UNIVERSITY OF SOUTHAMPTON

Hybrid Flow Shop Scheduling with Specialised Machines

Hugo Ranger Mills

Submitted for the degree of Doctor of Philosophy

Department of Mathematics

June 2002

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF MATHEMATICAL STUDIES

DEPARTMENT OF MATHEMATICS

Doctor of Philosophy

Hybrid Flow Shop Scheduling with Specialised Machines

by Hugo Ranger Mills

This study has its roots in the production system of a flour mill, manufacturing specialist flours for cakes, biscuits and batters. The aims are to develop new methods of scheduling, using the problems posed by the industrial system as motivation.

We analyse the industrial system and propose a model for the overall process. This model possesses several features of note – it has multiple processing stages, with parallel machines at each stage; jobs may be restricted in which machines can process them at each stage (*processing set restrictions*); jobs may omit stages; material flows from one stage to the next as a continuous stream; and there is highly flexible inter-process storage of interim products. A simplified model, extending the hybrid flow shop model from the literature, is also presented.

The scheduling of the system is approached in two steps. Firstly, we develop algorithms for solving the scheduling problems at each stage – parallel machines with processing set restrictions. We present efficient and exact algorithms for problems with unit-lengths jobs, across a wide range of the standard regular objective functions. We also suggest three heuristic algorithms for minimising the maximum lateness on identical parallel machines with processing set restrictions and general job lengths: Earliest Due Date (EDD); Jackson for Processing Sets (JPS); and Nested Jackson (NJ).

Secondly, we develop a generalised framework for hierarchical decomposition, aimed at solving hybrid flow shop problems. The framework has four separate major components: decomposition into sub-problems, ordering of sub-problems, sub-problem solution, and backtracking. For each of these we develop several alternative methods. The framework is then tested on a wide variety of problems using computational methods. We conclude that decomposition of the problem by execution sets, solving the sub-problems in stage order with the EDD solution method, and using multiple-pass backtracking produces the best quality solutions.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Celia Glass. Without her guidance and advice, often delivered in time-scales above and beyond the call of duty, I would not have been able to complete this Ph.D. I also thank Professor Chris Potts for being my advisor.

Many thanks are also due to all of the staff and management at Foster Mills – In particular Julian Garratt, Alan Whitcombe, Graham Bruce and David Plumb – for answering endless awkward questions about the mill. Also to Mary Garratt of DFI for starting the management of the project with enthusiasm and understanding, and Dr. Peter Sadd of RHMT for taking up Mary's mantle in similar style.

I must also thank my parents, Patrick and Jan Mills, for giving me encouragement at every stage of the process, and for providing me with a roof over my head for a year to allow me to complete this work.

Finally, thanks go to Jon Whitehead, Ian Woodward, Marcel Marré, Edgar Possani and Phillip Dippnall for their discussions on matters technical and non-technical, and, with David Sim and Jaime Kaminski, for helping keep me more sane than I would otherwise have been over the course of this Ph.D.

Chapter 1

Introduction

1.1 Background

The industrial system on which the work of this Ph.D. is based is a specialist flour mill, called Foster Mills and based in Cambridge, which mills and treats flour for use in industrial processes. The flour made at Foster Mills is used for three main purposes: biscuits, cakes, and batters, although a small amount of bread flour is also made there. To make these non-bread products on an industrial scale, the flour must be treated in various different ways after milling, in a specialist treatment plant. The treatments usually involve altering the biochemical makeup of the flour, for example breaking down starch grains with steam, or altering its acidity or moisture content. The wide variety of products required by different customers for different processes requires a uniquely flexible manufacturing process. However, the flexibility of the process also brings complexity, and the specialist treatment plant poses difficult problems for those running it.

One of the primary problems in the running of the specialist treatment plant is the scheduling of production. The production planner at Foster Mills produces daily a rolling three-day schedule. However, the schedule is updated throughout the day as changed information, both from the sales force and from the production side of the plant, is made available. The generation of the three-day schedule is an extremely taxing task for one person, and would be greatly helped by an algorithm, implemented as a computer program, to produce schedules for the specialist treatment plant, given a list of products which must be produced over the next three days.

1.2 Mathematical scheduling

Mathematical scheduling is a branch of combinatorial optimisation concerned with finding arrangements of tasks on one or more processing units which are optimal according to some predefined objective function.

The theoretical results of the mathematical research are applicable in a wide variety of areas including computing and roster management, but it is most commonly applied in a manufacturing or production context, and it is from manufacturing that the common terminology of the field has been taken. A typical problem will consider *n jobs*, to be scheduled on *m machines*. The machines may be arranged in many different ways, each implying different constraints on how the jobs are to be scheduled. Examples include:

- each job is to be processed on only one of the machines (parallel machines);
- each job is to be processed on all machines in a particular order (flow shop);
- each job is to be processed on all machines in a job-dependent order (job shop);
- or any other combination of parallel or sequential machines.

There is a notation (the three-field notation) which can be used to describe most common mathematical scheduling problems. This notation is described in more detail in the section below.

Research in mathematical scheduling goes back to at least the early 1950s, with the development of rules for producing optimal schedules for simple problems, usually on a single processing unit. For example, the shortest weighted processing time (SWPT) rule of Smith[74] was one such early result. Similar results, such as the SPSF rule presented in this thesis (see §5.4.2) are a common solution method for simple scheduling problems.

However, there were some problems – the flow shop with more than 3 machines being the most celebrated – which did not appear to have simple, easy to compute, solution methods. A simple exact solution to minimise the completion time of the last job (the makespan) for two machines or a special case of three machines had been known since Johnson in 1954[45], but all attempts to find a polynomial-time general solution failed. It was not until the arrival of complexity theory in the early 1970s that the failure of the scheduling community to develop simple algorithms for these problems was understood. The implications of complexity theory meant that there were some problems which appeared to be "difficult" to solve computationally to obtain an exact answer. Some exact techniques, such as branch-and-bound, dynamic programming, and integer programming, were usable, but often only for small problems. This lead to the development of methods intended to find a good answer rather than the best answer. These heuristic algorithms or *heuristics* can take many forms, among them rule-based heuristics; local search methods, including tabu search, genetic algorithms and simulated annealing; hierarchical decomposition; and polynomial time approximation schemes. It is the heuristic methods, and their application to the harder problems of mathematical scheduling, which have been the mainstay of scheduling research in the last thirty years.

1.2.1 Objectives

The great majority of the scheduling literature concentrates on a small set of generally well-behaved objective functions. These functions are all *regular* functions, being non-decreasing. The most common are:

- Maximum completion time, or makespan: the completion time of the last job to be completed. $C_{\max} = \max_{j=1}^{n} C_j$, where C_j is the completion time of job j.
- Total (weighted) completion time: $\sum (w_j)C_j$.
- Maximum lateness: $L_{\max} = \max_{j=1}^{n} L_j$, where $L_j = C_j - d_j$, and d_j is the due date of a job.

- Total (weighted) tardiness: $\sum(w_j)T_j$, where $T_j = \max(0, L_j)$.
- Total (weighted) number of late jobs: $\sum(w_j)U_j$, where $U_j = 1$ if $C_j > d_j$, and $U_j = 0$ otherwise.
- Total (weighted) flow time: $\sum(w_j)F_j$, where $F_j = C_j - r_j$, and r_j is the release date of the job (i.e. the earliest time that it can be scheduled).

The objective functions listed above have a relationships between them – the ability to solve a problem with one objective function may convey the ability to solve the same problem with a different objective function. For example, any unweighted summation objective (e.g. $\sum U_j$) may be reduced to its weighted equivalent (e.g. $\sum w_j U_j$), simply by setting all of the weights to 1. Similarly, a total (weighted) completion time problem may be reduced to a total (weighted) tardiness problem by setting the due dates of all of the jobs to zero.

A maximum lateness ($L_{\max}$) problem may be reduced to either a total tardiness problem ($\sum T_j$) or a number of late jobs problem ($\sum U_j$), by adding a constant, D, to the due date of every job, and then repeatedly solving the $\sum T_j$ (or $\sum U_j$) problem to find the minimum value of D for which $\sum T_j$ (or $\sum U_j$) is zero. Finally, a makespan problem may be reduced to a maximum lateness problem by setting the due date for all jobs to zero.

1.2.2 Three-field notation

One important advance in the field of scheduling is the development of a widely-accepted classification system for problems. This *three-field notation* was proposed by Graham, Lawler, Lenstra and Rinnooy Kan[31]. Their notation captures the general structure and type of a mathematical scheduling problem. The general form of a three-field problem description is $\alpha|\beta|\gamma$, where α specifies the arrangement of machines, β specifies job characteristics, and γ the objective function. The first field, α, is composed of three parts, $\alpha_1\alpha_2\alpha_3$, with the following possible values and meanings:

- α_1 indicates the general arrangement of machines

- α_1 empty: a single machine
- $\alpha_1 = P$: identical parallel machines
- $\alpha_1 = Q$: uniform (or related) parallel machines
- $\alpha_1 = R$: unrelated parallel machines
- $\alpha_1 = F$: flow shop – all jobs have the same fixed sequence of stages to be processed on
- $\alpha_1 = J$: job shop – each job has a fixed sequence of stages in which to be processed
- $\alpha_1 = O$: open shop – each job has a set of stages to be processed on in any order

- α_2 indicates the number of machines (for parallel machines) or number of stages (for sequential machines)
  - α_2 empty: an arbitrary or unknown number of machines or stages
  - $\alpha_2 = 1$: a single machine or stage
  - $\alpha_2 = m$: m machines in parallel (for $\alpha_1 \in \{P, Q, R\}$)
  - $\alpha_2 = s$: s stages (for $\alpha_1 \in \{F, J, O\}$)

- α_3 indicates the number of machines at each stage for multi-stage systems
  - α_3 empty: one machine, or one per stage
  - $\alpha_3 = (Pm)$: m parallel machines at each stage (for $\alpha_1 \in \{F, J, O\}$)
  - $\alpha_3 = (Pm_1, Pm_2, \ldots, Pm_s)$: m_k parallel machines at stage k (for $\alpha_1 \in \{F, J, O\}$)
  - $\alpha_3 = (P)$: arbitrary number of parallel machines at stage k (for $\alpha_1 \in \{F, J, O\}$)

The second field, β, indicates characteristics of the jobs in the problem. There are a large number of possible entries in β. These include the following, arranged by general classification:

- Processing times
  - $p_j = 1$ or $p_{ij} = 1$: all jobs (or operations) have unit processing times
  - $p_j = p$ or $p_{ij} = p$: all jobs have identical non-unit processing times
  - s_j: jobs have set-up times
  - s_{jk}: jobs have sequence-dependent set-up times
- Timing constraints
  - r_j: release dates: job j is not available until time r_j
  - $\bar{d}_j$: deadlines: job j *must* be complete by time $\bar{d}_j$
- Precedence constraints
  - chain: precedence constraints form non-branching chains of jobs
  - intree: precedence constraints form in-trees of jobs
  - outtree: precedence constraints form out-trees of jobs
  - prec: arbitrary precedence constraints exist
- Other characteristics
  - pmtn: operations may be preempted
  - M_j or M_{jk}: each job (or operation) may only be processed on some subset of the available parallel machines (at each stage, if on a multi-stage system)
  - M_j nested or M_{jk} nested: the subsets of machines used by all jobs exhibit a nesting property
  - missing: some jobs may omit some processing stages in multi-stage layouts

It should be noted that some of these characteristics are mutually exclusive (for example, the different forms of precedence), and some are not (release dates and due dates may be used in the same problem, for instance).

The third field specifies the objective function which is to be minimised. Some of the more common objective functions used in the literature are, as listed in §1.2.1:

- $C_{\max}$: maximum completion time, or makespan
- $\sum C_j$: total of the completion times of all jobs
- $\sum L_{\max}$: maximum lateness
- $\sum (w_j)T_j$: total (weighted) tardiness
- $\sum (w_j)U_j$: total (weighted) number of late jobs
- $\sum (w_j)F_j$: total (weighted) flow time

However, many other objective functions have been used, including discounted-cost earliness (related to warehousing costs), and work-in-progress on multi-stage problems.

1.2.3 Complexity theory

Another fundamental advance which has benefited the field of mathematical scheduling is the development of complexity theory. Complexity theory comes from the theory of computation and computability, and deals with the relationship between the time taken to perform a computational process and the size of the input to that process.

