MACHINE SCHEDULING PROBLEMS:
A BRANCH AND BOUND APPROACH

by

Tariq S. Abdul-Razaq

Ph. D. Thesis

Department of Mathematics

University of Keele

July, 1987

ACKNOWLEDGEMENTS

I would like to express my thanks to Dr. C. N. Potts, my supervisor, for his wonderful guidance and help throughout this thesis and for much advice and many suggestions about the research; and to Basrah University, Iraq for their financial support.

I would like also to thank members of the computer Center; and Dr. P. Avery in Mathematic Department; and Dr. A. Mahendrasingam in Physics Department for their help.

Finally, I would like to express my utmost appreciation to my family for their unfailing encouragement and support during my studies, and especially to my wife for her advice and help throughout.

ABSTRACT

All the material in this thesis is devoted to machine scheduling problems. It is presented in eight chapters.

The first three chapters are introductory in which we give various aspects of problem formulation, and we discuss the well-known methods of solution for machine scheduling problems.

The next four chapters contain original research, unless otherwise acknowledged, on various machine scheduling problem.

In chapter four we use branch and bound techniques to solve a one machine problem with release dates to minimize the weighted number of late jobs.

In chapter five machine sequencing to minimize total cost (not assumed to be a non-decreasing function of completion time) is considered. A dynamic programming formulation and relaxation of the problem is presented. Then we use branch and bound techniques to solve this problem, because the number of states required by this formulation is large.

In chapter six we provide a computational comparison of six algorithms which are used to solve the single machine sequencing to minimize the total weighted tardiness. Two algorithms use dynamic programming and four algorithms use branch and bound.

Chapter seven is devoted to use of branch and bound techniques to solve the two-machine flow shop problem to minimize the maximum completion time, when each job is processed first on machine A, is then transported to machine B, and lastly is processed on machine B.

Finally, chapter eight contains our conclusion together with some suggestions for future research.

CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 Background

A scheduling problem arises whenever we want to make a daily routine for any planned work. The analysis and study of such problems, particularly with respect to their optimality for various objectives and constraints, constitutes an exciting field known as scheduling theory. The terminology of scheduling theory arose in the processing and manufacturing industries. Thus we shall be talking about jobs and machines even though in some cases the objects referred to bear little relation to either jobs or machines. A basic problem in scheduling theory is the machine scheduling problem. This occurs whenever jobs, each of which consists of a given sequence of operations, have to be scheduled on one or more machines during a given period of time, such that a given objective function is minimized.

Recently, machine scheduling problems have received much attention. Two directions have always been very important in the investigation of the problem, namely:

(i) The search for algorithms that determine the optimal sequence efficiently, preferably in polynomial time.

(ii) The investigation of the computational complexity.

The first theoretical development in scheduling was made by Johnson [43], followed closely by results of Jackson [42] and Smith [91]. A natural way to attack machine scheduling problems is to formulate them as mathematical programming models. Apart from Wagner

[94], other simple integer programming formulations of scheduling problems are given by Bowman [9], Dantzig [16], Manne [62]. Both Conway et al. [14] and Baker [3] discuss integer programming formulations of scheduling problems. Rinnooy Kan [82] gives the most recent survey of such approaches, and also mentions some work of Nepomiastchy in which non-linear programming is used to obtain an approximate solution to general job shop problems.

The next important development was when branch and bound methods were applied to scheduling problems. They were first used by Ignall and Schrage [41], Lomnicki [61], Brown and Lomnicki [10] and McMahon and Burton [65].

Dynamic programming has also been proved to be a good approach for the solution of scheduling problem. In 1962 Held and Karp [38] followed the lead of Bellman, and applied dynamic programming to scheduling problems. Their method applies only to single machine problems.

Classifying scheduling problems according to their algorithmic complexity was first reported in [15] and [46]. If a problem is NP-hard then it is unlikely that it can be solved by an algorithm whose running time is bounded by a polynomial function of problem size [26]. In such cases two different approaches suggest themselves. The first method is to solve the problem to optimality using dynamic programming or branch and bound but this may require a time consuming search through the set of feasible solutions. The alternative approach is to use fast heuristic methods (methods that do not guarantee optimality) to find approximate solutions.

1.2 Contribution of this research

All the material in this thesis is devoted to machine scheduling problems. It is presented in eight chapters.

In chapter two various aspects of problem formulation, including notation, representation, optimality criteria, classification and computational complexity is given. In chapter three we discuss the well- known methods of solution for machine scheduling problems. We list in detail some of these methods, e.g. branch and bound and dynamic programming, because these two methods are amongst the most widely used methods of approach to solving scheduling problems, and because they are used throughout this thesis.

The next four chapters contain original research, unless otherwise acknowledged, on various machine scheduling problem. In each of these chapters new branch and bound algorithms are proposed and extensive computational testing is reported.

Chapter four demonstrates the properties and performance of the branch and bound technique on a single machine problem : single machine sequencing with release dates to minimize the number of late jobs and to minimize the weighted number of late jobs. Heuristics, dominance rules, and computational experience will also be included.

In chapter five machine sequencing to minimize total cost (not assuming that the cost is a non-decreasing function of completion time) is considered. A dynamic programming formulation and a relaxation by mapping the state-space onto a smaller state-space is presented. A heuristic and computational experience will also be included.

In chapter six single machine sequencing to minimize the total

weighted tardiness is studied. We provide a computational comparison of six algorithms. Two algorithms use dynamic programming and four algorithms use branch and bound. In both dynamic programming and branch and bound algorithms, dominance rules are used.

Chapter seven is devoted to demonstrating the performance of branch and bound algorithms on a two-machine problem: the two-machine flow shop problem to minimize the maximum completion time, when each job is processed first on machine A, then transported to machine B, and lastly is processed on machine B. A heuristic, dominance rules, and computational results will also be included.

Finally, chapter eight evaluates the contributions of this thesis. A discussion on the success of branch and bound algorithms for solving scheduling problems and some suggestions for future research are also given.

CHAPTER TWO

SCHEDULING PROBLEM FORMULATION

2.1 Introduction

In this chapter we deal with various aspects of problem formulation. In Section 2.2 we discuss problem representation and introduce notation to designate several concepts involving jobs, operations and machines, and several conditions such as restrictive assumptions on the jobs and machines are examined. In Section 2.3 we mention optimality criteria that have been used in previous research. Various job, machine and scheduling characteristics, are reflected by the classification with format $\alpha/\beta/\gamma$, are given in Section 2.4 . The theory of computational complexity is discussed in Section 2.5 .

2.2 Assumptions (Restrictions)

We shall introduce a number of basic definitions which explain the structure of scheduling problems. Here, we list assumptions for machines, jobs and other aspects of our problem.

2.2.1 Assumption about machines

M1 The number of machines is known and fixed.

M2 All machines are available at the same instant and are independent of each other.

M3 All machines remain available during an unlimited period of time.

M4 Each machine can be in one of three states : waiting for the next job, operating on a job or having finished its last

job.

M5 All machines are equally important.

M6 Breakdown or repair of any machine does not occur during the planning period.

M7 Any machine can process any job assigned to it.

M8 Each machine can process at most one job at the same time. This assumption is sometimes relaxed to obtain a lower bound on the objective function.

M9 There is only one of each type of machine (no machine group).

2.2.2 Assumption about jobs

J1 The number of jobs is known and fixed.

J2 All jobs are available at the same time and are independent. However we shall face some situations (see for example chapter 4) where each job i has a non-negative integer release date r_i at which it becomes available for processing. Further situations arise (see chapter 6) when jobs are not independent, i.e., precedence constraints among jobs exist. These precedence constraints on the jobs can be represented by a directed graph (digraph) G = (V,E) where V denotes the set of vertices and E the set of edges. The vertices of G represent the jobs and the edges join the vertices. Job i must be processed before job j on each machine if there exists a directed path from vertex i to vertex j in G [34].

J3 Each job be in one of three states : waiting for the next machine, being operated on by a machine, or having passed

its last machine.

J4 Each job has the same degree of importance. In many cases this assumption is dropped (see chapters 4,5 and 6) and to each job i is associated a non-negative integer w_i related to the importance of that job.

J5 Each job is processed by all the machines assigned to it.

J6 Each job can be processed by only one machine at the same time.

J7 Any operation which has started is not interrupted by other operations and is continued to its completion. This assumption is sometimes relaxed, i.e., job splitting (preemption) is allowed.

2.2.3 Other Assumption

* Each processing time is known and fixed. If not mentioned otherwise, set-up times, and transportation times of jobs between machines are assumed to be negligable.
* Any release date is known and fixed.
* All other quantities needed to define a particular problem are known and fixed, e.g. transportation times of the jobs between the machines (see chapter 7).

2.3 Performance Measures (Objectives)

Before we can define performance measures in precise mathematical terms, we need some definitions and notation. Let n denote the number of jobs. Also, we define for each job j (j=1,...,n):

r_j a release date at which job j become available for processing.

p_{ji} a processing time of its i^{th} operation, i=1,...,m_j, where m_j is the number of operations on job j, (if m_j=1 for all j, we shall write p_j instead of p_{j1}).

d_j a due date of job j, that is the time by which ideally we would like to have completed job j.

w_j the weight of job j (relative importance).

f_j a non-decreasing real cost function, measuring the cost $f_j(t)$ incurred if job j is completed at time t.

In general r_j, p_{ji}, d_j and w_j are given integer constants. Given a schedule, we can compute for each job j (j=1,...,n):

C_j the completion time of job j

L_j the lateness of job j . $L_j=C_j - d_j$

T_j the tardiness of job j . $T_j=\max\{ L_j , 0 \}$

E_j the earliness of job j . $E_j=\max\{ - L_j ,0 \}$

U_j the unit penalty of job j . $U_j=0$ if $C_j \leq d_j$, $U_j=1$ otherwise.

Here we note that there is a classification of performance measures into those that are regular and those that are not. A regular measure f is simply one that is non-decreasing in the completion times. Thus, a regular measure f is a function of

$C_1,\ldots,C_n$ such that

$$f(C_1,\ldots,C_n) < f(C_1',\ldots,C_n')$$

implies that $C_i < C_i'$ for at least one job i. These functions usually take one of the following forms :

(1) $f = f_{max} = \max \{ f_j(C_j) \}$

(2) $f = \Sigma f_j = \Sigma_{j=1}^{n} f_j(C_j)$.

The following specific criteria are used in later chapters.

$f=\Sigma U_j = \Sigma_{j=1}^{n} U_j$ number of late jobs.

Introducing weight we have:

$f=\Sigma w_jU_j = \Sigma_{j=1}^{n} w_jU_j$ weighted number of late jobs.

$f=\Sigma w_jT_j = \Sigma_{j=1}^{n} w_jT_j$ weighted sum of tardiness.

$f=\Sigma w_jC_j = \Sigma_{j=1}^{n} w_jC_j$ weighted sum of completion times.

One may also choose to minimize :

$f=C_{max} = \max_j \{ C_j \}$ maximum completion time.

$$f=\Sigma\, w_j e^{\lambda C_j} = \Sigma_{j=1}^{n}\, w_j e^{\lambda C_j} \quad (\lambda > 0)$$ total weighted

exponential function of completion times.

Additionally, if we do not assume that f is regular measure, then

$$f=\Sigma\, g_j(t) = \Sigma_{j=1}^{n}\, g_j(C_j)$$

where $g_j(t)$ is the cost of completing job j at time t, for any function g_j. One such function that is of interest is

$$f=\Sigma\, (h_j E_j + w_j T_j) = \Sigma_{j=1}^{n}\, (h_j E_j + w_j T_j)$$

where h_j and w_j are suitable weights for job j. In this case f measures the total weighted earliness and tardiness.

2.4 Problem Representation

Suppose that n jobs numbered 1,...,n have to be processed on m machines M_i (i=1,...,m), so as to minimize some objective function(s). We represent a machine scheduling problem, which involve well-defined set of jobs, machines and objective function(s), by the three parameter notation $\alpha/\beta/\gamma$, where α is the machine environment, β describes the job characteristics and γ is the objective function.

2.4.1 Machine environment (α)

We shall now describe the first parameter $\alpha=\alpha_1\alpha_2$, the machine environment. If $\alpha_1 \in \{ 1,P,Q,R \}$, each job j consists of a single operation that can be processed on any machine M_i with speed S_{ij}

(positive); the processing of job j on M_i requires a total time p_j/s_{ij}.

(P) Identical machines : All s_{ij} are equal. In this case we may assume that $s_{ij}=1$ for all i and j.

(Q) Uniform machines : $s_{ij}=s_{ik}$ for all machines i, and all jobs j and k. In this case each machine i performs all the jobs at the same speed s_i.

(R) Unrelated machines : There is no particular relationship between the s_{ij} values.

Hence we have :

$\alpha_1=1$ a single machine problem

$\alpha_1=P$ an m identical machine problem

$\alpha_1=Q$ an m uniform machine problem

$\alpha_1=R$ an m unrelated machine problem.

If $\alpha_1 \in \{ O,F,J \}$, each job j consists of a set of operations. Thus,

(O) $\alpha_1=O$, Open shop problem : There is no restriction on the order in which the operations are executed.

(F) $\alpha_1=F$, Flow shop problem : Each job has the same sequence of operations, but some job may overtake another job on some machine.

(J) α_1=J, Job shop problem : Each job has specified sequence of operation which may differ from the sequence of operation for other jobs.

The parameter α_2 denotes whether m is a constant or variable; if α_2 is a positive integer then the number of machines is constant and equal to α_2. If α_2 is not specified, then m is variable.

2.4.2 Job characteristics

The second parameter $\beta \in \{ \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7 \}$ indicates a number of jobs characteristics, which are defined as follows:

(1) $\beta_1 \in \{ \text{pmtn}, \phi \}$

β_1=pmtn : Preemption (job splitting) is allowed; the processing of any job may arbitrarily often be interrupted and resumed at the same time on a different machine or at a later time on any machine.

β_1=ϕ : No preemption is allowed.

(2) $\beta_2 \in \{ \text{prec}, \text{tree}, \phi \}$

β_2=prec : Arbitrary precedence relations between the jobs are given, where job i precedes job j implies that job i is completed before job j can start.

β_2=tree : Precedence relation between the jobs, such that the associated precedence graph G with vertices 1,...,n takes the form of a tree, i.e., G has outdegree or indegree at most one for all vertices.

β_2=ϕ : Jobs are independent.

(3) $\beta_3 \in \{ p_{ij}=1 , p_{ij} \leq p^* , \phi \}$

β_3=p_{ij}=1 : Each operation has unit processing time.

β_3=$p_{ij} \leq p^*$: Upper bound on all processing times.

β_3=ϕ : The processing times are arbitrary non-negative integers.

(4) $\beta_4 \in \{ \bar{d}_i , r_i , \phi \}$

β_4=$\bar{d}_i$: Jobs have deadlines.

β_4=r_i : Jobs have release dates.

β_4=r_i,$\bar{d}_i$: Jobs have release dates and deadlines

β_4=ϕ : Jobs are continuously available.

(5) $\beta_5 \in \{ p_i \leq p_j \longrightarrow w_i \geq w_j , \phi \}$

$\beta_5=p_i \leqslant p_j \longrightarrow w_i \geqslant w_j$: Agreeable weights.

$\beta_5=\phi$: Any w_i are arbitrary positive integers.

(6) $\beta_6 \in \{ r_i \leqslant r_j \longrightarrow d_i \leqslant d_j , \phi \}$

$\beta_6=r_i \leqslant r_j \longrightarrow d_i \leqslant d_j$: Agreeable due dates.

$\beta_6=\phi$: Any d_i are arbitrary non-negative integers.

(7) $\beta_7 \in \{ t_i , \phi \}$

$\beta_7=t_i$: Jobs have transportation times between machines.

$\beta_7=\phi$: No transportation times.

2.4.3 Objective function

The third parameter $\gamma \in \{ f_{max}, \Sigma f_j \}$ indicates the objective functions, which are defined as follows:

$$f_{max} \in \{ C_{max} , L_{max} , T_{max} \}$$

or

$$\Sigma f_j \in \{ \Sigma C_j, \Sigma w_j C_j, \Sigma T_j, \Sigma w_j T_j, \Sigma U_j,$$

$$\Sigma w_j U_j, \Sigma (h_j E_j + w_j T_j), \Sigma w_j e^{\lambda C_j} \},$$

as mentioned in section 2.3.

2.5 Theory of Computational Complexity

In this section we recall the basic concepts of theory of computational complexity. Since the computation time needed to solve a scheduling problem is very important, recent developments in the theory of computational complexity as applied to machine scheduling problems have aroused the interest of many researchers. The techniques of Cook [15] and Karp [46] have been applied extensively in this area, to locate the bordline between P the class of problems for which a polynomial bounded, good or efficient algorithm exists, and NP-complete, the class of problems for which the existence of such algorithm is very unlikely [34].

The class NP is very extensive, and it is clear that $P \subseteq NP$. Further insight into the relation between P and NP is obtained by introducing the following concepts:

(1) A problem π' is reduced to problem π ($\pi' \propto \pi$) if, for any instance (particular case of a problem) of π', an instance of π can be constructed in polynomial bounded time such that solving the instance of π will solve the instance of π' as well. The problem π' and π are equivalent if $\pi' \propto \pi$ and $\pi \propto \pi'$. When $\pi' \propto \pi$ and $\pi \in P$, this implies that $\pi' \in P$. Conversely, if $\pi' \propto \pi$ and $\pi' \notin P$, then $\pi \notin P$.

(2) The time complexity function f(I) of an algorithm gives the maximum number of operations that would be required to solve an instance I. An algorithm has a polynomial time complexity

if its time complexity function f(I) is O(P(I)) for some polynomial P(I). Otherwise it has exponential time complexity [26].

(3) A decision or recognition problem is a problem whose solution is either yes or no. In order to deal with the complexity of a combinatorial (scheduling) minimization problem, we transform it into the problem of determining the existence of a solution with value at most equal to y, for some threshold y [26].

(4) A decision problem π is called NP-complete if $\pi \in$ NP and $\pi' \propto \pi$ for every $\pi' \in$ NP. Thus the NP-complete problems form a subclass of NP. This subclass is formed from the hardest problems in NP. Hence, if one finds a polynomial time algorithm for any NP-complete problem, then all the problems in NP can be solved in polynomial time. This means that P=NP. Although this is still an open problem, the equality of P and NP is considered to be highly unlikely.

It has been shown, that many scheduling problem are NP-complete. Hence, for the present, there is no alternative method of solution for these problems than implicit enumeration algorithms.

Actually, NP-completeness concept is applied to recognition problems. Thus, whenever a scheduling problem is NP-complete, this refers to the recognition problem associated with it. Hence a polynomial algorithm exists either for both problems or for neither, but not for one alone.

If the recognition problem is NP-complete we say that the optimization problem is NP-hard.

(5) An algorithm has pseudo-polynomial time complexity if its maximum number of operations is bounded by a polynomial $q(I,\lambda)$, where I is the instance size, and λ is an upper bound of the data. For a recognition problem π and a polynomial P(I) of the problem size, consider a sub-problem π_p with restriction that all the data are bounded by P(I). Thus, if we have a pseudo-polynomial time algorithm for π with complexity $q(I,\lambda)$, π_p is answerable in $q(I,P(I))$ operations, that is in polynomial time. We say that π is NP-complete in strong sense if for some polynomial P(I), π_p is NP-complete [26]. Thus assuming that P≠NP, π can not be answered in pseudo-polynomial time if it NP-complete in strong sense.

CHAPTER THREE

METHODS OF SOLUTIONS

3.1 Introduction

In this chapter we discuss the well-known methods that have been used to solve machine scheduling problems. The optimal solution of a scheduling problem is to find a processing order of jobs on each machine so that some measure of performance achieves its optimal value. Since a machine scheduling problem is a combinatorial optimization problem, the objective in this kind of problem is to find an optimal schedule from a finite number of feasible schedules. To find one with the smallest value of the performance measure, we could search through all the possibilities, comparing all the schedules. This search of a finite set of schedules must eventually end, and the smallest value (optimal solution) is found. In this case of one machine non-preemptive scheduling problems, there exist n! possible processing orders of the jobs. Hence for the corresponding m-machines problem, there are $(n!)^m$ possible processing orders. This number, $(n!)^m$, is very large even for relatively small values of n and m. In practice, because of the constraints of the problem many of these schedules may be infeasible. However, searching for an optimal schedule among all feasible schedules using complete enumeration is still not suitable.

The best known methods of solution for machine scheduling problems [82] are as follows.

3.2 Complete Enumeration

Enumeration methods generate schedules one by one, searching for an optimal solution. These methods list all possible schedules and then eliminate the non-optimal schedules from the list, leaving those which are optimal. Nevertheless, elimination procedures can sometimes be used to see if the non-optimality of one schedule implies the non-optimality of others not yet generated. Thus, the methods may not search the complete set of feasible solutions. Clearly, searching for an optimal schedule among all possible schedules using complete enumeration is not suitable even for problems of small size.

3.3 Combinatorial Analysis

Combinatorial analysis may lead to very efficient algorithms that produce an optimal schedule in predictable number of steps, with this number increasing at most polynomially in the size of the problem. These methods examine the effect of a minor change in a particular sequence on the value of that sequence. As an example we use the adjacent job interchange argument for $1//\Sigma w_i C_i$ problem. Suppose we have a sequence $\pi_1 = \sigma ij\sigma'$ and consider $\pi_2 = \sigma ji\sigma'$ resulted from π_1 by interchanging job i and j. All the completion times are the same in π_1 and π_2 except for jobs i and j. If T is the completion time of σ, we obtain

$$\sum_{k=1}^{n} w_k C_k (\sigma ji\sigma') - \sum_{k=1}^{n} w_k C_k (\sigma ij\sigma')$$

$$= w_j(T+p_j) + w_i(T+p_j+p_i) - w_i(T+p_i) - w_j(T+p_i+p_j)$$

$$w_i p_j - w_j p_i \qquad (3.1)$$

Hence sequence i before j if (3.1) is not negative, i.e., if

$$w_i p_j - w_j p_i \geq 0 \quad \text{OR} \quad w_i/p_i \geq w_j/p_j.$$

3.4 Branch and Bound Method

We give in this section a general framework of a branch and bound method. Branch and bound methods are enumeration techniques which provide an approach to combinatorial optimization that applies to large class of problems. They were developed and first used by Eastman [19] for the travelling salesman problem and by Land and Doig [49] for integer programming. These methods were first applied to scheduling problems by Lomnicki [61] and Ignall and Schrage [41]. The branch and bound method is probably the solution technique most widely used in scheduling. This method is a typical example of the implicit enumeration approach, which can find an optimal solution by systematically examining subsets of feasible solution. The procedure is usually described by means of search tree with nodes that correspond to these subsets. To minimize an objective function f, for a particular scheduling problem, the branch and bound method successively partitions subsets using a branching procedure and computes bounds using a lower bounding procedure and by these procedures excludes the subsets which are found not to include any optimal solution: this eventually leads to at least one optimal solution.

A lower bound LB on the value of each solution in a subset is calculated. Define a node to be active if the associated lower bound

with this node is the smallest lower bound. If the lower bound calculated for a particular subset is greater than or equal to the upper bound UB, this subset is ignored since any subset with value less than UB can exist only in the remaining subsets. (This upper bound UB is usually defined as the minimum of the values of all feasible solutions currently found. On the other hand, i.e., if no feasible solution is known, UB is initially taken to be infinity until the first feasible solution is found.) These remaining subsets (if any) have to be considered one at a time. One of these subsets is chosen, according to some search strategy, from which to branch. When the branching ends at a complete sequence of the jobs, this sequence is evaluated and if its value is less than the current upper bound UB, this UB is reset to take that value.

The procedure is repeated until all nodes (subsets) have been considered (i.e.,lower bounds of all nodes in the scheduling tree are greater than or equal to the UB); a feasible solution with this UB is an optimal solution.

Hence the branch and bound method is determined by the following three procedures.

3.4.1 The bounding procedure

The bounding procedure indicates how to calculate a lower bound on the optimal solution of a given problem. Clearly the effectiveness of the bound is usually the most important factor, since it determines the efficiency of the complete algorithm. The well-known methods of obtaining lower bounds for machine scheduling problems (for details see Hariri [34] and Beloudah [7]) are as follows.

(1) Relaxation of constraints (see chapters 4 and 7).

(2) Lagrangean relaxation of constraints (see chapters 5,6 and 7).

(3) Dynamic programming state-space relaxation (see chapters 5 and 6).

(4) Relaxation of objective (see chapter 6).

3.4.2 The branching procedure

The branching procedure describes the method used to partition a subset of possible solutions. There five most usual ones used in scheduling are as follows.

(1) Forwards branching : The jobs are sequenced one by one from the beginning, (see chapters 4,5 and 7).

(2) Backwards branching : The jobs are sequenced one by one from the end, (see chapter 6).

(3) At every stage, a job is chosen to be sequenced either at the beginning or at the end according to some heuristic method based on the data of the problem, (see Hariri [34] chapter 8 and chapter 10).

(4) At every stage a job is chosen to be sequenced first, last, directly before or directly after another job (see Potts [74]).

(5) At every stage a job is sequenced either before or after another job. A heuristic can be used to determine this pair of jobs, (see Hariri [34]).

3.4.3 Search strategy

The search strategy describes the method of choosing a node of the search tree to explore. There are three commonly used methods to choose this node.

(1) Branching from the node with smallest lower bound.

(2) Branching from the recently created node.

(3) Branching from a node with smallest lower bound amongst the recently created nodes.

Strategy (1) is known as a frontier search and strategy (2) and (3) are a newest active node search. In chapters 4, 5, 6 and 7 we use strategy (3).

3.4.4 Other features

There are other factors one can introduce to improve the efficiency of the branch and bound method (see Hariri [34] and Potts [73]). For example, one might like to include a heuristic method to obtain an upper bound on the optimum. Also one can introduce dominance rules to specify whether a node can be eliminated before computing its lower bound. For the following reasons dominance rules are computationally useful (see Potts [73]).

(1) to reduce storage requirement on the computer,

(2) to reduce computation time by the avoidance of calculation for the dominated nodes and their successors.

3.5 Dynamic Programming Method

Dynamic programming (DP) is applicable to many optimization problems, and its computer storage requirements are often severe. The DP procedure applies to any problem which can be broken down into a sequence of nested problems, the solution of one being derived in a straightforward fashion from that of the preceding problem [26]. Hence the optimal solution is derived by recursive equations describing the optimal criterion function at any step in termsof previous ones. Held and Karp [38] followed Bellman's ideas and applied DP to machine scheduling problems. DP has also been proved to be good approach for the solution of scheduling problems. The DP is better than the complete enumeration methods. For example, if n=10, (for a particular single machine problem), we may have to examine 10!=3,628,800 possible sequences. On the other hand, if we use DP, we have only to consider $n(2^{n-1})$=5120 calculations in order to select the best sequence [6]. Below we give an example to illustrate the DP approach for the case $1/\ /\Sigma f_j$, that is to minimize the total cost $\Sigma_j f_j(C_j)$. This recursive equation was suggested by Held and Karp [38] and then by Lawler [50].

Let $S \subseteq \{1,\ldots,n\}=N$ be an arbitrary subset of jobs. Also define $f^*(S)$ be the minimum total cost incurred by scheduling the jobs of S in the period $[0,\Sigma_{i\in S} p_i]$. The objective is to find $f^*(N)$ by solving the recursion equations

$$f^*(S)=\min_{i\in S}\{f^*(S-\{i\}) + f_i(\sum_{i\in S} p_i)\} \qquad (3.2)$$

that are initialized by $f^*(\phi)=0$. Solving recursion equations (3.2) is equivalent to find the shortest path in a state-space graph [37].

3.6 Heuristic Methods

One approach to deal with the apparent difficulty of many scheduling problems is to devise computationally efficient algorithms that find schedules which are, for most problem instances, near optimal. This can be achieved by a heuristic method. Moreover, even if a branch and bound is adopted, the first step in such a method is usually the application of a heuristic, since it is well-known that computation can be reduced by using a heuristic method to find a good solution to act as an upper bound.

One method of studying the effectiveness of a heuristic method is to examine its worst-case behaviour. Let f^* denote the optimal solution to a given problem, and f^H denote the near optimal value obtained by using a certain heuristic H. If, whatever the problem data, $f^H \leq \rho f^* + \delta$ for specified constants ρ and δ, where δ is as small as possible, then ρ is called the worst-case performance ratio of heuristic H. This method serves to establish the maximum relative deviation between the optimal and near optimal solutions [76]. First results on the worst-case performance of heuristics were due to Graham [31,32]. Also, Garey, Graham and Johnson [27] give a review of the worst-case performance of scheduling heuristics.

