_{max} := LB(\mathcal{P}_0)$ and $\mathcal{P}_* := \mathcal{P}_{k_d}$
                  $Z := \min\{UB(\mathcal{P}_{k_d}), Z\}$
                  Determine the set $\mu_{k_d}$
                  activenodeslist := activenodeslist $\cup$ $\{N_{k_d}\}$
                end
            end
          Remove $N_k$ from activenodeslist
        end
    end
  stop with $C^*_{max} = Z$ and related schedule
end
```

may change. This position will at least be $i_{\rightarrow}$. Therefore we search for the smallest position t' in the interval $[i_{\rightarrow}, t-1]$ until $l(\mathcal{P}_k^{t'}) \leq l(\mathcal{P}_k^2) + p_c$. Once we determine the position t', we obtain $\mu_{k_t} = \{M_1, \ldots, M_{t'-1}, M_t, M_{t'}, \ldots, M_{t-1}, M_{t+1}, \ldots, M_{|\mu_k|}\}$ assuming that $l(\mathcal{P}_{k_t}^{t'}) = l(\mathcal{P}_k^t) + p_c < \lambda$ and $t' \neq 1$. Then we update $i_{\rightarrow} = t'$ for the next descendant. In this way the set μ_{k_d} for all $d = 1, \ldots, |\mu_k|$ is determined in $\mathcal{O}(m)$ time. Yet there are two cases in which the structure of μ_{k_d} is destroyed. If at some descendant N_{k_t} $l(\mathcal{P}_k^t) + p_c \geq \lambda$, the related machine is removed from μ_{k_t} and is placed in δ_{k_t}. Moreover if at some descendant N_{k_t} $l(\mathcal{P}_k^t) + p_c < \lambda$ and $l(\mathcal{P}_k^t) + p_c = l(\mathcal{P}_k^d)$ for some $d < t$, then the related machine will neither be in μ_{k_t} nor in δ_{k_t}. In such a case we insert the machine at the d-th position of the list but we do not consider it while branching.

(iii) For the fathoming conditions due to Propositions 3.1 and 3.2 two comparisons are required. For the conditions due to Propositions 3.3 through 3.5 we need to store number of unscheduled jobs, number of idle machines and number of discarded machines. At a node N_k these variables can be updated in linear time.

3.2 Bounding Scheme

In this section we will present a polynomial-time bounding scheme which yields tight lower and upper bounds on the scheduling problem encountered at a node N_k of the enumeration tree. The problem is to schedule all jobs $J_j \in \overline{\mathcal{P}}_k$, the set of unscheduled jobs, on machines $M_i \in \overline{\delta}_k$, the set of captured machines, such that the schedule length is minimized and each machine M_i processes the already allocated load $l(\mathcal{P}_k^i)$ nonpreemptively. At a node N_k, $|\overline{\mathcal{P}}_k| \leq n$ (some of the jobs may already be fixed) and $|\overline{\delta}_k| \leq m$ (some of the machines may already be discarded). In spite of this fact, for the ease of notation we assume that n denotes $|\overline{\mathcal{P}}_k|$ and m denotes $|\overline{\delta}_k|$.

A closely related decision problem is the bin packing problem with variable bin sizes, $\Pi^{\mathsf{d}}_{\mathsf{vbp}}$. In an instance $I^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ of this bin packing variant we are given

- a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of jobs (or, pieces) of sizes p_j for each $J_j \in \mathcal{J}$, and
- a set $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$ of bins of sizes s_i for each $B_i \in \mathcal{B}$.

The aim is to decide whether or not there exists a feasible packing. A feasible packing can be considered as a partition of the job set $\mathcal{J}$ into m disjoint sets, B_i, $i = 1, 2, \ldots, m$ where the total processing requirement of jobs in B_i is at most s_i for $i = 1, 2, \ldots, m$. If we let $\mathcal{J} = \overline{\mathcal{P}}_k$ where each job $J_j \in \mathcal{J}$ has a size p_j equivalent to its processing time, and $\mathcal{B} = \overline{\delta}_k$ where each bin has a size $s_i = \tau - l(\mathcal{P}^i_k)$ for some time τ, then the scheduling problem at node N_k can be viewed as determining minimum τ^* for which a feasible packing can be obtained.

If we had an polynomial-time procedure for solving the variable-sized bin packing problem, we could use it in a binary search to determine the minimum makespan schedule for the problem at node N_k and we would eliminate the need for branching at this node. Unfortunately, neither the bin packing problem with variable bin sizes nor the scheduling problem at node N_k can be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$. This result directly follows from Theorems 2.3 and 2.4 where the special cases, the bin packing problem with unit bin sizes $\Pi^{\mathsf{d}}_{\mathsf{bp}}$ and $1|P||C_{max}$ are shown to be unary $\mathcal{NP}$-hard, respectively. Nevertheless as before we can use the relation between the scheduling problem encountered at node N_k and the underlying variable-sized bin packing problem to determine tight bounds on the former.

For the bin packing problem with variable bin sizes [Hochbaum and Shmoys 1988] have provided a family of algorithms each of which is an ε-relaxed decision procedure (see Section 2.3.2.2). In their definition, a polynomial-time procedure, which given an instance $I^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ either

(i) produces an ε-relaxed packing, a feasible packing in which the size of each bin B_i, $i = 1, 2, \ldots, m$ is extended to $(1 + \varepsilon)s_i$; or

(ii) outputs "no feasible packing" indicating that there is no feasible packing.

is called as an ε-relaxed decision procedure. Let $\varepsilon - relaxed$ be an ε-relaxed decision procedure. Using the arguments presented in Section 2.3.2.2, it is straightforward to show that the procedure $\varepsilon - makespan$ (Procedure 3.2) determines schedules with their makespan being arbitrarily close to $(1+\varepsilon)C^*_{max}(\mathcal{J},\mathcal{B})$ in $\mathcal{O}$ (log $\lceil ub - lb \rceil$) time. Suppose that the following trivial bounds are used to initialize the binary search: $lb = l_{max}$; $ub = \max\{l_{max}, l_{min} + \sum_{J_j \in \mathcal{J}} p_j\}$ where $l_{min} = \min_{M_i \in \mathcal{B}} l(\mathcal{P}^i_k)$ and $l_{max} = \max_{M_i \in \mathcal{B}} l(\mathcal{P}^i_k)$. The lower bound lb is the maximum allocated load. The upper bound ub is the maximum of lb and the makespan of a schedule in which all jobs in $\mathcal{J}$ are assigned to a machine with minimum load. As we will discuss in Section 3.2.2, it is possible to improve these bounds. Nevertheless they are sufficient to show that the algorithm runs in $\mathcal{O}$ (log $\sum_{J_j \in \mathcal{J}} p_j$), a polynomial time in the binary encoding of the input. Therefore provided that the procedure $\varepsilon - relaxed$ is polynomial, the procedure $\varepsilon - makespan$ is also polynomial in the binary encoding of the input. The following theorem formalizes this result.

Theorem 3.2 *If the procedure $\varepsilon - makespan$ is executed with $\mathcal{O}$ $(\ell + \log\ m)$ iterations of the binary search, the resulting solution has makespan at most $(1+\varepsilon + 2^{-\ell})C^*_{max}(\mathcal{J},\mathcal{B})$.*

Proof:

Initially $ub - lb \le \sum_{J_j \in \mathcal{J}} p_j$ since $l_{max} \ge l_{min}$. After $\ell + \log\ m$ iterations

$$\begin{aligned} ub - lb &\le 2^{-\ell - \log\ m} \sum_{J_j \in \mathcal{J}} p_j \\ &\le 2^{-\ell} \sum_{J_j \in \mathcal{J}} p_j / m \\ &\le 2^{-\ell} C^*_{max}(\mathcal{J},\mathcal{B}) \end{aligned}$$

The schedule produced has makespan at most

Procedure 3.2: An approximate binary search procedure for solving the scheduling problem encountered at a node

arguments
I'_{pms}: an instance of the scheduling problem encountered at a node
lb: a lower bound on the scheduling problem encountered at a node
ub: an upper bound on the scheduling problem encountered at a node

procedure called
$\varepsilon - relaxed$: an ε-relaxed decision procedure

```
procedure $\varepsilon - makespan\ (I'_{\mathsf{pms}}, lb, ub)$
begin
    while $ub - lb \geq 1$
        begin
            $\tau := (ub + lb)/2$
            $s_i := \tau - l(\mathcal{P}^i_k)\ \forall\, i = 1, 2, \ldots, m$
            if $\varepsilon - relaxed\ (\mathcal{J}, \mathcal{B})$ outputs "no feasible packing"
              $lb := \tau$
            else
              $ub := \tau$
        end
    $s_i := \lceil lb \rceil - l(\mathcal{P}^i_k)\ \forall\, i = 1, 2, \ldots, m$
    if $\varepsilon - relaxed\ (\mathcal{J}, \mathcal{B})$ outputs "no feasible packing"
      begin
          output $ub$ and related schedule
          output $\lceil lb \rceil + 1$ as a lower bound
      end
    else
      begin
          output $\lceil lb \rceil$ and related schedule
          output $\lceil lb \rceil$ as a lower bound
      end
end
```

$$\begin{aligned}(1+\varepsilon)ub &= (1+\varepsilon)(ub-lb+lb)\\ &\leq (1+\varepsilon)(2^{-\ell}C^*_{max}(\mathcal{J},\mathcal{B})+C^*_{max}(\mathcal{J},\mathcal{B}))\\ &\leq (1+\varepsilon+2^{-\ell})C^*_{max}(\mathcal{J},\mathcal{B}) \text{ for } \varepsilon\leq 1\end{aligned}$$

In other words, after $\mathcal{O}$ (ℓ+log m) iterations, the resulting solution has makespan at most $(1+\varepsilon+2^{-\ell})C^*_{max}(\mathcal{J},\mathcal{B})$. ■

The ε-approximation scheme (a family of ε-relaxed decision procedures) proposed by [Hochbaum and Shmoys 1988] is not efficient both in terms of the large space requirements and the time complexity. As mentioned in Section 2.3.2.2, they provided a 1/2-relaxed decision procedure which runs in $\mathcal{O}$ (n) if the job sizes are sorted in advance. In Section 3.2.1, we will provide a 1/3-relaxed decision procedure which has the same time complexity as the above 1/2-relaxed decision procedure. Our algorithm is similar to the one due to [Hochbaum and Shmoys 1988], but not identical. We define a polynomial-time procedure (for $\varepsilon > 0$) as an ε-relaxed decision procedure if, given an instance $\mathrm{I}^{\mathrm{d}}_{\mathrm{vbp}} = (\mathcal{J},\mathcal{B})$ of the bin packing problem with variable bin sizes, it either

(i) produces an ε-relaxed packing, that is a feasible packing for the original instance in which the size of each bin B_i, $i = 1,2,\ldots,m$ is extended to $s_i + \varepsilon s_{max}$ for $s_{max} = \max_{B_i\in\mathcal{B}} s_i$; or

(ii) outputs "no feasible packing" indicating that there is no feasible packing.

Let $\varepsilon-relaxed-I$ and $\varepsilon-relaxed-II$ denote the ε-relaxed decision procedures as defined in [Hochbaum and Shmoys 1988] and as above, respectively. Note that any procedure that is $\varepsilon-relaxed-I$ is also an $\varepsilon-relaxed-II$ procedure, but not the vice versa. However our algorithm ($\varepsilon-relaxed-II$) has a better worst case bound than that of $\varepsilon-relaxed-I$. Implementing this procedure in a binary search, the schedules with makespan being at most $(4/3+2^{-\ell})C^*_{max}(\mathcal{J},\mathcal{B})$ can be obtained after $\mathcal{O}$ (ℓ + log m) iterations of the binary search in $\mathcal{O}$ ($n(\ell$ + log $n)$) time. Furthermore, upon its termination, the procedure determines a lower bound which is likely to be tighter than an initial one. In Section 3.2.2 we will discuss the ways of obtaining initial lower and upper bounds for the scheduling problem at node N_k of the enumeration tree. Section 3.2.3 discusses further refinements made in the 1/3-relaxed decision procedure and in the determination of initial bounds. The bounding operations will also be summarized in this Section.

3.2.1 A 1/3-Relaxed Decision Procedure

In the bin packing problem with variable bin sizes, the aim is to determine a feasible packing (if it exists). In a particular instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ of this problem we are given

- a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of jobs of sizes p_j for each $J_j \in \mathcal{J}$, and
- a set $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$ of bins of sizes s_i for each $B_i \in \mathcal{B}$.

In this section we present a 1/3-relaxed decision procedure which, given $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ either produces a 1/3-relaxed packing or, outputs "no feasible packing" in polynomial time. For some $\varepsilon > 0$, an ε-relaxed packing refers to a feasible packing in which each bin B_i $(i = 1, 2, \ldots, m)$ is filled with jobs of sizes totaling at most $s_i + \varepsilon s_{max}$ for $s_{max} = \max_{B_i \in \mathcal{B}} s_i$. Suppose in an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$, all of the processing times and bin sizes are sorted such that $p_1 \geq p_2 \geq \ldots \geq p_n$ and $s_1 \geq s_2 \geq \ldots \geq s_m$, respectively. Furthermore, without loss of generality, suppose that an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}}$ is scaled after dividing all processing times and bin sizes by $s_{max} = s_1$. Let $L[u_1, \ldots, u_k]$ denote the set of k distinct pieces $\{J_{j_1}, \ldots, J_{j_k}\}$, where J_{j_l} $(l = 1, \ldots, k)$ is the largest available job with $p_{j_l} \leq u_l$ and, where $u_1 \leq \ldots \leq u_k$ and $p_{j_1} \leq \ldots \leq p_{j_k}$.