Running time

Consider a general deterministic computational device of infinite capacity (such as a Turing machine or Minsky register machine). This machine contains a *program* which implements an algorithm. The machine will accept a sequence of symbols as an input, and may eventually stop, yielding a further sequence of symbols as output. For each input I of length n for which the machine halts, it will have taken a particular number of computational steps, $S(I)$. We say that the implementation of the algorithm *runs in $O(f(n))$ time* if there exists some value κ such that

$$S(I) < \kappa f(n),$$

for all inputs I of length n, and for all values of n. In other words, the function $f(n)$ is an upper bound on the behaviour of the running time of the algorithm as the size of the problem tends towards infinity.

For example, consider an algorithm for performing a linear search on an unsorted set of data for the largest value. To read a value and compare it to the largest currently known value will take no more than some constant (κ) number of operations. This must be repeated for every one of the n values, taking no more than κn operations. Thus, this linear search is an $O(n)$ algorithm. Performing a simple matrix multiplication of two square matrices, using the obvious algorithm, is an $O(n^3)$ process, since n^3 multiplications are required. (In fact, multiplication of two $n \times n$ matrices can be accomplished in approximately $O(n^{2.376})$ operations using an appropriate algorithm [16]).

In the field of combinatorial optimization, algorithms which have polynomial bounds on their computational time are considered "good", and those with larger-than-polynomial (e.g. exponential or larger) bounds are considered "bad". This should be compared to the computing-science viewpoint, where any algorithm with a running time of $O(n^2)$ or worse is generally considered "bad", and only $O(n \log n)$ or better algorithms are "good". It should also be noted that a polynomial time algorithm may not necessarily be useful – an algorithm with running time bounded by $10^{10} n \log n$ seconds is probably going to take too long to run to be practicable, and an algorithm bounded by $10^{-8} 2^{n/1000}$ seconds may well be useful for all real-life problems it is intended to solve. Also, it may be that the upper bound is only ever reached in a very few cases: the simplex method for solving linear programmes has an exponential upper bound on its running time (and problems exhibiting the behaviour can be constructed), but for most problems it behaves well, and takes few steps to complete. Similarly, the $O(n^2)$ worst-case running time of the quicksort algorithm has not prevented computer scientists and software engineers from using it extensively due to its ease of implementation and excellent average-case performance.

P vs NP

If a problem has an algorithm to solve it which has a polynomial running time, then that problem is said to be in the class **P**. However, there are some problems for which there is no known polynomial time solution method. Consider a *non-deterministic* machine, which may have multiple destination states from a given starting state and input – in other words, it has a feature of random behaviour built in to it in some way. It may be possible (although highly unlikely) for such a machine to find a solution in polynomial time to one of these problems, by "guessing" the right answer as the first solution tested. The class of problems for which there is a polynomial time algorithm running on a nondeterministic machine is known as **NP** (for nondeterministic polynomial). Note that problems in **NP** impose certain limits on their computation:

- The size of the solution must be polynomially bounded (else the solution could not be verified in polynomial time), and
- the solution verification procedure must itself be a polynomial time algorithm.

In fact, it is with the latter of these conditions that the more usual definition of **NP** is made. Clearly, all problems in **P** are also in **NP**. The remaining question is:

Are all problems in **NP** also in **P**?

The main result regarding this question was presented by Cook[14], who showed that the SATISFIABILITY problem was at least as hard as any other problem in **NP**. Specifically, he showed that any problem in **NP** can be reduced to SATISFIABILITY using a polynomial-time procedure. The SATISFIABILITY problem was dubbed an **NP**-complete problem. A year later, Karp[47] tidied up the definition of **NP**-complete, and proved for each of 21 other problems that SATISFIABILITY could be reduced to the problem using a polynomial time procedure. This proved that there were many other **NP**-complete problems. Since then, the number of known **NP**-complete problems has grown enormously – Garey and Johnson[25] in 1979 list 320 problems in various categories; many more have been added since. It is still an unsolved problem whether **P** = **NP**. In fact, there is a \$1m prize, posted by the Clay Mathematics Institute[1], for a proof of either **P** = **NP** or of **P** $\neq$ **NP**.

Not all **NP**-complete problems are created equal, however. There are two distinct groups of **NP**-complete problems. There are some problems whose most efficient optimal algorithm is bounded by a polynomial function $f(n, B)$, where n is the number of items in the input, and B is an upper bound on their magnitude. It would appear initially that this problem is in **P**, but this is not the case, for if the n input numbers are encoded in binary, they will require $n \log_2 B$ symbols to represent. Any running time involving a polynomial function of B is greater than polynomial in $\log B$. Problems with this behaviour are referred to as either ***NP*-*complete in the ordinary sense***, or *binary* ***NP*-*complete.*** If the input is encoded in "unary" (where a number of magnitude B is represented with a string of "1"s of length B), then the problem will run in polynomial time compared to its (now extremely inflated) input size. These problems are also sometimes referred to as *pseudo-polynomially solvable* problems, and can often be solved by dynamic programming.

On the other hand, there are problems which retain their **NP**-completeness even when their inputs are written in unary. These *unary* ***NP*-*complete*** or *strongly* ***NP*-*complete*** problems are in some way harder to solve than problems which are **NP**-complete in the ordinary sense. Examples of the two types of problems are the one-machine total tardiness problem, $1||\sum T_j$, which can be solved by a pseudo-polynomial dynamic programming process, as shown by Lawler [50]; and the one-machine total weighted tardiness problem, $1||\sum w_j T_j$, which is strongly **NP**-complete.

1.3 Aims and Objectives

The aim of this Ph.D. is to develop mathematical scheduling algorithms, using the problems presented by the specialist treatment plant at Foster Mills on which to base the work. In more detail, we will:

- identify new scheduling problems from the situation posed by the specialist treatment plant,

- produce a description of the plant which embodies all of the information relevant to developing a mathematical scheduling model,
- develop suitable algorithms for solving those scheduling problems, and
- implement, test and evaluate the algorithms.

The detailed description and analysis of the plant is done in chapters 2 and 3. Chapter 2 describes the features of the plant relevant to its scheduling problems, detailing the processes and machines, and the between-process storage bins, and showing how these components are connected together into an extremely complex industrial processing system. Chapter 3 takes the straightforward description of the plant from chapter 2 and reveals the structure implicit in the system. Although the specialist treatment plant is highly complex and flexible, there are biochemical, process and management reasons why some aspects of that flexibility are not exploited. Chapter 3 presents a series of increasingly simplified models for the machine layout of the plant, and shows how it may be modelled, using the concepts of the mathematical scheduling literature. The specialist treatment plant at Foster Mill is shown to be, broadly speaking, a hybrid flow shop, but with several interesting features most of which are not normally encountered in the mathematical scheduling literature: continuous flow processes, per-job machine processing restrictions (known as processing sets), a tree-structured hierarchy of product classes, and inter-process storage. We develop notation for describing each of these features as it is found in our industrial system.

In chapters 4–7, we develop various algorithms and techniques for solving the mathematical problems identified in chapter 3. Chapter 4 introduces disjunctive directed graphs, a technique widely used for modelling certain types of scheduling problem, and extends the technique so that it can be applied to our problem. Specifically, the disjunctive graph model is often used on problems of scheduling jobs each of which must pass through several machines before completion. The interaction of sequences of machines for each job, and jobs for each machine can be difficult to follow. The disjunctive graph model provides an efficient tool for rapid evaluation of schedules – both for feasibility and timing. We develop forms of disjunctive graph which are suitable for modelling the sort of inter-process storage we are interested in. We also develop timing constraints on scheduling systems with continuous flow processes and inter-process storage. These results are used to develop disjunctive graph models for such systems.

Chapters 5 and 6 develop algorithms for solving the general class of parallel machine problems which are found as sub-problems in the plant. In particular, we are interested in problems of parallel machine scheduling where each job can only be processed on a subset of the machines – the so-called *processing set restriction.* We identify a particularly well-behaved form of processing set restriction where the processing sets used across all jobs exhibit the property of being *nested*. Chapter 5 concentrates on problems where all jobs have identical (unit) processing times. We develop efficient polynomial algorithms for solving scheduling problems with unit-length jobs and processing set restrictions on parallel machines, across a wide variety of regular objective functions.

Chapter 6 examines similar problems to chapter 5, where the jobs do not necessarily have unit length. We concentrate specifically on heuristic algorithms for minimising the maximum lateness objective function, as these algorithms will be used in later chapters of the thesis as building blocks in an algorithm for solving problems on the hybrid flow shop. We propose three algorithms for minimising $L_{\max}$ on identical parallel machines with processing set restrictions.

Having identified the scheduling problem in chapter 3 as a hybrid flow shop, chapter 7 examines the theory of scheduling the hybrid flow shop in more detail. We extend the model by allowing processing set restrictions for operations at each stage. We develop a broad framework within which the hybrid flow shop problems of the type introduced in chapter 3 may be solved. The framework is a hierarchical decomposition method, similar to the shifting bottleneck method used to solve job shop scheduling problems. Our algorithm contains four main components – a decomposition method to break the problem down into sub-problems; a criticality measure to determine the order of solution of the sub-problems; a procedure to solve the sub-problems; and a backtracking method to determine which already-solved sub-problems to revisit in order to improve the solution quality. We describe several different approaches to each of these areas in chapter 7, with the exception of the sub-problem solution procedures, for which we will use the heuristics proposed in chapter 6.

Finally, in chapters 8 and 9 we generate test problem instances, perform computational tests, and analyse the results of those tests. This is to determine which of the components of the framework developed in chapter 7 is most important when generating schedules, and to find which of the components may be used in the framework is most effective. We analysed the behaviour of the algorithm for the $L_{\max}$ objective function, although the framework itself is suitable for any regular objective function. Chapter 8 describes the factors which may be varied in generating hybrid flow shop problems of different shapes and behaviours. It also describes in detail the techniques which we use to generate the test problem instances. Chapter 9 presents the statistical experiment design techniques which we use, and introduces the analysis method for the large data-sets which we generate through running the experiments. We then analyse the results of the experiments to draw conclusions about the performance of the solution framework.

Finally, in chapter 10 we summarise the work of this thesis, and point the direction for future research, both on ways of improving the work in this thesis, and highlighting open problems.

Chapter 2

Description of Industrial Process

In this chapter, we describe the production process at Foster Mills in 1998. The description is based on detailed observation and analysis of the plant, and discussion with the people working in the plant over a period of 9 months in 1998. Since the plant is continually being modified, we chose to work with a "snapshot" of the plant from the second half of 1998.

The primary difficulty encountered in doing the analysis was that the main sources of information – the people working at the plant – generally knew in great detail how the plant operated, in some cases down to individual valves on blowlines. However, they were unable to produce simpler abstractions of the plant which could give an overview of the connections between machines and bins without oversimplifying.

Foster Mills have adapted and used the results of this chapter. The flour recipe diagrams in figure 2.2 are used by the sales, planning and scheduling staff as quick-reference sheets. In addition, the various simplified process diagrams developed in the next chapter from the descriptions in this chapter are used for training new staff.

We cover the processes involved in making specialist flours, and how those processes related to each other with reference to the logistics of the plant. We give details of the scheduling characteristics of the processes. We also cover the inter-process storage bins in the plant, and how they are connected to the machines from a scheduling perspective. An important observation is the continuous-flow nature of the plant. Finally, we describe how interim products of processing may be made into several different products at each stage, leading to an out-tree structure of products.

2.1 Overview

The specialist treatment plant at Foster Mills in Cambridge in 1998 was operated on the basis of both make-on-demand and of make-to-stock. A few of the products are in extremely high demand, and can be made to stock to fill up any spare production capacity, since an order for them is likely to arrive in a short timescale. Most of the product lines made on the plant, however, are quite highly specialised products, sometimes produced for a single customer. There is little point in making these less frequently used products to stock. They are therefore made to order, with a lead time of approximately three days. When an order is placed, the due date is negotiated with the customer – due dates earlier than two days time, or later than 7 days time, are rare, with 3 days being the norm. Scheduling is performed on a rolling three-day horizon, with the next 24 hours being fixed and sent to the production staff. The objectives of the scheduling are to ensure that all orders are filled by their due date. The main key productivity indicator relating to scheduling which is measured at the mill is the number of late jobs.