CHAPTER FOUR

SCHEDULING JOBS WITH RELEASE DATES ON A SINGLE MACHINE TO MINIMIZE THE WEIGHTED NUMBER OF LATE JOBS

4.1 Introduction

The problem may be stated as follows. Consider the set of jobs $N=\{1,\ldots,n\}$ and one machine. The machine cannot process more than one job at a time. For each job i, let r_i, p_i, d_i and w_i denote the release date, processing time, due date and weight respectively. For a given processing order of the jobs the (earliest) completion time C_i and the variable U_i, where $U_i=0$ if $C_i \leq d_i$ indicating job i is *early* and $U_i=1$ if $C_i > d_i$ indicating job i is *late*, for job i ($i \in N$) can be computed. The objective is to find a processing order of the jobs that minimizes the weighted number of late jobs Σw_iU_i. We denote this problem by $1/r_i/\Sigma w_iU_i$. Note that once a job i is late ($U_i = 1$) it does not matter how late it is.

The problem $1/r_i/\Sigma w_iU_i$ is clearly equivalent to that of finding and scheduling a subset of jobs $B \subseteq N$ such that all the jobs in B can be completed on time (i.e., $U_i=0$ for all $i \in B$) and the total weight in B is maximum. The $1/r_i/\Sigma U_i$ problem has been shown to be NP-hard [58]. If this $1/r_i/\Sigma U_i$ problem is generalized to allow arbitrary weight w_i for each job i, then obviously the resulting weighted number of late jobs problem $1/r_i/\Sigma w_iU_i$ is NP-hard.

Several special cases yield polynomial algorithms. The $1//\Sigma U_i$ problem can be solved in $O(n \log n)$ steps by using Moore's algorithm [71]: let $\sigma = (\sigma(1),\ldots,\sigma(n))$ be the sequence obtained by ordering the jobs in non-decreasing order of their due dates. In this sequence

if there exists a job $\sigma(i)$ (with i as small as possible) that is completed after its due date, then one of the jobs sequenced in the first i positions and with largest processing time is selected to be late and removed from σ. The procedure continues until all jobs of σ are completed by their due dates.

The weighted version of this problem, $1//\Sigma w_i U_i$, is shown to be NP-hard [46], but it can be solved in pseudo-polynomial time $O(n \max\{d_i\})$ by the dynamic programming algorithm of Lawler and Moore [57]. However, if job weights are agreeable, i.e., there is a renumbering of the jobs so that

$$p_1 \leq p_2 \leq \ldots \leq p_n \text{ , and}$$
$$w_1 \geq w_2 \geq \ldots \geq w_n \text{ ,}$$

then the resulting problem can be solved in $O(n \log n)$ steps using Lawler's algorithm [51].

The $1/r_i/\Sigma U_i$ problem can be solved if there are agreeable due dates, i.e., there is a renumbering of the jobs so that

$$r_1 \leq r_2 \leq \ldots \leq r_n \text{ , and}$$
$$d_1 \leq d_2 \leq \ldots \leq d_n \text{ .}$$

in $O(n^2)$ steps by Kise et al. [47], and in $O(n \log n)$ steps by Lawler [55].

If preemption is permitted, then $1/r_i, pmtn/\Sigma w_i U_i$ problem is not NP-hard, since an $O(n^6)$ dynamic programming algorithm is presented by Lawler [55].

In Section 4.2 of this chapter problem reduction conditions and a heuristic method are described. For the case of unit weights,

various lower bounding schemes that can be used in branch and bound algorithms are derived in Section 4.3. A lower bound for the general case of this problem $1/r_i/\Sigma w_iU_i$ is given in Section 4.4. Dominance rules and a description of our branch and bound algorithms is contained in Section 4.5. Section 4.6 reports on computational experience with these branch and bound algorithms. Some concluding remarks are given in Section 4.7.

4.2 Problem Reduction and Heuristic Method

In the case of our problem $1/r_i/\Sigma w_iU_i$, there exist n! possible orders of the jobs. We first try to reduce the size of the problem by finding a job which precedes or succeeds all other jobs in an optimal schedule. Such a job is removed and hence the number of possibilities is reduced, if one of the following conditions is satisfied.

(1) If $r_i + p_i > d_i$ for any $i \in N$, then job i is late and discarded.

(2) If $r_i + p_i \leq r_j$ for each job $j \in N - \{ i \}$, then job i is early and discarded.

(3) If we have two jobs i and j which cannot both be early; i.e., $r_i + p_i + p_j > d_j$ and $r_j + p_j + p_i > d_i$, such that $p_i \geq p_j$, $w_i \leq w_j$ and $d_i - p_i \leq d_j - p_j$, then job i is late and discarded.

The first condition is clear. The second condition implies that the earliest completion time of job i is less than or equal to the

minimum release dates of job j, where $j \neq i$; hence fixing job i in the first position does not alter the optimal sequence. For the third condition, jobs i and j can not both be early. If job i were early and job j were late, then interchanging so that job j starts at the time job i previously started and job i is late gives a schedule at least as good as the original one. Thus, job i is selected to be late and removed from the set N.

When no further progress can be made with this reduction conditions we use the following heuristic. Henceforth we assume N has been modified to reflect any problem reduction which has taken place.

First we describe the heuristic for the case of unit weights. In the procedure below σ denotes a partial sequence of h early jobs the last of which is completed at time T and L denotes the set of late jobs, also UB =|L| is the number of late jobs. N is the set of jobs which remain to be either scheduled early in σ or included in L. The counter m prevents the procedure repeating an identical execution of Step(3).

Heuristic H

Step (1) : Set σ equal to the empty partial sequence, $L = \phi$, UB=0, h=0, m=0, $T=\min_{i \in N} \{ r_i \}$, and N={ 1,...,n).

Step (2) : If UB + h = n go to Step (7). Otherwise set m=0, if $r_i > T$ for all $i \in N$ set $T=\min_{i \in N} \{ r_i \}$. Among those jobs i with $r_i \leq T$ ($i \in N$), choose a job j with the smallest due date (if there is a tie, choose the one with smallest processing time.)

Step (3) : If job j early in the partial sequence σj (i.e., $\max\{ r_j, T \} + p_j \leq d_j$), go to Step(4). Otherwise set h=h+1, $\sigma(h)=j$ and N=N-{j}, discard a job (say k) from partial sequence σ such that

the current time (completion time) for the resulting sequence is minimum, delete i from σ, set h=h-1, reset T as the completion time of the last job of σ, set L=L U { i }, UB=|L| and go to Step(2).

Step (4) : If N - { j } = ϕ, set h=h+1, σ(h)=j and go to Step(7). Choose job k, where k $\in$ N - { j }, with smallest due date (if there is a tie, choose k with largest processing time). If max { r_k , T + p_j } + p_k > d_k go to Step(5). Otherwise, set h=h+1, set σ(h)=j, N=N -{j}, T=T + p_j, and go to Step(2).

Step (5) : m=m + 1. If m=1, set j=k, set T=max{r_k,T}, go to Step(3). If m=2 go to Step(6).

Step (6) : Set σ(h + 1)=k, σ(h + 2)=j, set h=h + 2 and set N=N-{j,k}. Discard a job (say i) from the partial sequence σ such that the current time for the resulting sequence is minimum, delete i from σ, set h=h-1, reset T as the completion time of the last job of σ, set L=L U { i }, UB=|L|, and go to Step(2).

Step (7) : Stop.

Example 4.1. Consider the number of late jobs problem with six jobs.

Table 4.1. Data for the example

i	1	2	3	4	5	6
r_i	4	8	2	3	6	1
p_i	3	5	3	7	4	3
d_i	10	13	9	12	16	7

The values obtained when heuristic H is applied to this example are given in Table 4.2. In the first iteration Step(2) chooses j=6 as the job with the smallest due date which is available at time t=1.

The procedure passes through Step(3) and then chooses k=3 in Step(4). Since j and k can both be sequenced early, Step(4) includes job 6 in σ. A similar pattern is also observed in the second iteration with j=3 and k=1 which yields σ= (6,3). In the third iteration the values j=1 and k=4 are selected in Step(2) and Step(4). In this case job j is early and job k is late in the partial sequence σjk. Step(5) resets j=4. On executing Step(3) again, job j is late in the partial sequence σj. Step(3) selects t=4 to leave σ=(6,3) and L={4}. The fourth iteration selects j=1 and k=2 in Step(2) and Step(4). In this iteration also job j is early and job k is late in the partial sequence σjk. Step(5) resets j=2. On executing Step(3) again job j is early in the partial sequence σj. Step(4) selects k=1 which shows that job k is late in the partial sequence σjk. On incrementing m in Step(5) we get m=2, now the procedure passes to Step(6). Hence it chooses t=2 to leave σ=(6,3,1) and L={2,4}. The fifth and final iteration chooses j=5 and sets σ=(6,3,1,5) in Step(4). Thus, we obtain UB=2.

Table 4.2.

Iteration	N	h	σ	L	UB	T	j	k	t	m
1	{1,2,3,4,5}	1	(6)	ϕ	0	4	6	3	-	0
2	{1,2,4,5}	2	(6,3)	ϕ	0	7	3	1	-	0
3	{1,2,5}	2	(6,3)	{4}	1	7	1	4	4	1
							4			
4	{5}	3	(6,3,1)	{2,4}	2	10	1	2	2	2
							2	1		
5	ϕ	4	(6,3,1,5)	{2,4}	2	14	5	-	-	0

In heuristic H the job j, selected in Step(2), has the smallest due date amongst those available whereas job k, selected in Step(4) may have a smaller due date than job j. Job k is sometimes used, where appropriate, to be scheduled next in σ rather than job j . It is easily verified that by the choice of k in Steps(3) and (6) the resulting partial sequence σ contains all early jobs.

For the case of arbitrary weights, Step(3) and Step(6) are modified, and in this case if there is a late job, then the main part of the new heuristic is a method for selecting one or two jobs with minimum weights of the jobs in σj in Step(3) or σjk in Step(6) to be late and removing it from the sequence such that the completion time T is minimum.

4.3 Lower Bounds

In this section, we derive four lower bounds for the number of late jobs. These bounds are each obtained by using a relaxation on the release date r_i of the original problem, and then applying Moore's algorithm to the relaxed subproblem.

4.3.1 Moore lower bound

To construct a lower bound $LB_M(N)$, we use a common release date $r^* = \min_{i \in N} \{ r_i \}$. We relax release dates to $r_i = r^*$ for each job i to give a $1/ /\Sigma U_i$ problem having due date $d_i = d_i - r^*$ for each job i ($i \in N$). This relaxed problem is solved by Moore's algorithm [71]: Jobs are first sequenced in non-decreasing order of d_i ($i \in N$). Assume the resulting sequence is $(1,\ldots,n)$. If there is no late job, then this sequence is optimal; otherwise find the first late job say k, and then one of the jobs sequenced in the first k positions and with

largest processing time is selected to be late and removed from the set N. The procedure is repeated until all the remaining jobs are completed by their due dates. Hence the lower bound $LB_M(N)$ is equal to the number of late jobs in the relaxed problem. The computation of the lower bound $LB_M(N)$ requires O(n log n) steps. From our initial computational experience we have seen that this bound is weak, and we modify it as described in the following sections.

4.3.2 Modified Moore lower bound

In this section a subproblem of the original problem is selected and Moore's algorithm is used for each subproblem to compute a lower bound. Suppose that some of the unscheduled jobs are ignored to leave the set of jobs N', where $N' \subseteq N$, and let $P(N')$ denote this subproblem. Then $LB_M(N')$ is computed for the remaining unscheduled jobs in N'. For a suitable choice of N', it will be the case that $LB_M(N') \geq LB_M(N)$. A suitable set N' will contain jobs i having large release date r_i. The following method is used to find N'.

Sequence the jobs in non-decreasing order of r_i ($i \in N$). Assume that the resulting sequence is $(\sigma(1),\ldots,\sigma(n))$. The set N' is obtained from the set N by successively choosing a common release date r^*. The search for the set N' starts with release date of the first job as a common release date (i.e., $r^* = r_{\sigma(1)}$). For this common release date, we choose $N_1 = \{ \sigma(1),\ldots,\sigma(n) \}$, and use the method of section 4.3.1 to find $LB_M(N_1)$ for the subproblem $P(N_1)$. Then set $r^* = r_{\sigma(2)}$, choose $N_2 = \{ \sigma(2),\ldots,\sigma(n)\}$, use the same method above to find $LB_M(N_2)$ for the subproblem $P(N_2)$ and continue up to $r^* = r_{\sigma(n-1)}$. This means we find the sets $N_v = \{ \sigma(v),\ldots,\sigma(n) \}$ and compute the lower bounds $LB_M(N_v)$ for each subproblem $P(N_v)$, $v=1,\ldots,n-1$. Hence N'

is the set of jobs that gives the maximum of $LB_M(N')$, i.e., $LB_M(N^*)=\max\{LB_M(N_1),\ldots,LB_M(N_{n-1})\}$. Clearly, the computation of the lower bound $LB_M(N^*)$ requires $O(n^2 \log n)$ steps. It is clear that this lower bound $LB_M(N^*)$ dominates the lower bound $LB_M(N)$ which is equal to $LB_M(N_1)$.

4.3.3 Kise lower bound

Now we shall construct the third lower bound LB_K by adjusting the release dates and due dates to satisfy the agreeability condition (if $r_i < r_j$, then $d_i \leq d_j$ for each $i,j \in N$). To relax the original problem, we either decrease the release date or increase the due date for some job i ($i \in N$), such that the condition of agreeability is satisfied. More precisely, if an ordering $\sigma=(\sigma(1),\ldots,\sigma(n))$ of the jobs is selected, release dates and due dates are reset using

$$r_{\sigma(i)}=\min_{j\in\{i,\ldots,n\}}\{r_{\sigma(j)}\} \text{ for } i=1,\ldots,n, \quad \text{and}$$

$$d_{\sigma(i)}=\max_{j\in\{1,\ldots,i\}}\{d_{\sigma(j)}\} \text{ for } i=1,\ldots,n,$$

so that after resetting

$$r_{\sigma(1)} \leq \ldots \leq r_{\sigma(n)} \quad \text{and}$$

$$d_{\sigma(1)} \leq \ldots \leq d_{\sigma(n)}.$$

We call σ an agreeability sequence.

An obvious method, which is a relaxation in the release date only, is to find the smallest release date r_i ($i \in N$) and set all the

release dates $r_j=r_i$ for $j \in M_i$, where $M_i = \{ j \in N: d_j \leq d_i \}$. Sequence the jobs of M_i in non-decreasing order of due dates to fill the first unoccupied positions of the agreeability sequence. Set $N=N-M_i$, and repeat the procedure above until N is empty.

Suppose the agreeability sequence is $\sigma=(\sigma(1),\ldots,\sigma(n))$. In this sequence if there exists a job $\sigma(k)$ (where k is as small as possible) that is completed after its due date, then one of these jobs sequenced in the first k positions is late. Otherwise all the remaining jobs are completed within their due dates. To select the late job, let $B=\{ \sigma(1),\ldots, \sigma(k) \}$ and let C(B) be the completion time for the jobs of B when sequenced in agreeability order. Let $\sigma(t)$ be the late job that is selected to be removed from the set B, chosen such that the completion time of jobs $B - \{\sigma(t)\}$ is minimal, i.e., if job $\sigma(t)$ satisfies

$$C(B - \{ \sigma(t) \}) \leq C(B - \{ \sigma(i) \}), \sigma(i) \in B,$$

then $\sigma(t)$ is selected to be late. This job $\sigma(t)$ is found applying a procedure of Kise [47]. The lower bound LB_K is the sum of these late jobs. It can be obtained in $O(n^2)$ steps, once the agreeability sequence is known. However, since our procedure for relaxing release dates and finding the agreeability sequence requires O(n log n) steps, the lower bound LB_K is computed in $O(n^2)$ steps.

It is also possible to devise a method of constructing an agreeability sequence for which an attempt is made to ensure that the reduction in release dates and the increase in due dates is as small as possible. However, initial experiments indicate that the increase in the lower bound over that obtained with the procedure described

above does not compensate for its extra computational requirements.

4.3.4 Linear programming lower bound

To construct the linear programming bound LB_{LP} for the number of late jobs, we use results obtained from the modified Moore bound $LB_M(N')$ given in section 4.3.2. Let x_i be a zero-one variable defined by

$$x_i = \begin{cases} 0 & \text{if } C_i \leq d_i \\ 1 & \text{otherwise,} \end{cases} \qquad (4.1)$$

where C_i is the completion time of job i. The objective of the problem is to find the minimum number of late jobs (i.e., min $\Sigma_{i \in N} x_i$).

To derive constraints for a zero-one programming problem, we consider the constraints obtained from jobs of the subset N_V, of the subproblem $P(N_V)$. Assume the jobs of N_V (V=1,...,n-1) are 1,...,n' where $d_1 \leq \ldots \leq d_{n'}$. Assume also that due dates are reset using $d_i = d_i - r^*$, where $r^* = \min_{j \in N_V} \{ r_j \}$ is the common release date used in the completion of $LB_M(N_V)$. Let k_1 be the first late job found by Moore's algorithm in the completion of the lower bound $LB_M(N_V)$, i.e.,

$$p_1 + \ldots + p_{k_1} > d_{k_1}$$

Then one of these first k_1 jobs must be late which gives the constraint

$$x_1 + \ldots + x_{k_1} \geq 1. \qquad (4.2)$$

In Moore's algorithm one of the jobs sequenced in the first k_1 positions and with largest processing time (say job j_1) is selected to be late and removed from the subset N_V. The procedure continues until all the remaining jobs are completed by their due dates. If k_2 is a second late job found by Moore's algorithm in the computation of the lower bound $LB_M(N_V)$, i.e.,

$$p_1 + \ldots + p_{k_1} + \ldots + p_{k_2} - p_{j_1} > d_{k_2}$$

where j_1 is the first selected late job. Then two of these first k_2 jobs must be late which gives the constraint

$$x_1 + \ldots + x_{k_1} + \ldots + x_{k_2} \geq 2. \qquad (4.3)$$

We use the same method above to find any other constraints for this subset N_V (V=1,...,n-1). Hence for each subset N_V given in section 4.3.2 either there is no constraint (i.e., there is no late job) or there is at least one constraint. The objective is to find the minimum number of late jobs satisfying the constraints that are obtained by applying Moore's algorithm to each subset $N_1,\ldots,N_{n-1}$.

Consider the following linear programming problem.

$$(LP_R) \qquad \begin{aligned} &\text{Min } e^T x \\ &\text{subject to} \\ &Ax \geq b, \\ &-x \geq -e, \\ &x \geq 0, \end{aligned}$$

where e is a column vector of ones, $x^T=(x_1, \ldots, x_n)$ is the vector of variables defined by (4.1) but with the zero-one restrictions relaxed (and replaced by $-x \geq -e$ and $x \geq 0$) and $Ax \geq b$ are all constrained of the form (4.2), (4.3), etc obtained from $P(N_1)$, ..., $P(N_{n-1})$. Let m denote the number of constraints contained in $Ax \geq b$.

Even with the zero-one restrictions on the variables, problem (LP_R) is not guaranteed to provide a feasible solution to our original scheduling problem. Nevertheless, since all constraints $Ax \geq b$ are valid for the original problem, it is a relaxation. Therefore, the solution of problem (LP_R) provides a lower bound.

Instead of solving this primal problem optimally to obtain the lower bound LB_{LP}, we consider the dual problem. Any feasible solution of the primal has a value greater than or equal to the value of any feasible solution of the dual. Hence to obtain a lower bound LB_{LP}, a heuristic method is used to find a feasible solution to the following dual problem:

$$\begin{aligned} &\text{Max } b^Ty - e^Tz \\ &\text{subject to} \\ &A^Ty - z \leq e, \\ &y \geq 0, \\ &z \geq 0, \end{aligned}$$

where $y^T=(y_1, \ldots, y_m)$ and $z^T=(z_1, \ldots, z_n)$.

The following heuristic is used to solve the dual problem above. We initially start with the feasible solution $y=0$ and $z=0$. Then set $y_i=1$, where i is selected so that $b_i=\max_j \{ b_j \}$. This gives a feasible solution for which the lower bound is b_iy_i. It is clear that

this feasible solution with value b_i is equal to the lower bound $LB_M(N')$ given in section 4.3.2.

To derive the general step of the heuristic, consider the t^{th} constraint of $A^T y - z \leq e$ and assume it is of the form

$$a_{1t}\, y_1 + \ldots + a_{it}\, y_i + \ldots + a_{mt}\, y_m - z_t \leq 1 \qquad (4.4)$$

If $a_{it}=0$, then increasing the value of y_i from 0 to 1 leaves the solution feasible with respect to this constraint. Consider now the alternative case $a_{it}=1$. (Recall that by the construction of the inequalities each element of A is zero or one). If (4.4) is satisfied as a strict inequality then increasing y_i from 0 to 1 leaves the solution feasible with respect to (4.4). Lastly if $a_{it}=1$ and (4.4) is satisfied as an equality, then to maintain feasibility with respect to (4.4) when y_i is increased from 0 to 1, it is necessary to increase z_t by 1 and hence decrease the objective function by 1. Therefore the contribution V_i that would be obtained by setting $y_i=1$, where previously $y_i=0$, to the lower bound is calculated as follows.

$$V_i = b_i - |\{\, t:\ a_{ti}^T=1;\ \textstyle\sum_{j=1}^{m} a_{tj}^T\, y_j - z_t = 1\}|.$$

The contribution V_i is added to the objective function if it is greater than zero. These contributions are computed for each i (i=1, ...,m) and whenever a positive contribution is found y, z and the lower bound are updated accordingly. The computation of the lower bound LB_{LP} requires $O(n^3)$ steps. The following example illustrates the linear programming bound LB_{LP} obtained using this heuristic method for solving the dual problem.

Example 4.2. Consider the number of late jobs problem with seven jobs.

Table 4.3. Data for the example

i	1	2	3	4	5	6	7
r_i	0	4	9	15	21	28	30
p_i	6	8	4	3	9	7	6
d_i	17	13	16	31	38	40	39

Using the techniques of linear programming bound given above we get the following constraints

$x_2 + x_3 + x_1 \geq 1$, for $r^*=0$,

$x_2 + x_3 + x_4 + x_5 + x_7 + x_6 \geq 1$, for $r^*=4$,

$x_5 + x_7 + x_6 \geq 1$, for $r^*=21$, and

$x_7 + x_6 \geq 1$, for $r^*=28$.

Hence the linear programming problem is

$$\text{Min } \sum_{i=1}^{7} x_i$$

subject to

$$\text{(Primal)} \quad \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$-x_i \geq -1 \quad i=1, \ldots,7,$$

$$x_i \geq 0 \quad i=1, \ldots,7.$$

The dual problem for this linear programming problem is

$$\text{Max } \sum_{i=1}^{4} y_i - \sum_{i=1}^{7} z_i$$

subject to

$$\text{(Dual)} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} - \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$y_i \geq 0 \quad i=1, \ldots,4,$$

$$z_i \geq 0 \quad i=1, \ldots,7.$$

A feasible solution to this dual problem is obtained by using the above heuristic.

Initially, $y_1=y_2=y_3=y_4=0$. Since in this case, all coefficients of y_i in the dual objective are equal to 1, we arbitrarily start with i=1. The current lower bound is now equal to 1. Since $V_2=-1$, the value $y_2=0$ is retained. However, $V_3=1$, so we set $y_3=1$ to give a current lower bound equal to 2. At this stage the lower bound cannot be improved further because $V_4=-1$. Hence the lower bound $LB_{LP}=2$ is obtained, which is better than the value $LB_M(N')=1$ obtained by using the modified Moore bound. Our bound $LB_{LP}=2$ is exact because an optimal sequence is (1,3,4,5,7,2,6) and the number of late jobs is 2.

We note that the second and third constraints of the primal are redundant. Such constraints could be removed without causing any reduction in value of the lower bound.

4.4 Lower Bounds for the Weighted Problem

In this section we derive two lower bounds for the weighted number of late jobs problem $1/r_i/\sum_{i=1}^{n} w_iU_i$. These bounds are each derived by using the same techniques of the lower bounds of $1/r_i/\sum_{i=1}^{n} U_i$ problem, given in section 4.3.2 and section 4.3.4.

4.4.1 Modified Moore lower bound

To construct a lower bound $LBW_M(N')$ for the weighted number of late jobs, we use the same techniques as for the modified Moore bound $LB_M(N')$ given in section 4.3.2. Since N' be the set of jobs that gives maximum number of late jobs in the relaxed problem, sequence the jobs of N' in non-decreasing order of w_i. Assume that the resulting sequence is $(1,\ldots,n')$, and the number of late jobs is k,

i.e., $k=LB_M(N')$. Hence a lower bound $LBW_M(N')$ is given by the k smallest weights of jobs of N', i.e., by

$$LBW_M(N') = \sum_{i=1}^{k} w_i.$$

The computation of the lower bound $LBW_M(N')$ requires $O(n^2 \log n)$ steps.

4.4.2 Linear programming bound

To construct this bound, we apply the procedure of linear programming bound given in section 4.3.4. The objective of the problem is to find the minimum weighted number of late jobs (i.e., min $\sum_{i=1}^{n} w_i x_i$). More precisely, let the variable x_i ($i \in N$) be defined by (4.1). Clearly the constraints $Ax \geq b$ that are valid in section 4.3.4 are also valid for our weighted problem. Therefore, a valid relaxation is given by the linear programming problem.

$$\begin{aligned} &\text{Min } w^T x \\ &\text{subject to} \\ \text{(Primal)}\quad &Ax \geq b, \\ &-x \geq -e, \\ &x \geq 0, \end{aligned}$$

where $w^T=(w_1,\ldots,w_n)$ is the weighted vector, A, b and x are defined in section 4.3.4.

To obtain a lower bound LBW_{LP}, a heuristic method is used to find a feasible solution to the following dual problem:

$$\text{Max } b^T y - e^T z$$

subject to

$$A^T y - z \leq w ,$$

(Dual)

$$y \geq 0 ,$$

$$z \geq 0 .$$

The heuristic solution to the dual problem is computed as follows. First let $b_i = \max_{j=1,\ldots,m} \{ b_j \}$ and set $y_j = 0$ for each j, where $j \neq i$. Also we set $y_i = w_k$ for some k that is chosen as follows. Recall that constraint ℓ^{th} of the dual is

$$a_{1\ell}\, y_1 + \ldots + a_{i\ell}\, y_i + \ldots . + a_{m\ell}\, y_m - z_\ell \leq w_\ell .$$

If $a_{i\ell} = 0$ or if $w_k \leq w_\ell$, then setting $y_i = w_k$ and $z_\ell = 0$ gives a feasible solution with respect to this constraint. Alternatively, if $a_{i\ell} = 1$ and $w_k > w_\ell$, then setting $y_i = w_k$ and $z_\ell = w_k - w_\ell$ also gives a feasible solution with respect to this constraint. Therefore, setting $y_i = w_k$ and

$$z_\ell = \begin{cases} w_k - w_\ell \text{ if } a_{i\ell} = 1 \text{ and } w_k > w_\ell, \ \ell = 1,\ldots,n \\ \\ 0 \quad \text{otherwise} \end{cases}$$

provides a contribution

$$G_k = b_i\, w_k - \Sigma_{\ell=1}^{n} z_\ell ,$$

to the dual objective. When G_j is computed for each j, we choose k such that $G_k = \max_{j=1,\ldots,n} \{ G_j \}$ to give a lower bound

$LBW_{LP} = G_k$.

Although it may be possible to perform further adjustments to the values of y and z, it is felt that the extra computation is unlikely to be worthwhile. The computation of the lower bound LBW_{LP} requires $O(n^3)$ steps.

The following example illustrates the modified Moore bound $LBW_M(N')$ and the linear programming bound LBW_{LP}.

Example 4.3. Consider the weighted number of late jobs problem with nine jobs.

Table 4.4. Data for the example

i	1	2	3	4	5	6	7	8	9
r_i	4	8	9	2	8	10	5	5	3
p_i	3	4	2	10	8	8	5	3	10
w_i	10	4	7	7	3	7	9	6	7
d_i	24	22	20	35	22	29	22	22	34

Applying the results of modified Moore bound $LBW_M(N')$ and the procedure of the linear programming bound LBW_{LP} yields the following.

Table 4.5.

r^*	Subset (N_V)	Constraints	
		$x_3+x_2+x_5+x_7+x_8$	≥ 1
2	N_1={1,2,3,4,5,6,7,8,9}	$x_3+x_2+x_5+x_7+x_8+x_1+x_6+x_9$	≥ 2
		$x_3+x_2+x_5+x_7+x_8+x_1+x_6+x_9+x_4$	≥ 3
3	N_2={1,2,3,5,6,7,8,9}	$x_3+x_2+x_5+x_7+x_8$	≥ 1
		$x_3+x_2+x_5+x_7+x_8+x_1+x_6+x_9$	≥ 2
4	N_3={1,2,3,5,6,7,8}	$x_3+x_2+x_5+x_7$	≥ 1
5	N_4={2,3,5,6,7,8}	$x_3+x_2+x_5+x_7$	≥ 1
5	N_5={2,3,5,6,8}	$x_3+x_2+x_5+x_8+x_6$	≥ 1
8	N_6={2,3,5,6}	$x_3+x_2+x_5+x_6$	≥ 1
8	N_7={3,5,6}	no constraints	
9	N_8={3,6}	no constraints	

It is clear from Table 4.5 that $N' = N_1$ and $k = LB_M(N') = 3$. The three jobs, chosen from N', with the smallest weights are 2, 5 and 8 hence $LBW_M(N') = w_2+w_5+w_8 = 13$.