Consider the recursive procedure $1/3 - \mathit{relaxed}$ below:

Procedure 3.3: A 1/3-relaxed decision procedure for the bin packing problem with variable bin sizes

arguments
$\mathcal{J}$: a set of unpacked jobs at the current level of recursion
$\mathcal{B}$: a set of unpacked bins at the current level of recursion
ℓ: the current level of recursion; initially $\ell = m$

procedures called
ordinarypack: a 1/3-relaxed decision procedure for the ordinary bin packing problem, $\Pi^{\mathrm{d}}_{\mathrm{bp}}$(see Procedure 3.4)
1/3 − *relaxed*: a recursive call

```
procedure  $1/3 - relaxed\,(\mathcal{J}, \mathcal{B}, \ell)$
begin
    if $\sum_{J_j \in \mathcal{J}} p_j \leq \sum_{B_i \in \mathcal{B}} s_i$ then
      begin
          if $s_\ell \geq 2/3$ then
            call  $ordinarypack\,(\mathcal{J}, \mathcal{B}, \ell)$
          else
            begin
                $\mathcal{J}_{sml} := \{J_j \in \mathcal{J} | p_j \leq s_\ell/3\}$
                $\mathcal{J}_{new} := \mathcal{J} \setminus \mathcal{J}_{sml}$
                $\mathcal{J}_{fit} := \{J_j \in \mathcal{J}_{new} | p_j \leq s_\ell\}$
                if $\mathcal{J}_{fit} \neq \emptyset$ then
                  begin
                      $\mathcal{J}_{pck} := L[0.5s_\ell, s_\ell]$ (pack this set in bin $B_\ell$)
                      $\mathcal{J}_{new} := \mathcal{J}_{new} \setminus \mathcal{J}_{pck}$
                  end
                $\mathcal{B}_{new} := \mathcal{B} \setminus \{B_\ell\}$
                if $\mathcal{J}_{new} \neq \emptyset$ then
                  call  $1/3 - relaxed\,(\mathcal{J}_{new}, \mathcal{B}_{new}, \ell - 1)$
                while there exists unpacked job $J_j \in \mathcal{J}$
                  find bin $B_i$ packed with $\leq s_i$ and add $J_j$ to $B_i$
            end
    end
  else
    output "no feasible packing"
end
```

Suppose for the time being that the **if** condition in the statement 5 never holds. That is, the difference between the size of a bin B_i ($i = 2, \dots, m$) and the size of the largest bin B_1 is $> 1/3$. The remaining program segment (statements 7 through 25) intends to pack a single bin at each recursive call (level of recursion) of the procedure $1/3 - \textit{relaxed}$ (Procedure 3.3). For example, at the first call the bin B_m is considered, at the second call the bin B_{m-1} is considered and so on. At a level of recursion we define three sets: (i) $\mathcal{J}_{sml}$ is the set of *small* jobs with respect to the bin considered, (ii) $\mathcal{J}_{new}$ is the set of *large* jobs with respect to the bin considered, (iii) $\mathcal{J}_{fit}$ is the subset of large jobs that can fit to the bin considered. Consider a level ℓ of the recursion. Suppose that the procedure $1/3 - \textit{relaxed}$ did not output "no feasible packing" at any one of the previous recursive calls. At this level, if the condition in the statement 3 is not satisfied, then the procedure outputs "no feasible packing" (statement 25) and returns back to the calling procedure. In such a case the calling procedure will also return to its calling procedure and so on. This fact is not indicated in the above procedure in order not to complicate the algorithm. But we assume that if the procedure encounters an output of "no feasible packing" in a return from the **call** statement (statement 19), it returns to the calling procedure with the same output. On the other hand, if the condition in the statement 3 is satisfied, then the current bin $B_{m-\ell+1}$ is packed as shown in statements 14 and 15 and another recursive call is made to pack bin $B_{m-\ell}$ (the case in which the condition in the statement 18 does not hold, will be explained later). As before this bin is either packed or "no feasible packing" is output. Suppose in the first $m - 1$ recursive calls, the procedure $1/3 - \textit{relaxed}$ does not output "no feasible packing" (otherwise the algorithm stops as described before). In such a situation the bins B_m through B_2 have already been packed. Therefore in the current call the bin B_1 is intended to be packed. If the condition in the statement 3 is not satisfied "no feasible packing" is output and the procedure terminates. On the other hand, if the condition $\sum_{J_j \in \mathcal{J}} p_j \le s_1$ is satisfied, then the bin B_1 is packed. Furthermore, in the statement 15 $\mathcal{J}_{new} = \emptyset$ since otherwise there would be no feasible packing at this level. Hence, at most m recursive calls are required before we start to pack the jobs in $\mathcal{J}_{sml}$. Suppose that m recursive calls were necessary to pack all of the large items. Then the jobs in $\mathcal{J}_{sml}$ are packed on the bin B_1 as described in statements 20 and 21. Afterwards the control returns to the statement 20 of the calling procedure and the related set $\mathcal{J}_{sml}$ of jobs is packed on bins B_1 and B_2 as described. The sêts $\mathcal{J}_{sml}$ of each level are packed in the same manner. Finally, at the first level the related set $\mathcal{J}_{sml}$ of jobs is packed on bins $B_1, B_2, \ldots, B_m$ and the procedure terminates.

Suppose that the previously ignored **if** statement (statement 5) holds at some level of recursion. That is, the difference between the size of a bin B_i and the size of the largest bin B_1 is $\leq 1/3$ for some $i = 1, 2, \ldots, m$. In this case, the procedure *ordinarypack*, an ε-relaxed decision procedure (for some $\varepsilon \leq 1/3$) for the ordinary bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, is called for the instance that consists of the remaining unpacked bins and jobs. Details of the procedure *ordinarypack* will be presented later in this section (see Procedure 3.4).

It is claimed that the procedure $1/3-$*relaxed* is a 1/3-relaxed decision procedure. In the proof we will consider two different cases that we mentioned when discussing the flow of the algorithm above. In the first case, we will assume that the condition in the statement 5 never holds, and in the second we assume it holds at some level of recursion. At the end we will state the main theorem combining the above cases. Throughout the proof we will make use of the following principle.

Lemma 3.1 [Hochbaum and Shmoys 1987] *In some feasible packing of an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$, if $\{J_{i_1}, J_{i_2}, \ldots, J_{i_k}\}$ are the only pieces in a bin and $J_{j_1}, J_{j_2}, \ldots, J_{j_k}$ are distinct pieces such that $p_{i_l} \leq p_{j_l}$ for all $l = 1, \ldots, k$, then the instance $\mathrm{I}^{\mathsf{d}\prime}_{\mathsf{vbp}}$ formed by removing $J_{j_1}, J_{j_2}, \ldots, J_{j_k}$ from I, remains feasible for one less number of bins.*

Proof: Take a feasible packing where $\{J_{i_1}, J_{i_2}, \ldots, J_{i_k}\}$ are the only pieces in some bin. Let J_{j_l} be some piece that is in the packing of the remaining bins. Replace J_{j_l} with J_{i_l}. Then the packing on the remaining bins must be feasible since $p_{j_l} \geq p_{i_l}$. As a result, after a finite number of these replacements we get a feasible schedule for the instance $\mathrm{I}^{d'}_{\mathsf{vbp}}$ using one less bin. ■

Case I: $s_\ell < 2/3$ at all levels of recursion

Lemma 3.2 *If an instance* $\mathrm{I}^{d}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ *has a feasible packing then the instance* $\mathrm{I}^{d'}_{\mathsf{vbp}} = (\mathcal{J}_{new}, \mathcal{B}_{new})$ *created by the procedure* $1/3 - relaxed$ *has a feasible packing.*

Proof: If $(\mathcal{J}, \mathcal{B})$ has a feasible packing, then certainly so does $(\mathcal{J} \setminus \mathcal{J}_{sml}, \mathcal{B})$. Consider any such feasible packing. Since all of the pieces in $\mathcal{J} \setminus \mathcal{J}_{sml}$ are greater than $s_m/3$ only two pieces can fit in bin m. This further implies that at least one of these pieces has the size $\leq s_m/2$. Hence, we can conclude that the decision given in the statement 14, packs two largest pieces that can be packed in any feasible packing of the bin B_m. Then due to Lemma 3.1, $\mathrm{I}^{d'}_{\mathsf{vbp}} = (\mathcal{J}_{new}, \mathcal{B}_{new})$ remains feasible. Clearly this proof will hold at all levels of recursion since at each level we will start with a feasible packing for the instance at that level. ■

Lemma 3.3 *If the procedure* $1/3 - relaxed$ *outputs "no feasible packing" then there is no feasible packing.*

Proof: Suppose for a contradiction that there were a feasible packing. Then, by Lemma 3.2, for each recursive call of $1/3 - relaxed$ there is a feasible packing of the specified instance. However, for the failure message to be printed, the last of these instances must have $\sum_{J_j \in \mathcal{J}} p_j > \sum_{B_i \in \mathcal{B}} s_i$. This is clearly a contradiction, since no instance that has greater total piece size than total bin size can have a feasible packing. ■

Lemma 3.4 *If the procedure* $1/3 - relaxed$ *does not output "no feasible packing" then it successfully packs all pieces in a 1/3-relaxed packing.*

Proof: There is only one statement in which the procedure $1/3 - relaxed$ could conceivably fail. In the statement 21, why should it always be possible to find a bin that is packed within its true capacity? If this were not possible, then all bins are packed beyond their true capacity, and thus $\sum_{J_j \in \mathcal{J}} p_j > \sum_{B_i \in \mathcal{B}} s_i$. But this is precisely the situation we have excluded in the case of the **if** statement (statement 3).

To show that the packing produced is 1/3-relaxed, consider two steps in the procedure in which pieces are packed. In the statement 14, we ensure that the size of the packing $\leq (3/2)s_\ell < (4/3)s_1 = (4/3)$ due to the fact that $(1/2)s_\ell < (1/3)$ (see the **if** statement indicated by 5), $s_1 = 1$ and $s_\ell \leq s_1$ (by assumption). In the statement 21, we always add to some bin B_i a piece of size $\leq s_\ell/3 \leq s_i/3$, and since bin B_i previously contained $\leq s_i$, afterwards it contains no more than $(4/3)s_i$. ■

Case II: $s_\ell \geq 2/3$ at a level of recursion

Suppose that the condition $s_\ell \geq 2/3$ of the statement 5 holds at some level of recursion, ℓ. At this level the bins $B_{\ell+1}, \ldots, B_m$ are filled with the related large jobs as described in the previous case. An instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ for the related variable-sized bin packing problem consists of the remaining unpacked bins (i.e. bins B_1 through B_ℓ) and jobs $J_j \in \mathcal{J}$. Since it is assumed that the bin sizes are sorted as $s_1 \geq s_2 \geq \ldots \geq s_m$ and normalized after dividing by the largest bin size s_1, the remaining unpacked bins have sizes $s_i \geq 2/3 \;\; \forall\, i = 1, 2, \ldots, \ell$.

In a related problem, given a finite number (say ℓ) of bins with equal (unit) sizes and a set $\mathcal{J}$ of jobs with sizes as before, it is aimed to determine a feasible packing if it exists. As introduced before, this is the (ordinary) bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$. Let *ordinarypack* be an ε-relaxed decision procedure. Given an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}} = (\mathcal{J}, \mathcal{B}) = (\mathcal{J}, \ell)$ (the last notation is valid since the bin sizes are unit) of this problem, the procedure *ordinarypack* either outputs "no feasible packing" indicating that there is no feasible packing or determines an ε-relaxed packing in which large jobs are packed feasibly (i.e. in a bin, sum of the sizes of jobs with $p_j > \varepsilon$ does not exceed the bin capacity).

Lemma 3.5 *At a level ℓ of the recursion, if $s_\ell \geq 1 - \varepsilon$ (for some $\varepsilon > 0$), then ordinarypack can be used as an ε-relaxed decision procedure for the bin packing problem with variable bin sizes.*

Proof: Suppose that the procedure *ordinarypack* outputs "no feasible packing" for an instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}} = (\mathcal{J}, \ell)$ of the ordinary bin packing problem. Consider the related instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}} = (\mathcal{J}, \mathcal{B})$ in which we are given the same set $\mathcal{J}$ of jobs and number of bins as before. If the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}$ has no feasible packing then so does the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}}$ since in the latter a bin B_i $i = 1, \ldots, \ell$ has a size $s_i \leq 1$. If the procedure *ordinarypack* outputs "no feasible packing" then there is no feasible packing for the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}}$.

On the other hand, suppose that the procedure *ordinarypack* does not output "no feasible packing". Then the condition $\sum_{J_j \in \mathcal{J}} p_j \leq \sum_{B_i \in \mathcal{B}} s_i$ of the statement 3 is satisfied and the procedure *ordinarypack* produces an ε-relaxed packing, $\mathcal{P}_\varepsilon$, for the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}$. Consider a set $\mathcal{J}' = \{J_{\alpha_1}, \ldots, J_{\alpha_{\ell-1}}\}$ of artificial jobs with processing times $p_{\alpha_1} = s_1 - s_\ell$, $p_{\alpha_2} = s_1 - s_{\ell-1}, \ldots, p_{\alpha_{\ell-1}} = s_1 - s_2$. For an artificial job J_{α_i} $(i = 1, \ldots, \ell - 1)$, the processing time $p_{\alpha_i} \geq 0$ since we assume that the bin sizes are ordered such that $s_1 \geq s_2 \geq \ldots \geq s_{\ell-1} \geq \ldots s_m$. By adding the processing times of all of the artificial jobs to the both sides of the condition of the statement 3 and by using the assumption that the bin sizes are normalized such that $s_1 = 1$, we obtain $\sum_{J_j \in \mathcal{J}} p_j + \sum_{J_j \in \mathcal{J}'} p_j \leq \ell$. The last inequality implies that in the packing $\mathcal{P}_\varepsilon$, artificial jobs will certainly find a bin containing ≤ 1 since otherwise we obtain a contradiction that $\sum_{J_j \in \mathcal{J}} p_j > \ell \geq \sum_{B_i \in \mathcal{B}} s_i$. Furthermore $p_{\alpha_i} = s_1 - s_{\ell-i+1} = 1 - s_{\ell-i+1} \leq 1 - s_\ell \leq \varepsilon \ \forall\, i = 1, \ldots, \ell - 1$. As a result even if we pack artificial jobs, $\mathcal{P}_\varepsilon$ remains as an ε-relaxed packing since each bin will contain no more than $1 + \varepsilon = (1 + \varepsilon)s_1$. Moreover artificial jobs can arbitrarily be packed on $\ell - 1$ distinct bins since the procedure *ordinarypack* is assumed to pack large jobs feasibly. Hence the packing $\mathcal{P}_\varepsilon$ is also an ε-relaxed packing for the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}}$ since a processing time p_{α_i} $(i = 1, \ldots, \ell - 1)$ is the extra capacity provided for the bin B_i in the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{bp}}$ as compared with the capacity of the same bin in the instance $\mathrm{I}^{\mathsf{d}}_{\mathsf{vbp}}$. ■

The above two cases are incorporated in the following theorem which is presented without proof since it is simply an application of previous cases.

Theorem 3.3 *Provided that the procedure ordinarypack is an ε-relaxed decision procedure for some $\varepsilon \leq 1/3$, the procedure $1/3-relaxed$ is a 1/3-relaxed decision procedure for the bin packing problem with variable bin sizes.*

The problem is then to find out a procedure *ordinarypack* as described in the above theorem. For such a procedure we could use one of the procedures developed by [Hochbaum and Shmoys, 1987] to solve the ordinary bin packing problem, Π^{d}_{bp}. However none of these algorithms guarantees a feasible packing of the large jobs which we require the procedure *ordinarypack* to do.

Procedure 3.4 is claimed to be a 1/3-relaxed decision procedure for the equal-sized bin packing problem. Suppose that we are given a normalized problem instance I^{d}_{bp} in which the bin sizes are unity. For the ease of reference we repeat the statement 3 of the procedure $1/3-relaxed$ in the statement 3 of the *ordinarypack* (Procedure 3.4).