The plant produced somewhere in the region of 150 to 170 products, processing flour from the mill on the same site. The exact number varied, depending on the product lines required by the mill's customers. Fewer than 10 distinct flours were made on the mill (the *base flours*), from about six different grists. A *grist* is the mixture of different grain types fed into the mill. Of the 150-170 products, about 70 were essentially distinguishable flours, with the remainder of the product types (the generic products) being made of the same 70 flours in different packaging and with different specifications of quality.

In order to make these distinct products, the base flours are passed through a number of different processes. There are 11 processes in all:

Milling is the process of turning a grist into flour. The mill produces many different products and by-products, although only the main product – the flour –

is used in the rest of the plant.

Classification takes a single stream of flour, and uses air blowers to split it into two parts: the *fine fraction* (FF), and the *coarse fraction* (CF). These are, respectively, the small particles and large particles from the flour.

Chlorination places the flour in a chlorine atmosphere for a short time. A small amount of the chlorine remains in the flour, and acts to reduce its pH value (i.e. it increases the acidity). Chlorinated flour was used for making cakes, typically in industrial processes, but has now been banned by EU regulations. Heat treated flours (see below) are now used in place of chlorinated flour.

Grinding is the reduction of the particle size of the flour using grinding rollers similar to those used in the mill.

Steam treatment holds the flour in live, pressurised steam for several minutes. This has two effects: it makes the flour extremely moist, and it breaks down the starch granules in the flour particles. Steamed flours are generally used for making batters.

Agglomeration is a process used to make pseudo-semolina (called *flour cones*). The flour is made wet, and then dried using a hot air blower. The individual flour particles stick together into small grains resembling semolina or couscous.

Drying uses hot air to reduce the water content of the flour, from 12-14% down to 2 or 3% in some cases.

Heat treatment holds the flour at over 100°C for up to an hour. This has much the same effect on the flour as chlorination, and heat-treated flours can be used as substitutes for chlorinated flours in some processes.

Mixing is the process of adding other powdered, particulate or ground materials to the flour. Normally either salt, for some batter mixes, or self-raising mixture (mostly sodium bicarbonate) are added in the mixing process. The machine used for this process is not suitable for blending two separate flours together.

Packing machines put flour into bags of between 10 and 32kg, and pack the bags on wooden palettes, usually in 1-tonne lots. Alternatively, the flour may be packed in a single 750kg or 1 tonne bag, known as an FIBC (*flexible intermediate bulk container*).

Bulk output blows the flour directly into 24-tonne bulk tankers for immediate road transport.

The above processes are used in different combinations with different grists to produce the wide range of final products. Each of the processes has different settings which affect the process and the resultant flour. For example, the angles of the fan blades in the classifier may be altered, changing the mix of CF and FF and altering the particle sizes found in each fraction. Similarly, the steam plants may be set to different temperatures and pressures to make different products; or the dryers may be set to make drier or moister flour. The implications of the different machine settings will be explored in more depth in §2.6.

Not all products need to pass through all of the processes listed above. In fact, most products pass through only a very small number of the available processes. The plant is capable of processing flour through the machines in many different combinations and directions (see §2.4 for details). However, it is a feature of the way that the plant is used that any products which require the same two processes undergo them in the same order. That order is the one listed above. The reasons for this are generally to do with the physics and chemistry of the processes involved. For example, steam treatment after drying would make no sense, as the dried flour would be re-moisturised during the steam process, so dry steamed flours are always steamed first and then dried. The only processes that all products must pass through are the milling process at the beginning, and one or other of the packing and bulk output processes at the end.

2.2 Machines

There are eighteen machines in the specialist treatment plant performing the 11 processes listed in the previous section. Duplication of machines occurs at the steam treatment, drying and packing stages. Most machines have a fixed processing rate, given as the number of tonnes of flour it can process per hour. Some, however, have a product-dependent processing rate. Similarly, most machines do not have a start-up period, but some do. Finally, some products can only be processed on certain machines – the best examples are the micro-clean flours, which can only be processed on steam plant 3 and the micro-clean dryer (WP2). A summary of the machines, processes and processing parameters is given in Table 2.1. We find it conceptually easier to group the machines into *processing stages*, which are also indicated in the table. More details on the processing stages and their role in making a mathematical model of the plant are given in chapter 3. Processes within a stage may be carried out simultaneously – for example, drying and heat treatment of the same flour may be done at the speed of the the heat treatment machine, which is the slower process.

The set-up time for the machines, where it is given, is the time taken for the machine to produce usable flour after it starts to process a batch. The steam plants in particular require anything up to 30 minutes to settle down for each batch of flour. The set-up times are not sequence dependent for any of the machines which require them, although the steam plants have a randomly distributed set-up time typically between 15 and 30 minutes. The set-up time for the bag packer is dependent on the bag size of the final product, as a fixed number of bags is discarded to flush the system of the previous flour. In all cases of set-up periods in the plant, flour is consumed at the normal rate during the set-up process. The output flour made during the set-up is discarded for recycling as lower-grade flour. Thus the production of each tonne of output flour requires more than one tonne of input flour.

The third steam treatment machine, Bok3, the second W-P dryer, WP2, and the second bag packing machine, Pack2, together form the *micro-clean plant* (MCP). This is a separate, slightly pressurised, section of the factory with microbial filters

Machine	Stage	Process	Speed (t/h)	Startup
Mill	(i)	Milling	13.5	-
Classifier	(ii)	Classification	9.0[a]	-
Chlorinator	(ii)	Chlorination	4.5	-
Grinder	(ii)	Grinding	4.5	-
Bok1[b]	(iii)	Steam Treat	2.0	15-30 mins
Bok2	(iii)	Steam Treat	2.0	15-30 mins
Bok3	(iii)	Steam Treat	2.0	15-30 mins
Agglom	(iv)	Agglomeration	1.0	-
TV[c]	(iv)	Drying	1.5	-
WP1[d]	(v)	Drying	2.0	-
WP2	(v)	Drying	1.7	-
Vrieco	(v)	Heat Treat	1.0	-
Batch Mix	(vi)	Mixing	4.0	-
Pack1	(vi)	Packing	varies	6 bags
Pack2	(vi)	Packing	varies	6 bags
FIBC1	(vi)	Packing	varies	-
FIBC2	(vi)	Packing	varies	-
Bulk	(vi)	Bulk Out	480	-

[a]Input stream. Output streams are 4.5tonnes/h each.
[b]Bokfard, or steam treatment plant.
[c]Thermo-Venturi dryer.
[d]Werner-Pflederer dryer.

Table 2.1: Summary of machines

on the air supply. Flour processed in the micro-clean plant is effectively sterilized by the steam treatment, and has a minimal micro-organism count when it is packed. The only way for flour to enter the MCP is through the Bok3 process.

The classifier machine has an interesting set of properties from a scheduling point of view. It produces two separate streams of product from a single input stream – two tonnes of input flour become one tonne each of coarse fraction and fine fraction. One problem encountered at the mill is that there is a large demand for products made from fine fraction, and a much smaller demand for products which require coarse fraction. However, there are some products which can be made with either unclassified flour or with (ground) coarse fraction. These are often made with coarse fraction to use up the surplus. The flours for which (ground) coarse fraction can be used as an alternative base are not shown in the diagram in figure 2.2, as it would make the figure considerably more difficult to read.

2.3 Storage

As the flour passes from one machine to the next, it does not move directly between the machines, but instead is blown into a storage bin before being sent to the second machine. Flour may remain in the storage bin for some time before being moved for processing on the second machine. This is the case for most processes and bins. However, in some instances, machines are also connected together directly, and flour can be passed between them without using a bin. Alternatively, flour may have to pass through two bins to move from one process to another. The details of the connections between bins and processes are given in the next section.

There are nearly 80 bins in the plant. These are of varying sizes, and each bin is connected to a different set of machines. There are some bins which feed only a single machine (or process), and are thus dedicated to that machine. Equally, there are some bins which can be fed from a large range of machines, and can act as general-purpose storage. The names and sizes (in metric tonnes) of the bins are given in table 2.2. The grouping of bin numbers in the table is due to functional differences between the bins, and reflects the layout of the bin stock sheets used for production control at the mill.

There are a few products made in the specialist treatment plant which are made by blending together two flours. The batch mix machinery is unsuitable for this task, and so bins are used to perform the mixing. If the two flours being mixed are blown into the same bin at the same time, they are mixed effectively and thoroughly. This is the only time that a bin is fed from two sources at the same time.

The usage of bins is limited in certain ways. The primary limitations on bin usage are that:

- a bin cannot be used to store two different types of flour at the same time (except when mixing),
- a bin cannot store more flour than its capacity, and
- there are some sets of bins from which two bins cannot feed different machines at the same time.

The latter condition is the most difficult constraint to model and explain in the whole system, but stems from the fact that some bins are linked to the same conveyor or feed line, and in order to feed a given machine, any bin from one of these groups must use that conveyor. This limitation is discussed in more detail in §2.4.2.

2.4 Connections

2.4.1 Blowlines

In order to move the flour between bins and processes, it is blown through pipes using a compressed air system. These *blowlines* are designed to have a greater capacity (i.e. transfer rate) than the processes at either end of the line. This means that the factor limiting the speed of any pair of machines (or machine and bin) is the machines involved and not the blowline. In addition, the transit time from one end of a blowline to the other is negligible. Blowlines may therefore be completely ignored from the point of view of modelling, as they have no appreciable effect on the timing of scheduling of the plant.

The connections between machines are generally mediated by bins. Blowlines and conveyors run from machine to bin and bin to machine, although there are some direct connections (gravity feed, or short dedicated blowlines) between machines. The way in which bins are used is therefore intimately connected to the flow of flour through the plant.

The connections between the processes and bins are extremely complex. One of the biggest challenges in this stage of the research project was identifying the logistics which governed the running of the plant. The plant was designed to

Name	Size (t)	Name	Size (t)	Name	Size (t)
1	15	7	15	10t	10
2	15	8	15	18t	18
3	15	9	15	SRF	1
4	15	10	15	BM	4
5	15	11	15		
6	15	12	15		
13	14	19	18	B1	24
14	14	20	18	B2	24
15	14	21	18	B3	24
16	14	22	18	B4	24
17	14	23	18		
18	14	24	18		
25	30	28	30	MB	9
26	30	29	30		
27	30	30	30		
31	28	42	28	W	23
32	28	43	28	X	23
33	28	44	28	Y	23
34	28	45	28	Z	23
35	28	46	28		
36	28	47	28		
37	28	48	28		
38	28	49	28		
39	28	50	28		
40	28	51	28		
41	28	52	28		
53	20	54	28	MCP1	
55	20	56	28	MCP2	
57	20	58	21	MCP3	
59	18	60	22	MCP4	
61	21	62	21		

Table 2.2: Summary of bins

be highly flexible in its operation, and thus has a vast number of possible paths through the system. The extent of the complexity involved may be appreciated from the original engineering diagram of the plant, which is included in appendix A.

A summary of the connections between bins and processes at the time that we performed our analysis in 1998 is given in table 2.3. This systematisation represents the first step in identifying the actual flow of flour through the machinery, as opposed to the possible flow implicit in the design and engineering of the plant.

Process	Bins feeding process	Bins fed by process
Mill	(Grain bins)	1, 2, 7, 8, 26-34, 38, 42, 44, 48-54, 56, 58, 60-62
Classifier	1, 2, 7, 8, 31, 42	FF: 57, 59, 60, (Chlorinator) CF: 28, 36, 47, 56, (Grinder)
Chlorinator	(Classifier, FF)	13-18, 26, 29, 52, 57-60
Grinder	(Classifier, CF)	37, 48, 53, 54
Bok1	26, 27	3, 4, 9, 10, 53, 61
Bok2	28-30	5, 6, 11, 12, 53, 61
Bok3	51, 52	39-41, 54, 61, MCP1-MCP4, (FIBC1)
Agglom	37, 48, 53, 54, 56-60	(TV)
WP1	37, 48, 53, 54, 56-60	19-24, 35, 55, 57, 60, (Vrieco)
WP2	MCP1, MCP3	MCP2
TV	37, 48, 53, 54, 56-60, (Agglom)	19-24, 35, 46, 55, 57, 60
Vrieco	(WP1)	19-24, 35, 55, 57, 60
Batch Mix	61, 62	(Pack1) (FIBC2)
Pack1	3-6, 9-24, 35, 38-41, (Mixer)	
Pack2	MCP1-MCP4	
FIBC1	3-6, 9-12, 19-24, 32-35, 38, 54, 56, 60-62, (Bok3)	
FIBC2	(Mixer)	
Bulk	3-6, 9-24, 32-35, 44, 46, 48-50, 54-58, 60-62	

Where processes are shown in brackets, there is a direct feed between the processes.