Note that it is sufficient to compute constraints only for each distinct value of r_i. Thus in Table 4.5 computation could be saved by not performing calculations for sets N_5 and N_7.

Note that from Table 4.5, if the redundant constraints are

removed then we get the following linear programming problem

$$\text{Min } 10x_1 + 4x_2 + 7x_3 + 7x_4 + 3x_5 + 7x_6 + 9x_7 + 6x_8 + 7x_9$$

subject to

$$\begin{aligned}
x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 &\geq 3\\
x_1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_8 + x_9 &\geq 2\\
x_2 + x_3 + x_5 + x_7 &\geq 1\\
x_2 + x_3 + x_5 + x_6 &\geq 1\\
-x_i &\geqslant -1 \quad i=1,\ldots,9\\
x_i &\geqslant 0 \quad i=1,\ldots,9.
\end{aligned}$$

The dual problem for this linear programming problem is

$$\text{Max } 3y_1 + 2y_2 + y_3 + y_4 - \sum_{i=1}^{9} z_i$$

subject to

$$\begin{bmatrix}
1 & 1 & 0 & 0\\
1 & 1 & 1 & 1\\
1 & 1 & 1 & 1\\
1 & 0 & 0 & 0\\
1 & 1 & 1 & 1\\
1 & 1 & 0 & 1\\
1 & 1 & 1 & 0\\
1 & 1 & 0 & 0\\
1 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix} y_1\\ y_2\\ y_3\\ y_4 \end{bmatrix}
-
\begin{bmatrix} z_1\\ z_2\\ z_3\\ z_4\\ z_5\\ z_6\\ z_7\\ z_8\\ z_9 \end{bmatrix}
\leqslant
\begin{bmatrix} 10\\ 4\\ 7\\ 7\\ 3\\ 7\\ 9\\ 6\\ 7 \end{bmatrix}$$

$$\begin{aligned}
y_i &\geqslant 0 \quad i=1,\ldots,4\\
z_i &\geqslant 0 \quad i=1,\ldots,9.
\end{aligned}$$

A feasible solution to this dual problem is now obtained by applying the above heuristic. In this case, the coefficient of y_1 in the dual objective is 3, which is the maximum coefficient of y_1. Hence set $y_2 = y_3 = y_4 = 0$. Since $G_1 = 0$, $G_2 = 11$, $G_3 = G_4 = G_6 = G_8 = G_9 = 13$, $G_5 = 9$, and $G_7 = 5$, hence the lower bound $LBW_{LP} = 13$. An optimal sequence is (1,7,3,8,6,4, 9,5,2) for this example, and the weighted number of late jobs is 14.

4.5 Branch and Bound Algorithm

We now give a description of our branch and bound algorithm and its implementation. Heuristic H given in section 4.2 is applied at the top of the search tree to yield an upper bound on the cost of an optimal schedule.

Our algorithm uses a forward sequencing branching rule for which nodes at level t of the search tree correspond to initial partial sequences in which early jobs are sequenced in the first t positions.

Dominance Rules

If it can be shown that an optimal solution can always be generated without branching from a particular node of the search tree, then that node is dominated and can be eliminated. Dominance rules usually specify whether a node can be eliminated before its lower bound is calculated. Clearly, dominance rules are particularly useful when a node can be eliminated which has a lower bound that is less than the optimum solution [35].

The first result is based on an idea of Villarreal and Bulfin [93].

Theorem (4.1). If $r_i \leq r_j$, $p_i \leq p_j$, $w_i \geq w_j$ and $d_i \geq d_j$ for $i, j \in N$, then if job j is early, then job i is early, and if job i is late, then job j late.

Proof. For any schedule in which job i is late and job j is early, by interchanging these two jobs a new schedule is produced for which the weighted number of late jobs is not increased. □

By repeated application of Theorem (4.1), define

$B(j)=\{i : i \in N, r_i \leq r_j, p_i \leq p_j, w_i \geq w_j \text{ and } d_i \geq d_j\}$ and

$A(j)=\{i: i \in N, j \in B(i)\}$.

Then the set of jobs B(j) must be early if job j is early, and A(j) is the set of jobs must be late if job j is late. These sets are used in branch and bound algorithm to fix certain jobs.

Let σ be an initial partial sequence of jobs, let S be the set of jobs not sequenced in σ and let $C(\sigma)$ denote the completion time of the last job of σ.

The following results show when any of the immediate successors of the node corresponding to an initial partial sequence σ are dominated. The earliest start time of an unsequenced job i is max $\{C(\sigma), r_i\}$.

The next of our dominance theorems is based on a result of Dessouky and Deogun [18] for the problem $1/r_i/\Sigma C_i$. It states that the machine should not be kept idle throughout a time interval within which another job can be completely processed.

Theorem (4.2). If there exists $i \in S$ such that the earliest completion time is

$C_i(\sigma i) = \max\{C(\sigma), r_i\} + p_i \leq d_i$ and $C_i(\sigma i) \leq r_j$ for any $i, j \in S$, then σj is dominated.

Proof. Any complete sequence beginning with the initial partial sequence σj can be modified by removing job i from its original position and inserting it directly before job j. This will not increase the weighted number of late jobs. □

Our final result is a consequence of dynamic programming. If the final two jobs of a partial sequence can be interchanged without increasing the time at which the machine become available to process the next unsequenced job, then this partial sequence is dominated. We assume that $\sigma = \sigma_1 h$, whenever σ is not empty.

Theorem (4.3). For $j \in S$, if we have two initial partial sequences of early jobs $\sigma_1 jh$ and $\sigma_1 hj$ such that $C(\sigma_1 jh) \leq C(\sigma_1 hj)$, then $\sigma_1 hj$ is dominated.

Note that if in Theorem (4.3), $C(\sigma_1 jh) = C(\sigma_1 hj)$ then either $\sigma_1 hj$ or $\sigma_1 jh$ (but not both) is discarded.

For the case that $r_i \leq C(\sigma)$ for each job $i \in S$, then the $1/r_i/\Sigma U_i$ problem is solved by Moore's algorithm, and $1/r_i/\Sigma w_i U_i$ problem is solved by the algorithm of Lawler and Moore [57].

For all nodes that remain after we apply the dominance theorems, we can use the procedure described in the previous section to compute a lower bound. If the lower bound for any node is greater than or equal to the current upper bound already computed, then this node is discarded. The lower bounding method described earlier is applied to

the unsequenced jobs S with release date reset using $r_j=\max\{r_j,C(\sigma)\}$ for each job $j \in S$.

Finally, the search strategy used in our branch and bound algorithm is the newest active node search.

4.6 Computational Experience with the Branch and Bound Algorithms

4.6.1 Test problems

The branch and bound algorithms were tested on the number of late jobs problems with 20,30,....,200 jobs as well as on the weighted number of late jobs problems with 10,20,30 jobs, that were generated as follows. For job j ($j \in N$), an integer processing time P_j is generated from the uniform distribution [1,100]. Problem 'hardness' is likely to depend on release dates r_j and due date d_j ($j \in N$). Let $T=\Sigma_{j \in N} P_j$ and select a value of R from the set { 0.2,0.5,0.8,1.1 }; an integer release date r_j is generated for each job j, from the uniform distribution [0,RT], where R controls the range of the distribution. An integer due date d_j is generated from the uniform distribution [DT,ET] for each job j, were D and E are selected from the sets { 0.3,0.6,0.9 } and { D+0.3,D+0.6,D+0.9} respectively . For the $1/r_j/\Sigma w_jU_j$ problem an integer weight w_j is generated from the uniform distribution [1,10] for each job j.

For every value of n, five $1/r_j/\Sigma U_j$ problems and five $1/r_j/\Sigma w_jU_j$ problems are generated for each of the 36 sets of values of R, D and E, yielding 180 problems of each type for each value of n.

4.6.2 Computational results

The algorithms of sections 4.3 and 4.4 were coded in FORTRAN IV and run on CDC 7600 computer. Whenever a problem remained unsolved within the time limit of 60 seconds, computation was abandoned for that problem. Comparative computational results for the algorithms of section 4.3 for n=20,30,40,50 are given in Table 4.6. This Table gives the average computation time in seconds or a lower bound on the average computation time when there are unsolved problems and lists the number of unsolved problems as well as number of problems solved that requires over 1000 nodes.

We start by discussing the results obtained using our lower bounds given in section 4.3. For all the bounds, the problems with small R are easiest, because for small R the release date become unimportant once a few jobs have been sequenced. A further analysis of the results indicates that the values of D and E do not significantly affect problem hardness.

Table 4.6. Comparison of branch and bound algorithms

	Algorithms											
	Moore			Modified Moore			Kise			Linear		
n	ACT	NS	NU	ACT	NS	NU	ACT	NS	NU	ACT	NS	NU
20	0.01	0	0	0.01	0	0	0.01	0	0	0.01	0	0
30	0.02	2	0	0.02	1	0	0.02	2	0	0.02	1	0
40	0.18	1	0	0.06	0	0	0.17	1	0	0.08	0	0
50	1.04	0	3	0.43	0	1	1.05	0	3	0.46	0	1

ACT: average computation time or a lower bound on average computation time with unsolved problems contributing 60

seconds.

NS : number of problems solved that require over 1000 nodes.

NU : number of unsolved problems.

From our initial computational experience with the lower bounds, and by the results of Table 4.6 we observe that Moore lower bound $LB_M(N)$ and Kise lower bound LB_K are rather weak and are clearly inferior to the other bounds $LB_M(N')$ and LB_{LP}. Thus the computational results for bounds $LB_M(N)$ and LB_K are abandoned. We now discuss the merits of algorithms $LB_M(N')$ and LB_{LP}. It is clear that the bound LB_{LP} is greater than or equal to the bound $LB_M(N')$. However, in practice, the bound LB_{LP} takes slightly more time to compute and gives identical search trees to those generated by $LB_M(N')$ for our test problems. Hence we adopted the modified Moore algorithm as the best algorithm for further investigation.

Table 4.7 gives the average computation time in seconds or a lower bound on the average computation time when there are unsolved problems, lists the number of unsolved problems and shows the numbers of solved problems that require not more than 100 nodes, that require over 100 and not more than 1000 nodes and that require over 1000 nodes, for algorithm $LB_M(N')$.

We first observe from Table 4.7 that for all values of n, less than 9% of problems are unsolved. Also, 90% of the problems are solved with no more than 100 nodes. The algorithm is clearly very effective when $n \leq 80$ with very few unsolved problems.

Table 4.7. Average computation times for $1/r_j/\Sigma U_j$ problem

Modified Moore Algorithm					
n	ACT	NS1	NS2	NS3	NU
60	0.69	164	12	3	1
70	0.70	165	6	9	0
80	2.03	165	2	11	2
90	2.22	166	4	6	4
100	3.52	166	1	5	8
110	3.83	167	0	7	6
120	5.45	162	1	4	13
130	4.32	167	0	1	12
140	4.29	166	1	2	11
150	4.60	167	0	0	13
160	4.81	166	1	0	13
170	5.82	164	0	0	16
180	6.16	163	0	1	16
190	5.85	164	0	1	15
200	5.85	165	0	0	15

ACT: average computation time or a lower bound on average computation time with unsolved problems contributing 60 seconds.

NS1: number of problems solved that require not more than 100 nodes.

NS2: number of problems solved that require over 100 nodes and not more than 1000 nodes.

NS3: number of problems solved that require over 1000 nodes.

NU : number of unsolved problems.

We now concentrate on the problem $1/r_1/\Sigma w_1U_1$. Results comparing the lower bounds $LBW_M(N')$ and LBW_{LP} of section 4.4 are given in Table 4.8. As in Table 4.7, average computation times and details of numbers of nodes required are given.

From our initial computational experience with the lower bounds, and by the results of Table 4.8 we observe that the bounds are rather weak. We also, observe that the bound LBW_{LP} appears to be slightly better than the bound $LBW_M(N')$ for the problems tested. Each algorithm appears capable of solving problems with up to 20 jobs satisfactorily.

Table 4.8. Average computation times for $1/r_1/\Sigma w_1U_1$ problem

n	Algorithm	ACT	NS1	NS2	NS3	NU
10	$LBW_M(N')$	0.01	180	0	0	0
	LBW_{LP}	0.01	180	0	0	0
20	$LBW_M(N')$	6.02	151	7	6	16
	LBW_{LP}	5.73	153	6	6	15
30	$LBW_M(N')$	10.18	143	5	2	30
	LBW_{LP}	9.53	144	6	2	28

ACT: average computation time or a lower bound on average computation time with unsolved problems contributing 60 seconds.

NS1: number of problems solved that require not more than 100 nodes.

NS2: number of problems solved that require over 100 nodes and not more than 1000 nodes.

NS3: number of problems solved that require over 1000 nodes.

NU : number of unsolved problems.

4.7 Concluding Remarks

For the $1/r_i/\Sigma U_i$ problem the branch and bound algorithm using modified Moore bound is the most efficient and is able to solve problems with up to 200 jobs. All other algorithms are satisfactory for solving problems of size up to 50 jobs

For the problem $1/r_i/\Sigma\ w_iU_i$, a sharper lower bound is needed to cut down the size of the search tree when the number of jobs exceeds 10. One way to improve the results might be to try a different method of finding a feasible solution for the dual problem.

The computational results (for both problems) indicate that the problems with small range of release dates are easiest, because for small R the release dates become unimportant once a few jobs have been sequenced. Also, the computational results (for both problems) show that the performance of the dominance rules was remarkably good.

CHAPTER FIVE

DYNAMIC PROGRAMMING STATE-SPACE RELAXATION FOR SINGLE MACHINE SCHEDULING

5.1 Introduction

The problem considered in this chapter of scheduling jobs on a single machine to minimize total cost is stated as follows. Each job of the set $N = \{1,\ldots,n\}$ is to be processed without interruption on a single machine that can handle only one job at a time. Job i ($i \in N$) becomes available for processing at time zero, requires an integer processing time p_i and incurs a cost $g_i(t)$ if it is completed at time t. Machine idle time is not permitted, i.e., all jobs are processed during the interval $[0,T]$, where $T = \sum_{i \in N} p_i$. The objective is to find a processing order of the jobs with associated completion times C_i ($i \in N$) that minimizes the total cost $\sum_{i \in N} g_i(C_i)$.

When, for each job i ($i \in N$), g_i is a non-decreasing function, Rinnooy Kan et al.[83] derive dominance rules that restrict the search for an optimal solution. Such rules can be used in dynamic programming algorithms (Schrage and Baker [86] and Lawler [53]) or in branch and bound algorithms (Rinnooy Kan et al. [83] and Fisher [24]). Computational results for the total tardiness problem in which $g_i(t) = \max\{t - d_i, 0\}$, where d_i is the due date of job i, indicate that Fisher's branch and bound algorithm is superior to that of Rinnooy Kan et al. For the total weighted tardiness problem in which $g_i(t) = w_i\max\{t - d_i, 0\}$, where w_i is the weight for job i, the special-purpose branch and bound algorithm of Potts and Van Wassenhove [80] that exploits the piecewise-linearity of g_i in its

lower bounding rule appears to be the most efficient algorithm.

It is not assumed that g_i ($i \in N$) is non-decreasing in this chapter. Thus, the dominance rules of Rinnooy Kan et al. cannot be used to generate precedence relations between jobs. Consequently, the dynamic programming algorithms of Schrage and Baker, and Lawler require excessive core storage when there are twenty or more jobs. Of those currently available, Fisher's branch and bound algorithm appears to be the only one that may be capable of solving problems of medium size.

In this chapter we investigate the use of the dynamic programming state-space relaxation technique for obtaining lower bounds as an alternative to the approach employed by Fisher. To obtain a lower bound, Fisher performs a Lagrangean relaxation of the capacity constraint that the machine can process only one job at a time during each of the $T = \sum_{i \in N} p_i$ unit time intervals for which the machine is required to be busy. The multipliers are obtained using the subgradient optimization technique [39]. The dynamic programming state-space relaxation method is developed by Christofides et al.[13] for various routing problems. In this method, a relaxed problem is obtained from a dynamic programming formulation by mapping the original state-space onto a smaller state-space and performing the recursion on this smaller state-space. Methods for improving this lower bound through the use of penalties and through the use of state-space modifiers are described.

Computational results to compare the performance of the Fisher and the dynamic programming state-space relaxation lower bounds are obtained with the total holding-tardiness cost problem for which g_i ($i \in N$) is defined as follows. Job i ($i \in N$) has a due date d_i when

it should ideally be shipped to the customer; if it completed before time d_i a holding cost h_i per unit time is incurred while the job waits for shipment at time d_i, whereas if it is completed after time d_i a tardiness cost of w_i per unit time is incurred as a penalty for late shipment. Thus, for this problem the cost of completing job i at time t is

$$g_i(t) = h_i \max(d_i - t, 0) + w_i \max(t - d_i, 0).$$

When $h_i = 0$ for all jobs i ($i \in N$), this total holding-tardiness cost problem becomes a total weighted tardiness problem.

Section 5.2 gives a lower bound derived by using Fisher's technique. In Section 5.3 the dynamic programming state-space relaxation bound and its modifications are derived. Section 5.4 gives the implementation of the lower bounds. Computational experience with the lower bounds is given in Section 5.5. Section 5.6 describes the branch and bound algorithms. A discussion of our computational experience with the branch and bound algorithms is given in Section 5.7. Section 5.8 gives a conclusion and some remarks.

5.2 Fisher Bound

The procedure used in this section to compute a lower bound is based on Lagrangean relaxation and uses Fisher's formulation of the problem.

Now consider the Lagrangean relaxation of the problem. Let $T=\Sigma_{i\in N} p_i$, and let C_i denote the completion time of job i ($i \in N$). Also, n_{it} ($i \in N$; $t=1,\ldots,T$) is a zero-one variable defined by

$$n_{it} = \begin{cases} 1 & \text{if job } i \text{ is being processed during the interval } [t-1,t]; \\ 0 & \text{otherwise.} \end{cases}$$

Then the problem of minimizing the sum of holding costs and tardiness costs for jobs completed before and after their due date respectively can be written as:

$$\min \{\Sigma_{i\epsilon N}\, g_i(C_i)\} \tag{5.1}$$

subject to

$$C_i \in \{p_i,\ p_i+1,\ \ldots,\ T\} \tag{5.2}$$

$$n_{it} = \begin{cases} 1 & \text{for } t=C_i-p_i+1,\ldots,\ C_i \\ 0 & \text{otherwise,} \end{cases} \tag{5.3}$$

$$\Sigma_{i\epsilon N}\, n_{it}=1,\ t=1,\ldots,\ T. \tag{5.4}$$

A Lagrangean relaxation of the constraints (5.4) (i.e., the machine capacity constraints) is performed. The machine can perform more than one job at each time in the following Lagrangean problem:

$$L(\lambda)=\min \{\Sigma_{i\epsilon N}\, g_i(C_i)\} + \Sigma_{t=1}^{T} \lambda_t\, (\Sigma_{i\epsilon N}\, n_{it} - 1)$$

subject to (5.2) and (5.3).

In the Lagrangean, $\lambda=(\lambda_1, \ldots, \lambda_T)$ is a vector of multipliers corresponding to (5.4). It is well known from the theory of Lagrangean relaxation (Fisher [25]) for any choice of multipliers, $L(\lambda)$ provides a lower bound. If λ is given, then using (5.3) the cost in the Lagrangean problem of scheduling job i in the interval $[C_i - p_i, C_i]$ is

$$g_i(C_i) + \sum_{t=C_i-p_i+1}^{C_i} \lambda_t \qquad (5.5)$$

Thus the Lagrangean problem is easily solved by considering each job independently of the others to give a lower bound $L(\lambda)$. This requires $O(nT)$ time.

We adopt the subgradient optimization method that is used by Fisher [24] as a suitable iterative method to find the value of λ which maximizes $L(\lambda)$, i.e., to find λ^* for which $L(\lambda^*)=\max_\lambda [L(\lambda)]$. Initially, multipliers $\lambda^{(0)}=0$, where $\lambda_t^{(0)}=0$ $(t=1,\ldots,T)$, are used. Thereafter, at the completion of iteration k-1 of the method for which $\lambda^{(k-1)}$ is the vector of multipliers, the value $L(\lambda^{(k-1)})$ is obtained. Let $n_t^{(k-1)}=\sum_{i\in N} n_{it}^{(k-1)}$ be the number of jobs are scheduled in the interval [t-1,t]. The updated multipliers $\lambda_t^{(k)}$ are computed using

$$\lambda_t^{(k)} = \lambda_t^{(k-1)} + \frac{h^{(k-1)}(UB - L(\lambda^{(k-1)}))\ (n_t^{(k-1)} - 1)}{\sum_{j=1}^{T} (n_j^{(k-1)} - 1)^2}$$

where $h^{(k-1)}$ is the step length at iteration k-1 and UB is an upper bound on the total cost that may be obtained by applying a heuristic

method. In our implementation here, we use an initial step length $h^{(0)}=2$ and thereafter $h^{(k)}$ is generated as follows. For $k \geq 1$ we set $h^{(k)}= h^{(k-1)}$. If the lower bound fails to give an overall improvement during five successive iterations, then set $h^{(k)}=h^{(k-1)}/2$.

The following example illustrates Fisher bound $L(\lambda)$ obtained using the subgradient optimization method.

Example 5.1. Consider the total holding-tardiness cost problem with 3 jobs having data as follows.

Job i	p_i	d_i	h_i	w_i
1	1	3	1	2
2	2	4	2	1
3	3	5	8	9

To demonstrate the method, let the vector of multipliers $\lambda=(5,6.5,8.5,10,14,0)$ be chosen. Then the cost of scheduling job 1, given by using the above Lagrangean problem, for each possible completion time C_1 is given by (5.5) is

C_1	1	2	3	4	5	6
cost	2+5	1+6.5	0+8.5	2+10	4+14	6+0

The first component of the cost is the holding cost or tardiness cost and the second component is $\sum_{t=C_1-p_1+1}^{C_1} \lambda_t$. The minimum cost of 6 occurs when $C_1=6$. Similarly, for job 2 we obtain the costs.

C_2	2	3	4	5	6
cost	4+11.5	2+15	0+18.5	1+24	2+14

The minimum cost of 15.5 occurs when C_2=2. Finally, for job 3 we obtain the costs

C_3	3	4	5	6
cost	16+20	8+25	0+32.5	9+24

The minimum cost of 32.5 occurs when C_3=5. Since the number of jobs scheduled in the interval [t-1,t], (t=1,...,6) is one, the sequence (2,3,1) is an optimal sequence with a cost of 10.

5.3 Dynamic Programming State-Space Relaxation

In this section we give a dynamic programming formulation of our single machine problem and the state-space relaxation method is then used to obtain lower bounds that can be embedded in branch and bound algorithms. Techniques for improving this lower bound through the use of penalties and through the use of state-space modifiers are discussed. An example is used to illustrate the above techniques.

5.3.1 Dynamic programming formulation

The dynamic programming formulation of Held and Karp [38] is given next. Let $S \subseteq N$ be an arbitrary subset of jobs. Also, define $f^*(S)$ as the minimum total cost when the jobs of S are sequenced in the first $|S|$ positions in the sequence. The objective is to find $f^*(N)$ by solving the recursion equations

$$f^*(S) = \min_{i \in S}\{f^*(S-\{i\}) + g_i(\Sigma_{j \in S} p_j)\} \qquad (5.6)$$

that are initialized by setting $f^*(\emptyset) = 0$. Solving recursion equations (5.6) is equivalent to finding the shortest path in a state-space graph G^* in which vertices correspond to subsets S and in which arcs correspond to a decision whereby the transition to a new state from a previous state is achieved by the scheduling of a job. The length of the arc directed from vertex S - {i} to vertex S ($i \in S$) is $g_i(\Sigma_{j \in S} p_j)$.

Clearly, there are 2^n vertices in the graph G^*. Since the number of vertices rapidly increases as n increases, the space required for the storage of values $f^*(S)$ exceeds available core storage unless n is small. Instead of applying recursion (5.6) directly to solve the problem, we propose to derive from (5.6) a lower bounding scheme that can be used in a branch and bound algorithm. These lower bounds are obtained by performing the recursion on a suitably relaxed state-space containing fewer states than in the original formulation.

5.3.2 Derivation of the lower bound

Suppose that the original state-space is relaxed by mapping states representing subsets of jobs onto states representing the total processing time of jobs in the subset, i.e., a state S is mapped onto a state $\Sigma_{i \in S} p_i$. (We assume that $T = \Sigma_{i \in N} p_i < 2^n$ to ensure that there are fewer states in the relaxed problem than in the original problem: if $T \geq 2^n$ it is more efficient to solve the original problem.) The relaxed problem is solved by computing $f_0(T)$ from the recursion equations

$$f_0(t) = \min_{i \in N}\{f_0(t - p_i) + g_i(t)\} \qquad (5.7)$$

that are initialized by setting $f_0(t) = \infty$ for $t < 0$ and $f_0(0) = 0$. The state-space graph G_0 for recursion (5.7) has vertices 0, 1, ...,T and, for each job i ($i \in N$), has an arc directed from vertex $t - p_i$ to vertex t for $t = p_i,\ldots,T$ of length $g_i(t)$ that corresponds to the scheduling of job i in the interval $[t-p_i,t]$. We may regard $f_0(t)$ as the minimum cost of scheduling jobs in the time interval [0,t]. The computation of $f_0(T)$ requires O(nT) time.

Note that our mapping is particularly convenient since in both the original and relaxed formulations the cost of scheduling job i is easily deduced from the state variable: for the original formulation the cost of scheduling job i after all other jobs of S is $g_i(\Sigma_{j \in S} p_j)$ and in the relaxed formulation the cost of scheduling job i to be completed at time t is $g_i(t)$. A mapping that does not associate total processing times with their corresponding sets does not appear to yield a recursion that can be used for computing lower bounds.

We show next that the solution of the relaxed problem provides a lower bound on the solution of the original problem.

Theorem (5.1). If $f^*(N)$ is obtained from (5.6) and if $f_0(T)$ is obtained from (5.7), then $f_0(T) \leq f^*(N)$.

Proof. Let $\sigma = (\sigma(1),\ldots,\sigma(n))$ be an optimal sequence. Thus, a shortest path in the state-space graph G^* successively passes through vertices $S_0,S_1,\ldots,S_n$, where $S_i = \{\sigma(1),\ldots,\sigma(i)\}$. The length of an arc from S_{i-1} to S_i is $g_{\sigma(i)}(\Sigma_{j \in S_i} p_j)$. In the relaxed state-space graph G_0, there exists a path that successively passes through

vertices 0, $p_{\sigma(1)}$, ..., $p_{\sigma(1)} + \ldots + p_{\sigma(n)}$, i.e., vertices $\Sigma_{j\in S_0} p_j$, $\Sigma_{j\in S_1} p_j$, ..., $\Sigma_{j\in S_n} p_j$. The length of an arc from vertex $p_{\sigma(1)} + \ldots + p_{\sigma(i-1)}$ to vertex $p_{\sigma(1)} + \ldots + p_{\sigma(i)}$ in G_0 is $g_{\sigma(i)}(p_{\sigma(1)} + \ldots + p_{\sigma(i)}) = g_{\sigma(i)}(\Sigma_{j\in S_i} p_j)$.
Therefore, there is a path in G_0 with the same length as the shortest path in G^*. Consequently, the shortest path in the relaxed graph G_0 is no longer than the shortest path in the original graph G^*. □

In recursion (5.6), the minimization is over all jobs i of S. However, in (5.7) the state variables provide no information as to which jobs are scheduled to give the value $f_0(t)$ and, consequently, the minimization is over all jobs i of N. In contrast to the state-space graph G^* therefore, some paths in the relaxed graph G_0 do not correspond to schedules in which each job is sequenced once. The decisions that generate the value $f_0(T)$ from recursion (5.7) form a 'sequence' $(\sigma(1),\ldots,\sigma(s))$ in which some jobs may appear more than once while others do not appear. For this reason the bound $f_0(T)$ is rather weak.

Following an example to illustrate the computation of $f^*(N)$ and $f_0(T)$, the next sections concentrate on methods to improve our lower bound by attempting to force every job to appear exactly once in the 'sequence' generated by the shortest path in the state-space graph for the relaxed problem.