Lemma 3.6 *If an instance $I = (\mathcal{J}, \ell)$ has a feasible packing then the instance $I_{new} = (\mathcal{J}_{new}, \ell - 1)$ created by the procedure largepack has a feasible packing.*

Proof: Consider the procedure *largepack*. If $(\mathcal{J}, \ell)$ has a feasible packing, then certainly so does the instance $(\mathcal{J} \setminus \mathcal{J}_{sml}, \ell)$. Consider any such feasible packing. Since all of the pieces in $\mathcal{J} \setminus \mathcal{J}_{sml}$ are greater than 1/3 only two pieces can fit in bin ℓ. This further implies that at least one of these pieces has the size $\leq 1/2$. If the condition in the statement 5 is satisfied then we will have at least one piece J_j with $p_j > 0.5$. In any feasible packing this piece can be packed with at most one other piece. $L[1 - p_j]$ is the largest piece that J_j fits with. Then due to Lemma 3.1, $I_{new} = (\mathcal{J}_{new}, m - 1)$ remains feasible. If the condition in the statement 5 is not satisfied, the two largest jobs we can pack is $L[0.5, 0.5]$. Again property 3.1 shows that $I_{new} = (\mathcal{J}_{new}, m - 1)$ remains feasible. Clearly this proof will hold at

Procedure 3.4: A 1/3-relaxed decision procedure for the ordinary bin packing problem, $\Pi^{\mathsf{d}}_{\mathsf{bp}}$

arguments
$\mathcal{J}$: a set of unpacked jobs at the current level of recursion
$\mathcal{B}$: a set of unpacked bins at the current level of recursion
ℓ: the current level of recursion

called from Procedure 3.3

procedure called
largepack: a recursive procedure which packs large pieces of the ordinary bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, feasibly

```
procedure ordinarypack ($\mathcal{J},\mathcal{B},\ell$)
begin
     if $\sum_{J_j \in \mathcal{J}} p_j \le \sum_{B_i \in \mathcal{B}} s_i$ then
       begin
            $\mathcal{J}_{sml} := \{J_j \in \mathcal{J} | p_j \le 1/3\}$
            $\mathcal{J}_{new} := \mathcal{J} \setminus \mathcal{J}_{sml}$
            if $\mathcal{J}_{new} \neq \emptyset$ then call largepack ($\mathcal{J}_{new},\ell$)
            while there exists unpacked $J_j \in \mathcal{J}$
                 find $B_i$ with $\le 1$ and add $J_j$ to $B_i$.
       end
     else
       output "no feasible packing"
end
```

all levels of recursion since at each level we will start with a feasible packing for the instance at that level. ■

Procedure 3.5: A recursive procedure which packs large pieces of the ordinary bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, feasibly

arguments
$\mathcal{J}$: a set of unpacked jobs at the current level of recursion
ℓ: the current level of recursion

called from Procedure 3.4

procedure called
largepack: a recursive call

```
procedure largepack ($\mathcal{J}, \ell$)
begin
      if $\sum_{J_j \in \mathcal{J}} p_j \leq \ell$ then
        begin
              if there exists $J_j \in \mathcal{J}$ with $p_j \in (0.5, 1]$ then
                $\mathcal{J}_{pck} := \{J_j\} \cup L[1 - p_j]$
              else
                $\mathcal{J}_{pck} := L[0.5, 0.5]$
              $\mathcal{J}_{new} := \mathcal{J}_{new} \setminus \mathcal{J}_{pck}$
              if $\mathcal{J}_{new} \neq \emptyset$ then call largepack ($\mathcal{J}_{new}, \ell - 1$)
        end
      else
        output "no feasible packing"
end
```

Lemma 3.7 *If the procedure largepack outputs "no feasible packing" then there is no feasible packing.*

Proof: Same arguments as in Lemma 3.3 applies. ■

Lemma 3.8 *If the procedure largepack does not output "no feasible packing" then it successfully packs all pieces in a feasible packing.*

Proof: In any one of the cases under the **if** statement (statement 5), a packing cannot contain greater than 1. ■

Corollary 3.1 *If the procedure ordinarypack does not output "no feasible packing" then it successfully packs all pieces in a 1/3-relaxed packing.*

Proof: Due to Lemma 3.5. ■

Given the bin and job sizes in sorted order, the procedure $1/3 - \textit{relaxed}$ with the above procedure *ordinarypack* runs in $\mathcal{O}(n)$. Consider the pointers to the sorted list of job sizes: (i) one to the largest piece that can fit on the current bin, (ii) one to the largest piece that is no longer than half of the current bin size, (iii) one to the largest piece that is no longer than one third of the current bin size and (iv) one to the largest piece that is no longer than 1 minus the size of the item pointed by (i). Thus the procedure $1/3 - \textit{relaxed}$ packs "large" pieces in bins of increasing bin size and packs "small" pieces in bins of decreasing bin size. As a result $\mathcal{O}(n)$ is required to maintain above pointers (first three of which applies to $1/3 - \textit{relaxed}$ and all applies to *largepack*). Given these pointers the procedure can easily be implemented in linear time.

3.2.2 Initial Bounds

As initial lower and upper bounds get tighter, the binary search requires less computation time. At a node N_k of the enumeration tree we determine an initial lower bound as the maximum of the three lower bounds. The first lower bound $lb_0(\mathcal{P}_{k_d})$ refers to the lower bound determined at the parent node N_k. It is assumed that at the root N_0, $lb_0(\mathcal{P}_{k_0} \geq lb = \max\{l(\mathcal{J})/m, \max_{J_j \in \mathcal{J}} p_j\}$ which is the lower bound given by [McNaughton 1959]. In this bound, each term of the outer maximum requires $\mathcal{O}\ (n)$ time. Therefore it can be determined in $\mathcal{O}\ (n)$ time. This lower bound gives the average time required to finish all jobs. Hence it ignores the variation in the processing time data.

For the time being consider node N_0 of the enumeration tree. The following lower bound is expected to be dominating when n mod $m \neq 0$ and processing time data has less variation:

Lemma 3.9 *Suppose that jobs are sorted in a nondecreasing order of their processing times.*

$$lb_1(\mathcal{P}_0) = \sum_{j=1}^{\lceil n/m \rceil} p_j \tag{3.2}$$

is a lower bound on $1|P||C_{max}$.

Proof: In an optimal schedule $\mathcal{P}_*$ to $1|P||C_{max}$ at least $\lceil n/m \rceil$ jobs will be scheduled on at least one of the m machines. Then for some machine M_i, the inequality $lb_1(\mathcal{P}_0) \leq l(\mathcal{P}_*^i) \leq C^*_{max}$ proves the statement of the lemma. ■

The main effort in determining the above lower bound is due to the sort step. Hence the time complexity of determining $lb_1(\mathcal{P}_0)$ is $\mathcal{O}\ (n \log\ n)$. Notice that $lb_1(\mathcal{P}_0) < lb$ when n mod $m = 0$ since $m(p_1+p_2+\ldots+p_{(n/m)}) < (p_1+p_2+\ldots+p_n)$.

In the lower bound below we intend to incorporate processing time variability. Let $J_{min} \in \overline{\mathcal{P}}_0$ be the job with minimum processing time and *lower* be a lower bounding procedure.

Lemma 3.10

$$lb_2(\mathcal{P}_0) = \max_{J_j \in \overline{\mathcal{P}}_0 \setminus \{J_{min}\}} \{ \min \{p_j + p_{min},\ lower(\overline{\mathcal{P}}_0 \setminus \{J_j\}, n-1, m-1) \} \} \tag{3.3}$$

is a lower bound on $1|P||C_{max}$.

Proof: In an optimal schedule $\mathcal{P}_*$ to $1|P||C_{max}$ either at least one job J_t is scheduled on the same machine that a particular job J_j ($J_j \in \overline{\mathcal{P}}_0$ and $J_t \in$

$\overline{\mathcal{P}}_0 \setminus \{J_j\})$ has been scheduled or none. In the former $C^*_{max} \geq p_j + p_{min}$ whereas in the latter $C^*_{max} \geq lower(\overline{\mathcal{P}}_0 \setminus \{J_j\}, n-1, m-1)$. Hence C^*_{max} is greater or equal to the minimum of these two quantities. This in turn implies that $lb_2(\mathcal{P}_0)$ is a lower bound. ■

The computational effort required to determine $lb_2(\mathcal{P}_0)$ depends on the time complexity of the lower bounding procedure *lower*. If the lower bounds proposed above are used then time complexity will be $\mathcal{O}\ (n \log\ n)$ since only one sort will be enough.

Corollary 3.2 $lb(\mathcal{P}_0) = \max\ \{\ lb_0(\mathcal{P}_0),\ lb_1(\mathcal{P}_0),\ lb_2(\mathcal{P}_0)\ \}$ *is a lower bound on* $1|P||C_{max}$.

Lower bound computations at node N_k $(k > 0)$ of an enumeration tree are similar. If we let M_{min} be the machine with least completion time at node N_k, then a lower bound at node N_k can be determined as shown in the following theorem.

Theorem 3.4 *Let* $\mathcal{A} = \{J_{\alpha_t} \mid J_{\alpha_t} \in \overline{\mathcal{P}}_k$ *with* $p_{\alpha_t} = l(\mathcal{P}^t_k) - l(\mathcal{P}^{min}_k)$ *for* $M_t \neq M_{min}\}$ *with* $|\mathcal{A}| \leq m-1$ *denote the set of artificial jobs at node* N_k. *Then*

$$lb(\mathcal{P}_k) \;=\; l(\mathcal{P}^{min}_k) + lb(\overline{\mathcal{P}}_k \cup \mathcal{A}) \qquad (3.4)$$

Proof: Without loss of generality consider a partial schedule $\mathcal{P}_k$ for a three-identical parallel machine scheduling problem as shown in Figure 3.3. In this figure, shaded parts of $\mathcal{P}^2_k$ and $\mathcal{P}^3_k$ are the extra processing allocated to machines M_2 and M_3, respectively. If we define two new jobs J_{α_1} and J_{α_2} with processing times $p_{\alpha_1} = l(\mathcal{P}^2_k) - l(\mathcal{P}^1_k)$ and $p_{\alpha_2} = l(\mathcal{P}^3_k) - l(\mathcal{P}^1_k)$, then by treating the set $\overline{\mathcal{P}}_k \cup \{J_{\alpha_1}, J_{\alpha_2}\}$ as if it was $\overline{\mathcal{P}}_0$ and by applying the previous corollary we obtain a lower bound $lb(\mathcal{P}_k)$. Hence Equation (3.4) is a lower bound at node N_k. ■

An upper bound $ub(\mathcal{P}_k)$ is assumed to be the incumbent value Z. Clearly $ub(\mathcal{P}_k)$ need not be an upper bound on the scheduling problem at node N_k. It only refers

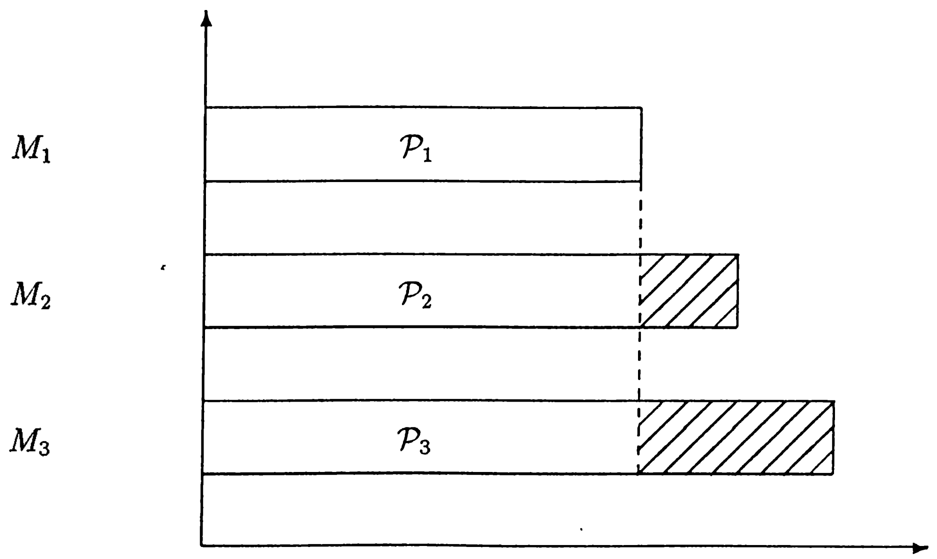


Figure 3.3: A partial schedule for a three-identical parallel machine problem

to a bin capacity for which no feasible packing may exist. For that reason at node N_k, the procedure $1/3 - relaxed$ (Procedure 3.3) is called to check whether it outputs "no" or not. If the answer is "no" then it is certain that we cannot obtain a schedule which finishes by time Z from any one of the descendants of the current node. Hence this node can be fathomed without applying the bounding procedure further. If else Z is assumed to be the initial upper bound for the binary search procedure. At node N_0 we determine the incumbent value by applying the list scheduling heuristic LPT (see Section 2.3.1).

3.2.3 Bound Computations at a Node

Procedure 3.6 determines the bounds on the scheduling problem encountered at node N_k. Clearly the run time of the procedure *bounds* is determined by that of $\varepsilon - makespan$ which is $\mathcal{O}$ $(n(k + \log\ n))$ for $\mathcal{O}$ $(k + \log\ m)$ iterations of the binary search.

The statement 3 of the procedure $1/3 - relaxed$ is important in determining tight

Procedure 3.6: The bounding procedure used at a node of the enumeration tree

arguments
$\mathcal{P}_k$: a set of jobs fixed (scheduled) at node N_k
$LB(\mathcal{P}_k)$: a lower bound on the scheduling problem encountered at node N_k
$UB(\mathcal{P}_k)$: an upper bound on the scheduling problem encountered at node N_k

called from Procedure 3.1

procedures called
$1/3-relaxed$: a 1/3-relaxed decision procedure for the bin packing problem with variable bin sizes (see Procedure 3.3)
$\varepsilon-makespan$: an approximate binary search procedure for solving the scheduling problem encountered at node N_k (see Procedure 3.2)

```
procedure  bounds $(\mathcal{P}_k, LB(\mathcal{P}_k), UB(\mathcal{P}_k))$
begin
    $s_i := Z - l(\mathcal{P}_k^i)\ \forall\, i = 1,2,\ldots,m$
    if $1/3-relaxed\,(\mathcal{J},\mathcal{B},m)$ outputs "no feasible packing" then
      fathom this node
    else
      call $\varepsilon - makespan\ (\mathrm{I}'_{\mathsf{pms}}, lb(\mathcal{P}_k), Z)$
end
```

lower bounds at a node N_k. The tighter the condition the better the quality of the lower bound on $1|P||C_{max}$. The reason is that if this condition is satisfied at an iteration of an binary search procedure $\varepsilon - relaxed$, then the lower bound is updated and thus becomes tighter. The condition in this statement can be considered as a generalization of the lower bound lb for $1|P||C_{max}$ (see Lemma 2.3). To improve both the quality of the lower bound and the run time of the algorithm we have used the following conditions in statement 3 in addition to the current condition:

(i) at a recursive call, if there are $> \ell$ (referring to the number of unpacked bins) large jobs (with $p_j > 1/2$) among unscheduled ones, then at least two large jobs have to be packed in one bin. Hence, there is "no feasible packing". This condition decreases the computational time requirement of the procedure $1/3 - relaxed$ since it may determine an infeasibility that the original condition in statement 3 determines after packing several bins.

(ii) at a recursive call, if there are $> 2\ell$ unscheduled medium jobs (with $1/3 < p_j \leq 1/2$), then at least three medium jobs will be packed in one bin. Hence, the output is "no feasible packing". This condition decreases the computational time requirement of the procedure $1/3 - relaxed$ in the same way as before.

(iii) at a recursive call, determine lb_1 (see Section 3.2.2) considering the remaining unscheduled jobs and unpacked bins. If lb_1 exceeds the size of the largest bin (=1), then in a packing at least one bin cannot have a feasible packing. Hence output "no feasible packing". This condition increases the quality of the bound and decreases the computation time of the algorithm.