Table 2.3: Bins connected to each process

2.4.2 Conveyors

To reach the blowlines from a bin, flour is let out of the bottom of the bin under gravity. It falls on to a short length of conveyor belt, which feeds the flour into the blowline system. Each conveyor handles the flour for several bins. Some conveyors also handle the flour from other conveyors. The practical result of this system is that two bins which use the same conveyor cannot feed processes at the same time, since their flours would be mixed on the conveyor.

The conveyors thus divide the bins into nested subsets of bins, where all bins inside any given subset are mutually exclusive when feeding flour into blowlines. We did not analyse the subsets in detail, as there are few occasions in practice when the conveyor constraints are encountered.

2.5 Continuous flow

The processes at Foster Mills are all continuous flow processes, where flour from the beginning of a batch is available for further processing (or storage) immediately and does not have to wait for the batch to complete. Thus, if a process runs slower than its predecessor, it may start processing the flour at the same time as its predecessor. If a process runs faster than its predecessor, then they may both finish at the same time (but no earlier). This continuous flow process is important for scheduling the industrial process, as the overlap between consecutive processes can yield a significant decrease in the total processing time of jobs.

2.6 Product differentiation

For many of the processes performed in the specialist treatment plant, one input flour may be converted into many different output flours, depending on the settings of the process. Most of the processes in the plant have several operational parameters which can be changed. For example, temperature and pressure may be changed in the steam plants; fan blade settings on the classifier can be altered to affect the ratio of coarse fraction to fine fraction produced; and the bag packer may be set up with different sized (or labelled) bags to suit different customers' requirements.

We refer to the combination of ingredients and the set of processes and their settings required to make a particular flour as that flour's *recipe*. Few of the machine settings for a recipe affect the timing or the scheduling of the plant. Those few settings that do affect the processing rates of machines can easily be modelled mathematically by using product-dependent processing times. Since we are primarily concerned with scheduling the plant, we do not concern ourselves with the details of product recipes. Machine settings do, however, have implications for the way that we consider products to be organised. The only relevant ingredient for a scheduling point of view is the base flour. We assume that the other ingredients (e.g. salt or self raising mixture) are always available.

As a consequence of having product-dependent machine settings, one base flour may be made into several different flours in its subsequent processing. Thus the flours have an out-tree structure to their recipes. The differentiation may result from different settings on the machine, or from the use of a different machine or process. Thus, product differentiation will not necessarily be apparent from the logistic data we work with regarding processing times. It is therefore necessary to record the hierarchy of flours explicitly. There are some minor exceptions to this out-tree structure, where two products are mixed together in between stages to give a third product.

Determining the product hierarchy was as complex a task as determining the physical structure of the plant, for many of the same reasons. There was a large amount of data available, but this data came from many different systems – variously, computerised, paper-based and kept in the heads of key personnel. It quickly became clear that there were many products which were identical. This was for a number of reasons: overlap between generic product ranges introduced at different times; convergence of requirements from different customers; and sales to niche markets. Determining definitively which products were actually duplicated was a much more difficult proposition. In particular, products would be referred to colloquially by different names depending on where in the plant they were being processed. In some cases, these names bore little or no resemblance to any existing product name. In addition, the official product numbering scheme was difficult to follow, as it originated in several slightly different systems, and gave no concept of the relationships between products. The product hierarchy given in this chapter is the result of a systematisation of the nomenclature, and groups products into *families* of identical flours.

An example of product differentiation is given in figure 2.1. Product names have been removed for commercial reasons. In the example, the base flour from the mill, *ST Base*, can be steam treated in two different ways (at two different temperatures, say) giving two resultant products. Of these, one is simply bagged and sold directly (*ST1*), and the other may be sold directly (as *ST2*) or further dried before being sold (*Dried ST2*). A full diagram of the product hierarchy is given in figure 2.2, again with the actual product names removed.

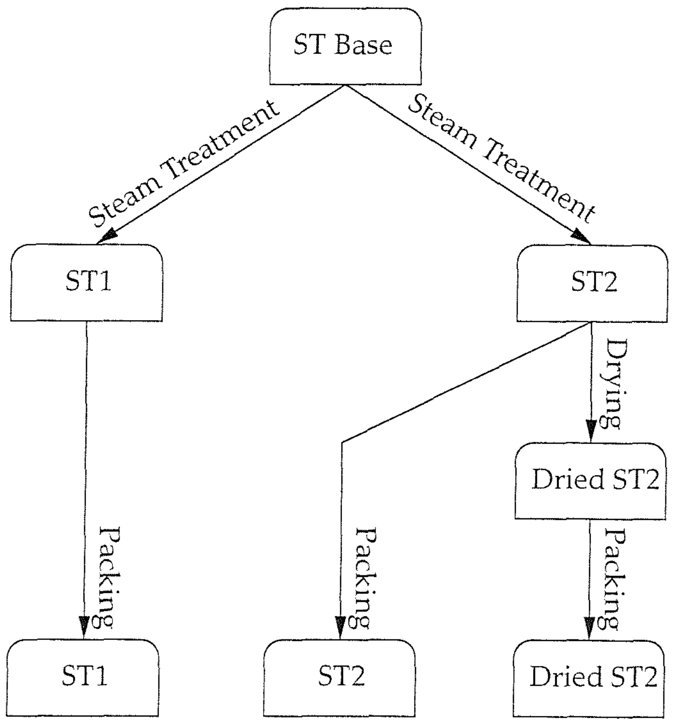


Figure 2.1: Example of part of product hierarchy

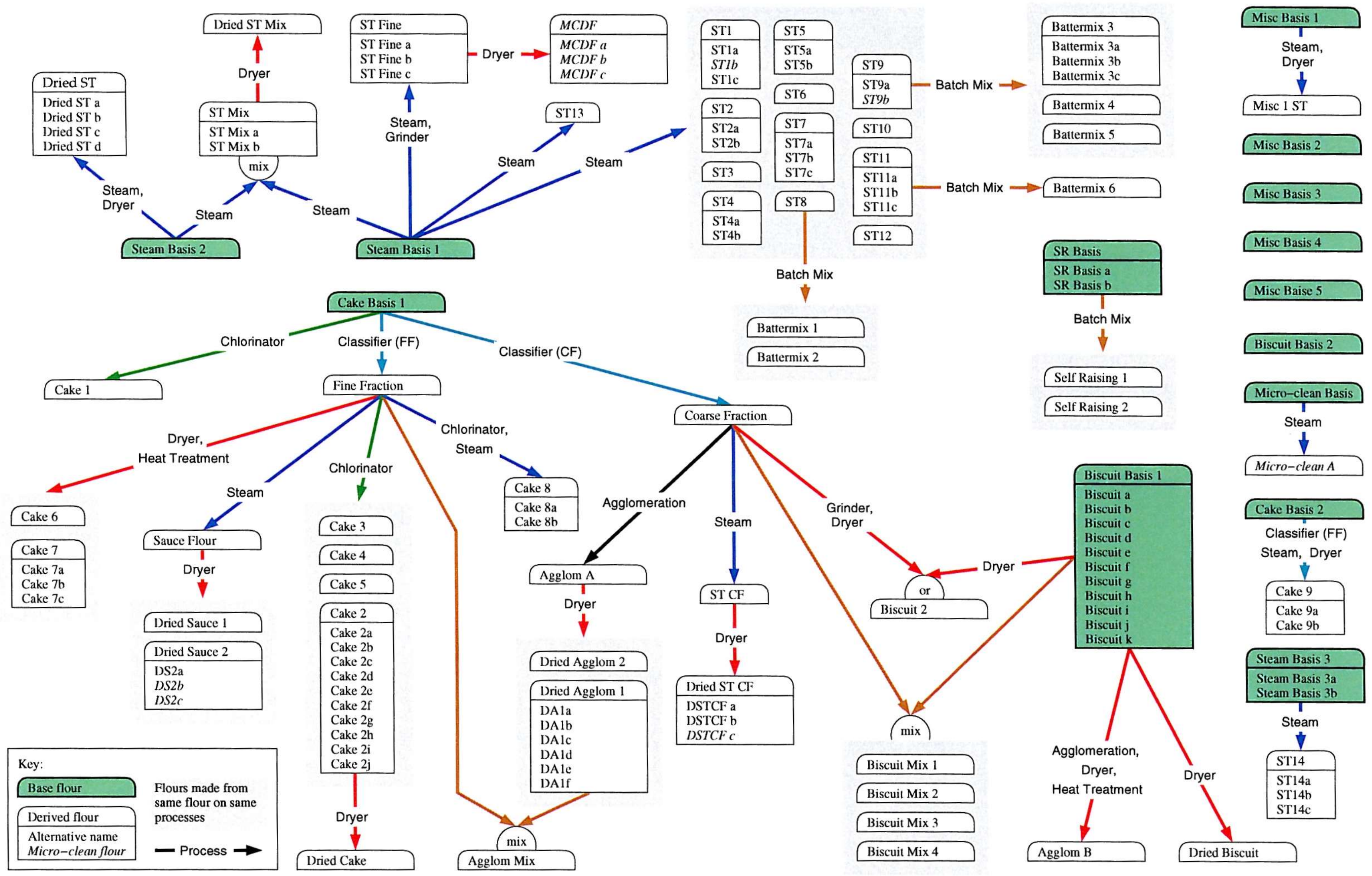


Figure 2.2: Product hierarchy at Foster Mills, Oct 1998

We distinguish between *interim flours* and *final flours*.

Interim flour refers to any flour made in the plant which may be used as the raw material for some other machine or process in the plant.

Final flours are those products which are the final output of the process – they are the saleable items.

Note that some products may be both interim flours and final flours at the same time. For example, see *ST9* in the product hierarchy in figure 2.2, which may be sold as ST9 (i.e. as a final flour), or used to make several distinct batter mixes (i.e. as an intermediate flour). The product hierarchy diagram in figure 2.2 does not show all of the interim flours; some are implied where there are several processes on an arrow.

The product types are closely tied in to the set-up periods of the machines. When a machine with set-ups (for example, one of the steam plants) processes a batch of flour, it must be set up for that batch, and run for a while to verify the resulting flour quality. Changing to a different output flour on the same machine requires another period of set-up. However, batches of the same flour may be processed together on the machine without the need for an additional set-up time between them. It is important to note that this applies to batches of the same interim flour being made together, regardless of whether both batches will become the same final flour. As an example, one steam plant may produce a batch of ST2 for packaging, followed by another batch of ST2 which will be dried and packaged as Dried ST2, without the need for a set-up period between the two batches since they are both the same flour *at that stage*. The problems of scheduling systems with this type of sequence-dependent set-up time are known in the mathematical literature as *product family* scheduling problems.

2.7 Current scheduling practice

The current scheduling process in the plant at Foster Mills is a manual process. As customer orders are placed with the sales department, the order requirements information is updated and passed to the person doing the scheduling. Each day, a three day schedule is created, where the next 24 hours are a fixed schedule, and the two days following are provisional plans. The schedule consists of a plan showing which products should be made on which machines and when, and which storage bins should be used.

2.8 Features of the plant

The plant as described above has a number of interesting features from a mathematical scheduling point of view. These characteristics are listed below:

1. each job is processed on two or more machines,
2. the machines are in the same fixed order for all jobs,
3. jobs may miss some stages (*missing operations*),
4. there are parallel machines at at least one point in the process,
5. a job may be limited to a subset of the available machines at a given stage (*specialist machines* or *processing set restrictions*),
6. there are storage bins between processes which interact with the processes as additional scheduling constraints,
7. processing times of jobs are both job- and machine-dependent,
8. some machines must be used together as a single unit (agglomerator and T-V dryer; W.P.1 and Vrieco; batch mix and packer 1 or FIBC 2),
9. a job may be processed in a continuous flow through the plant,
10. products become more differentiated as they move through the production process (giving an out-tree structure),
11. a few products are made by blending two products together in bins,
12. some machines have set-up times between jobs which make different products,

13. jobs are blown through compressed air pipes between processes,

14. some subsets of bins may not be used to feed different machines at the same time (the conveyor restrictions), and

15. one machine (classifier) produces two distinct products at the same time.