Example 5.2. Consider again the total holding-tardiness cost problem with 3 jobs having data as follows.

Job i	p_i	d_i	h_i	w_i
1	1	3	1	2
2	2	4	2	1
3	3	5	8	9

The state-space graph G^* is shown in Figure 5.1. The vertices of the

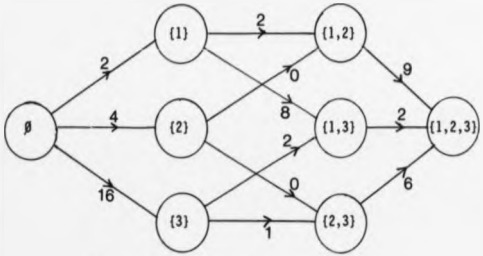

Fig. 5.1. The state-space graph G^*.

graph are the subsets of {1,2,3} and the costs of each decision are written against the arcs of the graph. The solution of equations (5.6) are $f^*(\{1\}) = 2$, $f^*(\{2\}) = 4$, $f^*(\{3\}) = 16$, $f^*(\{1,2\}) = 4$, $f^*(\{1,3\}) = 10$, $f^*(\{2,3\}) = 4$ and $f^*(\{1,2,3\}) = 10$. Thus, the shortest path in G^* is of length 10. Backtracing shows that this shortest path passes successively through the vertices ø, {2}, {2,3} and {1,2,3} and generates the optimal sequence (2,3,1).

The state-space graph G_0 is shown in Figure 5.2. The vertices

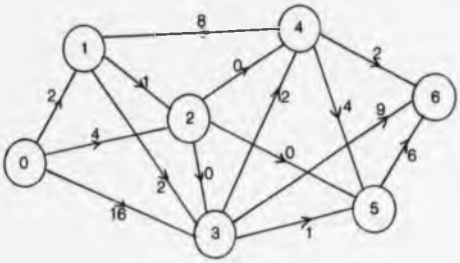

Fig. 5.2. The state-space graph G_0.

of the graph are the times 0, 1, 2, 3, 4, 5 and 6 at which jobs may be completed. This graph has an arc from vertex $t - p_i$ to vertex t ($t = 1,\ldots,6$) for each job i with $p_i \leq t$ that corresponds to scheduling job i in the interval $[t-p_i,t]$. Recursion (5.7) yields $f_0(1) = 2$, $f_0(2) = 3$, $f_0(3) = 3$, $f_0(4) = 3$, $f_0(5) = 3$ and $f_0(6) = 5$. Thus, the shortest path in the relaxed graph G_0 is of length 5. Backtracing shows that this shortest path passes successively through the vertices 0, 1, 2, 4 and 6 and generates the 'sequence' (1,1,2,2) in which jobs 1 and 2 appear twice and job 3 does not appear.

5.3.3 Constraints on successive jobs

In this section the first method of improving the lower bound is presented. We aim to modify recursion (5.7) so that no path exists in the corresponding state-space graph that generates a 'sequence' $(\sigma(1),\ldots,\sigma(s))$ with $\sigma(i-1) = \sigma(i)$ for any i ($i = 2,\ldots,s$). Thus,

'sequences' in which the same job appears in adjacent positions are avoided.

Let us define $f_1(t,j)$ ($j \in N$; $t = p_j,\ldots,T$) as the minimum cost of scheduling jobs in the interval $[0,t]$ where job j is scheduled in the interval $[t-p_j,t]$ and where the same job is not scheduled in adjacent intervals. For this formulation the problem is to find $\min_{j\in N}\{f_1(T,j)\}$ from the recursion equations

$$f_1(t,j) = \min_{i\in N-(j)}\{f_1(t - p_j,i) + g_j(t)\} \qquad (5.8)$$

that are initialized by setting, for each j ($j \in N$), $f_1(t,j) = -\infty$ for $t < 0$ and $f_1(0,j) = 0$. The following observation shows that recursion (5.8) can be performed without storing each of the values $f_1(t,j)$. Let us define $e(t,*)$ to be the job for which $f_1(t,e(t,*)) = \min_{j\in N}\{f_1(t,j)\}$. Also, let $f_1(t,*) = f_1(t,e(t,*))$ and $f_1(t,**) = \min_{j\in N-\{e(t,*)\}}\{f_1(t,j)\}$. We may regard $f_1(t,*)$ as the smallest and $f_1(t,**)$ as the second smallest of the values $f_1(t,j)$ ($j \in N$); $e(t,*)$ is the job for which $f_1(t,e(t,*)) = f_1(t,*)$. From the previously computed values $f_1(t - p_j,*)$, $e(t - p_j,*)$ and $f_1(t - p_j,**)$, we can find $f_1(t,j)$ ($j \in N$) using

$$f_1(t,j) = \begin{cases} f_1(t - p_j,*) + g_j(t) & \text{if } j \neq e(t - p_j,*) \\ f_1(t - p_j,**) + g_j(t) & \text{if } j = e(t - p_j,*) \end{cases}$$

from which $f_1(t,*)$, $e(t,*)$ and $f_1(t,**)$ are found. Clearly, the computation of $f_1(T,*)$ requires $O(nT)$ time. Implemented in this way, the computation and storage required to find $f_1(T,*)$ are little more

than that required to find $f_0(T)$ from recursion (5.7).

Let G_1 be the state-space graph for recursion (5.8). We establish next that $f_1(T,*)$ is a valid lower bound. This is followed by an example to demonstrate its computation.

Theorem (5.2). If $f^*(N)$ is obtained from (5.6) and if $f_1(T,*)$ is obtained from (5.8), then $f_1(T,*) \leq f^*(N)$.

Proof. Let $\sigma = (\sigma(1),\ldots,\sigma(n))$ be an optimal sequence which provides a shortest path in G^* that successively passes through vertices $S_0,S_1,\ldots,S_n$, where $S_i = \{\sigma(1),\ldots,\sigma(i)\}$. In G_1, a path exists that passes successively through vertices $(0,0)$, $(p_{\sigma(1)},\sigma(1))$, ..., $(p_{\sigma(1)+\ldots+p_{\sigma(n)}},\sigma(n))$. Using the arguments presented in the proof of Theorem (5.1), both of these paths have the same length which implies the required result. □

Example 5.3. Applying recursion (5.8) to the example of the previous section yields the following values.

t	$f_1(t,1)$	$f_1(t,2)$	$f_1(t,3)$	$f_1(t,*)$	$e(t,*)$	$f_1(t,**)$
1	2	∞	∞	2	1	∞
2	∞	4	∞	4	2	∞
3	4	4	16	4	1	4
4	6	∞	10	6	1	10
5	14	5	4	4	3	5
6	10	8	13	8	2	10

Thus, a lower bound $f_1(6,*) = 8$ is obtained. Backtracing shows that the corresponding 'sequence' is (1,2,1,2).

It is also possible to derive a recursion with the property that each path in the corresponding state-space graph generates a 'sequence' $(\sigma(1),\ldots,\sigma(s))$ for which $\sigma(i-2)$, $\sigma(i-1)$ and $\sigma(i)$ $(i = 3,\ldots,s)$ are three distinct jobs. However, it is uncertain whether the improvement to the lower bound compensates for the extra computation required to incorporate these extra constraints.

5.3.4 The use of job penalties to improve the lower bound

We define a penalty λ_i as an additional cost that is incurred when job i ($i \in N$) is scheduled. Thus, the total cost of scheduling job i to be completed at time t is $g_i(t) + \lambda_i$. The introduction of penalties yields an equivalent problem since the cost of every schedule is increased by $\Sigma_{i \in N} \lambda_i$, i.e., the length of every path in the state-space graph G^* is increased by $\Sigma_{i \in N} \lambda_i$. However, for a relaxed state-space graph, different paths are increased in length by different amounts when penalties are introduced and, consequently, the shortest path may change. Ideally, penalties would be chosen to force a shortest path in the relaxed state-space graph to define a feasible sequence. These penalties are analogous to the multipliers used in Lagrangean relaxation for integer programming. In this latter case, the aim is to find suitable values of the multipliers that generate a solution in which the relaxed constraints are satisfied.

Precise details of how penalties are used are given now. If $\lambda = (\lambda_1,\ldots,\lambda_n)$ is a given vector of penalties, then the original problem

is solved by computing $f^*(N;\lambda) - \Sigma_{i\in N}\lambda_i$ from the recursion equations

$$f^*(S;\lambda) = \min_{i\in S}\{f^*(S-\{i\};\lambda) + g_i(\Sigma_{j\in S} p_j) + \lambda_i\} \qquad (5.9)$$

that are initialized by setting $f^*(\emptyset;\lambda) = 0$. As is the case above, the original state-space is relaxed by mapping states S onto their total processing times to give a relaxed problem that is solved by computing $f_2(T;\lambda) - \Sigma_{i\in N}\lambda_i$ from the recursion equations

$$f_2(t;\lambda) = \min_{i\in N}\{f_2(t - p_i;\lambda) + g_i(t) + \lambda_i\} \qquad (5.10)$$

that are initialized by setting $f_2(t;\lambda)=\infty$ for $t < 0$ and $f_2(0;\lambda)=0$.

Let $G^*(\lambda)$ and $G_2(\lambda)$ be the state-space graphs for recursions (5.9) and (5.10). A proof that $f_2(T;\lambda) - \Sigma_{i\in N}\lambda_i$ is a valid lower bound is given next. An example demonstrating the use of penalties follows.

Theorem (5.3). If $f^*(N)$ is obtained from (5.6), if $f^*(N;\lambda)$ is obtained from (5.9) and if $f_2(T;\lambda)$ is obtained from (5.10), then

$$f_2(T;\lambda) - \Sigma_{i\in N}\lambda_i \leq f^*(N;\lambda) - \Sigma_{i\in N}\lambda_i = f^*(N).$$

Proof. We demonstrate first that $f^*(N;\lambda) - \Sigma_{i\in N}\lambda_i = f^*(N)$. Apart from the different arc lengths, the state-space graphs G^* and $G^*(\lambda)$ for recursions (5.6) and (5.9) are identical. Let $\sigma = (\sigma(1),\ldots,\sigma(n))$ be a sequence that defines the path P_σ that passes successively through vertices $S_0,S_1,\ldots,S_n$, where $S_i = \{\sigma(1),\ldots,\sigma(i)\}$, in G^* and $G^*(\lambda)$. In G^* it has length $L_\sigma = \Sigma_{i\in N}$

$g_{\sigma(i)}(\Sigma_{j\in S_i} p_j)$ and in $G^*(\lambda)$ it has length $\Sigma_{i\in N}(g_{\sigma(i)}(\Sigma_{j\in S_i} p_j) + \lambda_{\sigma(i)}) = L_\sigma + \Sigma_{i\in N} \lambda_i$. These path lengths differ by the constant $\Sigma_{i\in N} \lambda_i$, so P_σ is a shortest path in G^* if and only if P_σ is a shortest path in $G^*(\lambda)$. This proves that $f^*(N;\lambda)-\Sigma_{i\in N} \lambda_i = f^*(N)$.

Now suppose that σ is an optimal sequence so that P_σ is a shortest path in $G^*(\lambda)$ having length $L_\sigma + \Sigma_{i\in N} \lambda_i$. Applying the arguments used in the proof of Theorem (5.1), we deduce there is a path of length $L_\sigma + \Sigma_{i\in N} \lambda_i$ in $G_2(\lambda)$. This shows that $f_2(T;\lambda)-\Sigma_{i\in N} \lambda_i \leq f^*(N;\lambda) - \Sigma_{i\in N} \lambda_i$. □

Example 5.4. Consider again the example presented previously. Suppose that the vector of penalties $\lambda = (2, 2, -4)$ is chosen. The state-space graph $G_2(\lambda)$ is identical to the graph of Figure 5.2 except that the arc from vertex $t - p_i$ to vertex t ($t = 1,\ldots,6$), for each job i with $p_i \leq t$, is longer by λ_i. Recursion (5.10) yields $f_2(1;\lambda) = 4$, $f_2(2;\lambda) = 6$, $f_2(3;\lambda) = 8$, $f_2(4;\lambda) = 8$, $f_2(5;\lambda) = 2$ and $f_2(6;\lambda) = 10$. Thus, the lower bound of $f_2(6;\lambda) - \Sigma_{i=1}^{3} \lambda_i = 10$ is obtained. Backtracing shows that the shortest path passes successively through the vertices 0, 2, 5 and 6 to give a feasible sequence (2,3,1). When a feasible sequence is obtained, it is necessarily optimal.

The example above shows that it is possible to obtain a tight lower bound from recursion (5.10) provided that the penalties are chosen appropriately. We propose the subgradient optimisation method as a suitable iterative method to find λ^* for which $f_2(T;\lambda^*) = \max_\lambda\{f_2(T;\lambda)\}$. Initially, penalties $\lambda^{(0)}$, where $\lambda_i^{(0)} = 0$ ($i \in N$), are used. Thereafter, at the completion of iteration k-1 of the

method for which $\lambda^{(k-1)}$ is the vector of penalties, the value $f_2(T;\lambda^{(k-1)})$ is obtained and the corresponding 'sequence' is found by backtracing. Let $n_i^{(k-1)}$ be the number of times that job i ($i \in N$) occurs in this 'sequence'. The updated penalties $\lambda_i^{(k)}$ ($i \in N$) are computed using

$$\lambda_i^{(k)} = \lambda_i^{(k-1)} + \frac{h^{(k-1)}(UB - (f_2(T;\lambda^{(k-1)}) - \Sigma_{j\in N}\lambda_j^{(k-1)}))(n_i^{(k-1)} - 1)}{\Sigma_{j\in N}(n_j^{(k-1)} - 1)^2}$$

where $h^{(k-1)}$ is the step length at iteration k - 1 and UB is an upper bound on the total cost that may be obtained by applying a heuristic method. In our implementation, we use an initial step length $h^{(0)} = 2$ and thereafter halve the step length when the lower bound fails to give an overall improvement during five successive iterations. Also, we use the greedy heuristic to obtain UB. This heuristic successively schedules a job with the smallest cost in the first unfilled position in the sequence until all jobs are scheduled.

5.3.5 The use of state-space modifiers to improve the lower bound

In this section, we derive an alternative relaxed problem to that given by recursion (5.7) by employing a different mapping of the states S of recursion (5.6). We define the non-negative integer q_i as the state-space modifier for job i ($i \in N$). Also, let $Q=\Sigma_{i\in N} q_i$. A relaxed problem is now obtained from recursion (5.6) by mapping the original states S onto the states $(\Sigma_{i\in S} p_i, \Sigma_{i\in S} q_i)$. This relaxed problem is solved by computing $f_3(T,Q)$ from the recursion equations

$$f_3(t,q) = \min_{i\in N}\{f_3(t - p_i, q - q_i) + g_i(t)\} \qquad (5.11)$$

that are initialized by $f_3(t,q) = \infty$ for $t < 0$ or $q < 0$, and $f_3(0,0) = 0$. The computation of $f_3(T,Q)$ requires $O(nT(1 + Q))$ time.

The modifiers q_i ($i \in N$) play a similar role to the penalties λ_i ($i \in N$) of the previous section. Both are used with the aim of forcing the shortest path in the relaxed state-space graph to define a feasible sequence. For example, if for some j ($j \in N$) we have $q_j = 1$ and $q_i = 0$ for $i \in N$ where $i \neq j$, then all paths in the state-space graph for recursion (5.11) define 'sequences' in which job j appears exactly once. On the other hand, if $q_i = 1$ for all i ($i \in N$), each path in the state-space graph for recursion (5.11) defines a 'sequence' containing exactly n jobs.

Let G_3 be the state-space graph for recursion (5.11). We establish next that $f_3(T,Q)$ is a valid lower bound. An example demonstrating the use of modifiers is then given.

Theorem 5.4. If $f^*(N)$ is obtained from (5.6) and if $f_3(T,Q)$ is obtained from (5.11), then $f_3(T,Q) \leq f^*(N)$.

Proof. Let $\sigma = (\sigma(1),\ldots,\sigma(n))$ be an optimal sequence which provides a shortest path in G^* that successively passes through vertices $S_0, S_1, \ldots, S_n$, where $S_i = \{\sigma(1),\ldots,\sigma(i)\}$. In G_3 a path exists that passes successively through vertices $(0,0)$, $(p_{\sigma(1)}, q_{\sigma(1)})$, $\ldots$, $(p_{\sigma(1)}+\ldots+p_{\sigma(n)}, q_{\sigma(1)}+\ldots+q_{\sigma(n)})$. Using the arguments presented in the proof of Theorem (5.1), both of these paths have the same length which implies the required result. □

Example 5.5. Consider again the example presented previously. Suppose that the modifiers $q_1 = 0$, $q_2 = 1$ and $q_3 = 0$ are chosen.

Applying recursion (5.11) yields the following values.

t	$f_3(t,0)$	$f_3(t,1)$
0	0	-
1	2	-
2	3	4
3	3	4
4	5	3
5	3	4
6	9	7

Thus, a lower bound f(6,1) = 7 is obtained. Backtracing shows that the corresponding 'sequence' is (1,1,1,1,2).

Since there is no obvious alternative, we propose an iterative method to find values of the modifiers. Initially, we set $q_i^{(0)} = 0$ for all i ($i \in N$). Thereafter, at the completion of iteration k - 1 for which $q_i^{(k-1)}$ ($i \in N$) are the modifiers and $Q^{(k-1)} = \Sigma_{i \in N} q_i^{(k-1)}$, the lower bound $f_3(T,Q^{(k-1)})$ is obtained and the corresponding 'sequence' is found by backtracing. Let $n_i^{(k-1)}$ be the number of times that job i ($i \in N$) occurs in this 'sequence'. If $n_i^{(k-1)} = 1$ for all i ($i \in N$), then every job appears exactly once in the 'sequence' which is therefore feasible and hence optimal. Otherwise, our first approach is to find some job j ($j \in N$) for which $n_j^{(k-1)} > 1$ and increase by one the modifier for job j in an attempt to obtain $n_j^{(k)} = 1$ in the next iteration. More precisely, a job j satisfying

$$(n_j^{(k-1)} - 1)(q_j^{(k-1)} + 2) = \max_{i \in N}\{(n_i^{(k-1)} - 1)(q_i^{(k-1)} + 2)\} \quad (5.12)$$

is found and the modifiers are updated using

$$q_i^{(k)} = \begin{cases} q_i^{(k-1)} & \text{for } i \in N \text{ and } i \neq j; \\ q_i^{(k-1)} + 1 & \text{for } i = j. \end{cases} \quad (5.13)$$

One advantage of using (5.12) and (5.13) to update modifiers is that it yields $Q^{(k)} = k$. Since the computation required at iteration k is $O(nT(1 + Q^{(k)}))$, these relatively small $Q^{(k)}$ values are desirable. An alternative formula to (5.12) could be used to find job j. However, initial experiments with (5.12) produced satisfactory results. In our second approach, as an alternative to (5.12) and (5.13), modifiers are updated using

$$q_i^{(k)} = \max\{q_i^{(k-1)} + h^{(k-1)}(n_i^{(k-1)} - 1), 0\} \text{ for } i \in N, \quad (5.14)$$

where $h^{(k-1)}$ is the integer step length at iteration k - 1. In our implementation, we use the step lengths $h^{(k-1)} = 1$ for k = 1,...,10,16,... and $h^{(k-1)} = 2$ for k = 11,...,15, which produced good results in initial experiments. This second method of updating modifiers resembles the subgradient optimization technique that is used to update penalties. It has the disadvantage that $Q^{(k)}$ tends to be larger than in the first method although, possibly, the modifiers more quickly approach their optimal values than if (5.12) and (5.13) are used.

5.4. Implementation of the Lower Bounds

Obviously it is possible to apply all three of the lower bound improvement methods described above, i.e., to avoid the same job appearing in adjacent positions, to use penalties λ_i ($i \in N$) and to use state-space modifiers q_i ($i \in N$). Applying these methods, the lower bound is obtained by finding $\min_{j \in N}\{f(T,Q,j;\lambda)\} - \Sigma_{i \in N} \lambda_i$, where $Q = \Sigma_{i \in N} q_i$, from the recursion equations

$$f(t,q,j;\lambda) = \min_{i \in N-\{j\}}\{f(t - p_j, q - q_j, i;\lambda) + g_j(t) + \lambda_j\} \qquad (5.15)$$

that are initialized by setting, for each j ($j \in N$), $f(t,q,j;\lambda) = \infty$ for $t < 0$ or $q < 0$, and $f(0,0,j;\lambda) = 0$. This lower bound is computed in $O(nT(1 + Q))$ time.

Instead of storing $f(t,q,j;\lambda)$ for each j ($j \in N$), only the smallest value $f(t,q,*;\lambda)$, the job $e(t,q,*;\lambda)$ which yields the smallest value and the second smallest value $f(t,q,**;\lambda)$ are stored. It is also convenient to store the job $e(t,q,**;\lambda)$ which yields the second smallest value. If, additionally, the signs of $e(t,q,*;\lambda)$ and $e(t,q,**;\lambda)$ are used to indicate whether $f(t,q,*;\lambda)$ and $f(t,q,**;\lambda)$ respectively are computed, for some i ($i \in N$), from $f(t - p_i, q - q_i, *;\lambda)$ or from $f(t - p_i, q - q_i, **;\lambda)$, then backtracing can be performed using only the values $e(t,q,*;\lambda)$ and $e(t,q,**;\lambda)$ ($t = 1,\ldots,T$; $q = 0,\ldots,Q$). Since backtracing is necessary in the iterative schemes for updating penalties and modifiers to find the number of times each job appears in the current 'sequence', we store the $2T(1 + Q)$ values of $e(t,q,*;\lambda)$ and $e(t,q,**;\lambda)$.

The storage requirements for the function values $f(t,q,*;\lambda)$ and $f(t,q,**;\lambda)$ are discussed now. Let $q_{max} = \max_{i \in N}\{q_i\}$. For any $q >$

q_{max}, the values $f(t,q,j;\lambda)$ for $t = 0,\ldots,T$ and $j \in N$ are computed from the values $f(t',q',*;\lambda)$ and $f(t',q',**;\lambda)$ for $t' = 0,\ldots,T$ and $q' = q\text{-}q_{max},\ldots,q$. Thus, at any stage it is necessary to store at most $2(1 + T)(1 + q_{max})$ function values to compute the lower bound.

To summarize, state-space modifiers should be chosen so that Q and q_{max} are small enough to allow the $2T(1 + Q)$ values of $e(t,q,*;\lambda)$ and $e(t,q,**;\lambda)$ $(t = 1,\ldots,T;\ q = 0,\ldots,Q)$ and $2(1 + T)(1 + q_{max})$ function values to be stored.

To find suitable values of the penalties and modifiers, we can either first apply the iterative method for finding the penalties using $q_i = 0$ $(i \in N)$ and then perform iterations to update the modifiers or perform the modifier iterations before the penalty iterations. If the former approach is adopted, the computational requirement is O(nT) time per penalty iteration. If the latter approach is adopted in which the modifier iterations are performed first, a computational requirement of O(nT(1 + Q)) time per penalty iteration results. Since, in either case, the computational requirements for the modifier iterations are comparable, the method of performing the penalty iterations first is computationally much faster and therefore we adopt this approach.

Let $B(\lambda,q) = f(T,Q,*;\lambda) - \Sigma_{i \in N} \lambda_i$ denote the lower bound obtained from recursion (5.15) when first the penalties and then the modifiers are obtained using an iterative method. When the modifiers are updated from (5.12) and (5.13), the corresponding bound is denoted by $B(\lambda,q^1)$, whereas when the modifiers are updated from (5.14) the corresponding bound is denoted by $B(\lambda,q^2)$. When no penalty iterations are performed, i.e., when $\lambda_i = 0$ $(i \in N)$, the bound is denoted by $B(-,q^1)$ or $B(-,q^2)$, when no modifier iterations

are performed, i.e., when $q_i = 0$ ($i \in N$), the bound is denoted by $B(\lambda,-)$ and when no penalty or modifier iterations are performed, the bound is denoted by $B(-,-)$.

5.5 Computational Experience with the Lower Bounds

The lower bounds were tested on 20-job total holding-tardiness cost problems that were generated as follows. For each job i ($i \in N$), an integer processing time p_i, an integer unit holding cost h_i and an integer unit tardiness penalty w_i were generated from the uniform distribution [1,10]. Problem 'hardness' is likely to depend on parameters RDD and LF called the relative range of due dates and the average lateness factor. Having computed T, selected a value of RDD from the set {0.2, 0.4, 0.6, 0.8, 1.0} and selected a value of LF from the set {0.2, 0.4}, an integer due date d_i from the uniform distribution [T(1 - LF - RDD/2), T(1 - LF + RDD/2)] was generated for each job i ($i \in N$). Note that by the symmetry of the total holding-tardiness cost problem, problems with average lateness factors LF and 1 - LF are likely to be of similar difficulty. Hence problems with LF = 0.6 and LF = 0.8 were not generated. One problem was generated for each of the 10 pairs of values of RDD and LF.

Fisher's lower bound and the various lower bounds based on the dynamic programming state-space relaxation method were computed for each of the 10 problems. In all cases, a sufficiently large number of multiplier and penalty iterations was performed to obtain a lower bound that is close to the best possible for that particular method, although the number of modifier iterations was restricted because each such iteration is computationally expensive. More precisely, for Fisher's bound and for each bound that uses penalties, 100

iterations were performed to find suitable values of the multipliers and penalties. For each bound that uses modifiers, 10 iterations were performed to find suitable values for them.

Results comparing the various lower bounds are given in Table 5.1. The first column numbers the problems while the second column gives the corresponding parameters RDD and LF. The value of an optimal solution, found using the branch and bound algorithm to be described later, is given in the third column. The remaining columns give values of the various lower bounds. Cases for which a problem is solved by the lower bounding procedure, i.e., when backtracing generates a feasible sequence, are marked.

Table 5.2 is designed to show the increase of the Fisher, $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ lower bounds as the number of subgradient optimization iterations for multipliers and penalties increases and also to relate the computation time required to compute a bound to its value relative to the optimum. After 0, 20, 50 and 100 multiplier or penalty iterations and, where appropriate, after a subsequent 10 modifier iterations the average relative value, i.e., the average percentage of the optimum achieved, and the average computation time required in seconds using the FORTRAN V compiler on a CDC 7600 are listed for each bound. We note that for zero penalty iterations the bounds $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ reduce to $B(-,-)$, $B(-,q^1)$ and $B(-,q^2)$ respectively.

We first observe that the lower bound $B(-,-)$ only achieves on average 46.43% of the optimum. Even though it is quickly computed, such a weak lower bound is clearly unable to effectively restrict the search in a branch and bound algorithm. Next we discuss the relative merits of the bound $B(\lambda,-)$. Since in all cases after 50 penalty

Table 5.1. Comparison of values of lower bounds

Number	RDO,LF	Optimum	Fisher	$B(-,-)$	$B(\lambda,-)$	$B(-,q^1)$	$B(-,q^2)$	$B(\lambda,q^1)$	$B(\lambda,q^2)$
1	0.2,0.2	2406	2377	1143	2402	2009	2146	2402	2402
2	0.2,0.4	1441	1403	536	1419	1011	1193	1424	1419
3	0.4,0.2	1218	1213	468	1218*	988	1159	1218*	1218*
4	0.4,0.4	406	372	288	391	329	350	393	394
5	0.6,0.2	1054	1034	303	1046	583	692	1049	1046
6	0.6,0.4	905	899	605	905*	796	832	905*	905*
7	0.8,0.2	1976	1966	575	1976*	1079	1181	1976*	1976*
8	0.8,0.4	546	521	338	534	437	471	536	536
9	1.0,0.2	2143	2115	417	2133	859	1094	2140	2137
10	1.0,0.4	349	343	224	349*	312	322	349*	349*

*: indicates that the problem is solved by the lower bounding procedure.

iterations it gives a bound that is at least as large as $B(-,q^1)$ and $B(-,q^2)$ and since it is much faster to compute than these bounds that use modifiers, we conclude that penalties are more effective than modifiers in improving the lower bound. It is also apparent that for a given number of multiplier of penalty iterations $B(\lambda,-)$ provides a tighter lower bound than the method of Fisher. Table 5.2 shows that $B(\lambda,-)$ increases with the number of iterations much faster than Fisher's bound increases which is a major advantage when used in a branch and bound algorithm where the number of iterations performed at each node is limited. The advantage of $B(\lambda,-)$ is further demonstrated by noting that after 20 iterations it gives, on average, a superior bound to that obtained by Fisher's method after 50 iterations and it requires much less computation time. Similar comments apply to the performance of $B(\lambda,-)$ after 50 iterations compared to Fisher's bound after 100 iterations. These results indicate that $B(\lambda,-)$ is very likely to yield a superior branch and bound algorithm to one that employs Fisher's bound.