The use of the above rules does not affect the worst case bound of the algorithm $\varepsilon - makespan$. Furthermore together with the original condition in the statement 3, they can be used to update the lower bound of the parent node or to fathom the remaining nodes (which have the same parent with node N_k) that are not generated yet. Suppose that at an iteration of the binary search where we were packing the bin B_i, we have ended up with the message "no feasible packing". In this case if the index i is less than the machine index we are currently branching on (i.e. if i is the index of a machine that is branched on previously) then on the remaining nodes we will have the same infeasibility message due to the same bin B_i. If bin sizes are set Z, the incumbent value, then we can fathom the remaining nodes. If else we can update the lower bound on the remaining nodes which is equivalent to updating the lower bound of the parent node.

To improve the quality of the upper bound that the procedure $\varepsilon - makespan$ produces, a refinement in the algorithm is necessary. Although the procedure

$\varepsilon - makespan$ produces a schedule with makespan being arbitrarily close to $(1+\varepsilon)C^*_{max}(\mathcal{J},\mathcal{B})$, it might not output a better schedule even it determines such a schedule at an iteration of the binary search. This is due to the fact that ub in the procedure $\varepsilon - makespan$ refers to a particular bin capacity at some packing attempt but not necessarily to the makespan of a feasible schedule. In fact for a ub, a feasible schedule is obtained by the procedure $\varepsilon - relaxed$ and may have a makespan of at most $(1+\varepsilon)ub$. For that reason, for some ub_1 and ub_2 (where $ub_1 > ub_2$) that have been updated as such in two different iterations of the binary search, makespan of the schedule produced by ub_1 may be less than the one due to ub_2. In such a case the binary search procedure outputs the schedule produced by ub_2. However with a slight modification in the procedure $\varepsilon - makespan$, the best schedule generated so far can be stored and updated without any additional computation time. This will increase the mean performance of the procedure.

3.3 Search Strategy

The strategy used when searching the enumeration tree affects the performance of the branch and bound procedure. As pointed out in Section 3.1 the size of the tree is affected by the selection of a particular job for branching at a level. As a matter of fact, the example in the Figure 3.2 can be generalized to show that the number of leaf nodes is minimized if at each level of the enumeration tree the job with the maximum processing time, among the unscheduled jobs, is selected for branching. Moreover the performance of the procedure $1/3 - relaxed$ decreases if large pieces (with $p_j > 1/2$) and medium pieces (with $1/3 < p_j \leq 1/2$) exist at the same time to be packed into the same bin (see statement 14 of Procedure 3.3). For this reason selecting the job with maximum processing time among the unscheduled jobs, helps us to increase the quality of bounds at a node and to decrease the size of the enumeration tree. The procedure *selectajob* uses this rule.

After all of the descendant nodes is generated, a node has to be selected for further enumeration (at the beginning node N_0 is selected). The procedure *selectanode* applies backtracking strategy with the following hierarchy of selection rules:

1. $\min UB(\mathcal{P}_k)$, choose the node in descendants of which there is a feasible solution with its makespan being close to the incumbent value.
2. $\max LB(\mathcal{P}_k)$, break any tie in the first rule by choosing the node in descendants of which there is a feasible solution with its makespan being close to optimum.
3. choose the first node in the order that the branches were generated, break any tie in the second rule by choosing the node which has a more potential to generate optimal solution. The way we generate the nodes is such that the schedule associated with the first generated node has more chance to finish earlier.

3.4 Computational Experience

In $1|P||C_{max}$ there are three *factors* which seem to be the most important in affecting the performance of the algorithm:

(i) the proper divisibility of the ratio of number of jobs to number of machines (n/m ratio),

(ii) the magnitude of the n/m ratio,

(iii) the variability among processing times

The impact of the first factor is related with the deteriorating quality of the lower bound presented in Lemma 2.3. For problem instances in which n/m is not *divisible*, the lower bound gets worse. As a matter of fact in the literature the examples that are given to show the worst case ratio of an algorithm is selected among those instances in which n/m is *non-divisible*. For this factor what one may expect in terms of the performance of an optimizing algorithm is that the run time of the algorithm increases if this ratio becomes non-divisible since the algorithm will have a difficulty in proving the optimality.

The magnitude of the n/m ratio directly affects the size of an enumeration tree. In Table 3.1 it can be seen that the size of the tree decreases as the n/m ratio gets *low* or *high*. The problem becomes difficult as this ratio becomes *medium*. Hence, for instances in which the n/m ratio is medium the performance of an optimizing algorithm is expected to deteriorate.

The last factor is the processing time variability. As the variability decreases many alternative solutions to a problem exist. If a lower bounding scheme cannot differentiate among these alternatives then surely the performance of an optimizing algorithm decreases. If on the other hand, *high* variability is present in an instance then any wrong selection of a node causes the optimizing algorithm to spend much time in searching irrelevant parts of the enumeration tree. Hence, we expect the performance of an optimizing algorithm to improve in *medium* processing time variability cases.

The factors and the related levels that we consider in this study are shown in Table 3.2. According to the notation given in the table, NDHL represents a problem structure in which n/m is non-divisible, n/m ratio is high and process variability is low. Similarly DMH represents a structure in which n/m is divisible, n/m ratio is medium and process variability is high. For each of the 18 different problem structures, 50 problem instances are considered. As shown in Table 3.2, the problem parameters are generated from *uniform distributions*. Considering real world examples, it may be argued that the uniform distribution is not suitable to generate some of the problem parameters. For instance, it can be claimed that in many cases processing times are distributed *exponentially*. However, the aim of this empirical analysis is to test the performance of the algorithm for any problem instance without any assumption on its structure.

To be able to measure the performance of the branch and bound algorithm, for

Table 3.2: Factors and related levels considered in the experimental design

$x \sim U_d(a,b)$ shows that x is a *discrete random variable* distributed *uniformly* between a and b. x is generated as $a + \lfloor b - a + 1 \rfloor y$ where y is a *continuous random variable* distributed uniformly between 0 and 1

In "n/m divisible" case, if the generated n and m values are not divisible then n is increased till n/m becomes divisible. If n exceeds b then new n and m are generated and the above procedure is applied again.

In "n/m non-divisible" case if the generated n and m values are divisible then n is increased till n/m becomes non-divisible. If n exceeds b then new n and m are generated and the above procedure is applied again.

Factors	Levels		
n/m divisibility	divisible (D)	non-divisible (ND)	
n/m ratio	low (L) $n \sim U_d(100,120)$ $m \sim U_d(50,80)$	high (H) $n \sim U_d(100,120)$ $m \sim U_d(2,10)$	medium (M) $n \sim U_d(100,120)$ $m \sim U_d(20,50)$
processing variability	low (L) $p_j \sim U_d(5,10)$	high (H) $p_j \sim U_d(5,300)$	medium (M) $p_j \sim U_d(5,50)$

each problem structure we collect statistics on the following indicators

1. First encountered time: the *cpu* time elapsed to determine the solution that the algorithm delivers upon its termination.

2. First encountered node: the number of nodes enumerated to determine the solution that the algorithm delivers upon its termination.

3. Total cpu time.

4. Total number of nodes enumerated.

5. Initial gap: The minimum of the relative difference between the upper bound ub, and the lower bound lb (i.e. $(ub - lb)/lb$) determined at the root node and at the node N_1.

6. Ending gap: The relative difference between ub and lb upon the termination of the algorithm.

Table 3.3: A sample output

NDML			
Average:	4.72214		518.6818
Std.Dev.:	22.77753		688.9690
First Encountered Time (sec)		Total Time (sec)	
10	48	10	27
40	1	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	1	190	1
	0		22

The branch and bound algorithm has been coded in the computer language C. The experiments were conducted on the computer SUN SPARC SERVER 490 (22 MIPS). Results of the analysis are shown in Appendix C (Tables C.1 through C.3)

In these tables *mean* and *standard deviation* of each indicator are given. Moreover *percentile-like* information for each indicator is given. The information in Table 3.3 for example is read as, out of 50 problems, 48 times the first encountered time was less than 10 seconds, once it was between 10 and 40 seconds and, once it was between 160 and 190 seconds.

In summary, over 900 randomly generated problems

- 702 problems are solved in less than 10 seconds; in 15 problems the solution time (in cpu seconds) is in the range (10,40]; in 2 problems it is in (40,70]; in 4 problems it is in (70,100]; in 2 problems it is in (160,190] and in 7 problems it exceeds 190 seconds. 168 (19%) problems remain unsolved (since the algorithm stops after 10^6 nodes are enumerated).

- upon the termination of the algorithm the relative difference between the upper and lower bounds computed is always less than 10%. In 732 problems it is zero; in 143 problems it is in the range $(0, 0.02]$; in 17 problems it is in (0.02,0.06]; for 1 problem it is in $(0.06, 0.08]$ and in 3 problems it is in

(0.08, 0.1].

- upon the termination of the algorithm, the mean of the relative difference between the upper and lower bounds computed is at most 0.019 (the case NDML). Other than this case it is at most 0.0035.
- upon the termination of the algorithm the overall mean and standard deviation (for 900 problems) of the relative difference between the upper and lower bounds are 0.001935 and 0.008446, respectively.
- first encountered time is at most 505.263 cpu seconds (945,010 nodes). In 831 problems first encountered time is less than 10 seconds; in 44 problems it is in the range (10, 40]; in 3 problems it is in (40, 70] in 7 problems it is in (70, 100]; for 1 problem it is in (100, 130]; in 2 problems it is in (130, 160]; in 3 problems it is in (160, 190] and in 9 problems it exceeds 190.
- the first encountered time is at most in 38.55 seconds (the case DMH).
- the overall mean and standard deviation (for 900 problems) of the first encountered times are 6.33 and 32.23, respectively. The overall mean and standard deviation of the first encountered nodes are 13,273.07 and 74,668.58, respectively.

At the beginning of this section, we have provided an intuitive explanation of how each of the problem parameters is expected to affect the performance of the algorithm. In identifying whether or not the performance of the algorithm is sensitive to varying problem parameters (and if so, the direction of trends) with respect to each indicator, we make use of the *Three-Factor Fixed Effects ANOVA* model [Montgomery, 1984]

$$y_{ijkl} = \mu + \tau_i + \beta_j + \gamma_k + (\tau\beta)_{ij} + (\tau\gamma)_{ik} + (\beta\gamma)_{jk} + (\tau\beta\gamma)_{ijk} + \epsilon_{ijkl}$$

where $i = 1, 2$ (divisible, non-divisible); $j = 1, 2, 3$ (low n/m, high n/m, medium n/m); $k = 1, 2, 3$ (low variability, high variability, medium variability) and $l = 1, 2 \ldots, 50$ (sample size). ANOVA tables are presented in Appendix C (Tables

Table 3.4: The percentage of the standardized residuals that fall within the limits

The error terms ϵ_{ijkl} are $N(0, \sigma^2)$ (distributed normally with mean zero and variance σ^2). Hence, the standardized residuals $\epsilon_{ijkl}/\sqrt{MS_E}$ should be approximately normal with mean zero and unit variance. Thus, about 68% of the standardized residuals should fall within the limits ±1, about 95% of them should fall within ±2, and virtually all of them should fall within ±3. Each entry below·shows the percentage that falls within the specified limit for each of the indicator.

	Limits		
Factors	±1	±2	±3
First Encountered Time	93.1	97.9	98.7
Total Time	48.9	92.1	98.1
Initial Gap	6	48.7	73.7
Ending Gap	71.2	94.7	95.2
First Encountered Node	92.8	97.8	98.4
Total Node	37.8	55	89.1

C.4 through C.6). It is seen that the *normality* and *equal variance* assumptions about the error terms (ϵ_{ijkl}) are not satisfied perfectly (see Tables 3.4 and C.1 through C.3, respectively). Therefore, the results of the ANOVA should be loosely interpreted. However, as it is well-known, the F-test of the ANOVA is quite robust to *skewness* of the error distributions and a *balanced design* (that is, an experimental design with equal sample sizes) protects against unequal variances. Hence, the analysis still provides an insight to the problem which is consistent with the results obtained in Tables C.1 through C.3. In particular, the significant contrasts (which may be computed by using Tables C.1 through C.6) among the *treatment* levels agree with the results summarized in Tables C.1 through C.3. The following conclusions are drawn from these tables.

First encountered time (node) is insensitive to the changes in any one of the problem parameters. In fact, from Tables C.1 and C.3, it is possible to observe that the first encountered time (node) is slightly affected by the varying problem parameters. For instance, medium-n/m-ratio case is definitely worse than the other cases. However, in most of the cases, the differences among cpu seconds (and the related number of nodes) are practically negligible.

In terms of the total execution time (number of nodes), the performance of the algorithm varies with the different problem structures. The effect of the magnitude of the n/m ratio on the performance of the algorithm is summarized as $low \succ high \succ medium$ where $x \succ y$ shows that the execution time of the algorithm is shorter (and it enumerates less number of nodes) in case x than in y. The processing time variability affects the performance as $medium \succ low \succ high$. In terms of the n/m divisibility, there is no significant difference between the cases n/m divisible and n/m non-divisible. Except the last observation, these results agree with the intuitive explanations provided at the beginning of this section.

In 290 problems, the list scheduling heuristic *LPT* (see Section 2.3.1) is successful in determining the solution that the algorithm delivers (see Table C.3. The number corresponding to zero-first-encountered-node shows the number of problems in which *LPT* determines the ending solution). In 264 problems (out of 290), the ending solutions are proved to be optimal (see zero-total-#-of-nodes row of Table C.3). The performance of *LPT* is better in low-n/m-ratio and/or low-processing-time-variability cases (in 244 problems the ending solution is determined by *LPT* and 220 of them are proved to be optimal). Moreover, in low-n/m-ratio cases, the 1/3-dual approximation algorithm is successful in improving the solutions determined by *LPT* and in determining optimal solutions without any enumeration (see one-total-#-of-nodes row of Table C.3).

The initial gap is smaller in problems where n/m is divisible than that in n/m-non-divisible problems (see Table C.2). This result indicates that the initial lower bound gets worse (that is, its percent deviation from the upper bound increases) in an n/m-non-divisible case since as discussed above, no such tendency is observed in the initial upper bounds. Moreover, it is observed that the initial gap increases when the n/m ratio is medium which may either be due to the deteriorating performances of upper or lower bounds.

In terms of the ending gap, the behavior of the algorithm is slightly better in n/m divisible case. In n/m divisible cases, the ending gap is insensitive to changes in the parameters n/m ratio and/or processing variability. In n/m non-divisible cases, the ending gap increases when n/m is medium and/or when processing variability is low.

Considering the large (enough) number of jobs generated in a problem instance and, the results associated with the quality of solutions determined at the first encountered time and the length of this time, it can be concluded that the algorithm is robust and from the practical point of view, it solves the parallel machine scheduling problem in a reasonable time.

Chapter 4

Conclusions

The purpose of this study was to investigate the combinatorial aspects of a class of parallel machine scheduling problems, namely $1|P||C_{max}$, and develop a computationally feasible branch and bound algorithm for its exact solution.