The first five of these characteristics are extremely important to the scheduling of the plant. The next four are also important, but can be modelled as part of a more general model. The remaining six have a negligible effect on the scheduling of the plant. In the next chapter, we will develop simplified models of the plant which implement the first nine features of the above list in one way or another. We shall then use the simplified models as a framework on which to build a scheduling algorithm.

Chapter 3

Mathematical Formulation of Industrial Process

We aim to develop a mathematical model of the production process at Foster Mills described in the previous chapter. The model will describe, using the terms of the scheduling literature, a mathematical scheduling problem which we will then go on to solve. There are two stages towards achieving this aim: simplifying the production process at Foster Mills as described in chapter 2, and placing that simplified system in a formal mathematical framework.

In this chapter, we concentrate on simplifying the model, rather than solving the problem. To this end, our focus will be on the logistics of the plant. We develop progressively simpler models of the plant, by a process of modifying or incorporating the salient features identified earlier in chapter 2. Our aim is to identify which manufacturing features affect the logistics of the plant, from a scheduling perspective, and which do not. The aim of the process of progressive simplification is to produce a model which will then be placed in a mathematical framework in chapter 4. We eventually reduce the process at Foster Mills to a three-stage model for scheduling.

3.1 Scheduling objectives

The objectives for scheduling at the mill are due-date based. The main scheduling objective is to ensure that all jobs are ready on time – the key productivity indicator is actually the proportion of jobs shipped on time. Thus, the objective for scheduling is to minimise the number of late jobs, $\sum U_j$. A secondary consideration, particularly if all of the jobs can be completed on time, is to have as many jobs finished as long before their due date as possible, effectively minimising the maximum lateness, L_{max}.

3.2 Features not affecting scheduling

In this section, we discuss the modelling of some of the salient features of the manufacturing process at Foster Mills identified in chapter 2 and listed in §2.8. In particular, we describe several of the features in the list which have little or no effect on the scheduling of the system, and which we will not be modelling explicitly.

3.2.1 Blowlines

The blowlines in the plant at Foster Mills are high-pressure compressed air tubes which blow the flour from one place to another (feature 13 in §2.8). The rate of transfer of flour in a blowline is very high. In fact, it is much higher than the rate of processing of any of the machines in the plant. In addition, the time taken for a particle of flour to move along any given blow-line is negligible compared to the time taken for any other part of the process. Thus, the blowlines do not limit or delay any part of the manufacturing process. Therefore, we may safely ignore the effects of the blowlines on the timing of any schedule. The blowlines still affect the scheduling of the plant, but only in the sense that they connect machines and bins together, and thus restrict the overall flexibility of the plant.

3.2.2 Conveyor restrictions

The restrictions which stem from the feed conveyors from the bins (§2.4; feature 14 in §2.8) are extremely complex. As a result, they vastly increase the difficulty of the mathematical scheduling problem if they are included in the model. However, in the practical day-to-day operation of the plant, they do not affect the scheduling to any appreciable degree. We therefore decided to ignore entirely the limiting effects of the feed conveyors in both our initial analysis and in our models.

3.2.3 Start-up times and product hierarchy

We will not consider the machine set-up times (feature 12) in our model. Those set-up times which are non-zero in the plant are in practice either small compared with the typical production run (the steam plants), or on fast, non-bottleneck machines (the bulk output processes), or both (the bag packers). By not considering set-up times in the model, we also remove the benefit within the model of working with product families (feature 10). Thus, we treat each job as being distinct.

In addition, we will not deal with the problems posed by products made from blending together two flours (feature 11). There are only a very few such products made at the mill, and they are not made in large quantities. The blending process for these products is usually performed immediately before being sent to packing. Sometimes, the two flours are blended in the packing process – two separate flours are blown into and mixed in a bulk tanker at the same time. Therefore if necessary the scheduling of the blending operation can be performed manually after the main scheduling algorithm has been run.

3.3 Full process diagram

The blow-line connections in the plant at Foster Mill allow almost any order of processes in the plant, including re-entrant processes (see the process diagram in appendix A). However, in practice, this great flexibility is not used. The main reason for this is that some processes must follow others (for example, there is no point in drying a flour and then steaming it).

The eleven processes performed in the plant (listed in §2.1) can be broken up into six groups, according to the connections between the machinery. These six groups are, in order,

1. milling,
2. classification, chlorination and grinding,
3. steam treatment,
4. agglomeration and thermo-venturi (T-V) dryer,
5. Werner-Pflederer (W-P) dryers and heat treatment,
6. mixing, packing and bulk output.

Flour is always processed through the system in the order that the processes appear in the above list (feature 2 from §2.8), although most flours will not pass through all of the process groups (feature 3). From the observation that the overall production process is not re-entrant, and follows a fixed order of processes, we can draw a much simplified diagram of the plant, shown here in figure 3.1.

Note that in the diagram, the storage bins are shown as anonymous blocks. In practice, to move flour from any given processing stage to any other only a subset of all the bins can be used. The bins are shown this way to highlight the essentially uni-directional flow of processing in the plant. The use of the bins will be discussed in greater detail in chapter 4, where we will model them in terms of overlapping sets of bins which may be used to transfer material between two given machines. This allows us to include feature 6 from the list in our model.

In addition, a job may enter the "pool" of bins after processing on one stage, and exit the pool for further processing after skipping one or more stages. Thus, this model allows for the missing operations of feature 3.

Note also that there are four stages (2, 3, 5 and 6) with multiple parallel machines (feature 4). These parallel processing stages pose interesting scheduling problems in their own right – the best example of this is stage 3, where the newer steam plant 3 has better filters than the older two machines, and is also the only way of putting flour into the micro-clean section of the mill. Thus, for

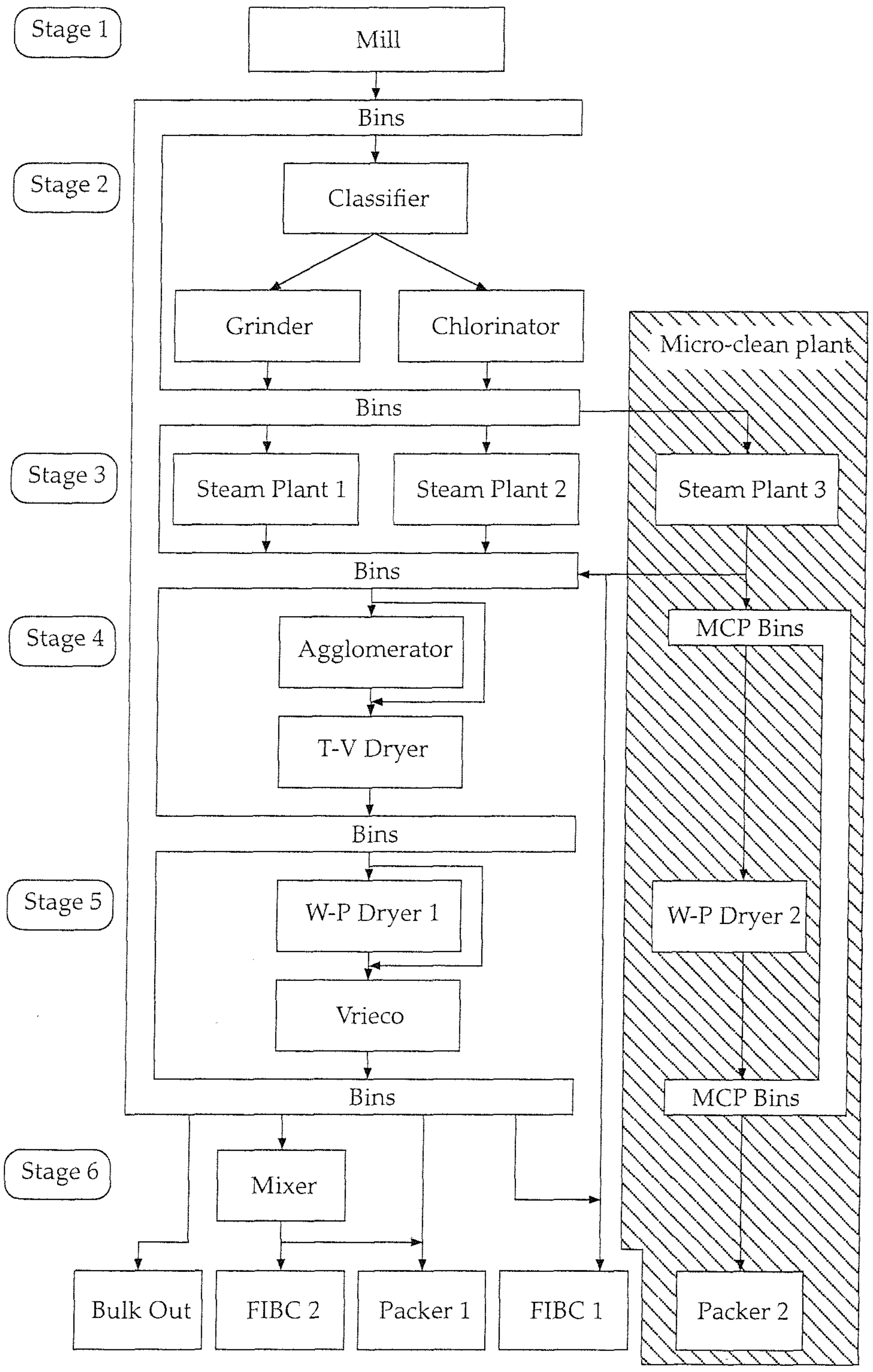


Figure 3.1: The six-stage model of the plant at Foster Mill

some flours (coarse, non-micro-clean products) the three machines are effectively interchangeable; for other flours (very fine flours or micro-clean products), only steam plant 3 can be used. Similarly, some products may be processed through either of the two W-P dryers at stage 5, some must be made on W-P dryer 1 (for heat treatment in the Vrieco), and some on W-P dryer 2. This structure of parallel machines with identical processing properties but restrictions on which jobs may be processed on them is a problem which has received little attention in the literature. The class of such problems is studied in chapters 5 and 6. We are therefore including feature 5 from §2.8 completely in our model.

3.4 The five-stage model

Although the six-stage model described in the previous section is a good description of the production system at Foster Mill, we can make some additional simplifications. These simplifications discard some of the features and structure of the specialist treatment plant, but only in places where the loss of precision is minimal.

Firstly, the usage of the batch mixer is very small – little of the total output of the plant passes through the batch mix process. Since the batch mix process is also one of the faster pieces of equipment in the plant, we can ignore the batch mixer for the purposes of any scheduling algorithm, relegating the scheduling of the machine to a post-processing stage.

Secondly, at stages 4 and 5, the pairs of machines in series (agglomerator-T-V dryer and W-P dryer-vrieco) may each be treated as a single machine, with product-dependent processing rates (i.e. at the speed of the slower machine). This means that feature 8 from the list in §2.8 can be successfully modelled. This also changes feature 7 into something more manageable – job-dependent but machine-independent processing times.

Few products, if any, need to pass through both the agglomerator-T-V dryer and the W-P dryer-vrieco units. We shall therefore model those two units as parallel machines, in the same stage as the W-P dryer in the micro-clean plant, reducing the total number of stages required. We may ensure that a product which requires (say) the agglomerator is processed on the appropriate machine of the three parallel units at that stage by using processing set restrictions, and limiting the job in question to being passed through only that machine.

After the above simplifications, we have a five-stage process. The simplified process diagram is shown in figure 3.2.