Comparing $B(\lambda,-)$ after 100 iterations with the value of the optimum shows that it is a strong lower bound and is exact for 4 of the problems. However, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ provide, at some computational expense, a slight improvement over $B(\lambda,-)$ for 5 and 3 of the other problems respectively. There are surprising figures in Table 5.2 for $B(\lambda,q^1)$ and $B(\lambda,q^2)$ which show that average computation times may decrease when the number of penalty iterations increases. This anomaly is explained by the observation that an increase in the number of penalty iterations may either enable an optimal solution to be generated in which case no time consuming modifier iterations are performed, or the 'sequence' corresponding to the bound $B(\lambda,-)$ is

Table 5.2. Relative performance and computation time of lower bounds

		Fisher	$B(\lambda,-)$	$B(\lambda,q^1)$	$B(\lambda,q^2)$
0 iterations	ARV	2.54	46.43	72.32	80.03
	ACT	0.02	0.03	1.46	5.51
20 iterations	ARV	77.48	94.07	95.06	96.12
	ACT	0.24	0.26	1.23	4.21
50 iterations	ARV	92.78	98.10	98.77	98.93
	ACT	0.60	0.63	1.52	3.79
100 iterations	ARV	97.67	99.12	99.30	99.25
	ACT	1.19	1.14	1.70	3.48

ARV: average relative value of bound, i.e., the average percentage of the optimum achieved by the bound.

ACT: average computation time in seconds to compute bound.

closer to a feasible solution in which case for $B(\lambda,q^2)$ the modifiers are smaller and consequently modifier iterations require less computation time.

Bearing in mind the inapplicability of the dominance rules of Rinnooy Kan et al., unless a strong lower bound is used in a branch and bound algorithm, large search trees will be generated. Since $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ appear to be the strongest lower bounds, their performance in a branch and bound algorithm is investigated.

5.6 Branch and Bound Algorithm

We now give the main features of our branch and bound algorithms. Prior to their application, the greedy heuristic is used to generate an upper bound on the cost of an optimal schedule. Also, at the root node of the search tree an initial lower bound on the cost of an optimal schedule is obtained from $B(\lambda,q^1)$ by first performing 100 penalty iterations and then performing 30 modifier iterations.

The algorithms use a forward sequencing branching rule for which nodes at level k of the search tree correspond to initial partial sequences in which jobs are sequenced in the first k positions. An adjacent job interchange rule is applied at each node of the search tree, except those at the first level in which only one job is sequenced, in an attempt to eliminate nodes through the dominance theorem of dynamic programming. At the current node, the adjacent job interchange rule compares the cost of the last two jobs of the initial partial sequence with the corresponding cost when the two jobs are interchanged: if the former cost is larger, then the current node is eliminated, while if both costs are the same, some convention

is used to decide whether the current node should be discarded. Potts and Van Wassenhove [80] obtain excellent computational results for the total weighted tardiness problem with this rule.

For nodes that are not eliminated by the adjacent job interchange rule, a lower bound $B(\lambda,-)$, $B(\lambda,q^1)$ or $B(\lambda,q^2)$ is computed. When $B(\lambda,-)$ is used the multipliers are updated from their values at the parent node using 10 penalty iterations. When $B(\lambda,q^1)$ or $B(\lambda,q^2)$ is used the multipliers are first updated from their values at the parent node using 5 penalty iterations (with modifiers set to zero) and then 2 modifier iterations are performed. A newest active node search is then used to select a node from which to branch.

Since all other features are identical, the three branch and bound algorithms are represented by the lower bounds $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ that they use.

5.7 Computational Experience with the Branch and Bound Algorithms

The branch and bound algorithms were tested on total holding-tardiness cost problems with 10, 15, 20 and 25 jobs. The 20-job problems are those used in Tables 5.1 and 5.2 to compare the lower bounds and the problems with 10, 15 and 25 jobs were generated similarly. The algorithms $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ were coded in FORTRAN V and run on a CDC 7600 computer. Whenever a problem could not be solved within the time limit of 100 seconds, computation was abandoned for that problem. Average computation times in seconds and average numbers of nodes (or lower bounds on the averages when there are unsolved problems) are given in Table 5.3.

There is no strong indication from the results of Table 5.3 that

Table 5.3 Comparison of branch and bound algorithms

	Algorithm $B(\lambda,-)$		Algorithm $B(\lambda,q^1)$		Algorithm $B(\lambda,q^2)$	
n	ACT	ANN	ACT	ANN	ACT	ANN
10	0.11	0	0.11	0	0.11	0
15	3.88	26	3.84	22	4.11	16
20	15.16	86	14.62	97	15.40	60
25	51.25*	187*	49.64**	204**	53.57**	126**

ACT: average computation time in seconds.

ANN: average numbers of nodes in the search tree.

* : a lower bound on the average because of one unsolved problem.

** : a lower bound on the average because of two unsolved problems.

one algorithm is superior to the others. Algorithm $B(\lambda,-)$ has the advantage that there is only one unsolved 25-job problem compared with two unsolved problems for the other algorithms. On the other hand, average computation times are slightly smaller for algorithm $B(\lambda,q^1)$. Comparing average numbers of nodes for algorithm $B(\lambda,q^2)$ with those for the other algorithms, it appears that the lower bounding scheme $B(\lambda,q^2)$ provides the tightest lower bounds. However, average computation times show that it is also computationally the most expensive scheme. Although on the evidence of the results of Table 5.3 we have a slight preference for algorithm $B(\lambda,-)$, varying the number of penalty and modifier iterations performed at each node of the search tree could lead us to a different conclusion.

Each algorithm appears capable of solving problems with up to 25 jobs satisfactorily. The application of the initial lower bound $B(\lambda,q^1)$ at the top of the search tree solves 10, 6, 4 and 2 of the 10 problems with 10, 15, 20 and 25 jobs respectively without requiring branching. This initial lower bound is within 1% of the optimum for all but 7 of the 40 problems and has a maximum deviation from the optimum of 3.2%. In spite of the tightness of the initial lower bound, experiments with 30-job problems showed that the majority could not be solved within the 100 second time limit. Further experimentation with numbers of penalty and modifier iterations to be performed at each node of the search tree and the use of a more effective heuristic is likely to yield slightly more efficient algorithms. Nevertheless, it would be unlikely for such an improved algorithm to be capable of solving problems with more than 30 jobs without requiring excessive computation time.

The influence of the parameters RDD and LF on problem hardness

deserves mention. There is some evidence problems with smaller due date range RDD tend to be harder than those with larger RDD. Also, problems with LF = 0.2 seem, on average, to be slightly harder than those with LF = 0.4. However, a larger sample of test problems is needed to draw any firm conclusions.

5.8 Concluding Remarks

The dynamic programming state-space relaxation method maps the states of a dynamic programming formulation of the problem onto a smaller set of states and performs the recursion on this smaller state-space. This smaller state-space provides only some of the information that is required to perform the recursion to obtain an optimal solution. The information loss may result either in infeasible solutions being generated as in the case of our single machine scheduling problem, or (for minimization problems) in the minimum costs evaluated for each state being a lower bound instead of an exact value. In the former case, penalties and state-space modifiers can be used in an attempt to enforce feasibility and thereby improve the lower bound. In the latter case the use of state-space modifiers may possibly increase the lower bounds for each state towards their exact value and consequently increase the overall lower bound.

For the problem of scheduling jobs on a single machine to minimize total cost, dynamic programming is the best solution method for small sized problems. For medium sized problems dynamic programming algorithms fail because of their storage requirements. For such problems branch and bound algorithms that employ lower bounds obtained from the dynamic programming state-space relaxation

method provide a satisfactory solution method when processing times are small. Their success is attributed to the ability of the penalty and state-space modifier iterations to strengthen the lower bounds. These lower bounds appear to be superior to those of Fisher that are derived by performing a Lagrangean relaxation of machine capacity constraints. Unfortunately, the lower bounds obtained by the dynamic programming state-space relaxation method are computationally too time consuming (they require pseudopolynomial time) to be used in a branch and bound algorithm that will satisfactorily solve large sized problems. Therefore the currently available lower bounds tend to be either rather weak or computationally expensive to compute; hence they do not perform ideally in branch and bound algorithms. It seem that there is no obvious bounding technique which would provide sharper bound than ours for solving problem with $n > 25$.

The bound $B(\lambda,-)$, $B(\lambda,q^1)$ and $B(\lambda,q^2)$ are valid lower bounds for $1//\Sigma w_i T_i$ problem which is a special case of our problem. We investigate the application of the dynamic programming state-space relaxation method to the total weighted tardiness problem in chapter 6.

CHAPTER SIX

A COMPUTATIONAL COMPARISON OF ALGORITHMS FOR THE SINGLE MACHINE TOTAL WEIGHTED TARDINESS SCHEDULING PROBLEM

6.1. Introduction

The single machine total weighted tardiness problem may be stated as follows. Each of n jobs (numbered 1,...,n) is to be processed without interruption on a single machine that can handle only one job at a time. Job i (i = 1,...,n) becomes available for processing at time zero, requires an integer processing time p_i, and has a positive weight w_i and a due date d_i. For a given processing order of the jobs the (earliest) completion time C_i and the tardiness $T_i = \max\{C_i - d_i, 0\}$ of job i (i = 1,...,n) can be computed. The objective is to find a processing order of the jobs that minimizes the total weighted tardiness $\sum_{i=1}^{n} w_i T_i$.

Emmons [22] derives several dominance rules that restrict the search for an optimal solution to the total weighted tardiness problem. These rules are used in both dynamic programming and branch and bound algorithms. The dynamic use of Elmaghraby's Lemma [21], which is a special case of one of Emmons' rules, also reduces the number of solutions that need to be generated. Computational experience with the dynamic programming algorithm of Schrage and Baker [86] indicates that it is able to solve problems with 20 jobs without requiring excessive core storage. Lawler [53] proposes an alternative dynamic programming algorithm with smaller core storage requirements than the Schrage-Baker algorithm. However, for the total weighted tardiness problem no computational experience with

Lawler's algorithm is reported in the literature. Of the branch and bound algorithms, those of Fisher [24] and Potts and Van Wassenhove [80] appear to be the most efficient. Although no computational experience with Fisher's algorithm applied to the total weighted tardiness problem is reported in the literature, results for the total tardiness problem (in which all weights are equal) indicate that it is an effective approach for problems with small processing times. The Potts-Van Wassenhove algorithm solves total weighted tardiness problems with 30 jobs without generating very large search trees.

This chapter provides a computational comparison of the Schrage-Baker and Lawler dynamic programming algorithms, and of the Fisher and Potts-Van Wassenhove branch and bound algorithms. These results are compared with those obtained from branch and bound algorithms that employ two new lower bounding schemes. The first of these new lower bounds is obtained from a total weighted exponential function of completion times problem and the second is derived using the dynamic programming state-space relaxation method.

In Section 6.2, Emmons' dominance rules are described. Section 6.3 gives a general description of the dynamic programming approach for solving the problem and provides details of the Schrage-Baker and Lawler algorithms (which has been done in collaboration with Potts and Van Wassenhove [80]). The various lower bounding schemes that can be used in branch and bound algorithms are described next. Section 6.4 describes the Potts-Van Wassenhove approach in which a lower bound is obtained from a total weighted completion time problem. A similar method of approach is also used in this section to derive a new lower bound from a total weighted exponential

function of completion times problem. Section 6.5 describes Fisher's lower bound that is based on Lagrangean relaxation and Section 6.6 derives a new lower bound that uses the dynamic programming state-space relaxation method. A description of our branch and bound algorithms is contained in Section 6.7 and Section 6.8 reports on computational experience with the dynamic programming and branch and bound algorithms. Some concluding remarks are given in Section 6.9.

To summarize, this chapter discusses six algorithms for the total weighted tardiness problem and provides extensive computational results. Two algorithms (Schrage and Baker, and Lawler) use dynamic programming and four algorithms use branch and bound. Of the latter, two algorithms (Potts and Van Wassenhove [80], and the one which employs the new weighted exponential function of completion times bound) are based on a quickly computed but possibly rather weak lower bound. These branch and bound algorithms rely heavily on dominance rules to restrict the size of the search tree. The other two branch and bound algorithms (Fisher and the one which employs the new dynamic programming state-space relaxation bound) invest a substantial amount of computation time at each node of the search tree in an attempt to obtain a tight lower bound and thereby generate only small search trees.

6.2 Dominance Rules

Emmons' dominance rules are described in this section. They play a major role in the algorithms that are described later. Emmons [22] developed three rules in which relationships between job variables are explored for the one machine total tardiness problem. By exploring these relationships, the problem size can be reduced

considerably in most cases. Using these rules, it is sometimes possible to assign the first few and the last few jobs of an optimal sequence. The basic advantage of Emmons' approach is its ability to reduce the number of solutions that need to be considered. This is why most recent researchers used his rules to construct as many precedence relationships as possible between the jobs, and then employed an implicit enumeration technique to sequence the remaining jobs. Suppose that the rules have already been applied to yield, for each job h, a set B_h and a set A_h of jobs which precede and succeed job h in at least one optimal sequence. Let N denote the set of all jobs.

Theorem (6.1) (Dominance Theorem) . There exists an optimal sequence in which job i is sequenced before job j if one of the following conditions is satisfied.

(a) $p_i \leq p_j$, $w_i \geq w_j$ and $d_i \leq \max\{d_j, \Sigma_{h\in B_j} p_h + p_j\}$;

(b) $w_i \geq w_j$, $d_i \leq d_j$ and $d_j \geq \Sigma_{h\in N-A_i} p_h - p_j$;

(c) $d_j \geq \Sigma_{h\in N-A_i} p_h$.

Elmaghraby's Lemma [21] follows from condition (c): if a job j with $d_j \geq \Sigma_{h\in N} p_h$ is found, then there exists an optimal sequence in which this job is sequenced last. In such a case job j is removed from the problem.

Whenever jobs i and j are found satisfying the conditions of the Dominance Theorem, an arc (i,j) is added to a precedence graph G_p together with any other arcs (h,k) that are implied by transitivity.

We refer to h as a predecessor of k and to k as a successor of h. The procedure is repeated until no further arcs can be added to G_p. At this stage, any job which is a successor of all remaining jobs is removed from the problem. Similarly, any job i which is a predecessor of all remaining jobs is removed from the problem and the due dates of remaining jobs are reduced by p_i. In many applications, including those that follow in the next section, it is convenient to regard G_p as representing precedence constraints on the jobs that must be satisfied.

Consider the transitive reduction of G_p which is obtained by removing all arcs of G_p that are implied by transitivity. For each arc (i,j) of the transitive reduction of G_p, i is an immediate predecessor of j and j is an immediate successor of i.

Using G_p, the earliest completion time $C_i^E = \Sigma_{h \in B_i} p_h + p_i$ and the latest completion time $C_i^L = \Sigma_{h \in N-A_i} p_h$ of job i (i = 1,...,n) are computed.

6.3 General Precedence Constrained Dynamic Programming Algorithms

6.3.1. The dynamic programming approach

Both Schrage and Baker, and Lawler use the same dynamic programming recursion equations which are as follows. Let f(R,j) be the minimum total weighted tardiness when the jobs of the set R - {j} are sequenced in the first |R| - 1 positions followed by job j in position |R|. Then we may define $f(R,*) = \min_{j \in R} \{f(R,j)\}$ as the minimum total weighted tardiness when the jobs of R are sequenced in the first |R| positions. For this formulation, the objective is to find f(N,*) (recall N = {1,...,n}) using the recursion equations

$$f(R,j) = \min_{i \in R-\{j\}} \{f(R-\{j\},i) + w_j \max(\Sigma_{k \in R} p_k - d_j, 0)\} \qquad (6.1)$$

that are initialized by setting $f(\emptyset,j) = 0$ $(j = 1,\ldots,n)$. Clearly, recursion (6.1) may be written in the equivalent, more usual form, in which the objective is to find $f(N,*)$ from the recursion equations

$$f(R,*) = \min_{j \in R} \{f(R - \{j\},*) + w_j \max(\Sigma_{k \in R} p_k - d_j, 0)\} \qquad (6.2)$$

that are initialized by setting $f(\emptyset,*) = 0$. Equations (6.2) define a forward recursion as an initial partial sequence corresponds to each pair R and $f(R,*)$. A backward recursion is derived as follows. Let $f'(\bar{R},*)$ be the minimum total weighted tardiness when the jobs of the set $\bar{R}$ are sequenced in the final $|\bar{R}|$ positions. We can regard $\bar{R}$ as the complement of the set R which occurs in (6.2), i.e., $\bar{R} = N - R$. Then the problem is to find $f'(N,*)$ from the recursion equations

$$f'(\bar{R},*) = \min_{j' \in \bar{R}} \{f'(\bar{R} - \{j'\},*) + w_{j'} \max(\Sigma_{k \in N-\bar{R}} p_k + p_{j'} - d_{j'}, 0)\} \qquad (6.3)$$

that are initialized by setting $f'(\emptyset,*) = 0$.

As mentioned in the previous section, the graph G_p, obtained by applying Emmons' dominance rules, is assumed to define precedence constraints on the jobs. As a consequence, equations (6.2) and (6.3) are modified as follows. Firstly, only feasible sets R and $\bar{R}$ are considered in (6.2) and (6.3) respectively, i.e., sets R for which all predecessors of jobs of R are contained in R and sets $\bar{R}$ for which all successors of jobs of $\bar{R}$ are contained in $\bar{R}$. Clearly, there is a one to one correspondence between the feasible sets R for (6.2) and

the feasible sets $\bar{R} = N - R$ for (6.3). Secondly, the minimization in (6.2) is restricted to jobs j such that R - {j} is a feasible set, i.e., only jobs j which have no successors in R are considered. Similarly, the minimization in (6.3) is restricted to jobs j^* which have no predecessors in $\bar{R}$.

In addition to Emmons' rules, Elmaghraby's Lemma may be applied at each stage of the dynamic programming recursion to improve efficiency. Consider a set R for which $j \in R$ with $d_j \geq \Sigma_{h \in R} p_h$ exists. If the jobs of R are sequenced in the initial positions, then, from the Lemma, there exists an optimal ordering of these jobs in which job j is sequenced last. In such a case, recursion (6.2) reduces to $f(R,*) = f(R - \{j\},*)$ which involves no minimization and thus reduces computation. Suppose that $\bar{R} = N - R$, where again $j \in R$ with $d_j \geq \Sigma_{h \in R} p_h$ exists. If there exists an optimal schedule in which the jobs of $\bar{R}$ are sequenced in the final positions, then, from the Lemma, there exists an optimal schedule in which the jobs of $\bar{R} \cup \{j\}$ are sequenced in the final positions. Thus, sets $\bar{R} \cup \{k\}$ $(k \neq j)$ may be regarded as infeasible thereby reducing the total number of feasible sets and, consequently, storage requirements.

The dynamic programming algorithms differ mainly in the order in which the feasible sets R (or $\bar{R}$) are generated and the way in which the values $f(R,*)$ (or $f^*(\bar{R},*)$) are stored. The two following algorithms are compared.

6.3.2. The Schrage-Baker algorithm

In the Schrage-Baker dynamic programming algorithm the feasible sets R are generated in lexicographic order , i.e., in increasing order of $\Sigma_{i \in R} 2^{i-1}$. Also, an integer label is assigned to each job

(the label given to a job is chosen so that it exceeds by one the sum of labels already assigned to jobs that are neither its predecessors nor its successors) so that each feasible set is given an address which is equal to the sum of labels of jobs in that set. The value f(R,*) is stored in a location corresponding to the address for the set R. It is often the case that there are addresses for which there is no feasible set in which case storage space is wasted. The storage space required by the algorithm is equal to the sum of all labels and is known before any of the recursion equations are solved. Also, since all the values f(R,*) are stored, once all recursion equations have been solved, a simple backtracing procedure allows the optimal sequence to be found. (Kao and Queyranne [46] give an alternative implementation in which the storage space is used cyclically to give a storage requirement which is equal to the maximum label: backup storage is used to find the optimal sequence.) In our implementation, if the sum of labels does not exceed 48000 then the problem is solved; otherwise computation is abandoned for that problem.

6.3.3 Lawler's algorithm

The algorithm described here is a variant of the dynamic programming algorithm of Lawler which is designed so that core storage requirements are kept to a minimum. Further to this aim, a backward recursion is used which allows Elmaghraby's Lemma to be applied dynamically thereby reducing the number of feasible sets that need to be considered. The feasible sets $\bar{R}$ are generated in cardinality order . More precisely, when all feasible sets $\bar{R}$ of cardinality $\bar{r}$ have been generated, all feasible sets of cardinality

$\bar{r}+1$ are of the form $\bar{R} \cup \{k\}$ where $k \notin \bar{R}$ and where all successors of k are in $\bar{R}$. Each set is represented by its incidence vector, where the incidence vector for set $\bar{R}$ is defined as $\Sigma_{i \in \bar{R}} 2^{n-i}$. If the incidence vectors for the sets of successors of each job k are stored, the tests of whether $k \in \bar{R}$ and whether all successors of k are in $\bar{R}$ can be performed in constant time. (In FORTRAN, this is achieved by performing a logical AND statement on the logical variables obtained from the incidence vectors through an EQUIVALENCE statement.) During the generation of feasible sets of cardinality $\bar{r} + 1$ from a list of feasible sets $\bar{R}$ of cardinality $\bar{r}$ that are stored in increasing order of incidence vectors, if $d_j \geq \Sigma_{i \in N-\bar{R}} p_i$ for some $j \in N - \bar{R}$, then Elmaghraby's Lemma eliminates sets $\bar{R} \cup \{k\}$, where $k \neq j$. The generation process starts by forming the list of feasible sets of the form $\bar{R} \cup \{1\}$ together with the corresponding total weighted tardiness values $f(\bar{R},*) + w_1 \max(\Sigma_{i \in N-\bar{R}} p_i - d_1, 0)$. The next step is to find feasible sets of the form $\bar{R} \cup \{2\}$ and compute their total weighted tardiness values. This list is merged with the list of feasible sets $\bar{R} \cup \{1\}$: when duplicate sets (i.e., sets with the same incidence vector) are found during the merge, only the entry with the smaller total weighted tardiness value is retained. Note that the list of feasible sets of the form $\bar{R} \cup \{2\}$ does not need to be constructed explicitly since forward pointers to the appropriate entries in the list of feasible sets of cardinality $\bar{r}$ allow the necessary information to be accessed. Feasible sets of the form $\bar{R} \cup \{3\}, \ldots, \bar{R} \cup \{n\}$ are successively created and merged to give a complete list of feasible sets of cardinality $\bar{r} + 1$ stored in increasing order of incidence vectors. At this stage, the list of sets of cardinality $\bar{r}$ is discarded and the process of generating sets of cardinality $\bar{r} + 2$

commences.

In our implementation, one word of storage is used for each incidence vector. The total weighted tardiness value occupies a second word and the processing time $\sum_{i \in N-R} p_i$ occupies a third word. The forward pointers used in the generation phase are also stored in the third word together with the processing time in the form of a string. The storage space required by the algorithm needs to be sufficient to store all sets of cardinality $\bar{r}$ and all sets of cardinality $\bar{r} + 1$ simultaneously ($\bar{r} = 1,\ldots,n-1$). In our implementation, if during the course of applying the algorithm it becomes apparent that the number of sets of cardinality $\bar{r}$ plus the number of sets of cardinality $\bar{r} + 1$ ($\bar{r} = 1,\ldots,n-1$) exceeds 16000, then computation is abandoned for that problem. It is a disadvantage of the algorithm that the storage requirements of a particular problem cannot be predicted before any recursion equations are solved. Another disadvantage is that it is not simple to find the optimal sequence after the recursion equations have been solved unless, as is the case in our implementation, backup storage is used.

6.4. Lower Bounds by Reducing the Total Weighted Tardiness

6.4.1. Reduction to a linear function

Although the Potts-Van Wassenhove lower bound [80] is originally obtained using Lagrangean relaxation, it may also be derived by reducing the objective function as follows. Clearly, for job i (i = 1,...,n) we have

$$w_i T_i = w_i \max\{C_i - d_i, 0\} \geq u_i \max\{C_i - d_i, 0\} \geq u_i (C_i - d_i),$$

where $w_i \geq u_i \geq 0$. Let $u = (u_1,\ldots,u_n)$ be a vector of linear weights (i.e., weights for the linear functions $C_i - d_i$ $(i = 1,\ldots,n)$). Then a lower bound is given by the linear function

$$LB_{LIN}(u) = \min\{\Sigma_{i=1}^{n} u_i (C_i - d_i)\} \leq \min\{\Sigma_{i=1}^{n} w_i T_i\}.$$

This shows that the solution of a total weighted completion time problem provides a lower bound on the total weighted tardiness problem. Given u, the weighted completion time problem is solved by Smith's shortest weighted processing time rule [91] in which jobs are sequenced in non-increasing order of u_i/p_i. The best possible such lower bound is given by

$\max_{0<u<w} \{LB_{LIN}(u)\}$, where $w = (w_1,\ldots,w_n)$. To obtain this best possible bound, the subgradient optimization method [25,28] could be used to find u. However, since it is computationally expensive to apply, we prefer to use the following non-iterative heuristic method of Potts and Van Wassenhove [80] to determine u.

Suppose that a heuristic method is first applied to obtain a sequence and job completion times C_i^H $(i = 1,\ldots,n)$. Suppose also that the jobs are renumbered so that the heuristic sequence is $(1,\ldots,n)$. Then the vector of linear weights u is chosen to maximize $LB_{LIN}(u)$ subject to the condition that the heuristic sequence is an optimal solution of the total weighted completion time problem, i.e., u is a solution of the linear programming problem (P_{LIN}) defined by

$$
\begin{aligned}
& \text{maximize } LB_{LIN}(u) = \Sigma_{i=1}^{n} u_i(C_i^H - d_i) \\
(P_{LIN}) \qquad & \text{subject to } u_i/p_i \geq u_{i+1}/p_{i+1} \qquad (i = 1,\ldots,n-1) \\
& 0 \leq u_i \leq w_i \qquad (i = 1,\ldots,n).
\end{aligned}
$$

An algorithm which solves problem (P_{LIN}) in linear time is described later. First, however, we derive our new lower bound by reducing the weighted tardiness to an exponential function rather than the linear function used in this section.

6.4.2. Reduction to an exponential function

Our new lower bound is derived as follows. It is assumed that $P = \Sigma_{j=1}^{n} p_j > d_i$ for job i $(i = 1,\ldots,n)$. (If $P \leq d_i$, then job i is sequenced last by Elmaghraby's Lemma and discarded.) For any positive α, we aim to find a non-negative weight v_i for job i $(i = 1,\ldots,n)$ that satisfies

$$w_i T_i = w_i \max\{C_i - d_i, 0\} \geq v_i(e^{\alpha(C_i - d_i)} - 1). \qquad (6.4)$$

Clearly, (6.4) holds when $C_i \leq d_i$. When $d_i < C_i \leq P$, then (6.4) also holds provided that

$$v_i \leq w_i(P - d_i)/(e^{\alpha(P - d_i)} - 1). \qquad (6.5)$$

Let $v = (v_1,\ldots,v_n)$ be a vector of exponential weights (i.e., weights for the exponential functions $e^{\alpha(C_i - d_i)} - 1$ $(i = 1,\ldots,n)$). Then, if α is chosen, a lower bound is given by the exponential function

$$LB_{EXP}(v) = \min\{\Sigma_{i=1}^{n} v_i(e^{\alpha(C_i - d_i)} - 1)\} \leq \min\{\Sigma_{i=1}^{n} w_i T_i\}.$$

This shows that the solution of a total weighted exponential function of completion times problem provides a lower bound on the total weighted tardiness problem. When α and v are known, the total weighted exponential completion time problem is solved by Rothkopf's rule [84] in which jobs are sequenced in non-increasing order of $v_i/(e^{\alpha d_i}(1 - e^{-\alpha p_i}))$.

For the reasons indicated in the Section 6.4.1, we propose a non- iterative method to determine v rather than subgradient optimization. As before, suppose that a heuristic method is first applied to obtain a sequence $(1,\ldots,n)$ (where the jobs are suitably renumbered) and job completion times C_i^H $(i = 1,\ldots,n)$. Then the vector of exponential weights v is chosen to maximize $LB_{EXP}(v)$ subject to the condition that the heuristic sequence is an optimal solution of the total weighted exponential function of completion times problem, i.e., v is a solution of the linear programming problem (P_{EXP}) defined by

$$\text{maximize } LB_{EXP}(v) = \sum_{i=1}^{n} v_i(e^{\alpha(C_i^H - d_i)} - 1)$$

subject to

$$(P_{EXP}) \quad v_i/(e^{\alpha d_i}(1 - e^{-\alpha p_i})) \geq v_{i+1}/(e^{\alpha d_{i+1}}(1 - e^{-\alpha p_{i+1}})) \qquad (i=1,\ldots,n-1)$$

$$0 \leq v_i \leq w_i(P - d_i)/(e^{\alpha(P-d_i)} - 1) \qquad (i=1,\ldots,n).$$

We describe next an algorithm that solves the linear programming problems (P_{LIN}) of Section 6.4.1 and (P_{EXP}) in linear time.