After a brief discussion of machine scheduling problems, in Chapter 1, the formal definition of the problem to be investigated in this study was given. This specific class of problems is $1|P||C_{max}$, in which n independent jobs have to be scheduled on m identical parallel machines with the objective of minimizing the schedule length. The characteristics of an optimum solution to $1|P||C_{max}$ were presented in Chapter 2. These characteristics are crucial in developing a branch and bound algorithm. Also in Chapter 2, the previous approaches to this problem and their main drawbacks were discussed. The main chapter, Chapter 3, presented a detailed development of a branch and bound algorithm for $1|P||C_{max}$. The branching scheme was discussed in Section 3.1. It enumerates the set of all nonpreemptive schedules in which none of the machines are idle, without any repetition (see Theorem 3.1). Moreover due to Lemmas 2.2 and 2.4, the length of each schedule generated by the branching scheme is strictly less than twice the lower bound given in Lemma 2.3 even if none of the fathoming rules are applied. The bounding scheme, as discussed in Section 3.2, uses the relationship between

$1|P||C_{max}$ and the bin packing problem Π^d_{bp}. The former problem can be viewed as the bin packing problem with the objective of determining the minimum bin size for which there is a feasible packing. Hence a binary search procedure, such as Procedure 3.2, can used to search a range of possible optimum makespan values. In Section 3.2.1, a (1/3)-relaxed decision procedure was developed to solve the bin packing problem associated with the subproblems that arise in each node of the enumeration tree. This procedure, when used in a binary search, provides tight lower and upper bounds at a node of the tree. Furthermore, this algorithm has applications in its own right to a specific class of parallel machine scheduling problems which are discussed later in this chapter. The search strategy used for the branch and bound tree was presented in Section 3.3. It is basically the depth-first strategy applied with the selection rule: select a node among the ones in the deepest (active) level with the minimum upper bound. In this strategy, the size of the active nodes list remains constant since once a node is selected for branching it is removed from the list. This resolves the memory problem associated with the computer code of the algorithm. The motivation behind the selection of a node with the minimum upper bound is to determine the part of the tree in which the feasible schedule with its makespan being equal to the incumbent value lies. Since the initial incumbent value is determined by the list scheduling algorithm *LPT* (see Section 2.3.1), the incumbent Z, is expected to be close to the minimum makespan. Hence by changing assignments of a few number of jobs in the *LPT* schedule, which is done by the branching scheme, an optimum solution may be found. At a selected node, a job with the maximum processing time among the unscheduled jobs is scheduled (fixed) to reduce the size of the tree. A detailed empirical study was the concern of Section 3.4. In 900 randomly generated problems, it has been observed that 168 (19%) problems remain unsolved. 702 problems were solved in less than 10 cpu seconds. In 831 problems the ending solution (the solution that the algorithm returns upon its termination) was found in less than 10 cpu seconds. In 732 problems, the ending solution turned out to be optimal. In 143 problems, the relative deviation of the ending solution from the ending lower bound was less than 2% and only in 4 problems this deviation is in the interval [6%, 10%]. Further analysis of these results showed that the performance of the algorithm (both in terms of the solution time and the quality of the ending solution found) was not affected by a change in any one of the problem parameters. Hence, it can be concluded that for all practical purposes the branch and bound algorithm solves $1|P||C_{max}$. The classification scheme is given in Appendix A for deterministic scheduling problems. Appendix B provides a glossary of some of the complexity theoretic concepts that were used in the study. The summary of the computational results is given in Appendix C in tabular format.

This chapter mainly deals with the significance and the importance of the results of this study and possible directions for future research.

It is well documented that the classical job shop scheduling problem, $J|||C_{max}$, is one of the most difficult combinatorial problems. In this problem we are given a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ each has to be processed on s machines (stages) $M_1, M_2, \ldots, M_s$. Each job J_j consists of a sequence of o_j operations $\{O_{1j}, \ldots, O_{o_j j}\}$; O_{ij} being the processing of job J_j on machine m_{ij} (one of the machines $M_1, M_2, \ldots, M_s$, which is specified to perform the operation O_{ij}) with $m_{(i-1)j} \neq m_{ij}$ during an uninterrupted time p_{ij}. Then the problem is to determine a processing order on each machine M_ℓ such that the makespan is minimized.

[Conway *et al.* 1967] asserted that "many proficient people have considered this problem, and all have come away essentially empty-handed. Since this frustration is not reported in the literature, the problem continues to attract investigators who just cannot believe that a problem so simply structured can be so difficult until they have tried it". In a similar pessimistic assertion, [Adams *et al.* 1988] stated that "job shop scheduling is among the hardest combinatorial optimization problems. Not only it is $\mathcal{NP}$-hard [Garey and Johnson 1979], but even among members of the latter class it belongs to the worst in practice; we can solve exactly randomly generated traveling salesman problems with 300–400 cities (over 100,000 variables) or set covering problems with hundreds of constraints and thousands of variables, but we are typically unable to schedule optimally ten jobs on ten machines". The history of the so called "notorious" 10×10 job shop problem is quite interesting. The specific instance of this classical job shop problem was given in [Muth and Thompson 1963]. There are 10 jobs each having 10 different routings through 10 machines. This problem instance defied all solution attempts until 1989 when [Carlier and Pinson 1989] solved the problem in an unreasonably large computation time (about four hours).

At the present time the only viable approximation algorithm for the classical job shop problem seems to be the "shifting bottleneck" procedure of [Adams *et al.* 1988] which uses the one-machine lower bound of [Lageweg *et al.* 1978]. Such a bound is determined by relaxing the capacity constraints on all machines except the machine M_i, $i = 1, 2, \ldots, s$. For each operation on machine M_i we can determine a *head* being the earliest start time of this operation, a *tail* being the earliest completion time of all the operations that follow the current operation and a *body* being the processing requirement of this operation. Then treating operations as jobs we obtain a three-stage flow shop problem $F3||\text{nonbottleneck 1st and 3rd stages}|C_{max}$ which is equivalent to $1||r_j, d_j|L_{max}$ [Lenstra 1977; Rinnooy Kan 1976]. This latter problem is usually referred as *the one-machine scheduling problem*. An optimal solution to the related $1||r_j, d_j|L_{max}$ provides a lower bound $LB(M_i)$ on the general problem class $J|||C_{max}$. Then a tighter bound can be determined each time by assuming a different machine as the bottleneck and letting $LB = \max_{1 \leq i \leq s} LB(M_i)$ (See [Roy and Sussmann 1964; Lenstra 1977; Rinnooy Kan 1976; Adams *et al.* 1988] for details, specifically the *disjunctive graph* representation of $J|||C_{max}$ and the related lower bounding issues). Unfortunately $1||r_j, d_j|L_{max}$ is known to be unary $\mathcal{NP}$-hard [Lenstra 1977; Rinnooy Kan 1976]. However [Carlier 1982] has developed a branch and bound algorithm which solves this problem in a reasonable time. This branch and bound procedure is used in the shifting bottleneck procedure of [Adams *et al.* 1988]. If we let $\mathcal{M}_f$ denote the set of machines that are sequenced (initially $\mathcal{M}_f = \emptyset$) then their procedure can be summarized as follows:

Step 1 Solve the one-machine scheduling problem for each $M_i \notin \mathcal{M}_f$. Call the machine M_k with maximum objective (which is makespan for $F3||$nonbottleneck 1st and 3rd stages$|C_{max}$ or equivalently maximum lateness for $1||r_j, d_j|L_{max}$) as the *critical machine*. Fix the optimum sequence on machine M_k. Let $\mathcal{M}_f := \mathcal{M}_f \cup \{M_k\}$. Go to step 2.

Step 2 Reoptimize the sequence of each critical machine $M_i \in \mathcal{M}_f$ while keeping the other (previously fixed) sequences fixed. If $\mathcal{M}_f = \mathcal{M}$ then stop. Otherwise go to step 1.

Rather than identifying one machine as critical at each iteration of the algorithm, [Adams *et al.* 1988] have considered the first k machines selected from a list sorted in nonincreasing order of the objective values of the related one machine problems. Performing this step for each of these k machines, an enumeration tree is obtained. The modified heuristic of [Adams *et al.* 1988] searches the truncated enumeration tree in which the number of critical machines k, considered at a level of the tree decreases as the level increases according to some function which depends on the problem size. This version of the shifting bottleneck heuristic is famous since it solves the 10×10 problem in about five minutes (without proving the optimality).

Clearly, the realistic version of the classic job shop scheduling problem is the multi-stage (flow or job shop) parallel machine scheduling problem. In the contemporary manufacturing environment, CNC machining cells consisting of identical parallel machines (CNCs) and each cell functioning as a stage in a multi-stage manufacturing are common occurrences. Speed of operation and high investment in these modern manufacturing systems make it absolutely necessary to be able to schedule these systems in (almost) real time and with high machine utilization. In this respect, makespan minimization seems quite acceptable in these systems since it can be shown that minimizing makespan results in a maximization in machine utilization levels. Unfortunately none of these problems are solvable in polynomial-time unless $\mathcal{P}$=$\mathcal{NP}$. Even for the simplest production environment flow shops, the following complexity results are obtained.

- $F2|P2, P1|\text{nonbottleneck 2nd stage}|C_{max}$ is $\mathcal{NP}$-hard [Akyel and Benli 1988],
- $F2|Pm_1, Pm_2|\max\{m_1, m_2\} > 1|C_{max}$ is unary $\mathcal{NP}$-hard [Gupta 1988],
- $F2|P2, P1|pmtn|C_{max}$ is $\mathcal{NP}$-hard [Lenstra 1988],
- $F2|Pm_1, Pm_2|no - wait,\ m_j \geq 1 \ \ \forall \ \ j = 1, 2|C_{max}$ is unary $\mathcal{NP}$-hard [Sriskandarajah and Ladet 1986].

The solution procedures proposed in the literature for the multi-stage parallel machine scheduling problems indicate that the problem area is still open (see [Akyel and Benli 1988] for a review). The two conflicting aims require, if exact algorithms are timewise infeasible, good approximate algorithms with acceptable mean or worst case behaviors, that can operate in real time while giving good machine utilization levels.

On the other hand, availability of exact algorithms for multi stage parallel machine scheduling problems are essential for a number of reasons. For one, it is important to have a benchmark to empirically compare the heuristics being developed. Moreover any truncated search of the enumeration tree provides both lower and upper bounds on a particular problem instance using which we can rate the quality of the schedule determined.

The analysis of the solution methodologies for $J|||C_{max}$ suggests that in any extension of the multi-stage single machine problem to identical parallel machines in each stage, the problem $1|P|r_j, d_j|L_{max}$ becomes important. [Carlier 1987] has developed a branch and bound algorithm for this problem. From what has been reported, this procedure is not computationally promising. Hence computationally feasible means of determining bounds for $1|P|r_j, d_j|L_{max}$ are required. The algorithm developed in this study, we believe, provides such means.

Even when we consider the single stage identical parallel machine scheduling case, there are a number of important application areas. As it was mentioned earlier, in contemporary manufacturing systems it is common to come across machining cells consisting of identical CNC machines operating in parallel under the control of a central computer. At any given point in time, the central computer assigns (or, schedules) the jobs that are already in the queue of the cell to the CNCs under its control (usually using makespan minimization as the objective with the intent of improving machine utilization). Reassignment (or, rescheduling) of jobs to CNCs is required at least under two circumstances: when one or more new jobs join the queue, and when machine failure occurs in one or more of the CNCs. In either of the circumstances, the situation is basically the bin packing problem with variable bin sizes for which (1/3)-dual approximation algorithm presented in Section 3.2.1 gives, to our knowledge, the best worst case bound.

Assuming nonpreemptions, when new jobs arrive, the jobs already being processed on the CNCs must continue processing. That is, each CNC will become available at different times in the future. This corresponds to bins of different sizes in the corresponding bin packing problem. The situation is identical when one or more CNCs fail at any point in time. It is customary to assume deterministic repair times for CNCs. Hence, when the repair times are treated as pseudo jobs that tie up the failed CNCs, the problem reduces to the previous case (see Figure 4.1).

Although the (1/3)-dual approximation algorithm gives the best worst case performance bounds for this problem, a rigorous computational study is needed in order to compare its mean behavior against possible other heuristics, such as *LPT*. It should also be noted that the makespan minimization is used only as a surrogate for maximization of machine utilizations. Hence, what we really want is not the optimum makespan, but avoid, possibly rare, worst case occurrences. This mode of operation is akin to the rolling horizon concept in production planning: we do not really expect the stated makespan value, computed at a particular point in time, to be realized, but long before that either new jobs will join the queue, or some CNCs will fail, or both.

In conclusion, the three significant aspects of this study are: (i) a rigorous

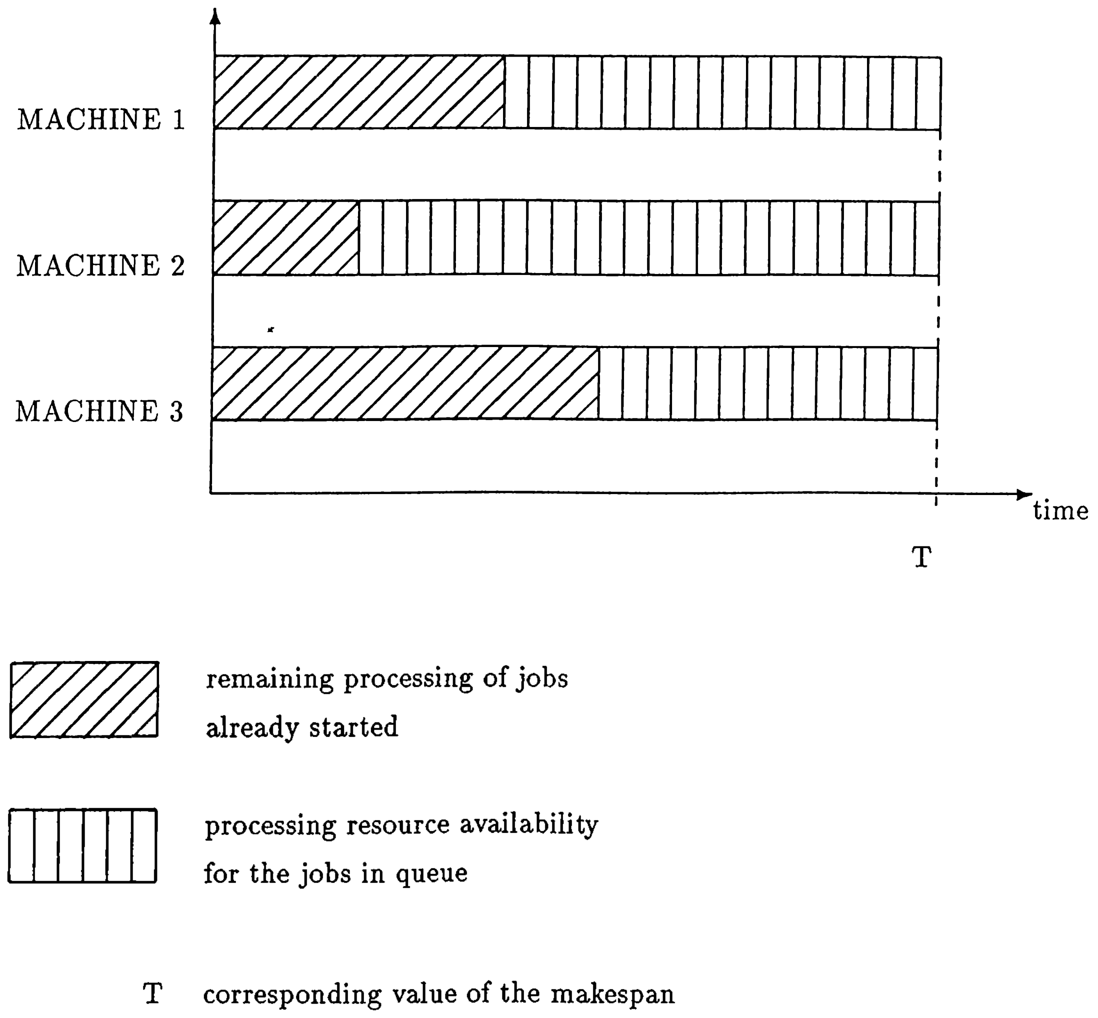

MACHINE 1
MACHINE 2
MACHINE 3
time
T
remaining processing of jobs already started
processing resource availability for the jobs in queue
T corresponding value of the makespan

Figure 4.1: $1|P||C_{max}$ with one or more jobs fixed

complexity theoretic treatment of the class of problems: $1|P||C_{max}$, (ii) development of a (1/3)-dual approximation algorithm for a specific class of problems in $1|P||C_{max}$, and (iii) development of a computationally feasible exact algorithm that effectively utilizes the branching and bounding schemes, and the search strategies in a branch and bound procedure for the class of problems, $1|P||C_{max}$.