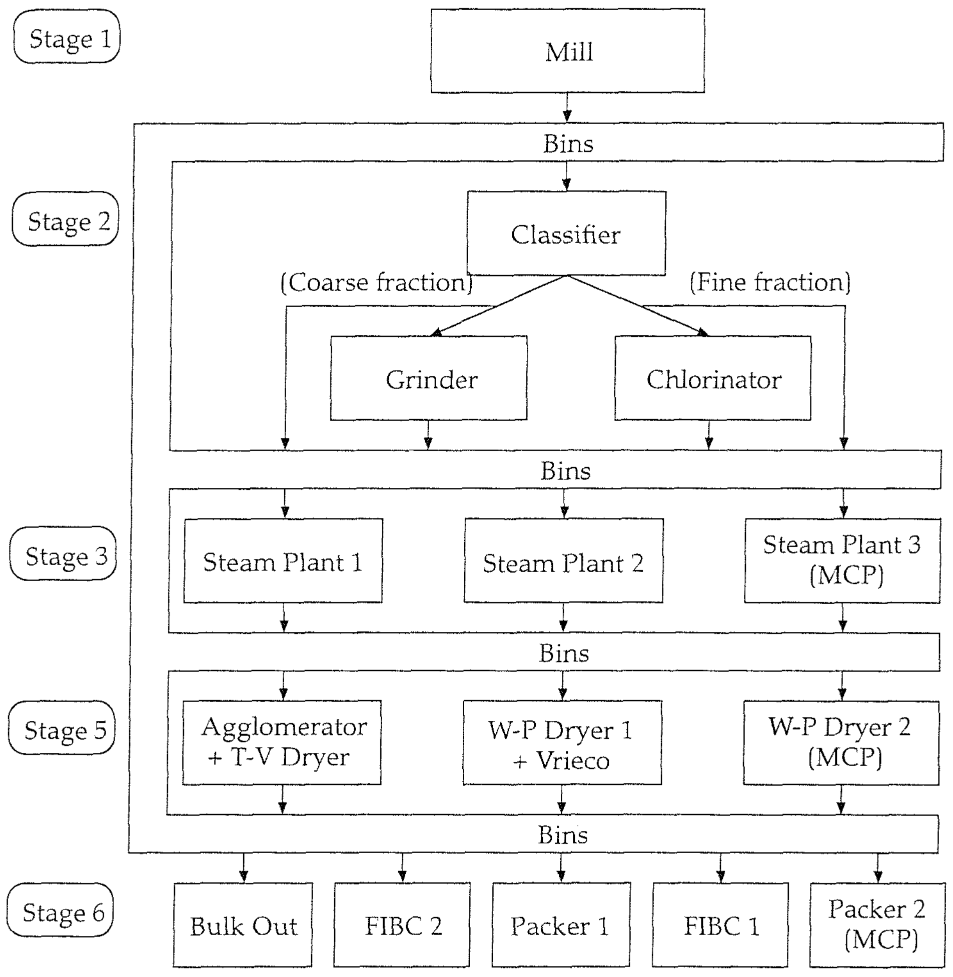


Figure 3.2: The five-stage model of the plant at Foster Mill

3.5 Core scheduling model

In practice, the characteristics of the industrial plant and its current usage patterns allows a further stage of simplification. The bins between the mill and the rest of the plant, and between the classifier/chlorinator and subsequent processes, are sufficiently large and numerous that in practice the first two stages of the plant may be scheduled effectively independently of the rest of the system. Moreover, that these stages contain the fastest two machines in the plant is also a factor.

The first two production stages may be scheduled after the rest. This removes the problem posed by the awkward classifier machine (feature 15). We have thus reduced the problem of scheduling the entire plant to that of scheduling a three-stage system with several parallel machines at each stage, as shown in figure 3.3.

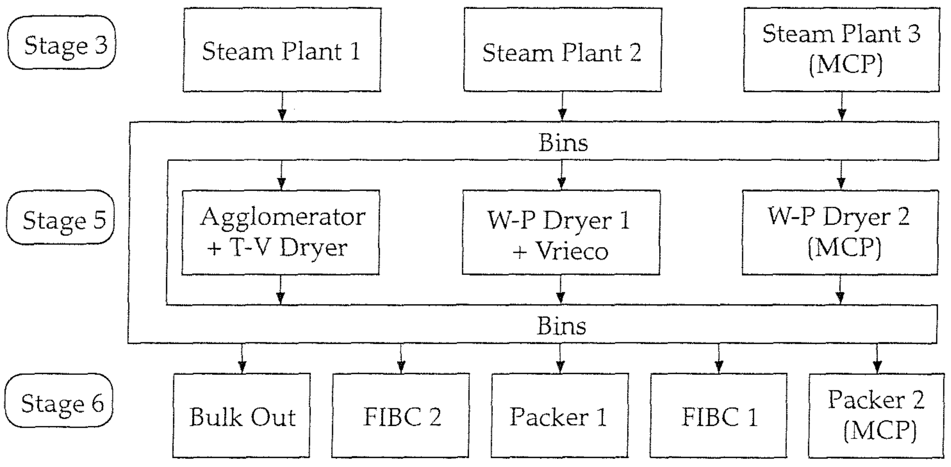


Figure 3.3: The core scheduling model of the Foster Mill plant

3.5.1 Hybrid flow shop

In the scheduling literature, the *hybrid flow shop* (sometimes called a *flexible flow line*) is a multi-stage scheduling model with parallel machines at one or more stages. Jobs are processed through the stages strictly in stage order, and a job is processed on at most one machine at any given stage. The literature of the hybrid flow shop will be discussed in more detail in chapter 7. The 3-stage core scheduling model which we have arrived at in this chapter resembles a hybrid flow shop, with four additional features:

- continuous flow processes,
- processing set restrictions at each stage,
- product families, and

- inter-process storage bins.

We now introduce terminology and notation for the hybrid flow shop problem and each of the additional features listed above.

We shall assume that there are s processing *stages* in the system which we are modelling, with the stages indexed by k $(1 \leq k \leq s)$. At each stage, there are m_k identical *machines*, numbered $1, \dots, i, \dots, m_k$. When referring to machines at different stages, we shall use (k, i) to denote machine i at stage k. Machine (k, i) has a product-dependent *processing rate* $\rho_{k\mathbf{F}}$ and requires a time $s_{k\mathbf{F}}$ for *set-ups* (see §3.5.4 for details on the product type notation, $\mathbf{F}$). In our model, the set-up period requires the presence of the next operation, and consumes material at the normal processing rate of the machine. The material processed during the set-up period is discarded.

There are n *jobs* $(1, \dots, j, \dots, n)$, each consisting of up to s operations. Each job has a *due date*, d_j, and a product type, $\mathbf{F}_j$. A job may also have a *release date*, r_j, before which processing of the job cannot start. An *operation* of a job is the processing of that job at a given stage. The operation of job j at stage k is denoted O_{jk}. An operation has a *length*, p_{jk}, a *size*, β_{jk}, which is a measure of the amount of storage it will require after processing, and a *processing set*, M_{jk} (see §3.5.3), as properties. Jobs may not necessarily have operations at all stages.

A summary of this notation, and of all other notation used throughout this thesis, can be found in appendix C.

3.5.2 Continuous flow

The continuous-flow nature of the process (feature 9) does not affect the analysis and simplification that we perform in this chapter. It does, however, affect how we perform the mathematical modelling of the process. We will investigate the implications of scheduling continuous-flow processes in chapter 4.

3.5.3 Processing sets

Processing sets are restrictions on which machines may process a particular job. For example, at Foster Mills, some jobs must be processed through the micro-clean plant, which requires the job to pass through the third steam plant; alternatively, a job which is to be packed in bags must pass through one of the bag packers and not through the bulk output equipment at the final stage. We model these restrictions as a set of machines for each operation of each job in the problem. We denote the processing set for operation O_{jk} by M_{jk}.

It is also convenient to talk about the unique processing sets which are (or may be) encountered at any given stage. At stage k, there are Q_k possible values that M_{jk} might take. We index these possible values with q, and denote them by $M^{(qk)}$ (where $1 \leq q \leq Q_k$). We refer to the set of unique processing sets, $\{M^{(1k)}, \ldots, M^{(Q_k k)}\}$, as the processing sets *available* at stage k.

If we place no restrictions on the possible processing sets which may be used at stage k, say, then there exist $2^{m_k} - 1$ possible sets, these being the members of the power set of the set of machines, less the empty set. We describe this case as *general* processing sets. As an example, with three machines (1, 2 and 3), there are seven possible general processing sets: $\{1,2,3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1\}$, $\{2\}$ and $\{3\}$.

Alternatively, we may restrict the allowable processing sets to be nested. Interestingly, all of the stages in our 3-stage core model of Foster Mills exhibit nested processing sets. Formally, a set of processing sets available at stage k, $\{M^{(1k)}, \ldots, M^{(Q_k k)}\}$, is *nested* if, and only if, for each pair of sets $M^{(qk)}$, $M^{(q'k)}$, one of the following is true:

$$\begin{aligned} M^{(qk)} &= M^{(q'k)}, \\ M^{(qk)} &\subset M^{(q'k)}, \\ M^{(qk)} &\supset M^{(q'k)}, \text{ or} \\ M^{(qk)} \cap M^{(q'k)} &= \emptyset. \end{aligned}$$

In other words, either one set is completely contained within the other, or they are totally disjoint. This structure, which imposes a partial order on the processing sets, is important, as it gives an ordered, hierarchical structure to the parallel machines problem, and makes it considerably more tractable as a result. A final important observation on sets of nested processing sets is that at a stage k with m_k parallel machines, there can be at most $2m_k - 1$ sets. We prove this result below.

Let Q be a set of nested sets constructed from some set of machines $S = \{1, \dots, m\}$. We wish to find the least upper bound on the size of Q. We denote this least upper bound by $N(m)$ where m is the size of the "parent" set S.

Each set in a collection of nested sets is either a "leaf" set, (i.e. it has no further sub-division), or it has nested subsets. There are many ways in which nested sets may be constructed. For example, two possible ways of making nested sets for seven machines are

$$\{\{1\}, \{2,3\}, \{5,6\}, \{7\}, \{1,2,3,4\}, \{5,6,7\}\},$$

and

$$\{\{1,2,4\}, \{3\}, \{5\}, \{6,7\}, \{1,2,3,4\}, \{1,2,3,4,5,6,7\}\}.$$

However, the number of subsets in a nested set is bounded, as we now demonstrate.

Theorem 1 *Given a set S of size m, let $N(m)$ be the largest number of nested subsets that may be constructed from S. Then*

$$N(m) = 2m - 1.$$

Proof: Firstly, observe that when considering a division scheme to generate the maximum number of nested subsets, we need only consider those sub-divisions which partition each set into two subsets. If we divide a set into more than two partitions, $\mathcal{P}_1, \dots, \mathcal{P}_k$, for example, then by forming $\mathcal{P}'_{k-1} = \mathcal{P}_{k-1} \cup \mathcal{P}_k$, and then subdividing $\mathcal{P}'_{k-1}$ into its two components $\mathcal{P}_{k-1}$ and $\mathcal{P}_k$, we may add one set to the overall total.

We proceed by induction. Firstly, $N(1) = 1$ is trivial. Secondly, assume that $N(m) = 2m - 1 \; \forall \; m < M$. Then

$$\begin{aligned} N(M) &= \max_{1 \le i \le M/2} \{N(i) + N(M-i) + 1\} \\ &= \max_{1 \le i \le M/2} \{2M - 1\} = 2M - 1, \end{aligned}$$

thus proving the theorem. □

As an example, consider the set $S = \{1, \dots, 7\}$, representing seven machines. Then $N(7) = 13$, and we may divide S into up to 13 nested subsets. The following is an example of precisely 13 nested subsets:

Subdivision scheme	No. of sets	Nested sets
$1\;2\;3\;4\;5\;6\;7$	1	$\{1,2,3,4,5,6,7\}$
$1 \mid 2\;3\;4\;5\;6\;7$	2	$\{1\},\{2,3,4,5,6,7\}$
$2\;3\;4 \mid 5\;6\;7$	2	$\{2,3,4\},\{5,6,7\}$
$2\;3 \mid 4 \mid 5\;6 \mid 7$	4	$\{2,3\},\{4\},\{5,6\},\{7\}$
$2 \mid 3\quad 5 \mid 6$	4	$\{2\},\{3\},\{5\},\{6\}$
	13	

3.5.4 Product hierarchy

The out-tree nature of the product hierarchy poses an interesting problem for modelling, in that not only does each job have a product type, but so does each individual operation. The product type of a job defines the types of each of its operations. Moreover, the type of any given operation of a job defines the types of the preceding operations, but not the succeeding ones. We present here a suitable notation/ data structure for modelling such a structure.