6.4.3. An algorithm to solve (P_{LIN}) and (P_{EXP})

We first observe that problems (P_{LIN}) and (P_{EXP}) can be written in the general form

$$\begin{aligned} & \text{maximize } LB(z) = \sum_{i=1}^{n} a_i z_i \\ (P) \qquad & \text{subject to } b_i z_i \geq b_{i+1} z_{i+1} \qquad (i=1,\ldots,n-1) \quad (6.6) \\ & 0 \leq z_i \leq c_i \qquad (i=1,\ldots,n), \quad (6.7) \end{aligned}$$

where a_i $(i = 1,\ldots,n)$ is a constant and b_i and c_i $(i = 1,\ldots,n)$ are non-negative constants. When $a_i = C_i^H - d_i$, $b_i = 1/p_i$ and $c_i = w_i$ $(i = 1,\ldots,n)$ problem (P) reduces to problem (P_{LIN}). Similarly, when $a_i = e^{\alpha(C_i - d_i)} - 1$, $b_i = 1/(e^{\alpha d_i}(1 - e^{-\alpha p_i}))$ and $c_i = w_i(P - d_i)/(e^{\alpha(P-d_i)} - 1)$ $(i = 1,\ldots,n)$ problem (P) reduces to problem (P_{EXP}).

We observe that for any jobs h and i $(h,i = 1,\ldots,n)$ where $h \leq i$, constraints (6.6) and (6.7) yield

$$b_i z_i \leq b_h z_h \leq b_h c_h. \qquad (6.8)$$

Let us define

$$c_i' = \min_{h \in \{1,\ldots,i\}} \{b_h c_h\}/b_i \qquad (i = 1,\ldots,n).$$

In view of (6.8) adding the constraints

$$0 \leq z_i \leq c_i' \qquad (i = 1,\ldots,n) \qquad (6.9)$$

to problem (P) does not alter its solution. Since $c_i' \leq c_i$ $(i = 1,\ldots,n)$, these new constraints imply the original constraints (6.7)

which may therefore be dropped.

An algorithm to solve problem (P) is now presented. It is a generalization of the algorithm derived by Potts and Van Wassenhove [80] for problem (P_{LIN}). The proof of optimality follows that given by Potts and Van Wassenhove for (P_{LIN})(see Appendix). In the algorithm the variable D indicates whether z_k is set to its lower bound value given by (6.6) or its upper bound value given by (6.9) and the variable LB provides the lower bound.

Algorithm LP

Step 1 . Set D = 0, LB = 0 and k = 1.

Step 2 . Set D = D + a_k/b_k. If D $\leq$ 0, go to Step 4.

Step 3 . Set LB = LB + Db_kc_k and set D = 0.

Step 4 . If k = n, stop. Otherwise set k = k+1 and go to Step 2.

Clearly, Algorithm LP solves problem (P) in O(n) time.

To obtain the lower bounds LB_{LIN} and LB_{EXP} a heuristic method is required to sequence the jobs. In our implementation a heuristic method, which is applied at the top of the branch and bound search tree, provides the sequence initially. Thereafter, the sequence which currently corresponds to the best solution found by the branch and bound algorithm is used to generate the lower bounds LB_{LIN} or LB_{EXP}.

The best value of the non-negative α in (P_{EXP}) is found by using golden section search over the interval [0,1]. If the initial interval is [0,1], the two points are placed at 1-β and β, (i.e., at approximately 0.382 and 0.618). However, when the interval is reduced, one of the old points will then be in the correct position

with respect to the new interval. With golden section search, there is a constant reduction of the interval at every step such that the length of the interval converges linearly to zero [29]. In our implementation, 20 iterations of golden section search are performed to find a 'good' value α^* at the top of the search tree. A heuristic method is used to find a value α to be used within the search tree since it is computationally too expensive to apply the golden section search at each node of the search tree. Based on the results of numerous tests with various rules to determine α, the following heuristic is adopted. For $\alpha^* < 0.0001$ set $\alpha = 1000\alpha^*$ and for $\alpha^* \geq 0.0001$ set $\alpha = 6\alpha^*/(5RDD + TF - 1)$ if $5RDD + TF > 1.5$ and $\alpha = 12\alpha^*$ if $5RDD + TF \leq 1.5$, where $RDD = (\max_{i=1,\ldots,n} \{d_i\} - \min_{i=1,\ldots,n} \{d_i\})/\sum_{i=1}^{n} p_i$ is the relative range of due dates and $TF = 1 - \sum_{i=1}^{n} d_i/(n\sum_{i=1}^{n} p_i)$ is the average tardiness factor.

6.5 A Lower Bound from Lagrangean Relaxation

In this section Fisher's lower bound is described. It is obtained by performing a Lagrangean relaxation of the machine capacity constraints which restrict the machine to processing only one job at a time. In this approach a multiplier is associated with each of the $P = \sum_{i=1}^{n} p_i$ unit time intervals during which the machine is required to be busy.

The problem may be formulated using the zero-one variables n_{it} $(i = 1,\ldots,n;\ t = 1,\ldots,P)$ where

$$n_{it} = \begin{cases} 1 \text{ if the machine processes job } i \text{ in the time interval } [t-1,t]; \\ 0 \text{ otherwise.} \end{cases}$$

Recalling the notation in section 6.2 the problem may be stated as

$$\text{minimize } \sum_{i=1}^{n} w_i \max\{C_i - d_i, 0\}$$

$$\text{subject to } C_i \in \{C_i^E, \ldots, C_i^L\} \qquad (i=1,\ldots,n) \quad (6.10)$$

$$n_{it}=1 \text{ for } t=C_i-p_i+1,\ldots,C_i \qquad (i=1,\ldots,n) \quad (6.11)$$

$$n_{it}=0 \text{ for } t=1,\ldots,C_i-p_i,C_i+1,\ldots,P \quad (i=1,\ldots,n) \quad (6.12)$$

$$\sum_{i=1}^{n} n_{it}=1 \qquad (t=1,\ldots,P) \quad (6.13)$$

$$C_i + p_j \leq C_j \text{ for each arc } (i,j) \text{ of } G_p. \qquad (6.14)$$

To obtain a lower bound some of constraints (6.14) are relaxed and a Lagrangean relaxation of constraints (6.13) is performed. More precisely, let G_p' be a subgraph of the graph G_p obtained by deleting arcs from G_p until each job has at most one immediate successor in the resulting graph. Arcs are deleted so that longest paths in the network are retained. It is assumed that G_p' is connected: if not a dummy job n+1 with a large due date is added to generate arcs (i,n+1) for $i = 1,\ldots,n$ that ensure that G_p' is connected. Constraints (6.14) are replaced by

$$C_i + p_j \leq C_j \text{ for each arc } (i,j) \text{ of } G_p'. \qquad (6.15)$$

The lower bound is obtained by performing a Lagrangean relaxation of constraints (6.13). Applying (6.11) and (6.12), the resulting bound is given by

$$LB_{LR}(\mu) = \min\{\Sigma_{i=1}^{n} (w_i \max\{C_i - d_i, 0\} + \Sigma_{t=C_i-p_i+1}^{C_i} \mu_t) - \Sigma_{t=1}^{P} \mu_t\}$$

subject to (6.10) and (6.15),

where $\mu = (\mu_1,\ldots,\mu_P)$ is a vector of multipliers corresponding to constraints (6.13). We discuss next the problem of solving this Lagrangean problem for a given μ to obtain the lower bound.

Let β_i be the set of immediate predecessors of job i (i = 1,...,n) in graph G_p'. Assume that the jobs are renumbered so that i < j for each arc (i,j) of G_p'. The Lagrangean problem is solved by a dynamic programming recursion defined on $F_i(t)$ which represents the minimum total cost of scheduling job i and its predecessors to be completed not later than time t, where the contributions to the total cost for each job are its weighted tardiness and a multiplier for each time period in which it is scheduled. The following properties of G_p' are used to find the lower bound $LB_{LR}(\mu)$.

(a) The graph G_p' is connected,

(b) job n has all other jobs as its predecessors,

(c) each job i, where i≠ n, has exactly one immediate successor,

(d) the jobs are renumbered so that i < j for each arc (i,j) ε G_p'.

This lower bound is given by

$$LB_{LR}(\mu) = F_n(P) - \Sigma_{t=1}^{P} \mu_t,$$

where $F_n(P)$ is found from the recursion equations

$$F_i(t) = \begin{cases} \infty & \text{for } t=0,\ldots,C_i^E-1 \\ \min\{F_i(t-1),\ \Sigma_{k\in\beta_i} F_k(t-p_i) + \\ \quad w_i\max(t - d_i, 0) + \Sigma_{\tau=t-p_i+1}^{t} \mu_\tau\} & \text{for } t=C_i^E,\ldots,C_i^L \\ F_i(C_i^L) & \text{for } t=C_i^L+1,\ldots,P \end{cases}$$

To find the job completion times C_i^* (i = 1,...,n) which correspond to the value $F_n(P)$ that is generated, the following backtracing

procedure is performed. Using the properties of G_p^* and the job renumbering, job n has no successors while all other jobs have exactly one immediate successor. Select C_n^* as large as possible subject to $C_n^* \leq P$ and $F_n(C_n^*) < F_n(C_n^*-1)$. Thereafter, having determined $C_{i+1}^*,\ldots,C_n^*$ $(i = 1,\ldots,n-1)$, C_i^* is chosen as large as possible subject to $C_i^* \leq C_j^* - p_j$, where j is the immediate successor of i, and $F_i(C_i^*) < F_i(C_i^*-1)$.

The subgradient optimization method is used to find values of the multipliers $\mu_1,\ldots,\mu_p$. For each iteration of this method the Lagrangean problem is solved in O(nP) time. In our implementation of this bound in a branch and bound algorithm, 100 subgradient optimization iterations are performed at the root node of the search tree whereas at other nodes the multipliers are updated from their values at the parent node by performing 10 iterations.

6.6 A Lower Bound from Dynamic Programming State-Space Relaxation

Following the techniques outlined in section 5.3 one can show that a lower bound can be obtained by mapping states representing feasible sets of jobs (in recursion equations (6.1)) on to states representing the total processing time of jobs in the set, (i.e., a state R is mapped on to a state $\Sigma_{i\in R}\, p_i$). Hence in the resulting relaxed problem the objective is to find $f_1(P,*)=\min_{j\in N}\{f_1(P,j)\}$, using the recursion equations

$$f_1(t,j)=\begin{cases} \infty & \text{for } t=1,\ldots,C_j^E-1, C_j^L+1,\ldots,P \\ \min_{i\in N-\{j\}}\{f_1(t-p_j,i)\} + w_j\max\{t-d_j,\,0\} & \text{for } t=C_j^E,\ldots,C_j^L \end{cases} \qquad (6.16)$$

that are initialized by setting $f_1(t,j) = \infty$ for $t < 0$ and $f_1(0,j) = 0$ $(j = 1,\ldots,n)$. Recursion (6.16) is derived in a similar way to recursion (5.8) of the previous chapter. However, in this case we consider only those schedules for which $C_j^E \leq C_j \leq C_j^L$ $(j\epsilon N)$, where C_j^E and C_j^L are obtained from Emmons' rules, in an attempt to strengthen the bound.

We present here the method of improving our lower bound as developed in section 5.3.4. Assuming that the penalties are present, the lower bound is obtained by finding $LB_{SSR}(\lambda)=\min_{j\epsilon N}\{f(P,j;\lambda)\} - \Sigma_{i\epsilon N}\lambda_i$, from the improved recursion equations

$$f(t,j;\lambda)=\begin{cases} \infty & \text{for } t=1,\ldots,C_j^E-1,C_j^L+1,\ldots,P \\ \min_{i\epsilon N-\{j\}}\{f(t-p_j,i;\lambda) + w_j\max\{t-d_j,0\} + \lambda_j\} & \text{for } t = C_j^E,\ldots,C_j^L \end{cases} \tag{6.17}$$

that are initialized by setting $f(t,j;\lambda)= \infty$ for $t < 0$ and $f(0,j;\lambda)= 0$ $(j = 1,\ldots,n)$. Recursion (6.17) is analogous to recursion (5.15) when all modifiers are zero. This lower bound is computed in $O(nP)$ time. Initially all penalties are set to zero and thereafter they are updated by using an iterative method given in section 5.3.4.

It is worth mentioning here that we did not use the state-space modifiers q_i given in section 5.3.5, because that we observed from our initial computational experience that the computationally expense of including the state-space modifiers q_i in recursion equations (6.17) is not justified.

The subgradient optimization method is used to find values of the penalties $\lambda_1,\ldots,\lambda_n$ (see section 5.3.4). For each iteration of this method the dynamic programming recursion equations are solved in $O(nP)$ time. The lower bound LB_{SSR} is implemented in a branch and

bound algorithm in a similar way to LB_{LR}: at the root node 100 subgradient optimization iterations are performed whereas the penalties are updated from their values at the parent node by performing 10 iterations at other nodes of the search tree.

6.7. Branch and Bound Algorithms

This section describes a branch and bound algorithm which may employ any of the lower bounding schemes described above. The general framework for our algorithm follows that of Potts and Van Wassenhove [80].

Initially, the precedence graph G_p is constructed from Emmons' dominance rules as described in Section 6.2. Also at the root node of the search tree two heuristic methods are used to schedule the jobs. The better of the two heuristic sequences is used to provide an initial upper bound. The first heuristic method selects a job with no successors in G_p to be sequenced in the last unfilled position in the sequence: when there is a choice, one is chosen for which its weighted tardiness when sequenced in this last position is as small as possible. The selected job is deleted and the process is repeated until all jobs are scheduled. The second heuristic is a straightforward generalization to the case of weighted tardiness of the method of Wilkerson and Irwin [95].

The generalized version of the Wilkerson and Irwin heuristic method involves two partial sequences, σ and π for scheduled and unscheduled jobs respectively. Let the jobs of π be sequenced in EDD (earliest due date) order. Using the partial ordering σ, let $w_i T_i(\sigma)$ is the weighted tardiness of job i of σ. The statement of the method will now be given.

Generalized Wilkerson and Irwin Heuristic

Step 1. Choose $\pi=(\pi(1),\ldots,\pi(n))$ where $d_{\pi(1)} \leq \ldots \leq d_{\pi(n)}$ and σ is the empty partial sequence.

Step 2. Let i and j the first two jobs in π.

Step 3. If $w_iT_i(\sigma ij) + w_jT_j(\sigma ij) > w_jT_j(\sigma ji) + w_iT_i(\sigma ji)$, then set i=j go to Step 4. Otherwise add job i to the end of σ, remove i from π and set i=j. If job i is the only job in π sequence job i last in σ and stop. If there exist more than one job in π, let j be the second job in π and repeat Step 3.

Step 4. If σ is empty, sequence job i first in σ, and go to Step 2. Let k be the last job in σ and let σ' be the partial sequence obtained by removing job k from σ. If $w_kT_k(\sigma' ki) + w_iT_i(\sigma' ki) > w_iT_i(\sigma' ik) + w_kT_k(\sigma' ik)$, go to Step 5. Otherwise, add job i to the end of σ, remove job i from π and go to Step 2.

Step 5. Remove k from σ, return it to π in EDD order and go to Step 4.

The branch and bound algorithms use a backward sequencing branching rule which generates a search tree for which nodes at level ℓ correspond to final partial sequences in which jobs are sequenced in the last ℓ positions. A newest active node search selects a

node from which to branch.

A branch of the search tree in which a job is added to a final partial sequence is discarded unless all successors of that job in G_p appear in the final partial sequence. Further nodes are eliminated using Elmaghraby's Lemma: if in any subproblem it is possible to sequence a job last so that it has zero tardiness, then a single node is added to the search tree which sequences that job last in the subproblem. A further attempt is made to eliminate nodes using two tests which are based on the dominance theorem of dynamic programming. The first of these tests uses an adjacent job interchange to compare the sum of weighted tardiness for the two jobs most recently added to the final partial sequence with the corresponding sum when these two jobs are interchanged in position: if the former sum is larger than the latter, then the current node is eliminated, while if both sums are the same, some convention is used to decide whether the current node should be discarded. The second test uses the job labelling procedure of Schrage and Baker to construct an address for each subset of jobs that can form a final partial sequence of jobs which is consistent with the precedence graph G_p. Storage space limits us to the comparison of only those final partial sequences with an address of L or less: in our implementation the value of L depends on the lower bounding scheme adopted, although in all cases $25000 \leq L \leq 34000$. Using the labelling scheme, we can easily check whether such a final partial sequence can be compared with one that has been previously generated and, if so, whether the current node is dominated. When it is not dominated, the total weighted tardiness of jobs of the current partial sequence is stored, replacing any previously stored quantity

in that address.

For all nodes that remain after the dominance tests are applied, we compute one of the lower bounds LB_{LIN}, LB_{EXP}, LB_{LR} or LB_{SSR}. If the lower bound for any node is greater than or equal to the smallest of the previously generated upper bounds, then that node is discarded.

6.8. Computational Experience

The algorithms were tested on problems with 20, 30, 40 and 50 jobs that were generated as follows. For each job i, an integer processing time p_i and an integer weight w_i were generated from the uniform distribution [1,10]. Problem 'hardness' is likely to depend on parameters RDD and TF called the relative range of due dates and the average tardiness factor. Having computed $P = \sum_{i=1}^{n} p_i$ and selected a value of RDD and TF from the set {0.2, 0.4, 0.6, 0.8, 1.0}, an integer due date d_i from the uniform distribution [P(1 - TF - RDD/2), P(1 - TF + RDD/2)] was generated for each job i. Three problems were generated for each of the 25 pairs of values of RDD and TF, yielding 75 problems for each value of n. Note that these problems are generated in the same way as those of Potts and Van Wassenhove [80] except that in the latter case integer processing times are generated from the uniform distribution [1,100]. Our method yields smaller processing times with the result that the branch and bound algorithms that use the lower bounds LB_{LR} and LB_{SSR} which require pseudopolynomial time are favoured by these test problems. Also, these problems with a small number of distinct processing times tend to be easier because the conditions of Emmons' dominance rules are easier to fulfil.

The dynamic programming algorithms of Schrage and Baker, and Lawler, denoted by DPSB and DPLAW respectively, and the branch and bound algorithms BBLIN, BBEXP, BBLR and BBSSR which use the linear, exponential, Lagrangean relaxation and dynamic programming state-space relaxation lower bounds respectively, were coded in FORTRAN V and run on a CDC 7600 computer. For the four branch and bound algorithms, whenever a problem was not solved within a time limit of 60 seconds, computation was abandoned for that problem. Also, due to the storage limits of the two dynamic programming algorithms, problems are unsolved if the sum of labels for algorithm DPSB exceeds 16000 and if the number of feasible sets with cardinalities differing by at most one for algorithm DPLAW exceeds 48000. Algorithms BBLR and BBSSR were not tested on the problems with 40 and 50 jobs because of the discouraging results obtained for n = 30.

Results comparing the performances of the algorithms are given in Tables 6.1 and 6.2. For each value of n, Table 6.1 lists the median computation time in seconds required by each algorithm to solve the test problems and also, when any exist, gives the number of unsolved problems. For algorithm DPSB when n = 40 and n = 50 and for algorithm DPLAW when n = 50, however, because over half of the problems are unsolved the median cannot be computed. Table 6.2 lists the median sum of labels for Algorithm DPSB, the median numbers of feasible sets generated for algorithm DPLAW (excluding the entry for n = 50 where the median cannot be computed) and the median numbers of nodes in the search tree for the branch and bound algorithms.

Before giving an overall comparison, it is appropriate to discuss the algorithms in pairs. The dynamic programming algorithms

Table 6.1

Median computation time in seconds and numbers of unsolved problems

n	DPSB	DPLAW	BBLIN	BBEXP	BBLR	BBSSR
20	0.03	0.06	0.03	0.05	2.48	1.65
30	0.15:12	0.47: 2	0.14	0.24	6.91: 2	7.96: 3
40	- :43	4.75:25	0.67: 4	1.28: 3	-	-
50	- :45	- :39	2.03:16	5.05:19	-	-

Table 6.2

Median sums of labels for DPSB, median numbers of feasible sets for DPLAW and median numbers of nodes for the branch and bound algorithms

n	DPSB	DPLAW	BBLIN	BBEXP	BBLR	BBSSR
20	504	434	91	104	53	28
30	7820	3799	319	456	135	112
40	73253	33873	1162	1781	-	-
50	422340	-	2717	4128	-	-

are compared first, followed by the branch and bound algorithms BBLIN and BBEXP which use quickly computed lower bounds. Lastly, algorithms BBLR and BBSSR that use tighter lower bounds which require pseudopolynomial time are discussed.

We first observe from Table 6.1 that although algorithm DPLAW is able to solve several problems that are unsolved when algorithm DPSB is applied, computation times are generally larger than those for algorithm DPSB. These results are in accordance with those obtained by Potts and Van Wassenhove [81] for the total tardiness problem but at variance with those given by Kao and Queyranne for the assembly line balancing problem where Lawler's algorithm is found to require less computation time than the Schrage-Baker algorithm. It would be misleading to suggest from our results that one of these algorithms is clearly superior to the other since algorithm DPSB is fast and easy to code whereas algorithm DPLAW is able to solve larger problems using the same amount of core storage.

Tables 6.1 and 6.2 indicate that algorithm BBLIN is superior to algorithm BBEXP. However, the results for $n = 40$ show that there is one less unsolved problem for BBEXP, so our new exponential bound does have some merits. At the root node of the search tree where the golden section search is used to find α^*, the bound LB_{EXP} is sometimes substantially better than LB_{LIN} for problems with small TF ($TF \leq 0.6$) and tends to be only slightly worse for problems with large TF ($TF \geq 0.8$). Unfortunately, the bounds are not tight enough to justify the use of the golden section search at each node of the search tree. In spite of much initial experimentation with various heuristic methods to compute values of α within the search tree, we are unable to find a method which yields smaller search trees than

those generated by algorithm BBLIN.

We next observe from Table 6.1 that algorithm BBLR appears to be slightly better than algorithm BBSSR for the problems tested, even though the median numbers of nodes in the search tree are larger. However, in both cases computation times are large. Results of Abdul-Razaq and Potts [1] for the problem in which jobs have costs for earliness as well as for tardiness, where there are no dominance rules analogous to those of Emmons, show that algorithm BBSSR is superior. The most likely explanation of the difference is that the lower bound LB_{LR} uses some of the constraints of G_p whereas LB_{SSR} relaxes all such constraints after the earliest and latest completion times are computed. Initial experiments with a modified version of LB_{SSR} which retains some of the precedence constraints of G_p, indicate that the improvement to the lower bound is insufficient to compensate for its extra computational requirements.

Lastly, we give an overall comparison of the algorithms. Algorithm BBLIN is the most efficient and is able to effectively solve problems with up to 40 jobs. Ignoring BBLIN, our new algorithm BBEXP is the most effective. The dynamic programming algorithms require too much core storage to compare favourably with these branch and bound algorithms although computation times are small for algorithm DPSB. The tighter lower bounds employed in algorithms BBLR and BBSSR successfully limit the size of search trees but by an insufficient amount to justify their heavy computational requirements. For all algorithms, unsolved problems tend to lie in those classes which have traditionally been considered the hardest.

6.9. Concluding Remarks

In this chapter we discuss and compare existing and new algorithms for the single machine total weighted tardiness problem. The branch and bound algorithm of Potts and Van Wassenhove [80] (BBLIN) which obtains a lower bound from a linear function of completion times problem is the most efficient and is able to solve problems with up to 40 jobs. A new but similar algorithm (BBEXP) which replaces the linear function with an exponential function also yields reasonable results. For the other branch and bound algorithms which use Lagrangean relaxation of machine capacity constraints (BBLR) and dynamic programming state-space relaxation (BBSSR), the computational requirements of the lower bounds are too time consuming to yield a competitive algorithm. Results for the Schrage-Baker algorithm (DPSB) show that dynamic programming algorithms can yield small computation times, although for larger problems dynamic programming is limited by computer core storage requirements, even when special attempts are made to minimize storage (DPLAW).

To solve larger problems, a tighter lower bound than that obtained from a linear function of completion times is needed. Ideally, it would require polynomial time, although a non-iterative pseudopolynomial scheme would not be ruled out. Unfortunately, there is no obvious way to approach the derivation of lower bounds having these desired characteristics.

CHAPTER SEVEN

THE TWO-MACHINE FLOW SHOP PROBLEM WITH TRANSPORTATION TIME BETWEEN THE MACHINES

7.1 Introduction

The problem may be stated as follows. Consider n jobs (numbered $1,\ldots,n$) and two machines (labelled A and B). Neither of the machines can process more than one job at a time. Each job is processed first on machine A, then is transported to machine B, and lastly is processed on machine B. For each job i, a_i and b_i denote the processing times on machines A and B respectively and t_i denotes the transportation time. The objective is to find a schedule that minimizes the maximum completion time on machine B. Denote this problem by $F2\ /t_i/C_{max}$.

When each transportation time is zero, Johnson [43] shows that there exists an optimal schedule in which the processing orders on the two machines are identical. He also shows that the problem can be solved in O(n log n) steps by sequencing jobs i with $a_i \leq b_i$ first in non-decreasing order of a_i followed by the remaining jobs i (with $a_i > b_i$) sequenced in non-increasing order of b_i.

This result of Johnson has been extended to the two machine flow shop problems with arbitrary time lags (Mitten [68]). Mitten defines the start lag $h_i \geq 0$ of job i ($i=1,\ldots,n$) as the minimum time interval between starting job i on machine A, and starting it on machine B. Similarly the stop lag $h_i' \geq 0$ of job i is the minimum time interval between completing job i on machine A and completing it on machine B. Assuming the same processing order on each machine, this

generalized problem is solved as follows. Defining $h_i^* = \max(h_i - a_i, h_i - b_i)$ and applying Johnson's algorithm to the processing times $(a_i+h_i^*, b_i+h_i^*)$ will produce an optimal schedule. Johnson [44] has discussed the same problem presenting another proof of optimality.

It is interesting to examine some variations of the problem with zero transportation times. When there are precedence constraints on the jobs (i.e., if job i has precedence over job j, then for each machine job j can not be processed on that machine before job i has been processed on that machine) Kurisu [48] presents an algorithm for the case of parallel chain precedence constraints and Sidney [90] generalizes this algorithm to solve problems with series-parallel precedence constraints. For the case of general precedence constraints, Monma [70] shows that the problem is NP- hard. Hariri and Potts [36] derive a lower bound based on Lagrangean relaxation and use it in branch and bound algorithms based on three different branching rules.

An extension of Johnson's algorithm to the two-machine flow shop scheduling problem with setup times, in which the processing orders on the two machines are identical, is given by Yoshida and Hitomi [96]. Also their result and Mitten's result are extended by Szwarc [92] to the case in which each job has a setup time, a processing time and removal time. (If job i is sequenced immediately before job j on one of the machines, then after job i is completed on that machine a removal time for job i followed by a setup time for job j are required before job j can start its processing. For the first operation on a machine only a setup time is needed and for the final operation only a removal time is needed.)

When there is a non-negative release date r_i for each job i,

Lenstra et al. [60] have shown that the problem is NP-hard, and branch and bound algorithm for the problem proposed by Grabowski [30]. Three heuristic methods are presented by Potts [77] of which two have a worst-case performance ratio of 2, while the third one is modified to give an improved worst-case performance ratio of 5/3.

For the case of transportation times, Lenstra [58] claims that the problem is NP-hard, which indicates that the existence of a polynomial bounded algorithm to solve the problem is unlikely and an enumerative approach is required.

In this chapter, a branch and bound algorithm is used. Some basic results, a heuristic method and its worst-case analysis are given in Section 7.2, Section 7.3 contains a description of our branching rule together with a derivation of single and two-machine lower bounds. An alternative lower bound based on Lagrangean relaxation is given in Section 7.4. The use of dynamic programming dominance for this problem is discussed in Section 7.5. The algorithms for solving this problem are given in Section 7.6. Computational experience is presented in Section 7.7. Concluding remarks are given in Section 7.8.

7.2 Heuristic Method

7.2.1 Some basic results

In the case of our problem $F2/t_i/C_{max}$, there exist $(n!)^2$ possible orderings of the jobs on machines A and B. The following Lemma reduces the number of possible schedules that have to be considered to n! only. Let C_i^A be the completion time of job i on machine A.

Lemma (7.1). If σ is an optimal sequence on machine A, then an optimal sequence σ' on machine B is obtained by ordering the jobs according to non-decreasing arrival times.

Proof. Let $r_i = C_i^A + t_i$ for every $i \in \sigma$, is the arrival time at machine B. Then the sequence σ' on machine B which is obtained by ordering the jobs according to non-decreasing r_i is an optimal sequence as required. □

For a special case of the two-machine flow shop scheduling problem with transportation times, an optimal sequence minimizing the maximum completion time is given by the following result.