References

1. Adams, J., E. Balas and D. Zawack (1988) "The Shifting Bottleneck Procedure for Job Shop Scheduling", *Man. Sci*, Vol.34, No.3

2. Akyel, C. and Ö.S. Benli (1988) "On Scheduling in Parallel Machine Flow Shops", *Working paper*, IEOR-8803, Department of Industrial Engineering, Bilkent University

3. Baker, K.R. (1974) *Introduction to Sequencing and Scheduling*, John Wiley

4. Bellman, R., A.O. Esogbue and I. Nabeshima (1982) *Mathematical Aspects of Scheduling and Applications*, Pergamon Press

5. Blazewicz, J. (1987) "Selected Topics in Scheduling Theory", *Ann. of Disc. Mat.*, Vol.31

6. Blazewicz, J., W. Cellary, R. Slowinski and J. Weglarz (1986) *Scheduling Under Resource Constraints - Deterministic Models, Annals of Oper. Res.*, Vol.17, J.C. Baltzer AG

7. Bratley, P., M. Florian and P. Robillard (1975) "Scheduling with Earliest Start and Due Date Constraints on Multiple Machines", *Nav. Res. Log. Quar.*, Vol.22

8. Bratley, P., M. Florian and P. Robillard (1971) "Scheduling with Earliest Start and Due Date Constraints", *Nav. Res. Log. Quar.*, Vol.18

9. Carlier, J. (1982) "The One-Machine Sequencing Problem", *E.J.O.R.*, Vol.11

10. Carlier, J. (1987) "Scheduling Jobs With Release Dates and Tails on Identical Machines to Minimize the Makespan", *E.J.O.R.*, Vol.29

11. Carlier, J. and E. Pinson (1989) "An Algorithm for solving the Job Shop Problem", *Man. Sci.*, Vol.35, No.2

12. Cheng, T.C.E. and C.C.S. Sin (1990) "A State-of-the-Art Review of Parallel-Machine Scheduling Research", *E.J.O.R.*, Vol.47

13. Coffman, JR.E.G., M.R. Garey and D.S. Johnson (1978) "An Application of Bin-Packing to Multiprocessor Scheduling", *SIAM J. Comput.*, Vol. 7, No. 1

14. Coffman, JR.E.G., G.S. Lueker and A.H.G. Rinnooy Kan (1988) "Asymptotic Methods in the Probabilistic Analysis of Sequencing and Packing Heuristics", *Man. Sci.*, Vol.34, No.3

15. Conway, R.W., W.L. Maxwell and L.W. Miller (1967) *Theory of Scheduling*, Addison Wesley

16. Fisher, M.L. (1982) "Worst-Case Analysis of Heuristic Algorithms for Scheduling and Packing", in: Dempster, M.A.H., J.K. Lenstra and A.H.G. Rinnooy Kan (eds.), *Deterministic and Stochastic Scheduling*, Reidel

17. French, S. (1982) *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*, Ellis Horwood

18. Friesen, D.K. (1978) "Sensitivity Analysis for Heuristic Algorithms", *Report No. UIUCDCS-R-78-939*, Department of Computer Science, University of Illinois at Urbana-Champaign

19. Friesen, D.K. and M.A. Langston (1983) "Bounds for MULTIFIT Scheduling on Uniform Processors", *SIAM J. Comput.*, Vol. 12, No.1

20. Garey, M.R., R.L. Graham and D.S. Johnson (1978) "Performance Guarantees for Scheduling Algorithms", *Oper. Res.*, Vol.26

21. Garey, M.R. and D.S. Johnson (1979) *Computers and Intractability*, Freeman

22. Garey, M.R. and D.S. Johnson (1981) "Approximation Algorithms for Bin Packing Problems: A Survey", in: Ausiello, G. and M. Lucertini (eds.), *Analysis and Design of Algorithms in Combinatorial Optimization*, Springer-Verlag

23. Graham, R.L. (1966) "Bounds for Certain Multiprocessing Anomalies", *Bell Syst. Tech. J.*, Vol.45

24. Graham, R.L. (1969) "Bounds on Multiprocessing Timing Anomalies", *SIAM J. Appl. Math.*, Vol.17, No.2

25. Graham, R.L., E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan (1979) "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey", *Ann. Discrete Math.*, Vol. 5

26. Gupta, J.N.D. (1988) "Two-Stage Hybrid Flow Shop Scheduling Problem", *J. Opl. Res. Soc.*, Vol.39, No.4

27. Hochbaum, D.S. and D.B. Shmoys (1987) "Using Dual Approximation Algorithm for Scheduling Problems: Theoretical and Practical Results", *J. ACM*, Vol. 34, No. 1

28. Hochbaum, D.S. and D.B. Shmoys (1988) "A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach", *SIAM J. Comput.*, Vol. 17

29. Lageweg, B.J., J.K. Lenstra and A.H.G. Rinnooy Kan (1978) "A General Bounding Scheme for the Permutation Flow Shop Problem", *Oper. Res.*, Vol.26, No.1

30. Lageweg, B.J., J.K. Lenstra, E.L. Lawler and A.H.G. Rinnooy Kan (1982) "Computer-Aided Complexity Classification of Combinatorial Problems", *Comm. ACM*, Vol.25, No.11

31. Lawler, E.L. and J. M. Moore (1969) "A Functional Equation and Its Application to Resource Allocation and Sequencing Problems", *Man. Sci.*, Vol. 16

32. Lawler, E.L., J.K. Lenstra and A.H.G. Rinnooy Kan (1982) "Recent Developments in Deterministic Sequencing and Scheduling: A Survey", in: Dempster, M.A.H., J.K.L. Lenstra and A.H.G. Rinnooy Kan (eds.), *Deterministic and Stochastic Scheduling*, Reidel

33. Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan, and D. Shmoys, (1989) "Sequencing and Scheduling: Algorithms and Complexity", to appear in: Graves, S.C., Rinnooy Kan, A.H.G. and Zipkin, P. (eds.), *Handbooks in Operations Research Volume 4: Logistics for Production and Inventory*, North Holland

34. Lenstra, J.K. (1977) *Sequencing by Enumerative Methods*, Mathematisch Centrum

35. Lenstra, J.K. (1988) *Private Communication*

36. McNaughton, R. (1959), "Scheduling with Deadlines and Loss Functions", *Man. Sci.*, Vol.12, No.1

37. Minyi, Y. (1989) "On the Exact Upper Bound for the MULTIFIT Processor Scheduling Algorithm", *Report No. 88547-OT*, Institut für Ökonometrie und Operations Research, Universität Bonn

38. Montgomery, D.C. (1984) *Design and Analysis of Experiments*, John Wiley & Sons

39. Muth, J.F. and G.L. Thompson (1963) *Industrial Scheduling*, Prentice Hall

40. Rinnooy Kan, A.H.G. (1976) *Machine Scheduling Problems*, Martinus Nijhoff

41. Rothkopf, M.H. (1966) "Scheduling Independent Tasks on Parallel Processors", *Man. Sci.*, Vol. 12

42. Roy, B. and B.G. Sussmann (1964) "Les problèmes d'ordonnancement avec constraintes disjonctives", *Note DS No.9 bis*, SEMA, Montrouge

43. Sahni, S.K. (1976) "Algorithms for Scheduling Independent Tasks", *J. ACM*, Vol.23, No.1

44. Sriskandarajah, C. and P. Ladet (1986) "Some No-Wait Shops Scheduling Problems: Complexity Aspect", *E.J.O.R.*, Vol.24

Appendix A

A Classification Scheme for Machine Scheduling Problems

The development of formal scheduling models dates back to the time of the World War I. At that time H.L. Gantt invented a graphical tool, *Gantt Chart*, which is used to represent which job is loaded on which machine and at what time, resulting in a *schedule.* A Gantt Chart can be used to compare two different schedules with respect to some criteria. It was the first work that helps to identify the benefits obtained from scheduling. Later [Roy and Sussmann 1964] proposed another representation tool, *disjunctive graph representation*, which is essentially same as the Gantt chart except that the graph notation introduced.

The development of formal scheduling models and the concern of the scheduling theory can be followed chronologically in [Conway *et al.* 1967; Baker 1974; Rinnooy Kan 1976; Lenstra 1977; Graham *et al.* 1979; Bellman *et al.* 1982; French 1982; Lawler *et al.* 1982; Blazewicz *et al.* 1986; Lawler *et al.* 1989].

Scheduling problems may arise whenever n jobs J_j $(j = 1, \ldots, n)$ have to be processed on s stages S_ℓ $(\ell = 1, \ldots, s)$ each of which may have m_ℓ parallel machines $M_{k\ell}$ $(k = 1, \ldots, m_\ell)$. We assume that

- machines of a stage are not shared with any other stage i.e., if $\mathcal{M}_\ell$ denotes the set of machines at stage S_ℓ (where $|\mathcal{M}_\ell| = m_\ell$) then $\mathcal{M}_\ell \bigcap \mathcal{M}_h = \emptyset \ \forall \ \ell \neq h$,
- each machine $M_{k\ell}$ of a stage S_ℓ can process at most one job J_j at a time and,
- unless otherwise stated each job J_j can be processed on at most one machine $M_{k\ell} \in \mathcal{M}_\ell$ and stage S_ℓ at a time.
- following data can be specified for each job J_j:
  - a number of operations o_j,
  - a sequence of operations $\{O_{1j}, \ldots, O_{o_j,j}\}$, where O_{ij} has to be processed on one of $m_{s_{ij}}$ parallel machines of a stage s_{ij} with $s_{(i-1)j} \neq s_{ij} \ \forall i = 2, \ldots, o_j$,
  - a processing requirement p_{kij} of each O_{ij} on k-th ($k = 1, \ldots, m_{s_{ij}}$) machine of s_{ij},
  - a *ready time* or *release date* r_j on which J_j becomes available for processing,
  - a *due date* d_j by which J_j should ideally be completed,
  - a *deadline* $\overline{d}_j$ by which J_j must be completed,
  - a *weight* w_j indicating the relative importance of J_j,
  - a nondecreasing real function f_j of the completion time C_j, indicating the cost $f_j(C_j)$ incurred if J_j is completed at C_j.

Given such an instance, a scheduling problem can be modeled as determining the schedule $\mathcal{S}$ that minimizes f_{max} or $\sum f_j$ such that in $\mathcal{S}$

1. $f_{max} = \max_{1 \leq j \leq n} \{f_j(C_j)\}$ and $\sum f_j = \sum f_j(C_j)$,
2. prescribed $\{O_{1j}, \ldots, O_{o_j,j}\}$ for each job J_j is preserved,

3. each of $m_{s_{ij}}$ parallel machines in stage s_{ij} processes one operation O_{ij} at a time,
4. each operation O_{ij} requiring the stage s_{ij} are processed on one and only one of $m_{s_{ij}}$ parallel machines at a time,
5. some other characteristics of each job and/or shop are satisfied (the ones that are commonly used in the literature will be explained later in this section).

Variety of scheduling problems defined by the above formulation can be identified by the terminology similar to the one used in [Graham *et al.* 1979]. In this notation, each scheduling problem is represented by a 4-tuple $\alpha \mid \beta \mid \gamma \mid \delta$:

- α identifies the production environment. Following four different configurations are defined in the literature:
  - $\alpha = 1$: a single stage problem.
  - $\alpha = Fs$: a *flow shop* problem in which $o_j \leftarrow s$ and $s_{ij} \leftarrow s_i \;\; \forall \;\; J_j$ and O_{ij}. If s is not given the general class of flow shop scheduling problems will be represented.
  - $\alpha = Js$: a *job shop* problem which is the general case defined at the beginning of this section.
  - $\alpha = Os$: an *open shop* problem which is same as the flow shop problem except in this case the order of operations is immaterial, i.e. $\{O_{1j}, \ldots, O_{o_j,j}\}$ represents a set of operations but not necessarily their sequence.
- β identifies the machine environment at each stage of production. If we let o denote the *empty symbol* then the possible configurations are:
  - $\beta = \text{o}$: the problem with single machine at each stage of production.
  - $\beta = 1$: single machine at a particular stage of production; $p_{1ij} \leftarrow p_{ij}$.

- $\beta = Pm_\ell$: Identical m_ℓ parallel machines at stage S_ℓ; $p_{kij} \leftarrow p_{ij} \;\; \forall \;\; M_{k\ell} \in \mathcal{M}_\ell$. If m_ℓ is not specified then the general class of problems in which there is an arbitrary number (m_ℓ) of parallel machines at stage S_ℓ, is implied.
- $\beta = Qm_\ell$: Uniform m_ℓ parallel machines at stage S_ℓ; $p_{kij} \leftarrow p_{ij}/t_{k\ell}$ for a given speed $t_{k\ell}$ of machine $M_{k\ell} \in \mathcal{M}_\ell$.
- $\beta = Rm_\ell$: Unrelated m_ℓ parallel machines at stage S_ℓ.

- γ identifies further assumptions of the scheduling problem such as:
  - $\gamma = pmtn$: job preemption is allowed, i.e. the processing of any operation may be interrupted and resumed at a later time.
  - $\gamma = strm$: lot streaming is allowed. It may occur in two different ways: (i). any operation of a job may be processed on different machines of a stage at the same time ($in - stage\;\; strm$), (ii). before an operation is entirely completed, some portion of the work can be moved ahead to begin next operation ($inter - stage\;\; strm$). Clearly *strm* is different from *pmtn* and it violates the third assumption stated at the beginning of the section.
  - $\gamma = nonbottleneck\;\; \ell - th\;\; stage$: stage S_ℓ is assumed to have an infinite capacity.
  - $\gamma = dominated\;\; \ell - th\;\; stage$: processing times of operations requiring a stage S_ℓ are such that the stage S_ℓ can be considered as *nonbottleneck*, i.e. in any schedule for a given problem instance, stage S_ℓ will be *nonbottleneck*.
  - $\gamma = no - wait$: no job is allowed to wait in between stages.
  - $\gamma = r_j$: ready times that may differ for each job are specified.
  - $\gamma = o_j \leq o$: constant upper bound on number of operations for all J_j is specified (valid only if $\alpha = Js$).
  - If any one of the above characteristics is not possessed, $\circ$ is used instead of it.