We index products, whether interim or final, with an $(s+1)$-element ordered list (or vector), $\mathbf{F} = (F_0, \ldots, F_s)$. Element F_k (for $1 \leq k \leq s$) corresponds to the output of the process at stage k. The element F_0 distinguishes between different types of raw material to be fed to the first process. A zero entry in this list indicates that the product was not processed at the corresponding stage. A non-zero element in the vector means that the product was processed at that stage, and is used to distinguish between the different output products made from the same input product at the corresponding stage. The products are numbered so that the product $(F_0, \ldots, F_k, 0, \ldots, 0)$ is the parent of all product types with the same entries in the first k positions. As an example, we might index the products shown in figure 2.1 according to the scheme in table 3.1. Note that in our 3-stage core scheduling model of the plant at Foster Mills, all final products have a non-zero entry in the last element of the product index vector, and all interim products a zero, since all final products must be packed in some way.

We also define several functions and operators, for ease of manipulation of these product indexes. Firstly, for a product $\mathbf{F}$, we define the last stage at which it

Product	Index
ST Base	(1, 0, 0, 0)
ST1	(1, 1, 0, 0)
ST2	(1, 2, 0, 0)
Dried ST2	(1, 2, 1, 0)
ST1 (Packed)	(1, 1, 0, 1)
ST2 (Packed)	(1, 2, 0, 1)
Dried ST2 (Packed)	(1, 2, 1, 1)

Table 3.1: Example of product indexing scheme

was processed,

$$\chi(\mathbf{F}) = \min_{1\leq k\leq s} k \quad \text{such that } F_k \neq 0 \text{ and } F_{k'} = 0 \, \forall k' > k.$$

Secondly, we can define parent-child relations between products to navigate the out-tree hierarchy. A product $\mathbf{F}$ has an ancestor at stage k if it was at some point processed on stage k. Thus, if $\chi(\mathbf{F}) \geq k$ and $F_k \neq 0$, then the ancestor of $\mathbf{F}$ at stage k is

$$P_k(\mathbf{F}) = (F_0, \ldots, F_k, 0, \ldots, 0).$$

Alternatively, we may view the tree hierarchy as being a partial order on the product types. We define a partial order, $\prec$, on product types, where $\mathbf{F} \prec \mathbf{F}'$ if and only if $\mathbf{F} = P_k(\mathbf{F}')$ for some $k < \chi(\mathbf{F}')$.

We define the parent of a product as being the last ancestor of the product before the current stage

$$P(\mathbf{F}) = (F_0, \ldots, F_k, 0, \ldots, 0), \text{ where } k = \max_{k'<\chi(\mathbf{F}) \text{ and } F_{k'}\neq 0} k'.$$

Note that these definitions mean that, where it is defined, $\chi(P_k(\mathbf{F})) = k$ for all $\mathbf{F}$ and k. However, $\chi(P(\mathbf{F}))$ is not necessarily $\chi(\mathbf{F}) - 1$, since a product may miss a processing stage. It is also the case that, given a job j with product type $\mathbf{F}_j$, the product type of the output flour of operation O_{jk} is $P_k(\mathbf{F}_j)$.

3.5.5 Inter-process storage bins

In the earlier parts of this chapter, we have brushed over any detailed discussion of modelling the bins at Foster Mills. We now remedy that omission, and discuss here a suitable mathematical model for the plant.

The flexibility of the production plant requires a flexible and rather general model. We model the storage at Foster Mills as a shared pool of bins, with restrictions on which bins may be used to hold flour between any pair of machines. We denote the set of all bins by $\mathcal{B}$, and use b to index the members of this set. The capacity of bin b we write as c_b. Between any arbitrary pair of machines, say (k, i) and (k', i'), the set of bins which may be used to hold flour is $B_{(k,i),(k',i')} \subseteq \mathcal{B}$.

In some cases, a bin or bins may be specific to a particular flour (by convention rather than required by the plant's engineering), in which case we must define bin sets which depend on the flour type as well, for which we use the notation $B_{(k,i),(k',i'),\mathbf{F}}$, where $\mathbf{F}$ is the flour type.

In this chapter we have developed a scheduling model of the plant at Foster Mills which incorporates all of the features necessary for solving the scheduling problems at Foster Mills. The simplified model takes the highly complex and flexible system described in chapter 2, and presents it as a hybrid flow shop. This presentation supports the current and likely future practises of operation at the plant. In the next chapter, we present methods of modelling and evaluating schedules using the model of this chapter.

Chapter 4

Network formulations of hybrid flow shops

In this chapter, we discuss how to model hybrid flow shop processes such as those, corresponding to Foster Mills, identified in the previous chapter. We pay particular attention to the makespan and maximum lateness objective functions. We develop disjunctive graph models for scheduling the hybrid flow shop with and without inter-process storage. In doing so, we also investigate some of the implications of the continuous-flow processes seen in the industrial plant, as compared to the discrete-flow processes more commonly encountered in the mathematical scheduling literature.

4.1 Literature on inter-process storage

In the mathematical scheduling literature on multi-stage systems such as (hybrid) flow shops, it is usually assumed that there is unlimited storage capacity available between processing steps for any job. However, there is also an extensive literature on two-stage flow shops with limited (or no) storage available between machines.

In models with inter-process storage, it is usually assumed that each job in the problem occupies the same amount of space, and that each machine has an output buffer into which completed jobs are placed. When a buffer is full, the machine cannot complete any further operations, and is said to be *blocked*. A blocked machine must wait for one of the jobs in its output buffer to be started on some other machine before it can complete its current operation. There are effectively three distinct cases in flow shop scheduling: buffers of unlimited size ($b \geq n$), no buffers ($b = 0$), and buffers of finite size ($0 < b < n$). Surveys of the literature in this area include one by Hall and Sriskandarajah[35], and a smaller review by Dudek, Panwalkar and Smith[21], which makes interesting comments on the practical utility of the existing research into flow shop scheduling.

The first case above, with unlimited buffer capacity, is the ordinary flow shop. This was introduced as 2- and 3-machine problems by Johnson[45], who showed that for 2- and 3-machine makespan problems there exists a permutation schedule (i.e. the jobs are processed in the same order on each machine) which is optimal. It is from Johnson's original paper that the extensive literature of flow shop scheduling is descended.

The second case, with $b = 0$, is known as the *no-wait* flow shop, where jobs have to be processed from start to finish with no pauses between machines. The first major result for this problem was that of Gilmore and Gomory[28], who showed that two machines can effectively be modelled as a single machine, and the scheduling problem can be solved in polynomial time as a special case of the travelling salesman problem.

The third, more general case, with $0 < b < n$, was shown to be strongly **NP**-complete by Papadimitriou and Kanellakis[63], who also developed a heuristic scheduling method for the case when $b = 1$ with worst-case bound $\frac{3}{2}$.

4.2 Disjunctive graph representations

With complex scheduling problems such as that which we are investigating, one common approach in the literature is to formulate the problem as a *disjunctive graph*. The first use of disjunctive graphs in mathematical scheduling was for the job shop problem in Roy and Sussmann's 1964 paper[69]. The concept has been used by many people since, most usually for the job shop problem[3, 64, 4], but also for other problems, such as the hybrid flow shop[79]. Similar graph techniques have been used as evaluation tools for more complex problems such as hybrid job shops[54] and even for applications which do not fit into the standard scheduling problem classification such as an automated chemical laboratory[36].

In a disjunctive graph formulation, a directed graph represents the relationships between the individual operations in the problem. For a flow shop, some arcs in the digraph have fixed direction, namely those representing the scheduling constraints between operations of the same job. Other arcs in the digraph form sets of arc pairs, from which one arc of each pair must be chosen. These disjunctive arcs represent the allocation of operations to machines, and the sequencing of operations on those machines. The nodes linked by disjunctive arcs, corresponding to the processing at a single stage, form a clique (complete subgraph) of nodes. By making a suitable selection of one arc from each pair of disjunctive arcs, sequences of operations can be constructed at each stage. It is possible to create several sequences of operations at a single stage by removing arcs to make several disjoint subgraphs. This allows sequences on independent parallel machines to be modelled.

In such a network, the nodes of the graph represent individual time-points, such as the time at which an operation starts, and the arcs represent minimum-time constraints between events. Each arc on the graph has a length and corresponds to a constraint of the form $t_a - t_b \geq d$, where t_a and t_b are the times of events a and b, associated with the start and end nodes of the arc respectively, and d is the minimum delay between the two events. For example, when operation 4 may not start until the previous operation, 3, has completed on the same machine, we have $t_{\text{op 4 start}} - t_{\text{op 3 end}} \geq 0$. Similar constraints may be used to specify each relationship between the operations on a machine, and between the successive operations of a job. The length of an arc may be thought of as the minimum time allowed between two events, which are the nodes at each end of the arc.

There are normally two special nodes in the graph, a start node A, and a finish node Z. The longest path from the start node to any given node in the graph is the earliest time at which the corresponding event may happen, counting the start event as time zero. Such a longest path may be found for every node in the graph simultaneously, in $O(nm)$ time, using Dijkstra's algorithm [20, 15] (see appendix B).

4.2.1 Discrete flow, unlimited inter-process storage

The simplest case for which to draw the disjunctive graph is the discrete-flow case where there are no limits on inter-process storage. This is the type of problem most commonly seen in the mathematical scheduling literature. The basic structure of such a digraph is shown in figure 4.1, for an example with three processing stages (numbered 1-3) and four jobs (numbered 5-8 to avoid confusion in the notation). Note that job 6 has a missing operation at stage 1, and job 7 at stage 3. The nodes in this diagram represent the start of the corresponding operation. The two types of arcs are constructed as follows:

Fixed arcs starting at O_{jk} have length p_{jk}. Other fixed arcs (e.g. starting at A) have length zero. These arcs represent the finish-start constraints between operations of the same job.

Disjunctive arcs starting at O_{jk} have length p_{jk}, and represent the finish-start constraints between operations of different jobs on individual machines. Where operations are performed on different machines at the same stage, the disjunctive arcs between them are removed entirely, as described below.

To generate a schedule, a subset of the disjunctive arcs is removed. With a single-machine stage, one arc of each pair is removed so as to give a complete acyclic subgraph of the relevant clique. With m machines, arcs are removed to give m disjoint complete acyclic subgraphs. In either case, each complete subgraph has a unique topological sort order. A *topological sort* of a digraph is an ordering of the nodes in which no node in the list is reachable from any node later in the list. A topological sort of a graph may be found, if one exists, in $O(pq)$ time, where the graph has p nodes and q arcs, using an appropriate representation of the graph. See the description of Dijkstra's algorithm in appendix B for such a topological sort algorithm.

In each case, the ordering of the nodes for a machine is equivalent to the ordering of the operations on that machine. Given an ordering of operations, the

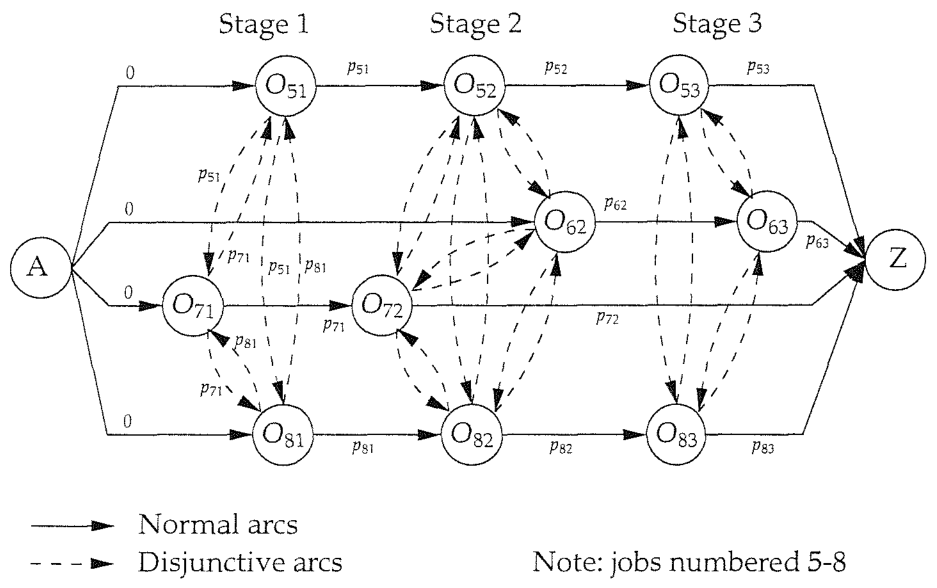


Figure 4.1: Example of disjunctive graph for 4 jobs and 3 stages, discrete flow and unlimited inter-process storage

set of arcs to use can be determined, and vice versa. Taking the earlier example, if the operations are processed on the machines and in the order shown in table 4.1, then the corresponding arc selections would be as shown in figure 4.2. The allocation and sequencing of operations can be achieved by a suitable algorithm, such as one of those discussed in chapters 5 and 6.