Theorem (7.1). If in an $F2/t_i/C_{max}$ problem we have $t_i \leq a_j + t_j$ for every i, j, then there exists an optimal schedule in which the two processing orders are identical. Furthermore the common processing order is obtained from the optimal processing order for the $F2//C_{max}$ problem having processing times (a_i+t_i, b_i+t_i).

Proof. Assume that we have a sequence on machine A such that job i is sequenced before job j. We want to show that job i precedes job j on machine B. (i.e., we want to show that the arrival time of job i on machine B is less than or equal to the arrival time for job j.) If C_i^A and C_j^A are the completion times of job i and job j respectively, then $C_i^A + t_i$ and $C_j^A + t_j$ are their respective arrival times, and we have

$$C^A_{i+1_i} \leq C^A_{i+a_j+1_j} \text{ since } 1_i \leq a_j+1_j \qquad \forall\ i,j.$$

Hence $C^A_{i+1_i} \leq C^A_{j+1_j}$ as required. Johnson [43] shows that by considering the processing times $(a_i+1_i,\ b_i+1_i)$ for an F2/ $/C_{max}$ problem, an optimal sequence is obtained. □

Theorem (7.1) shows that Johnson's rule guarantees an optimal solution for the case $1_i \leq a_j+1_j$ for every i,j. If the processing times a_i and b_i are interchanged (i=1,...,n), then an equivalent inverse problem results.

Corollary (7.1). If in an $F2/1_i/C_{max}$ problem we have $1_i \leq b_j+1_j$ for every i,j, then there exists an optimal schedule in which the two processing orders are identical. Furthermore the common processing order is obtained from the optimal processing order for the $F2//C_{max}$ problem having processing times (a_i+1_i, b_i+1_i).

Proof. We prove this corollary by applying theorem (7.1) to the inverse problem. □

We shall use Theorem (7.1) and corollary (7.1) later in deriving a lower bound. They also indicate an ordering of the jobs in the heuristic method of the next section.

7.2.2 Description of heuristic method

It is well-known that the computation can be reduced by using a heuristic to act as un upper bound on the maximum completion time prior to the application of a branch and bound algorithm.

An obvious heuristic is to order the jobs by applying Johnson's algorithm to the processing times $(a_i+\ell_i, b_i+\ell_i)$ and use this processing order on each machine. We denote this heuristic by J.

With the help of Lemma (7.1), and Theorem (7.1) which are given above we can improve heuristic J as follows. The procedure starts by applying Johnson's algorithm to the processing times $(a_i+\ell_i, b_i+\ell_i)$ to obtain a processing order for machine A. If we use Lemma (7.1) above we obtain a sequence on machine B by ordering the jobs according to non-decreasing arrival times. If Theorem (7.1) above is not satisfied, this means that the schedule obtained is not optimal in general. Then, we may possibly decrease the maximum completion time C_{max} by considering the inverse problem: the sequence obtained for the (original) machine B is reversed and used as a sequence for the first machine in the inverse problem. A sequence for the second machine in the inverse problem (machine A for the original problem) is obtained by Lemma (7.1). Now we can repeat the same technique on machine A, continuing until the maximum completion does not decrease from one iteration to the next. Note that the sequence of C_{max} values obtained by this procedure is necessarily non-increasing. Denote this heuristic by H.

7.2.3 Worst-case analysis

Suppose that C^*_{max} denotes the minimum value of the maximum completion time while C^J_{max} (C^H_{max}) denotes the maximum completion time when the jobs are sequenced using heuristic J (heuristic H).

Theorem (7.2). $C^J_{max} / C^*_{max} < 2$ and this bound is the best possible.

Proof. If the heuristic J generates a sequence $(\sigma(1),\ldots,\sigma(n))$, then it is well known from flow shop theory that the corresponding maximum completion time can be written for some $u \in \{1,\ldots,n\}$, as

$$C^J_{max} = \Sigma_{i=1}^{u} a_{\sigma(i)} + \ell_{\sigma(u)} + \Sigma_{i=u}^{n} b_{\sigma(i)}.$$

Assume that

$$a_{\sigma(u)} \leq b_{\sigma(u)} \quad \text{(the proof is similar if } a_{\sigma(u)} > b_{\sigma(u)} \text{)}.$$

Hence

$$C^J_{max} \leq \Sigma_{i=1}^{u} b_{\sigma(i)} + \ell_{\sigma(u)} + \Sigma_{i=u}^{n} b_{\sigma(i)}$$

$$= b_{\sigma(u)} + \ell_{\sigma(u)} + \Sigma_{i=1}^{n} b_{\sigma(i)}$$

$$< 2\, C^*_{max}, \text{ (since } C^*_{max} > \Sigma_{i=1}^{n} b_{\sigma(i)}, \text{ and } C^*_{max} \geq a_{\sigma(u)} + \ell_{\sigma(u)} + b_{\sigma(u)} > b_{\sigma(u)} + \ell_{\sigma(u)}).$$

To complete the proof, we present the following examples to show that the bound of Theorem (7.2) is the best possible.

Example 7.1. Consider the (n+1)-job problem (n is even number) specified by the data in Table 7.1, where $k > 0$, ε arbitrary small positive number.

Table 7.1. Data for example 7.1

i	1	...	n/2	n/2 +1	...	n	n+1
a_i	k	...	k	k+ε	...	k+ε	ε
t_i	0	...	0	0	...	0	(n+1)k
b_i	k+ε	...	k+ε	k	...	k	ε

An optimal sequence on machine A is (n+1,1,...,n) with C^*_{max} =(n+1)k as ε⟶ 0. The sequence which is obtained by Johnson's algorithm is (1,...,n/2,n+1,n/2+1,...,n) with C^J_{max}=(2n+1)k as ε⟶ 0. Hence C^J_{max}/C^*_{max} =2 - 1/(n+1) which can be arbitrary close to 2 as required. □

We also observe, that if heuristic J is applied to the inverse of the problem in example 7.1, then C^J_{max}/C^*_{max} can be arbitrarily close to 2. Thus, in terms of worst-case performance there is no advantage in applying heuristic J to the original problem and its inverse and taking the better solution.

Example 7.2. We show that, for arbitrary n the bound 2 is the best possible. Consider the n-job problem specified by the data in table 7.2.

Table 7.2. Data for example 7.2

i	1	...	n-1	n
a_i	k	...	k	ε
t_i	0	...	0	nk
b_i	k+ε	...	k+ε	ε

Clearly (n,1,...,n-1) is an optimal sequence on machine A, with C^*_{max} =nk as $\varepsilon \longrightarrow 0$. When the heuristic J is applied, the sequence (1,...,n) with C^J_{max} =(2n-1)k as $\varepsilon \longrightarrow 0$. Hence C^J_{max}/C^*_{max} =2-1/n which can be arbitrary close to 2.

The example shows that when the sequence (1,...,n) is used on machine A, a non-decreasing order of arrival sequence on machine B is also (1,...,n). Thus, in terms of worst-case performance, there is no advantage in applying heuristic J to obtain an ordering on machine A and then using Lemma (7.1) to obtain an ordering on machine B.

For the heuristic H in which a sequence of schedules is generated, we have $C^H_{max}/C^*_{max} < 2$ from Lemma (7.1). However, in this case we suspect the bound is not the best possible. The following example provides the largest value of C^H_{max}/C^*_{max} that we found.

Example 7.3. The 3-job problem specified by the data in table 7.3.

Table 7.3. Data for example 7.3.

i	1	2	3
a_i	1	1	1
t_i	0	1	2
b_i	1	1	1

Johnson's algorithm gives the sequence (1,2,3) on machine A and Lemma (7.1) gives the sequence (1,2,3) on machine B yield a maximum completion time of 6. If we use the heuristic H (i.e., sequence (3,2,1) for reverse problem), then we obtain the same schedule . An optimal schedule is (3,1,2) on machine A, (1,3,2) on machine B which gives C^*_{max}=5. Hence C^H_{max}/C^*_{max}=6/5.

7.3 Single and Two-Machine Bounds

We use a forward branching rule in which each node of the search tree corresponds to an initial partial sequence of jobs. Each new branch that is added represents the addition of another job to the corresponding initial partial sequence.

In this section, we shall be interested in deriving a lower bound on the maximum completion time. Let S be the set of unsequenced jobs, σ be an initial partial sequence of jobs, the last of which is completed on machine A at time $C^A(\sigma)$.

7.3.1 Single-machine bounds

To construct a lower bound LB_A on machine A, we relax the capacity constraints on machine B (i.e., machine B can process more than one job at a time). A $1//L_{max}$ problem results having due dates $d_i = -(t_i+b_i)$. If $q_i = t_i+b_i$, this problem is solved by the EDD rule: jobs are sequenced in non-increasing order of q_i ($i \in S$). Calculate completion times C_i^A for this sequence, and for jobs in σ, the lower bound LB_A is given by

$$LB_A = \max_i \{C_i^A + q_i\}$$

Now we shall construct the second single-machine bound LB_B on machine B. If we relax the capacity constraints on machine A, (i.e., machine A can process more than one job at a time) for jobs of S, then the problem is equivalent to $1/r_i/C_{max}$. For a job i in σ then r_i is straightforwardly written as $r_i = C_i^A + t_i$; otherwise $r_i = C^A(\sigma) + a_i + t_i$. To obtain a lower bound on the maximum completion time, sequence the jobs in non-decreasing order of r_i. Assuming that the jobs are renumbered so that the resulting sequence is $(1,\ldots,n)$, we compute

$$C^B_1 = r_1 + b_1 \text{ , and}$$

$$C^B_i = \max \{C^B_{i-1}, r_i\} + b_i \text{ , } i=2,\ldots,n.$$

Hence $LB_B = C^B_n$.

7.3.2 Two-machine bounds

If the transportation times correspond to the conditions of Theorem (7.1) (or corollary (7.1)), then the problem is solved. However, if they do not, some of them can be reduced in value until the conditions are satisfied at which stage a lower bound is obtained. To satisfy the conditions of Theorem (7.1) (or corollary (7.1)), let $e=\min_{i\in S} \{a_i+t_i\}$, $f=\min_{i\in S} \{b_i+t_i\}$, and $g=\max \{e,f\}$. Reduce the value of t_i to g (i.e., for each job i with $t_i > g$, set $t_i=g$). Let $t'_i=\min_{i\in S} \{t_i,g\}$, and apply Johnson's algorithm to the relaxed problem with processing times $(a_i+t'_i, b_i+t'_i)$ to obtain $C^B_{max}(S)$. Hence the lower bound $LB_1(S)$ is given by

$$LB_1(S) = C^A(\sigma) + C^B_{max}(S).$$

For the special case when σ is empty, we shall show later that this bound $LB_1(S)$ is greater than or equal to the single machine bound LB_B.

Suppose that some of the unscheduled jobs are ignored and $LB_1(S')$ is computed for the remaining unscheduled jobs in S'. For a suitable choice of S', it may be the case that $LB_1(S') > LB_1(S)$. A suitable set S' will contain jobs i having relatively large t_i. The following heuristic is used to find S'.

Let us first define a block as a period of continuous

utilization of the machine B (when we calculated LB_B in section 7.3.1) such that the last job in the block completes its processing at a time t, which is less than the ready time of the first job in the new block. Assume we obtain the blocks $S_1,\ldots,S_m$ and let S' be the set of jobs in block S_m. Instead of immediate computing $LB_1(S')$, we investigate the possibility of computing $LB_1(S^*)$ where $S^* \supset S'$. The set S^* is obtained from S' by successively introducing job j when the conditions below are satisfied. The search for such jobs starts with block S_{m-1} proceeding from the last job to the first job in this block. The blocks $S_{m-2},\ldots,S_1$ respectively are searched in a similar way. The job j is introduced to give $S^*=S' \cup (j)$ if it appears likely that $LB_1(S^*) > LB_1(S')$. Firstly, by introducing job j into the set S' an additional contribution to the lower bound of a_j or b_j will result. However, the reduction in the transportation times need to satisfy the condition of Theorem (7.1) (or corollary (7.1)) may be large after job j is introduced. Let $t_*=\max_{i\in S'}\{t_i\}$, $e'=\min_{i\in S'}\{a_i+t_i\}$, $f'=\min_{i\in S'}\{b_i+t_i\}$, $g'=\max\{e',f'\}$, $e^*=\min\{e', a_j+t_j\}$, $f^*=\min\{f',b_j+t_j\}$, and $g^*=\max\{e^*, f^*\}$. Then when the set S' is used the maximum reduction in transportation time is given by $\max\{t_*-g',0\}$. The corresponding reduction to the transportation time when the set S^* is used is given by $\max\{t_*-g^*,0\}$. Thus, if the difference between these reductions is less than $\min\{a_j,b_j\}$, the set S^* should be used in preference to S'. Thus, a job j is introduced if

$$\max\{t_*-g^*, 0\} - \max\{t_*-g', 0\} \leq \min\{a_j, b_j\}.$$

If this condition is satisfied, then we reset $S'=S^*$ and attempt to augment S' further. If the test is not satisfied for any job j, a

lower bound based on the current S' is computed.

Now apply the same techniques of the lower bound $LB_1(S)$ on the set of jobs S'. Hence the modified lower bound $LB_1(S')$ is given by

$$LB_1(S') = C^A(\sigma) + C^B_{max}(S').$$

Theorem (7.3). When σ is empty, then $LB_1(S_m) \geq LB_B$.

Proof. For the calculation of LB_B the jobs are sequenced in non-decreasing order of r_i, where $r_i = a_i + t_i$. Assuming that the resulting sequence is $(\pi(1),\ldots,\pi(n))$. Let $\pi(u)$ be the first job of the block S_m. Then

$$LB_B = a_{\pi(u)} + t_{\pi(u)} + \Sigma_{i \in S_m} b_{\pi(i)}.$$

For $LB_1(S_m)$ for each $i \in S_m$, $t_i' = \min\{t_i,\ a_{\pi(u)} + t_{\pi(u)}\}$.

Let $j \in S_m$ be the first job sequenced in the calculation of $LB_1(S_m)$. Then

$$\begin{aligned} LB_1(S_m) &\geq a_j + t_j' + \Sigma_{j \in S_m} b_j \\ &= a_j + \min\{t_j,\ a_{\pi(u)} + t_{\pi(u)}\} + \Sigma_{j \in S_m} b_j \\ &\geq \min\{a_j + t_j,\ a_{\pi(u)} + t_{\pi(u)}\} + \Sigma_{j \in S_m} b_j \\ &= a_{\pi(u)} + t_{\pi(u)} + \Sigma_{j \in S_m} b_j \\ &= LB_B. \quad \square \end{aligned}$$

Although we have no proof that $LB_1(S') \geq LB_1(S)$ and $LB_1(S') \geq LB_1(S_m)$, initial experiment have shown $LB_1(S')$ to compare favourably

with $LB_1(S)$ and $LB_1(S_m)$.

7.4. Lagrangean Relaxation

In this section we shall use the Lagrangean relaxation technique to construct a lower bound $L(\lambda)$. In this method appropriate constraints are introduced into a Lagrangean function using multipliers λ_i. Consider the problem formulated as follows. The variable C^B_{max} represents the maximum completion time, C^A_i, C^B_i are given in section 7.3.1. Now the problem is

Minimize C^B_{max}

subject to

$$C^B_i \leq C^B_{max} \qquad \text{for } i=1, \ldots,n \qquad (7.1)$$

$$C^B_i \geq C^A_i + t_i + b_i \qquad \text{for } i=1, \ldots,n \qquad (7.2)$$

where the minimization is over all processing orders on the two machines with C^A_i and C^B_i subject to machine capacity constraints. If we perform a Lagrangean relaxation of constraint (7.2) using non-negative multipliers specified by the vector $\lambda=(\lambda_1,\ldots,\lambda_n)$, then the Lagrangean function becomes.

$$L(\lambda)=\min \{C^B_{max} + \Sigma^n_{i=1} \lambda_i (C^A_i + t_i + b_i - C^B_i)\}$$

subject to (7.1). This may be written as

$$L(\lambda)=\min\ \{\Sigma_{i=1}^{n}\ \lambda_i C_i^A + C_{max}^B - \Sigma_{i=1}^{n}\lambda_i C_i^B\} + \Sigma_{i=1}^{n}\ \lambda_i(t_i + b_i)$$

subject to (7.1).

In this Lagrangean problem there is no constraint involving both C_i^A and C_i^B. Thus, the Lagrangean problem becomes

$$L(\lambda)=\min\ \{\Sigma_{i=1}^{n}\ \lambda_i C_i^A\} + \min\ \{C_{max}^B - \Sigma_{i=1}^{n}\ \lambda_i C_i^B\} + \Sigma_{i=1}^{n}\ \lambda_i(t_i + b_i)$$

subject to (7.1).

Next we discuss how the Lagrangean problem is solved when λ is given. The first term in the Lagrangean function is minimized by applying Smith's rule [91] (sequencing the jobs in non-increasing order of λ_i/a_i). To minimize the second term $(C_{max}^B - \Sigma_{i=1}^{n}\ \lambda_i\ C_i^B)$, we observe that $C_i^B \leq C_{max}^B$ for all i. Therefore, $\Sigma_{i=1}^{n} - \lambda_i\ C_i^B$ is minimized, also by applying Smith's rule: the jobs are sequenced in non-increasing order of $-\lambda_i/b_i$ (i.e., in non-decreasing order of λ_i/b_i) and scheduled (with no idle time between jobs) so that the last job is sequenced to be completed at time C_{max}^B, where C_{max}^B is yet to be determined. Let $(\sigma(1),\ldots,\sigma(n))$ denote the sequence determined by λ_i/b_i. Thus, for our schedule

$$C_{max}^B - \Sigma_{i=1}^{n}\ \lambda_i C_i^B = C_{max}^B - \lambda_{\sigma(n)}C_{max}^B -$$

$$\lambda_{\sigma(n-1)}(C_{max}^B - b_{\sigma(n)}) - \ldots - \lambda_{\sigma(1)}(C_{max}^B - b_{\sigma(n)} - \ldots - b_{\sigma(2)})$$

which can be written as

$$C^B_{max} - \Sigma^n_{i=1} \lambda_i C^B_i = C^B_{max}(1 - \lambda_n - \ldots - \lambda_1) + V,$$

where V is non-negative constant. If $\Sigma^n_{i=1} \lambda_i > 1$, then $(C^B_{max} - \Sigma^n_{i=1} \lambda_i C^B_i)$ can be made arbitrary small by increasing C^B_{max}. Consequently, we use multipliers which satisfy the condition $\Sigma^n_{i=1} \lambda_i \leq 1$. Thus $(C^B_{max} - \Sigma^n_{i=1} \lambda_i C^B_i)$ is minimized by setting C^B_{max} to be as small as possible, subject to the constraint that job $\sigma(1)$ can not have a negative start time. However an improved lower bound is obtained by using the improved constraint $C^B_{max} \geq T$, where $T=LB_B$ (given in section 7.3.1). Then $(C^B_{max} - \Sigma^n_{i=1} \lambda_i C^B_i)$ is minimized by scheduling the jobs in the interval $[T - \Sigma^n_{i=1} b_i, T]$. The third term in the Lagrangean function is constant. The value of the multipliers $\lambda=(\lambda_1,\ldots,\lambda_n)$ can be found using the subgradient optimization technique. The general framework for our subgradient optimization follows that of section 5.3.4. Initially, multipliers $\lambda^{(0)}$, where $\lambda^{(0)}_i=1/n$ for each i, are used. Thereafter, at the completion of iteration k-1 of the method for which $\lambda^{(k-1)}$ is the vector of multipliers, the value $L(\lambda^{(k-1)})$ of a known feasible solution is obtained. Let $f_i=C^A_i + r_i + b_i - C^B_i$, $f=(f_1,\ldots,f_n)$. The updated multipliers $\lambda^{(k)}_i$ $(i=1,\ldots,n)$ are computed using

$$\lambda^{(k)}_i = \lambda^{(k-1)}_i + \frac{h^{(k-1)}(UB - L(\lambda^{(k-1)}))\, f^{(k-1)}_i}{\Sigma^n_{j=1} (f^{(k-1)}_j)^2}$$

where $h^{(k-1)}$ is the step length at iteration k-1 (initial step length $h^{(0)}= 2$) and UB is an upper bound, given in section 7.2. If $\lambda_i < 0$ $(i=1,\ldots,n)$, we set $\lambda_i=0$ and scale the multipliers (i.e., $\lambda_i = \lambda_i/\Sigma^n_{j=1} \lambda_j$).

7.5. Dynamic Programming Dominance

Dominance rules usually specify whether a node can be eliminated before its lower bound is calculated. Clearly, dominance rules are particularly useful when a node can be eliminated which has a lower bound that is less than the optimal solution.

Prior to applying the branching rule at the second level of the search tree or below, the dynamic programming dominance theorem is applied in which an initial partial sequence σ containing r (≥ 2) jobs, and the initial partial sequence π obtained by interchanging the last two jobs of σ are compared.

Let $C_i^A(\sigma)$ the completion time of job i of the partial sequence σ on machine A, and $h_i = C_i^A + t_i$ is the arrival time at machine B for each job i of σ. Let $T = C^A(\sigma) + \min_{i \in S} (a_i + t_i)$ (where S is the set of unsequenced jobs) be the minimum time at which an unsequenced job $i \in S$ is ready to start processing on machine B. Suppose the jobs of σ and π are sequenced according to their arrival times h_i, such that the resulting sequences are $\sigma' = (\sigma'(1),\ldots,\sigma'(r))$ and $\pi' = (\pi'(1),\ldots,\pi'(r))$. Let $C_i^B(\sigma)$ the completion time of job i on machine B.

We are interested in idle time on machine B which occurs after time T. For instance job $\sigma'(i)$ $(i=1,\ldots,r-1)$ for which $C^B_{\sigma'(i)} \geq T$, contributes an idle time $\max\{h_{\sigma'(i+1)} - C^B_{\sigma'(i)}, 0\}$. For any time $t \geq T$ the idle time number for σ, $I_\sigma(t)$, is the total time that machine B is idle, during the interval $[T,t]$, when job $\sigma'(i)$ $(i=1,\ldots,r)$ is scheduled to be completed on machine B at time $C^B_{\sigma'(i)}$. The idle time number $I_\pi(t)$ for π is defined similarly.

Theorem 7.4. π dominates σ, if $I_\pi(t) \leq I_\sigma(t)$ for $t \geq T$.

Proof. We show that the maximum completion time resulting from the sequence $\pi\gamma$ on machine A is not greater than the maximum completion time resulting from the sequence $\sigma\gamma$ on machine A. Choose job k from γ so that its arrival time on machine B is as small as possible. Consider the ordering, on machine B, obtained by sequencing the jobs in non-decreasing order of the arrival times resulting from the sequence $\sigma\gamma$. In this ordering, the jobs of σ are sequenced according to σ. Suppose that job k is scheduled to be processed in some interval $[t_1, t_1+b_k]$, where, by the definition of T, $t_1 \geq T$.

For the sequence $\pi\gamma$ on machine A, consider the partial schedule in which each job i of π has completion time $C^B_{\pi(i)}$. Insert job k into this schedule so that k is processed during the interval $[t_2, t_2+b_k]$ where t_2 is chosen as small as possible so that $t_2 \geq t_1$ and so that the machine is idle at time t_2 or a job is completed at time t_2 (i.e., t_2 is the first point in time, at or after time t_1, when the machine is not processing a job). Consider the idle time numbers for these new partial schedules resulting from inserting k on machine B. For the schedule based on σ the idle time numbers are given by $I'_\sigma(t) = I_\sigma(t)$ for $t \leq t_1$, and

$$I'_\sigma(t) = \max \{I_\sigma(t) - b_k, I_\sigma(t_1)\} \text{ for } t > t_1.$$

For the schedule based on π the idle time numbers are given by $I'_\pi(t) = I_\pi(t)$ for $t \leq t_2$, and

$$I'_{\pi}(t) = \max \{I_{\pi}(t) - b_k,\ I_{\pi}(t_2)\} \text{ for } t > t_2.$$

Since, during the interval $[t_1, t_2]$ there is no machine idle time, even when k is not included, we have

$$I_{\pi}(t) = I_{\pi}(t_1) \text{ for } t_1 \leq t \leq t_2. \text{ Thus,}$$

$$I'_{\pi}(t) = \max \{I_{\pi}(t) - b_k\ ,\ I_{\pi}(t_1)\}.$$

Since $I_{\pi}(t) \leq I_{\sigma}(t)$ for all $t \geq T$, it follows that $I'_{\pi}(t) \leq I'_{\sigma}(t)$ for all $t \geq T$. Repeating this argument, we obtain a schedule on machine B for the sequence σ_T in which the jobs are sequenced in order of arrival having idle time numbers $I^*_{\sigma}(t)$ where $t \geq T$. The schedule produced from π_T has idle time numbers $I^*_{\pi}(t)$ where $I^*_{\pi}(t) \leq I^*_{\sigma}(t)$ for $t \geq T$. Let $C^B_{max}(\sigma_T)$ and $C^B_{max}(\pi_T)$ be the completion times of these schedules. If $C^B_{max}(\pi_T) > C^B_{max}(\sigma_T)$, then,

for $t = C^B_{max}(\pi_T) - 1$, $I_{\pi}(t) > I_{\sigma}(t)$ which is a contradiction.

Therefore, $C^B_{max}(\pi_T) \leq C^B_{max}(\sigma_T)$ and π_T dominates σ_T. □

The example below shows the importance of Theorem (7.4)

Example 7.4. Consider 7-jobs problem

Table 7.4. Data for example 7.4

i	1	2	3	4	5	6	7
a_i	1	3	4	7	6	2	5
t_i	30	5	12	10	10	5	6
b_i	4	8	3	5	1	2	1

Let σ =(1,2,3,4), π =(1,2,4,3), S ={5,6,7}. $T = C^A(\sigma) + \min_{i \in S}\{a_i + t_i\} = 15+7 = 22$. Sequence the jobs according to their arrival times h_i on machine B, hence $\bar{\sigma}$ =(2,3,4,1), $\bar{\pi}$ =(2,4,3,1).

i	2	3	4	1
$\bar{h}_i$	9	20	25	31
$C^B_{\sigma}(i)$	17	23	30	35

i	2	4	3	1
$\bar{h}_i$	9	21	27	31
$C^B_{\pi}(i)$	17	26	30	35

We see that $I_{\sigma}(t)=0$ for t=22,23, $I_{\sigma}(t)=1$ for t=24, $I_{\sigma}(t)=2$ for t=25,...,30, $I_{\sigma}(t)=3$ for t=31,...,35 and $I_{\sigma}(t)=t-35$ for $t \geq 36$. Similarly $I_{\pi}(t)=0$ for t=22,...,26, $I_{\pi}(t)=1$ for t=27,...,30, $I_{\pi}(t)=2$ for t=31,...,35 and $I_{\pi}(t)=t-35$ for $t \geq 36$. Clearly, $I_{\sigma}(t) \geq I_{\pi}(t)$ for $t \geq T$.

Thus, the conditions of Theorem (7.4) are satisfied. Consequently σ is dominated.

7.6 The Algorithms

In this section we shall give a complete description of the two algorithms which are used to solve this problem. The general form of these algorithms consists of the heuristic method, dynamic programming dominance, the single and two-machine bounds, or the

single and Lagrangean relaxation bound. As is often the case in flow shop scheduling, if $\sum_{i=1}^{n} a_i > \sum_{i=1}^{n} b_i$, the inverse problem is solved instead of the original problem.

7.6.1 Algorithm(1)

Algorithm(1) starts by applying the heuristic method given in section 7.2 to obtain a sequence, which gives us the completion time of each job in this sequence. The value of the completion time of the last job in this sequence forms an initial upper bound UB on C^B_{max}. We also compute at the top of the search tree the set of jobs S^* given in section 7.3.2.

The branch and bound procedure is then started. For each node the dynamic programming dominance (Theorem (7.4)) is used (at the second level of the search tree or below). The lower bounding procedure is then applied (assuming that the node is not eliminated) as follows. First we find LB_B given in section 7.3.1, and $LB_1(S^*)$ given in section 7.3.2. Having found $LB = \max \{LB_B, LB_1(S^*)\}$, for each immediate successor of the node from which we are branching, the minimum lower bound is then found. If it is not less than the current upper bound, this node is eliminated. Otherwise it is selected for our next branching.

The branch and bound procedure continues in a similar way. Whenever a complete sequence is obtained, this sequence is evaluated and the upper bound is altered if the new value is less than the old one.