- δ identifies the optimality criterion of the scheduling problem. Commonly used performance measures are:

$f_j(C_j)$	f_{max}	$\sum f_j$
C_j	C_{max} *(makespan)*	$\sum w_j C_j$ *(mean weighted flowtime)*
$C_j - d_j$	L_{max} *(maximum lateness)*	—
$\max\{0, C_j - d_j\}$	—	$\sum w_j T_j$ *(mean weighted tardiness)*
$\begin{cases} 0 & \text{If } C_j \le d_j, \\ 1 & \text{otherwise.} \end{cases}$	—	$\sum w_j U_j$ *(mean weighted number of tardy jobs)*

These performance measures are called *regular* in the sense that each δ is a monotone function of the completion times $C_1, C_2, \ldots, C_n$. That is $C_j \le C'_j \ \forall\ j \Rightarrow \delta(C_1, C_2, \ldots, C_n) \le \delta(C'_1, C'_2, \ldots, C'_n)$.

A.1 Examples

$1|P||C_{max}$: refers to a class of scheduling problems in which n jobs are scheduled on m identical parallel machines so as to minimize makespan.

$1|Pc||C_{max}$: refers to a class of $1|P||C_{max}$ problems in which the number of machines is a constant c.

$1|Q||C_{max}$: refers to a class of scheduling problems in which n jobs are scheduled on m uniform parallel machines so as to minimize makespan.

$1|P|r_j, \overline{d_j}|C_{max}$: refers to a class of scheduling problems in which n jobs are to be scheduled on m identical parallel machines so as to minimize makespan. In a feasible schedule no job can start before its ready time r_j and each job must be completed by its deadline $\overline{d_j}$.

$J|||C_{max}$: refers to a class of job shop scheduling problems in which the aim is to minimize makespan. It is assumed that in the job shop there is a single machine at each stage.

A.2 Reducibility Among Scheduling Problems

Reducibility among scheduling problems have been showed in Lenstra (1977) and Rinnooy Kan (1976) and can easily be adapted for parallel machine multi-stage problems. The results can be summarized as in Figure A.1. In this figure each

$$\begin{tikzcd}
R & & \\
Q \arrow[u] & & J \\
 & O & \\
P \arrow[uu] & & F \arrow[uu] \\
 & 1 \arrow[ul] \arrow[uu] \arrow[ur] & \\
 & \mathcal{G}_1 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \sum w_jT_j & \sum w_jU_j \\
\sum w_jC_j \arrow[ur] & \sum T_j \arrow[u] & \sum U_j \arrow[u] \\
\sum C_j \arrow[u] \arrow[ur] & L_{max} \arrow[u] \arrow[ur] & \\
 & C_{max} \arrow[u] & \\
 & \mathcal{G}_2 &
\end{tikzcd}$$

$$\begin{tikzcd}
pmtn \\
\circ \\
\mathcal{G}_3
\end{tikzcd}
\qquad
\begin{tikzcd}
r_j \\
\circ \arrow[u] \\
\mathcal{G}_4
\end{tikzcd}
\qquad
\begin{tikzcd}
\circ \\
o_j \leq o \arrow[u] \\
\mathcal{G}_5
\end{tikzcd}$$

Figure A.1: Reducibility among scheduling problems (excerpted from [Lawler *et al.* 1982]

graph $\mathcal{G}_i$ represents a different characteristic of a scheduling problem and a 7-tuple $(v_o, \ldots, v_6)$, where v_i is a vertex of graph $\mathcal{G}_i$, represents a particular scheduling problem. In $\mathcal{G}_i$, the directed path from Π' to Π shows the reducibility in terms of the characteristic v_i. The computer program MSPCLASS due to [Lageweg *et al.* 1982] uses these reducibility graphs and the known complexity results to determine the complexity class of a problem. A recent study due to [Lawler *et al.* 1989] showed that over 4,536 scheduling problems (problem classes) defined in the literature, only 416 were solvable in polynomial-time. 3,817 problems were shown to be $\mathcal{NP}$-hard (3,582 of them were unary $\mathcal{NP}$-hard). The status of 303 was unknown at the time the study was done.

Appendix B

Terminology

In this appendix, a glossary of the basic complexity theoretic concepts and definitions that are used in this study are presented. For details see [Garey and Johnson 1979].

Binary Encoding Consider an instance of the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$, defined in Section 2.1: $n = 4$, $m = 2$, $p_1 = 5$, $p_2 = 3$, $p_3 = 1$, $p_4 = 4$ and the bin size $\tau = 6$. In a binary encoding, all numbers are written in binary and *separated* by commas in the following way: number of pieces, number of bins, size of a bin, sizes of pieces (At the end "." is used to specify the end of the input). Thus the above instance is represented as: "100,10,110,101,11,1,100." The input length obtained from the above encoding scheme is

$$\lfloor \log_2 n \rfloor + \lfloor \log_2 m \rfloor + \sum_{j=1}^{n} \lfloor \log_2 p_j \rfloor + \lfloor \log_2 \tau \rfloor + n + m + 1 = 24$$

A binary encoding, as a *reasonable* encoding scheme, should be uniquely decodable and concise (that is, it should not allow artificial growth in the input length such as "100,10,,,,,,,110,,,101,11,1,100,,." for the above example).

Fully Polynomial ε-Approximation Scheme A family of approximation algorithms A_ε each of which has the worst case bound $1+\varepsilon$ for some $\varepsilon > 0$ and each has the time polynomial-time complexity function. Furthermore the computational requirement of algorithms grow polynomially both in the input length and $1/\varepsilon$.

Input Length See Input Size.

Input Size is determined by the amount of input data needed to describe an instance (input length) ν, and the magnitude of the numbers involved in an instance θ. Input data is assumed to be encoded in Binary Encoding (or in some other encoding scheme other than unary).

Mean Behavior of an approximation algorithm refers to the expected deviation of the solution delivered by this algorithm from the optimal. It can either be determined by an empirical analysis or by a probabilistic analysis.

$\mathcal{NP}$ Class consists of decision problems for which both a feasible solution can be guessed and checked whether or not it provides a "yes" answer in polynomial time. Clearly $\mathcal{P}$ is a subset of $\mathcal{NP}$. The conjecture that $\mathcal{P} \neq \mathcal{NP}$ is still open.

(Binary) $\mathcal{NP}$-complete The decision problem Π_2 is called $\mathcal{NP}$-complete if $\Pi_2 \in \mathcal{NP}$ and $\Pi_1 \propto \Pi_2$ (that is, Π_1 is reducible to Π_2) for every $\Pi_1 \in \mathcal{NP}$. Π_2 is the hardest problem in $\mathcal{NP}$. This definition implies that when proving $\mathcal{NP}$-completeness of Π_2, it is enough to find a $\mathcal{NP}$-complete problem $\Pi_1 \propto \Pi_2$ for $\Pi_2 \in \mathcal{NP}$.

(Binary) $\mathcal{NP}$-hard The problem Π_2 is called $\mathcal{NP}$-hard if the $\mathcal{NP}$-complete decision problem $\Pi_1 \propto \Pi_2$ (that is Π_1 is reducible to Π_2). Informally, the optimization problem is called $\mathcal{NP}$-hard if the associated decision problem is $\mathcal{NP}$-complete.

$\mathcal{P}$ Class consists of all problems for which algorithms with polynomial-time complexity function have been found.

Polynomial-Time Algorithm is defined to be the one with the time-complexity function being $\mathcal{O}$ $(f(\nu))$ for some polynomial function f and the input length ν.

Polynomial ε-Approximation Scheme A family of approximation algorithms A_ε such that for a fixed $\varepsilon > 0$ each has the worst case bound $1 + \varepsilon$ and the polynomial-time complexity function. The computational requirement of algorithms grow polynomially in the input length but exponentially in $1/\varepsilon$.

Pseudo Polynomial-Time Algorithm is defined to be the one with the time-complexity function $\mathcal{O}$ $(f(\nu, \theta))$ for some polynomial function f, input length ν and an upper bound θ, on the magnitude of each of the data. That is, any algorithm which is polynomial in the unary encoding is a pseudo ,polynomial. By definition, any polynomial-time algorithm is also a pseudo-polynomial time algorithm since it runs in time bounded by a polynomial in the input length.

Reducibility A problem Π_1 is reducible to another problem Π_2 ($\Pi_1 \propto \Pi_2$) if for any instance of Π_1 an instance of Π_2 can be constructed in polynomial-time such that solving the instance of Π_2 will solve the instance of Π_1 as well. The reducibility of Π_1 to Π_2 implies that Π_1 can be considered as a special case of Π_2.

Time-Complexity Function $f(x)$ of an algorithm gives the maximum number of operations that would be required to solve an instance of size x. $f(x)$

is $\mathcal{O}\ (g(x))$ implies that there exists a constant c such that $|f(x)| \leq c|g(x)|$ for all values of $x \geq 0$.

Turing Reducibility Consider a *search problem* Π that consists of a set D_Π of *instances.* For each instance $I \in D_\Pi$, the set $S_\Pi(I)$ is called *solutions* for I. An algorithm $\mathcal{A}$ is said to *solve* a search problem Π if, given as input any instance $I \in D_\Pi$, it returns the answer "no" whenever $S_\Pi(I)$ is empty and otherwise returns some solution s belonging to $S_\Pi(I)$. A Turing reduction from a search problem Π_1 to a search problem Π_2 is an algorithm$\mathcal{A}$ that solves Π_1 by using a hypothetical subroutine $\mathcal{S}$ for solving Π_2 such that, if $\mathcal{S}$ were a polynomial-time algorithm for Π_2, then $\mathcal{A}$ would be a polynomial time algorithm for Π_1.

Unary $\mathcal{NP}$-complete A decision problem Π is called unary $\mathcal{NP}$-complete if the subproblem Π_ρ defined as the restriction of Π in which magnitudes of all data is bounded by a polynomial $\rho(\nu)$ of the input length, is $\mathcal{NP}$-complete. If Π is unary $\mathcal{NP}$-complete then it cannot be answered with a pseudo polynomial-time algorithm. In the literature decision problems in this class are sometimes referred to as *$\mathcal{NP}$-complete in the strong sense.*

Unary $\mathcal{NP}$-hard Definition is similar to unary $\mathcal{NP}$-complete, except that the restricted problem is $\mathcal{NP}$-hard.

Unary Encoding Consider an instance of the bin packing problem $\Pi^{\mathsf{d}}_{\mathsf{bp}}$defined in Section 2.1: $n = 4, m = 2, p_1 = 5, p_2 = 3, p_3 = 1, p_4 = 4$ and the bin size $\tau = 6$. In a unary encoding, all numbers are based on 1 and *separated* by commas in the following way: number of pieces, number of bins, size of a bin, sizes of pieces (At the end "." is used to specify the end of the input). Thus the above instance is represented as: "1111,11,111111,11111,111,1,1111." The input size obtained from the above encoding scheme is

$$n + m + \sum_{j=1}^{n} p_j + r + n + m + 1 = 32$$

A unary encoding should be uniquely decodable and concise (that is it should not allow artificial growth in the input such as "1111,,11,111111,11111,,111,1,1111." for the above example).

Worst-Case Bound For a minimization problem Π the worst-case bound of an approximation algorithm $\mathcal{A}$ is defined as

$$R_A = inf\{r \geq 1 | \mathcal{A}(I)/OPT(I) \leq r \text{ for all instances } I \text{ of } \Pi\}$$

where for an instance I of Π, $\mathcal{A}(I)$ denotes the solution value returned by the approximation algorithm $\mathcal{A}$ and $OPT(I)$ denotes the minimum value. A worst-case bound is called *tight* if it is attainable.