7.6.2. Algorithm(2)

Algorithm(2) is the same as algorithm(1) apart from the lower bounds used. At the top of the search tree, we compare $LB_1(S^*)$ with $L(\lambda)$, where λ is obtained by performing 100 subgradient optimization iterations, and select the better bound. If $LB_1(S^*)$ is better we use the lower bound of algorithm(1) above henceforth. Otherwise, $L(\lambda)$ given in section 7.4 is used henceforth where λ is found using a heuristic described below.

When $L(\lambda)$ is used, we construct two values of λ from the multipliers λ^0 used at the parent node and select the one which gives the better lower bound. Our first value is $\lambda^1 = \lambda^0$. To construct λ^2, the second value of λ, assume that job j is the last sequenced job. We set $\lambda^2_j = 0$ and $\lambda^2_i = \lambda^0_i / \sum_{k \neq j} \lambda^0_k$ for i=1,...,n; $i \neq j$. The lower bound used is LB=max{ $L(\lambda^1)$, $L(\lambda^2)$} and we store λ^1 or λ^2 according to which gives the better bound.

7.7. Computational Experience

It is well-known that the number of jobs and any correlation between the two processing times for each job are likely to affect the efficiency of a branch and bound algorithm. Problems with 10, 15,20,25, and 30 jobs are generated as follows. For each value of n, uncorrelated problems with integers processing times a_i and b_i (i=1,...,n) were generated from the uniform distribution [1,100], and correlated problems with integers processing times a_i and b_i (i=1,...,n) were generated from the uniform distribution [1 + $20e_i$, 20 + $20e_i$] for e_i randomly selected from {1,2,3,4,5}. This method of processing time generation follows that given in [36]. The transportation times t_{ij} for the uncorrelated and correlated problems

were generated from the uniform distribution [1, R] for R selected from {100,250,500, 1000,1500,2000,2500,3000}. For each value of n we have 5 uncorrelated problems, and 5 correlated problems for each value of R. This yields 40 uncorrelated, and 40 correlated problems for each value of n.

The two algorithms given in section 7.6 were coded in FORTRAN IV and run on a CDC 7600 computer. Whenever a problem was not solved within the time limit of 60 seconds, computation was abandoned for that problem. The algorithms were not tested on the problems with 25 and 30 jobs having correlated processing time because of the discouraging results obtained for n= 20. Computational results for the uncorrelated and correlated problems are given in Tables 7.5 and 7.6 respectively.

For each algorithm, Tables 7.5 and 7.6 show average computation times, number of unsolved problems and the numbers of solved problems that require not more than 250 nodes, that require over 250 and not more than 1000 nodes and that require over 1000 nodes.

The results given in Tables 7.5 and 7.6 for problems with uncorrelated and correlated processing times indicate the weakness of the lower bounds. The large number of unsolved problems for fairly small n (n=15 and n=20) shows that introducing transportation times into a flow shop greatly increases the problem difficulty. An analysis of unsolved problems indicates that those with small and large R are relatively easy whereas those with R=500,1000, 1500, and 2000 are the hardest.

The other factors that are likely to affect the efficiency of a branch and bound algorithm are the dynamic programming dominance (Theorem (7.4)). We also tested the algorithms but with the dynamic

programming dominance omitted. The results show a substantial increase in the number of nodes when the dominance check is not used, although average computation times are comparable. These results are not very surprising since it is computationally time consuming to test the conditions for dynamic programming dominance.

The results given in Tables 7.5 and 7.6 for problems with uncorrelated and correlated processing times are different. These correlated problems are clearly very much harder than the uncorrelated problems.

There is no evidence from our results that one algorithm is clearly superior to the other. Overall, algorithm(1) performs slightly better than algorithm(2) but differences in performance are very small.

Table 7.5. Computational results for problems with uncorrelated processing time

	Algorithm(1)					Algorithm(2)				
N	ACT	NPS1	NPS2	NPS3	NU	ACT	NPS1	NPS2	NPS3	NU
10	0.17	26	9	5	0	0.20	26	9	5	0
15	13.32	22	1	9	8	13.39	22	1	9	8
20	26.68	12	3	8	17	26.79	12	3	8	17
25	30.95	13	2	6	19	31.93	13	2	6	19
30	40.84	11	0	2	27	40.86	11	0	2	27

Table 7.6. Computational results for problem with correlated processing time

	Algorithm(1)					Algorithm(2)				
N	ACT	NPS1	NPS2	NPS3	NU	ACT	NPS1	NPS2	NPS3	NU
10	4.83	16	2	22	0	5.77	16	2	22	0
15	38.76	8	2	6	24	40.45	8	2	5	25
20	55.03	1	0	3	36	56.05	1	0	3	36

Variables at Tables 7.5 and 7.6

ACT : Average completion time in seconds for the 40 problems (included the 60 seconds for the unsolved problems)

NPS1: Number of problems solved that require not more than 250 nodes.

NPS2: Number of problems solved that require over 250 nodes and not more than 1000 nodes.

NPS3: Number of problems solved that require over 1000 nodes.

NU : Number of unsolved problems.

7.8. Concluding Remarks

Both of our algorithms are satisfactory for solving small sized problems, and their results for our test problems are nearly the same. However, a sharper lower bound is needed to cut down the size of the search tree when the number of jobs exceeds 10. The computational results indicate that, for both algorithms, problems with small and large range of transportation times having up to 30 jobs can usually be solved quite rapidly, although excessive computation times will occasionally arise.

Originally we included LB_A as a lower bound, together with LB_B and $LB_1(S^*)$, but the computational results showed that exclusion of LB_A did not affect overall the lower bound. In addition, further improvements in the lower bound have been obtained through the use of the set S^* to modify the lower bound $LB_1(S_m)$. Moreover the Lagrangean relaxation was used in an attempt to improve the algorithms. In spite of all of the improvements, unfortunately the lower bound still remains weak.

The computational results for this problem have also shown the importance of dynamic programming dominance within a branch and bound scheme even if this only involves the two most recently added jobs.

Lower bound for the m-machine general flow shop problem can be obtained by relaxing the capacity constraints on all except two machines. The lower bound for the resulting two-machine subproblems is obtained using the procedure of section 7.4. The difficulty of the $F2/t_i/C_{max}$ problem, that we found supports the view that the general flow shop is extremely hard to solve.

CHAPTER EIGHT

CONCLUSIONS AND REMARKS

In this thesis, we have tried to present some interesting results for selected machine scheduling problems. We have looked at the following problems. Firstly, the $1/r_i/\Sigma w_i U_i$ problem appeared to have a structure which would lead to fairly efficient algorithm. Furthermore, no algorithm for this problem has previously appeared in the literature. Secondly, we have looked at the $1/\ /\Sigma(h_i E_i + w_i T_i)$ problem, because it is interesting to discover whether objectives that are not non-decreasing functions of completion times are substantially harder than those which are non-decreasing. Our results indicate that problems with non-decreasing objective functions are slightly easier. Thirdly, we looked at the $1/\ /\Sigma w_i T_i$ problem because of its practical importance and the challenging nature of the problem, which has a long history. Lastly, we looked at the $F2/L_i/C_{max}$ problem, because this problem is a subproblem of the general flow shop problem and the general flow shop is special case of the general job shop. Hence good lower bounds for our problem could be used for the general flow shop problem. On the other hand if $F2/L_i/C_{max}$ problem is difficult, it confirms that the general flow shop and job shop are very difficult.

We have discussed and compared existing and new branch and bound algorithms for these problems. The efficiency of these algorithms depends on the lower bounds, which are obtained by using the following techniques: relaxation of constraints; Lagrangean relaxation of constraints; dynamic programming state-space

relaxation; relaxation of objective and linear programming theory. Our results for 1/ /$\Sigma w_i T_i$ problem show that the non-iterative heuristic method is computationally more effective than the subgradient optimization method for deriving lower bound. Also, our results for the $\Sigma(h_i E_i + w_i T_i)$ problem show that branch and bound algorithms that employ lower bounds obtained from the dynamic programming state-space relaxation method provide satisfactory results. It seems likely that this idea of computing lower bounds by using dynamic programming state-space relaxation could be effectively applied to other machine scheduling problems such as problem of minimizing total weighted tardiness in a two-machine flow shop. In view of the lack of any algorithm in this area, this approach seems an interesting topic for future research. Lastly, our results for the $1/r_i/\Sigma w_i U_i$ problem show that lower bound can be obtained by applying a heuristic to the dual of a linear programming relaxation of the problem. Again, this method could benefit from further development. Also, dominance rules are very important in restricting the search tree particularly if the lower bound used is not strong. Clearly, new dominance rules and sharper lower bounds are needed to cut down the size of the search tree if large problems are to be solved.

It is interesting to discuss the success of the algorithms presented in the previous chapters relative to branch and bound algorithms for other NP-hard scheduling problems. For selected problems, Table 8.1 below shows the size of problem that can be solved in a reasonable amount of computation time.

Table 8.1

Range of n	Problem	Reference
	$1/r_i/\Sigma w_i U_i$	Chapter four
$n \leq 20$	$F2/t_i/C_{max}$	Chapter seven
	$Fm/\ /C_{max}$	Potts [75]
	$1/\ /\Sigma w_i T_i$	Chapter six
$20 < n \leq 50$	$1/\ /\Sigma(h_i E_i + w_i T_i)$	Chapter five
	$1/r_i/\Sigma w_i C_i$	Hariri and Potts [35]
	$1/r_i/\Sigma U_i$	Chapter four
$50 < n \leq 100$	$1/\ /\Sigma T_i$	Potts and Van Wassenhove [81]
	$1/Prec/\Sigma w_i C_i$	Potts [74]
$n > 100$	$1/r_i/L_{max}$	Carlier [11]
	$1/\ /\Sigma w_i U_i$	Potts and Van Wassenhove [79]

We observe from Table 8.1 that algorithms for $1/r_i/\Sigma U_i$ problem are fairly successful although, in contrast, the $1/r_i/\Sigma w_i U_i$ problem appears very difficult. The $1/\ /\Sigma w_i T_i$ and $1/\ /\Sigma(h_i E_i + w_i T_i)$ problems appear challenging although they are easier than the problems appearing in the first three lines of Table 8.1. It is noticeable that there are very few NP-hard scheduling problems which can be routinely solved for more than 100 jobs.

In conclusion, branch and bound algorithms are moderately successful in solving many scheduling problems. The main reason for

their present success seems to be the simplicity of the basic principles, combined with easy implementation. Prospects for the future there are fairly optimistic: computer hardware is becoming cheaper and more effective and new lower bounding techniques and dominance rules are being developed. The stage is slowly being reached where branch and bound algorithms are sufficiently effective that they can be applied to problems of practical size. However, it appears that much research remains to be done before this stage is reached.

APPENDIX

In this appendix, the proof of optimality for algorithm LP to problem (P) is given:

Let $k_1,\ldots,k_r$ be the indices for which Step(3) of algorithm LP is executed. Let $K=\{k_1,\ldots,k_r\}$ be the set of indices which is used to define the solution of problem (P) using the following result:

Theorem. Problem (P) is solved by setting $z_i=z_i^*$ for $i=1,\ldots,n$ where

$$z_i^*=b_{i+1}z_{i+1}^*/b_i \qquad i \notin K,\ i\neq n \qquad (1)$$

$$z_i^*=c_i \qquad i \in K \qquad (2)$$

$$z_n^*=0 \qquad \text{if } n \notin K \qquad (3)$$

Proof. Consider any optimal solution $z_i=z_i^*$ for $i=1,\ldots,n$ to problem (P), z_i are chosen so that $\Sigma_{i=1}^n z_i$ is as small as possible.

We first prove that $b_1z_1^*=b_{i+1}z_{i+1}^*$ for $i=1,\ldots,k_1-1$. Suppose that $b_jz_j^* \neq b_{j+1}z_{j+1}^*$ for $j=1,\ldots,k_1-1$, choose j is as small as possible such that $b_jz_j^* \neq b_{j+1}z_{j+1}^*$. Let $b_1z_1^*=\ldots=b_jz_j^*=b_{j+1}z_{j+1}^*+\alpha$, where $\alpha>0$, we have

$$\Sigma_{i=1}^n a_iz_i^*=\Sigma_{i=1}^j a_i\,(b_{j+1}z_{j+1}^*+\alpha)/b_i+\Sigma_{i=j+1}^n a_iz_i^*$$

$$<\Sigma_{i=1}^j (a_i/b_i)(b_{j+1}z_{j+1}^*)+\Sigma_{i=j+1}^n a_iz_i^*.$$

Setting

$$z_i' = \begin{cases} (b_{j+1}/b_j)\, z_{j+1}^* & i=1,\ldots,j \\ z_i^* & i=j+1,\ldots,n. \end{cases}$$

We have $\Sigma_{i=1}^n a_i z_i^* \leq \Sigma_{i=1}^n a_i z_i'$. Since $z_i = z_i'$ $(i=1,\ldots,n)$ is a feasible solution to problem (P), it is also an optimal solution, because $z_i = z_i^*$ for $i=1,\ldots,n$ is an optimal solution. Furthermore $\Sigma_{i=1}^n z_i^* > \Sigma_{i=1}^n z_i'$ contradicting the assumption that the original solution has minimal $\Sigma_{i=1}^n z_i$. Therefore, $z_i^* = b_{i+1} z_{i+1}^* / b_i$ for $i=1,\ldots,k_1-1$ and consequently, $z_i^* = (1/b_i)\, z_{k_1}^* b_{k_1}$ for $i=1,\ldots,k_1$.

We next show, by contradiction that $z_\ell^* = c_\ell$ where $\ell = k_1$. Suppose $z_\ell^* < c_\ell'$. Then

$$\Sigma_{i=1}^n a_i z_i^* = \Sigma_{i=1}^{\ell} (a_i/b_i)\, b_\ell z_\ell^* + \Sigma_{i=\ell+1}^n a_i z_i^*$$

$$< \Sigma_{i=1}^{\ell} (a_i/b_i)\, b_\ell\, c_\ell' + \Sigma_{i=\ell+1}^n a_i z_i^*.$$

The solution,

$$z_i = \begin{cases} (b_\ell/b_i)\, c_\ell' & i=1,\ldots,\ell \\ z_i^* & i=\ell+1,\ldots,n \end{cases}$$

is feasible since $b_\ell c_\ell' \geq b_{\ell+1} z_{\ell+1}^*$ (This is true because, $b_{\ell+1} z_{\ell+1}^* \leq b_{\ell+1} c_{\ell+1}' \leq b_\ell c_\ell'$). Furthermore, this solution has a larger objective value. Therefore $z_\ell^* = c_\ell'$.

Since our choice of $z_1,\ldots,z_\ell$ does not restrict the choice of z_i for $i=\ell+1,\ldots,n$, the argument is repeated, thus establishing (1) and

(2).

To complete the proof, we need to establish (3). If $k_r \neq n$, we can prove that $z_i^* = z_n^* b_n / b_i$ for $i = k_r+1, \ldots, n$ using the argument presented in the first part of the proof. Then, we have

$$\Sigma_{i=1}^{n} a_i z_i^* = \Sigma_{i=1}^{k_r} a_i z_i^* + \Sigma_{i=k_r+1}^{n} (a_i / b_i) b_n z_n^*.$$

Now, since $\Sigma_{i=k_r+1}^{n} (a_i / b_i) b_n \leq 0$, $\Sigma_{i=1}^{n} a_i z_i^*$ achieves its maximum value when $z_n^* = 0$. □

REFERENCES

[1] T.S. Abdul-Razaq and C.N. Potts, Dynamic programming state-space relaxation for single machine scheduling, J. Oper. Res. Soc. (to appear).

[2] T.S. Abdul-Razaq, C.N. Potts and L.N. Van Wassenhove, A computational comparison of algorithms for the single machine total weighted tardiness scheduling problem, (to appear).

[3] K.R. Baker, Introduction to sequencing and scheduling, John Wiley & Sons, (1974).

[4] K.R. Baker, Sequencing with due-dates and early start to minimize maximum tardiness, Naval Res. Logist. Quart. 21 (1974) 171-176.

[5] K.R. Baker, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints, Oper. Res. 24 (1983) 381-387.

[6] R. Bellman, A.O. Esobue and I. Nabesima, Mathematical aspects of scheduling & application, Pergamon press, Oxford (1982).

[7] H. Belouadah, Scheduling, to minimize total cost, Thesis, University of Keele, (1985).

[8] J.Blazewicz, J.k. Lenstra and A.H.G. Rinnooy Kan, Scheduling subject to resource constraints: classification and complexity, Discrete App. Math. 5 (1983) 11-24.

[9] E.H. Bowman, The scheduling-sequencing problem, Oper. Res. 7 (1959) 621-624.

[10] A.P.G. Brown and Z.A. Lominicki, Some applications of the branch and bound algorithm to machine scheduling problems, Oper. Res. Quart. 17 (1966)173-186.

[11] J. Carlier, The one-machine sequencing problem, European J. Oper. Res. 11 (1982) 42-47.

[12] N. Christofides, A. Mingozzi, P. Toth and C. Sandi, Combinatorial optimization, John wiley & Sons, Chichester (1979).

[13] N. Christofides, A. Mingozzi and P. Toth, State-space relaxation procedures for the computation of bounds to routing problems, Networks 11 (1981) 145-164.

[14] R.W. Conway, W.L. Maxwell and L.W. Miller, Theory of scheduling, Addison Wesley, Reading, MA, (1967).

[15] S.A. Cook, The complexity of theorem-proving procedures, Proc. 3rd Annual ACM Symp. Theory Comput. (1971) 151-158.

[16] G.B. Dantzig, A machine-job scheduling model, Management Sci. 6 (1960) 191-196.

[17] M.I. Dessouky, and C.R. Margenthaler, The one-machine sequencing problem with early starts and due dates, AIIE Trans. 4 (1972) 214-222.

[18] M.I. Dessouky, and J.S. Deogun, Sequencing jobs with unequal ready times to minimize mean flow time, SIAM J. Comput. 10 (1981) 192-202.

[19] W.L. Eastman, A solution to the travelling salesman problem, Econometrica 27 (1959)282.

[20] W.L. Eastman, S. Even, and I.M. Isaacs, Bounds for the optimal scheduling of n jobs on m processors, Management Sci. 11 (1964) 268-279.

[21] S.E. Elmaghraby, The one-machine sequencing problem with delay costs, J. Indust. Eng. 19 (1968) 105-108.

[22] H. Emmons, One-machine sequencing to minimize certain functions of job tardiness, Oper. Res. 17 (1969) 701-715.

[23] H. Emmons, One machine sequencing to minimize mean flow time with minimum number tardy, Naval Res. Logist. Quart. 22 (1975) 585-592.

[24] M.L. Fisher, A dual algorithm for the one-machine scheduling problem, Math. Programming 11 (1976) 229-251.

[25] M.L. Fisher, The Lagrangian relaxation method for solving integer programming problems, Management Sci. 27 (1981) 1-18.

[26] S. French, Sequencing and scheduling : An introduction to the mathematics of the job-shop, John Wiley & Sons, New York (1982).

[27] M.R. Garey, R.L. Graham, and D.S. Johnson, Performance guarantees for scheduling algorithms, Oper. Res. 26 (1978)3-21.

[28] A.M. Geoffrion, Lagrangean relaxation for integer programming, Math. Programming Study 2 (1974) 82-114.

[29] P.E. Gill, W. Murray and M.H. Wright, Practical optimization, Academic Press, London, (1981).

[30] J. Grabowski, On two-machine scheduling with release and due dates to minimize maximum lateness, Opsearch 17 (1980) 133-154.

[31] R.L. Graham, Bounds for certain multiprocessing anomalies, Bell Sys. Tech. J. 45 (1966) 1563-1581.

[32] R.L. Graham, Bounds on multiprocessing anomalies, SIAM J. Appl. Math. 17 (1969) 416-429.

[33] S. Gupta, and T. Sen, Minimizing the range of lateness on a

single machine, J. Oper. Res. Soc. 35 (1984) 853-857.

[34] A.M.A. Hariri, Scheduling : using branch and bound techniques, Ph.D. thesis, University of Keele (1981).

[35] A.M.A. Hariri, and C.N. Potts, An algorithm for single machine sequencing with release dates to minimize total weighted completion time, Report BW 143/81 Mathematish Centrum, Amsterdam (1981).

[36] A.M.A. Hariri, and C.N. Potts, Algorithms for two-machine flow-shop sequencing with precedence constraints, European J. Oper. Res. 17 (1984) 238-248.

[37] N.A.J. Hastings, Dynamic programming with management applications, Butterworths, London (1973).

[38] M. Held and R.M. Karp, A dynamic programming approach to sequencing problems, J. SIAM 10 (1962) 196-210.

[39] M. Held, P. Wolfe and H.P. Crowder, Validation of subgradient optimization, Math. Programming 6 (1974) 62-88.

[40] W.A. Horn, Some simple scheduling algorithms, Naval Res. Logist. Quart. 21 (1974) 177-185.

[41] E. Ignall and L. Scrage, Application of the branch and bound technique to some flow-shop scheduling problems, Oper. Res. 13 (1965) 400-412.

[42] J.R. Jackson, Scheduling a production line to minimize maximum tardiness, Research, Report 43, Management Science Research Project, University of California, Los Angeles (1955).

[43] S.M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Res. Logist. Quart. 1 (1954) 61-68.

[44] S.M. Johnson, Sequencing n jobs on two machines with arbitrary

time lags, Managment Sci. 5 (1959) 299-303.

[45] E.P.C. Kao and M. Queyranne, On dynamic programming methods for assembly line balancing problems, Oper. Res. 30 (1982) 375-390.

[46] R.M. Karp, Reducibility among combinatorial problems. In: R.E. Miller & J.W. Thatcher (eds.) Complexity of computer computations, Plenum Press, New York (1972) 85-103.

[47] H. Kise, T. Ibarki and H. Mine, A solvable case of the one-machine scheduling problem with ready and due times, Oper. Res. 26 (1978) 121-126.

[48] T. Kurisu, Two-machine scheduling under required precedence among jobs, J. Oper. Res. Soc. Japan 19 (1976) 1-13.

[49] A.H. Land and A.G. Doig, An automatic method for solving discrete programming problems, Econometrica 28 (1960) 497-520.

[50] E.L. Lawler, On scheduling problems with deferral costs, Management Sci. 11 (1964) 280-288.

[51] E.L. Lawler, Sequencing to minimize the weighted number of tardy jobs, Rairo Rech. Oper. 10.5 Suppl. (1976) 27- 33.

[52] E.L. Lawler, A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness, Ann. Discrete. Math. 1 (1977) 331-342.

[53] E.L. Lawler, Efficient implementation of dynamic programming algorithms for sequencing problems, Report BW106, Mathematisch Centrum, Amsterdam (1979).

[54] E.L. Lawler, Recent results in the theory of machine scheduling, Mathematical Programming: The state of the Art, Ed. A. Bachem, M. Grotsthel, B. Korte, Springer-Verlag, Bonn (1982) 203-234.

[55] E.L. Lawler, Scheduling a single machine to minimize the number of late jobs, (to appear).

[56] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, Recent developments in deterministic sequencing and scheduling: A survey. In: M.A.H. Dempster et al. Deterministic and Stochastic Scheduling. Proceding of the NATO Advanced study and research institute on theoritical approaches to scheduling problems, Duram, England (1981) 35-73.

[57] E.L. Lawler and J.M. Moore, A functional equation and its application to resource allocation and sequencing problems Management Sci. 16 (1969) 77-84.

[58] J.K. Lenstra, Sequencing by enumerative methods, Mathematisch Centrum, Amsterdam (1977).

[59] J.K. Lenstra, and A.H.G. Rinnooy Kan, Computational complexity of discrete optimization problems, Ann. Discrete Math. 4 (1979) 121-140.

[60] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker, Complexity of machine scheduling problems, Ann. Discrete Math. 1 (1977) 343-362.

[61] Z.A. Lomnicki, A branch-and-bound algorithm for the exact solution of the three-machine scheduling problem, Oper. Res. Quart. 16 (1965) 89-100.

[62] A.S. Manne, On the job-shop scheduling problem, Oper. Res. 8 (1960) 219-223.

[63] W.L. Maxwell, The scheduling of single machine systems: a review, The International J. prod. Res. 3 (1964) 177-199.

[64] W.L. Maxwell, On sequencing n jobs on one machine to minimize the number of late jobs, Management. Sci. 16

(1970) 295-297.

[65] G.B. McMahon and P.G. Burton, Flow-shop scheduling with the branch-and-bound method, Oper. Res. 15 (1967) 473-481.

[66] G. McMahhon, and M. Florian, on scheduling with ready times and due dates to minimize maximum lateness, Oper. Res. 23 (1975) 475-483.

[67] P. Mellor, A review of job shop scheduling, Oper. Res. Quart. 17 (1966) 161-171.

[68] L.G. Mitten, A scheduling problem, J. Indust. Eng. 10 (1959) 131-135.

[69] L.G. Mitten, Sequencing n jobs on two machines with arbitrary time lags, Management Sci. 5 (1958) 293-298.

[70] C.L. Monma, Sequencing to minimize the maximum job cost, Oper. Res. 28 (1980) 942-951.

[71] J.M. Moore, An n job, one machine sequencing algorithm for minimizing the number of late jobs, Management Sci. 15 (1968) 102-109.

[72] S.S. Panwalkar, Job shop sequencing problem on two machines with time lag constraints, Management Sci. 19 (1973) 1063-1066.

[73] C.N. Potts, The Job-machine scheduling problem, Ph.D. Thesis, University of Birmingham (1974).

[74] C.N. Potts, A Lagrangean based branch and bound algorithm for single machine sequencing with precedence constraints to minimize total weighted completion time, Management Sci. 31 (1985) 1300-1311.

[75] C.N. Potts, An adaptive branching rule for the

permutation flow-shop problem, European J. Oper. Res. 5 (1980)19-25.

[76] C.N. Potts, Analysis of a heuristic for one machine sequencing with release dates and delivery times, Oper. Res. 28 (1980) 1436-1441.

[77] C.N. Potts, Analysis of heuristics for two-machine flow-shop sequencing subject to release dates, Report BW 150/81 Mathematisch Centrum, Amsterdam (1981).

[78] C.N. Potts, and L.N. Van Wassenhove, A decomposition algorithm for the single machine total tardiness problem, Oper. Res. Letters 1 (1982) 177-181.

[79] C.N. Potts, and L.N. Van Wasenhove, Algorithms for scheduling a single machine to minimize the weighted number of late jobs (to appear).

[80] C.N. Potts and L.N. Van Wassenhove, A branch and bound algorithm for the total weighted tardiness problem, Oper. Res. 33 (1985) 363-377.

[81] C.N. Potts and L.N. Van Wassenhove, Dynamic programming and decomposition approaches for the single machine total tardiness problem, European J. Oper. Res. (to appear).

[82] A.H.G Rinnooy Kan, Machine scheduling problems: classification, complexity and computations, Martinus Nijhoff, The Hague, Holland (1976).

[83] A.H.G. Rinnooy Kan, B.J. Lageweg, and J.K. Lenstra, Minimizing total costs in one-machine scheduling, Oper. Res. 23 (1975) 908-927.

[84] M.H. Rothkopf, Scheduling independent tasks on parallel

processors, Management Sci. 12 (1966) 437-447.

[85] M.H. Rothkopf and S.A. Smith, There are no undiscovered priority index sequencing rules for minimizing total delay costs, Oper. Res. 32 (1984) 451-456.

[86] L. Schrage and K.R. Baker, Dynamic programming solution of sequencing problems with precedence constraints, Oper. Res. 26 (1978) 444-449.

[87] T. Sen, and S.K. Gupta, A state-of-art survey of static scheduling research involving due dates, OMEGA The Int. J. Management Sci. 12 (1984) 63-76.

[88] J.G. Shanthikumar, Scheduling n jobs on one machine to minimize the maximum tardiness with minimum number tardy, Comput. & Oper. Res. 10 (1983) 255-266.

[89] J.B. Sidney, An extension of Moore's due date algorithm, Symposium on the theory of scheduling and its applications: In S.R. Elmagehraby, (1973) 393-397.

[90] J.B. Sidney, The two-machine maximum flow time problem with series prallel precedence relations, Oper. Res. 27 (1979) 782-791.

[91] W.E. Smith, Various optimizers for single-stage production, Naval Res. Logist. Quart. 3 (1956) 59-66.

[92] W. Szwarc, Flow shop problems with time lags, Management Sci. 29 (1983) 477-481.

[93] F.J. Villarreal, and R.L. Bulfin, Scheduling a single machine to minimize the weighted number of tardy jobs, AIIE Trans. 15 (1983)337-343.

[94] H.M. Wagner, An intger linear-programming model for machine scheduling, Naval Res. Logist. Quart. 6

(1959) 131-140.

[95] L.J. Wilkerson and J.D. Irwin, An improved algorithm for scheduling independent tasks, AIIE Trans. 3 (1971) 239-245.

[96] T. Yoshida, and K. Hitomi, Optimal two-stage production scheduling with setup times separated, AIIE. Trans. 11 (1979) 261-263.