Appendix C

Computational Results in Tabular Form

Table C.1: First encountered time and total time

DLL			
Average:	0.0017		0.00238
Std.Dev.:	0.0051		0.005898
First Encountered Time (sec)		Total Time (sec)	
10	50	10	50
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		0

DLH			
Average:	2.12692		107.9296
Std.Dev.:	3.134036		228.3357
First Encountered Time (sec)		Total Time (sec)	
10	49	10	39
40	1	40	1
70	0	70	0
100	0	100	1
130	0	130	0
160	0	160	0
190	0	190	0
	0		9

DLM			
Average:	0.98606		107.6828
Std.Dev.:	3.074165		367.9076
First Encountered Time (sec)		Total Time (sec)	
10	49	10	45
40	1	40	1
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		4

Table C.1: First encountered time and total time (continued)

DHL			
Average:	0.00034		0.00034
Std.Dev.:	0.00238		0.00238
First Encountered Time (sec)		Total Time (sec)	
10	50	10	50
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		0

DHH			
Average:	2.33028		89.04114
Std.Dev.:	9.808269		127.0046
First Encountered Time (sec)		Total Time (sec)	
10	49	10	33
40	0	40	0
70	0	70	0
100	1	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		17

DHM			
Average:	0.154		0.15434
Std.Dev.:	0.254839		0.254645
First Encountered Time (sec)		Total Time (sec)	
10	50	10	50
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		0

Table C.1: First encountered time and total time (continued)

DML			
Average:	0.03068		171.7558
Std.Dev.:	0.145388		593.2152
First Encountered Time (sec)		Total Time (sec)	
10	50	10	46
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		4

DMH			
Average:	38.5465		451.2136
Std.Dev.:	94.04424		146.6622
First Encountered Time (sec)		Total Time (sec)	
10	27	10	1
40	16	40	1
70	1	70	0
100	2	100	0
130	0	130	0
160	1	160	0
190	0	190	0
	3		48

DMM			
Average:	7.83614		33.49444
Std.Dev.:	33.05876		109.2993
First Encountered Time (sec)		Total Time (sec)	
10	48	10	45
40	0	40	0
70	1	70	1
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	1		4

Table C.1: First encountered time and total time (continued)

NDLL			
Average:	0.00814		0.00814
Std.Dev.:	0.009092		0.009092
First Encountered Time (sec)		Total Time (sec)	
10	50	10	50
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		0

NDLH			
Average:	0.12238		11.52626
Std.Dev.:	0.486959		80.03702
First Encountered Time (sec)		Total Time (sec)	
10	50	10	49
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		1

NDLM			
Average:	0.0434		12.6719
Std.Dev.:	0.277174		88.58846
First Encountered Time (sec)		Total Time (sec)	
10	50	10	49
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		1

Table C.1: First encountered time and total time (continued)

NDHL			
Average:	25.67562		84.35964
Std.Dev.:	56.08327		134.1312
First Encountered Time (sec)		Total Time (sec)	
10	35	10	28
40	8	40	7
70	0	70	0
100	3	100	2
130	0	130	0
160	0	160	0
190	1	190	1
	3		12

NDHH			
Average:	1.90066		50.77232
Std.Dev.:	4.239113		106.1439
First Encountered Time (sec)		Total Time (sec)	
10	48	10	39
40	2	40	2
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		9

NDHM			
Average:	0.68638		0.68704
Std.Dev.:	0.362379		0.363126
First Encountered Time (sec)		Total Time (sec)	
10	50	10	50
40	0	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	0	190	0
	0		0

Table C.1: First encountered time and total time (continued)

NDML			
Average:	4.72214		518.6818
Std.Dev.:	22.77753		688.9690
First Encountered Time (sec)		Total Time (sec)	
10	48	10	27
40	1	40	0
70	0	70	0
100	0	100	0
130	0	130	0
160	0	160	0
190	1	190	1
	0		22

NDMH			
Average:	14.13172		392.9053
Std.Dev.:	27.24027		261.3126
First Encountered Time (sec)		Total Time (sec)	
10	32	10	7
40	15	40	3
70	0	70	0
100	1	100	0
130	1	130	0
160	0	160	0
190	1	190	0
	0		40

NDMM			
Average:	14.54648		53.25626
Std.Dev.:	47.76684		218.8059
First Encountered Time (sec)		Total Time (sec)	
10	46	10	44
40	0	40	0
70	1	70	1
100	1	100	1
130	0	130	0
160	0	160	0
190	0	190	0
	2		4

Table C.2: Initial gap and ending gap

DLL			
Average:	0		0
Std.Dev.:	0		0
Initial Gap	(relative)	Ending Gap	(relative)
0	50	0	50
0.02	0	0.02	0
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

DLH			
Average:	0.015478		0.000960
Std.Dev.:	0.021945		0.002252
Initial Gap	(relative)	Ending Gap	(relative)
0	28	0	41
0.02	5	0.02	9
0.06	14	0.06	0
0.08	3	0.08	0
0.1	0	0.1	0
	0		0

DLM			
Average:	0.020758		0.001429
Std.Dev.:	0.022477		0.004848
Initial Gap	(relative)	Ending Gap	(relative)
0	21	0	46
0.02	11	0.02	4
0.06	17	0.06	0
0.08	0	0.08	0
0.1	1	0.1	0
	0		0

Table C.2: Initial gap and ending gap (continued)

DHL			
Average:	0		0
Std.Dev.:	0		0
Initial Gap	(relative)	Ending Gap	(relative)
0	50	0	50
0.02	0	0.02	0
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

DHH			
Average:	0.001292		0.000149
Std.Dev.:	0.001278		0.000252
Initial Gap	(relative)	Ending Gap	(relative)
0	5	0	33
0.02	45	0.02	17
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

DHM			
Average:	0.000724		0
Std.Dev.:	0.001247		0
Initial Gap	(relative)	Ending Gap	(relative)
0	35	0	50
0.02	15	0.02	0
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

Table C.2: Initial gap and ending gap (continued)

DML			
Average:	0.004978		0.003478
Std.Dev.:	0.013542		0.011795
Initial Gap	(relative)	Ending Gap	(relative)
0	44	0	46
0.02	0	0.02	0
0.06	6	0.06	4
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

DMH			
Average:	0.044103		0.003114
Std.Dev.:	0.027782		0.002117
Initial Gap	(relative)	Ending Gap	(relative)
0	0	0	2
0.02	10	0.02	48
0.06	24	0.06	0
0.08	10	0.08	0
0.1	5	0.1	0
	1		0

DMM			
Average:	0.031864		0.000571
Std.Dev.:	0.021734		0.002316
Initial Gap	(relative)	Ending Gap	(relative)
0	2	0	47
0.02	20	0.02	3
0.06	20	0.06	0
0.08	7	0.08	0
0.1	1	0.1	0
	0		0

Table C.2: Initial gap and ending gap (continued)

NDLL			
Average:	0		0
Std.Dev.:	0		0
Initial Gap	(relative)	Ending Gap	(relative)
0	50	0	50
0.02	0	0.02	0
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

NDLH			
Average:	0.001331		0.000196
Std.Dev.:	0.005287		0.001372
Initial Gap	(relative)	Ending Gap	(relative)
0	47	0	49
0.02	1	0.02	1
0.06	2	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

NDLM			
Average:	0.001898		0.000384
Std.Dev.:	0.008769		0.002692
Initial Gap	(relative)	Ending Gap	(relative)
0	47	0	49
0.02	2	0.02	1
0.06	1	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

Table C.2: Initial gap and ending gap (continued)

NDHL			
Average:	0.20529		0.001696
Std.Dev.:	0.013552		0.003694
Initial Gap	(relative)	Ending Gap	(relative)
0	2	0	41
0.02	26	0.02	9
0.06	22	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

NDHH			
Average:	0.002064		0.000107
Std.Dev.:	0.001756		0.000241
Initial Gap	(relative)	Ending Gap	(relative)
0	1	0	41
0.02	49	0.02	9
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

NDHM			
Average:	0.005389		0
Std.Dev.:	0.003549		0
Initial Gap	(relative)	Ending Gap	(relative)
0	1	0	50
0.02	49	0.02	0
0.06	0	0.06	0
0.08	0	0.08	0
0.1	0	0.1	0
	0		0

Table C.2: Initial gap and ending gap (continued)

NDML			
Average:	0.101193		0.019249
Std.Dev.:	0.058377		0.027030
Initial Gap	(relative)	Ending Gap	(relative)
0	1	0	29
0.02	0	0.02	0
0.06	12	0.06	17
0.08	8	0.08	1
0.1	4	0.1	3
	25		0

NDMH			
Average:	0.040239		0.003032
Std.Dev.:	0.032811		0.002564
Initial Gap	(relative)	Ending Gap	(relative)
0	5	0	10
0.02	13	0.02	40
0.06	16	0.06	0
0.08	9	0.08	0
0.1	6	0.1	0
	1		0

NDMM			
Average:	0.047070		0.000476
Std.Dev.:	0.033386		0.002391
Initial Gap	(relative)	Ending Gap	(relative)
0	2	0	48
0.02	10	0.02	2
0.06	21	0.06	0
0.08	5	0.08	0
0.1	8	0.1	0
	4		0

Table C.3: First encountered node and total number of nodes

DLL			
Average:	0.28		0.28
Std.Dev.:	0.448998		0.448998
First Encountered Node		Total # of Nodes	
0	36	0	36
1	14	1	14
500	0	500	0
1000	0	1000	0
5000	0	5000	0
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		0

DLH			
Average:	1818.22		186415.7
Std.Dev.:	2469.589		382978.6
First Encountered Node		Total # of Nodes	
0	5	0	4
1	24	1	24
500	0	500	1
1000	0	1000	0
5000	12	5000	5
10000	9	10000	6
50000	0	50000	0
100000	0	100000	0
	0		10

DLM			
Average:	916.92		80879.48
Std.Dev.:	3035.137		271050.7
First Encountered Node		Total # of Nodes	
0	8	0	0
1	21	1	21
500	1	500	9
1000	5	1000	3
5000	14	5000	12
10000	0	10000	0
50000	1	50000	1
100000	0	100000	0
	0		4

Table C.3: First encountered node and total number of nodes (continued)

DHL			
Average:	0		0
Std.Dev.:	0		0
First Encountered Node		Total # of Nodes	
0	50	0	50
1	0	1	0
500	0	500	0
1000	0	1000	0
5000	0	5000	0
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		0

DHH			
Average:	8426.2		341147.2
Std.Dev.:	47840.91		472913.4
First Encountered Node		Total # of Nodes	
0	6	0	5
1	0	1	0
500	15	500	11
1000	14	1000	8
5000	12	5000	8
10000	0	10000	0
50000	2	50000	1
100000	0	100000	0
	1		17

DHM			
Average:	146.96		146.96
Std.Dev.:	245.5789		245.5789
First Encountered Node		Total # of Nodes	
0	35	0	35
1	0	1	0
500	9	500	9
1000	5	1000	5
5000	1	5000	1
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		0

Table C.3: First encountered node and total number of nodes (continued)

DML			
Average:	14.06		80014.06
Std.Dev.:	70.30801		271289.0
First Encountered Node		Total # of Nodes	
0	48	0	44
1	0	1	0
500	2	500	2
1000	0	1000	0
5000	0	5000	0
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		4

DMH			
Average:	78228.68		960631.7
Std.Dev.:	202849.0		192874.3
First Encountered Node		Total # of Nodes	
0	0	0	0
1	0	1	0
500	0	500	0
1000	0	1000	0
5000	13	5000	0
10000	18	10000	1
50000	10	50000	1
100000	2	100000	0
	7		48

DMM			
Average:	15833.36		75763.88
Std.Dev.:	80493.70		246960.8
First Encountered Node		Total # of Nodes	
0	3	0	2
1	0	1	0
500	0	500	0
1000	1	1000	1
5000	43	5000	41
10000	1	10000	1
50000	0	50000	0
100000	0	100000	0
	2		5

Table C.3: First encountered node and total number of nodes (continued)

NDLL			
Average:	0.94		0.94
Std.Dev.:	0.237486		0.237486
First Encountered Node		Total # of Nodes	
0	3	0	3
1	47	1	47
500	0	500	0
1000	0	1000	0
5000	0	5000	0
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		0

NDLH			
Average:	140.92		20091.5
Std.Dev.:	566.5401		139987.6
First Encountered Node		Total # of Nodes	
0	43	0	43
1	4	1	4
500	0	500	0
1000	0	1000	0
5000	3	5000	2
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		1

NDLM			
Average:	40.08		20012.88
Std.Dev.:	278.8460		139998.1
First Encountered Node		Total # of Nodes	
0	37	0	37
1	12	1	10
500	0	500	2
1000	0	1000	0
5000	1	5000	0
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		1

Table C.3: First encountered node and total number of nodes (continued)

NDHL			
Average:	72989.32		245206.8
Std.Dev.:	161173.5		386581.6
First Encountered Node		Total # of Nodes	
0	5	0	2
1	0	1	0
500	4	500	2
1000	8	1000	7
5000	10	5000	9
10000	2	10000	2
50000	11	50000	11
100000	1	100000	1
	9		16

NDHH			
Average:	4116.12		183514.4
Std.Dev.:	12743.18		382744.9
First Encountered Node		Total # of Nodes	
0	1	0	1
1	0	1	0
500	11	500	11
1000	15	1000	12
5000	18	5000	13
10000	0	10000	0
50000	4	50000	3
100000	1	100000	1
	0		9

NDHM			
Average:	632.44		632.44
Std.Dev.:	353.8562		353.8562
First Encountered Node		Total # of Nodes	
0	1	0	1
1	0	1	0
500	22	500	22
1000	18	1000	18
5000	9	5000	9
10000	0	10000	0
50000	0	50000	0
100000	0	100000	0
	0		0

Table C.3: First encountered node and total number of nodes (continued)

NDML			
Average:	6375.32		432360.7
Std.Dev.:	36676.61		486516.8
First Encountered Node		Total # of Nodes	
0	9	0	1
1	0	1	0
500	12	500	6
1000	21	1000	16
5000	6	5000	4
10000	0	10000	0
50000	1	50000	0
100000	0	100000	0
	1		23

NDMH			
Average:	28208.86		801755.7
Std.Dev.:	86223.19		396553.1
First Encountered Node		Total # of Nodes	
0	0	0	0
1	5	1	5
500	0	500	0
1000	0	1000	0
5000	16	5000	1
10000	16	10000	1
50000	9	50000	2
100000	1	100000	1
	3		40

NDMM			
Average:	21026.7		60979.36
Std.Dev.:	74892.32		205748.4
First Encountered Node		Total # of Nodes	
0	0	0	0
1	2	1	2
500	0	500	0
1000	3	1000	2
5000	40	5000	39
10000	1	10000	1
50000	0	50000	0
100000	1	100000	1
	3		5

Table C.4: The Analysis of Variance Table for the Three-Factor Fixed Effects Model: CPU Time

Source	SS	d.f.	MS	F
A	268.1024	1	268.1024	0.282195
B	25048.81	2	12524.40	13.18277[a]
C	5782.018	2	2891.009	3.042980[a]
AB	6752.673	2	3376.336	3.553819[a]
AC	13757.61	2	6878.808	7.240405[a]
BC	32728.07	4	8182.017	8.612119[a]
ABC	12414.47	4	3103.619	3.266766[a]
Error	837951.6	882	950.0585	
TOTAL	934703.4	899		

First Encountered Time

Source	SS	d.f.	MS	F
A	74341.71	1	74341.71	1.071978
B	10717654	2	5358827.	77.27217[a]
C	3419756.	2	1709878.	24.65576[a]
AB	1041565.	2	520782.7	7.509482[a]
AC	1832625.	2	916312.5	13.21286[a]
BC	5078385.	4	1269596.	18.30707[a]
ABC	827717.9	4	206929.4	2.983841[a]
Error	61166723	882	69350.02	
TOTAL	84158769	899		

Total Time

A stands for the n/m divisibility
B stands for the n/m ratio
C stands for the processing variability

[a]Significant at 5%

Table C.5: The Analysis of Variance Table for the Three-Factor Fixed Effects Model: Gap

Source	SS	d.f.	MS	F
A	0.028065	1	0.028065	59.49738[a]
B	0.306437	2	0.153218	324.8118[a]
C	0.002398	2	0.001199	2.542688
AB	0.083037	2	0.041518	88.01639[a]
AC	0.087998	2	0.043999	93.27521[a]
BC	0.019168	4	0.004792	10.15886[a]
ABC	0.063479	4	0.015869	33.64297[a]
Error	0.416053	882	0.000471	
TOTAL	1.006639	899		

Initial Gap

Source	SS	d.f.	MS	F
A	0.000662	1	0.000662	12.38346[a]
B	0.004193	2	0.002096	39.20675[a]
C	0.002142	2	0.001071	20.03270[a]
AB	0.001414	2	0.000707	13.22577[a]
AC	0.001897	2	0.000948	17.74290[a]
BC	0.004365	4	0.001091	20.40808[a]
ABC	0.002358	4	0.000589	11.02266[a]
Error	0.047171	882	0.000053	
TOTAL	0.064206	899		

Ending Gap

A stands for the n/m divisibility
B stands for the n/m ratio
C stands for the processing variability

[a]Significant at 5%

Table C.6: The Analysis of Variance Table for the Three-Factor Fixed Effects Model: Number of Nodes

Source	SS	d.f.	MS	F
A	2.2E+09	1	2.2E+09	0.429117
B	9.0E+10	2	4.5E+10	8.805632[a]
C	2.8E+10	2	1.4E+10	2.754640
AB	5.0E+10	2	2.5E+10	4.879816[a]
AC	7.7E+10	2	3.8E+10	7.469016[a]
BC	1.8E+11	4	4.5E+10	8.694404[a]
ABC	6.9E+10	4	1.7E+10	3.370123[a]
Error	4.5E+12	882	5.1E+09	
TOTAL	5.0E+12	899		

First Encountered Node

Source	SS	d.f.	MS	F
A	4.3E+09	1	4.3E+09	0.055275
B	2.0E+13	2	1.0E+13	129.5529[a]
C	2.3E+13	2	1.2E+13	147.8270[a]
AB	7.6E+11	2	3.8E+11	4.810473[a]
AC	5.0E+12	2	2.5E+12	31.54767[a]
BC	1.7E+13	4	4.2E+12	53.86708[a]
ABC	9.3E+11	4	2.3E+11	2.947135[a]
Error	6.9E+13	882	7.9E+10	
TOTAL	1.4E+14	899		

Total Number of Nodes

A stands for the n/m divisibility
B stands for the n/m ratio
C stands for the processing variability

[a]Significant at 5%

Vita

H. Cemal Akyel was born in Ankara, on 1 January 1961. He attended Department of Industrial Engineering, Middle East Technical University (METU) in August 1978 and graduated with honors in September 1983. From that time to September 1986 he was a research assistant at Department of Industrial Engineering, METU. During that period he worked with Professor Ömer S. Benli on *Production Scheduling in Two-Stage Parallel Machine Flow Shops* and got his M.Sc. degree in February 1986. In September 1986 he joined to Department of Industrial Engineering, Bilkent University as a research assistant to continue his graduate study with Professor Ömer S. Benli. In October 1989 he was appointed as an instructor at Department of Industrial Engineering, Bilkent University. Currently, he is an instructor at the same department.