Machine	Stage 1 ($m_1 = 1$)	Stage 2 ($m_2 = 2$)	Stage 3 ($m_3 = 3$)
1	O_{51}, O_{71}, O_{81}	O_{72}	O_{63}
2		O_{62}, O_{82}, O_{52}	O_{53}
3			O_{83}

Table 4.1: Example allocation and sequencing of operations (see figure 4.2)

4.2.2 Discrete flow, limited inter-process storage

Now consider a system with limited inter-process storage. When an operation is finished, it is placed into one of a limited number of bins until it is started by the next machine. For example, a Gantt chart showing a single job moving between two processes is shown in figure 4.3. In that figure, whilst the length of each of

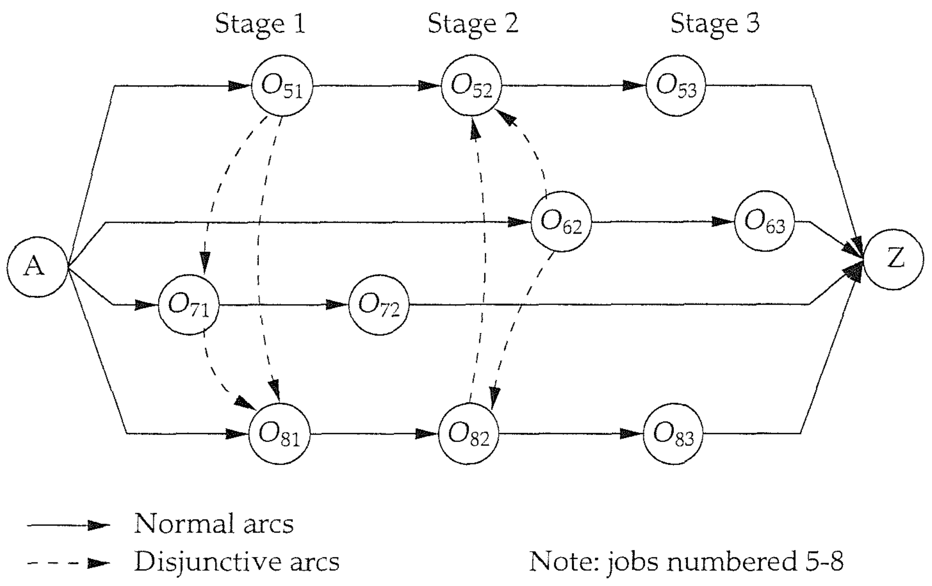


Figure 4.2: Example with selected subset of disjunctive arcs (see table 4.1)

the two operations is fixed, the length of time for which the bin is in use may be any non-negative value.

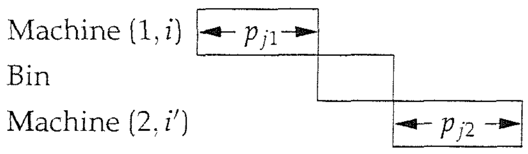


Figure 4.3: Gantt chart of a discrete-flow system with a bin

In the system shown above, there are five constraints for every pair of two machines, A and B, at different stages with a bin between:

- the operation on machine A is of an exact length,
- the operation on machine B is of an exact length,
- the job starts in the bin when it finishes on machine A,
- the job finishes in the bin when it starts on machine B, and
- the bin must be used for a non-negative length of time.

To model this type of system, we introduce an additional set of nodes into the digraph to represent the bin usage. These nodes, labelled B_{jk}, represent the time at which job j enters a bin following the processing of operation O_{jk}. An example of a system with two stages and a single stage of bins separating them is shown in figure 4.4. Note that there are four types of arc in the diagram.

The type 1 arcs represent the sequence of operations within a job (the fixed arcs) and the sequence of operations on machines and bins (the disjunctive arcs). The type 3 arcs represent the movement of a job to the next processing stage. The type 4 arcs prevent operations from being placed into a bin before the previous operation in the bin has moved on to the next processing stage, and form disjunctive pairs similar to the disjunctive type 2 arcs. These types of arc have the following properties:

Type 1 arcs starting at node O_{jk} have length p_{jk}. These represent the length constraint for an operation: an operation must have a fixed length, and the job enters the bin when it has completed processing.

Type 2 arcs starting at bin node B_{jk} also have length p_{jk}. As in figure 4.1, these arcs are disjunctive arcs representing the finish-start constraints on operation of different jobs performed on the same machine.

Type 3 arcs have length zero, and are fixed (i.e. always present). These arcs represent the minimum length of stay of the operation in the bin – this is always zero, but may be longer.

Type 4 arcs have length zero, and form disjunctive pairs. For every pair of nodes B_{jk} and $B_{j'k}$ with subsequent operations $O_{j\ell}$ and $O_{j'\ell'}$ respectively, there is a pair of disjunctive type 4 arcs, one from $O_{j\ell}$ to $B_{j'k}$ and one from $O_{j'\ell'}$ to B_{jk}. These arcs perform the same function for operations in bins as the Type 2 arcs perform for operations on machines. They represent the constraint that a batch cannot finish processing at the previous stage and be placed into a bin until the previous batch in that bin has started processing at the next stage.

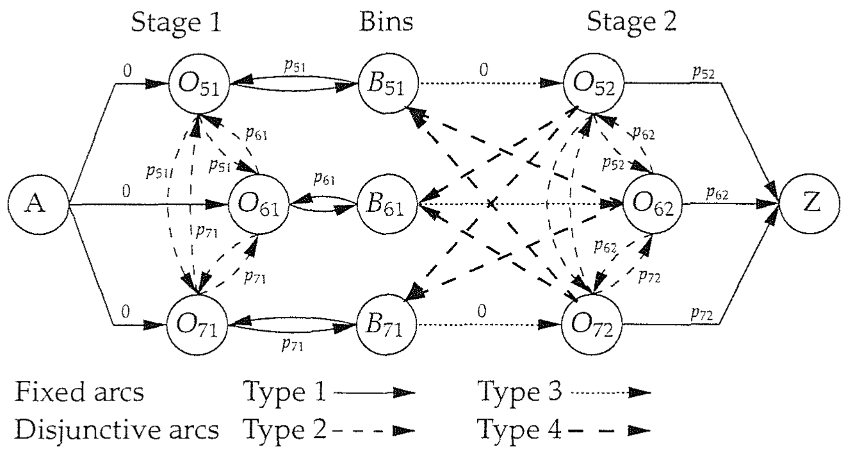


Figure 4.4: Example of disjunctive graph for 4 jobs, 2 stages and one bin stage with discrete flow

Consider also the problem with missing operations which could be affected by bin scheduling. For example, consider the problem given in table 4.2, with bin storage following each stage of machines. Note that there is a missing operation at stage 2 for job 2. The disjunctive digraph for this problem is shown in figure 4.5.

Job	Stage 1	Stage 2	Stage 3
5	3	8	1
6	5	(missing)	6
7	6	2	4

Table 4.2: Processing times for an example problem with missing operations and inter-process storage

4.3 Discrete and continuous flow

Much of the existing mathematical (shop) scheduling literature works on the assumption that an operation cannot be started on any machine before the previous operation is finished on the previous machine. This was the assumption used in the disjunctive graphs of the previous section. One major feature found in our original industrial problem is that jobs *flow* through the system, rather than being

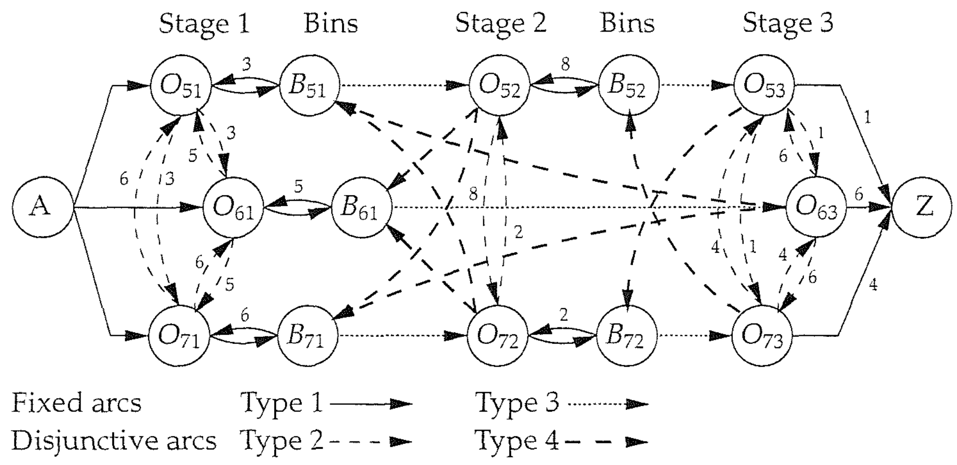


Figure 4.5: Example of disjunctive graph for problem with missing operations

processed in a single block, as they are in the majority of the scheduling literature. As a result, the second operation of a job may be started before the first operation has finished. When bins are also added to the system, its behaviour changes in several ways. The differences between the two types of system (discrete and continuous flow) are shown in figures 4.6 and 4.7.

Obviously, the distinction between the two types of system is only evident when there are multiple stages of production. On a single production stage (one machine or parallel machines), there is no modelling difference between discrete and continuous systems.

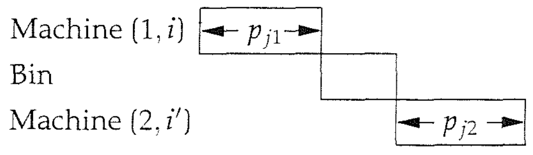


Figure 4.6: Gantt chart of a discrete-flow system

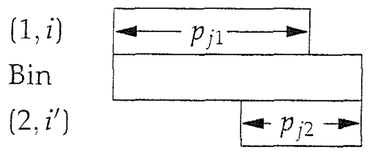


Figure 4.7: Gantt chart of a continuous-flow system

Modelling a system as discrete-flow gives rise to a single finish-to-start constraint for each job and pair of machines. In modelling a continuous-flow system, the single constraint from the "classical" mathematical scheduling problem may be replaced by a pair of constraints, governing the start and end of each operation with respect to the next operation.

In adding continuous flow to the model, we change significantly the structure of the problem. Since few algorithms have been developed for this kind of problem in the mathematical scheduling literature, there is nothing against which we may compare any algorithm we develop.

4.3.1 Batch Size Constraints

When considering inter-process storage bins, discrete-flow systems generally put a simple limit on the number of jobs that may be stored in each bin. However, the nature of continuous-flow systems means that in practice they will often process jobs of differing sizes, and the storage bins will be limited in the size of job (or jobs) they can hold. In our model, a bin holds a single job, but has a limited size.

Since continuous-flow systems allow the second operation of a job to start before the first operation has finished, there exists the possibility of processing a job through a bin smaller than the job. In this section, we are interested in the size of the largest batch that may be processed through a bin. We will consider here only bins with a fixed maximum capacity, holding only a single job at any one time. We are also interested in the leeway that we have in scheduling the start times of a job on the different machines on which it must be processed. We would like to develop a set of limits which may be used in both of these tasks. In this section, we derive such limits for the basic case of two machines with a single dedicated bin between them.

Consider a single job, j, as it moves between stage k and k' in some system: it is processed on a machine (machine (k,i)), and the resulting material is placed in a bin (bin b), followed by processing on another machine at the next stage (machine (k',i')). The two machines have processing rates ρ_k and $\rho_{k'}$ respectively, and set-up times s_k and $s_{k'}$. The bin will hold up to c_b units of material. A batch is to be processed on this system, first on machine (k,i), then the intermediate product is placed in bin b, and finally taken out of the bin and processed on machine (k',i'). The processing run on (k,i) starts at a time t after the start of processing on machine (k',i').

Setting up machine (k,i) requires $\rho_k s_k$ units of material. As a result, to make

$\beta_{jk'}$ units at the end of the second operation in our simple two-machine example, $\beta_{jk'} + \rho_{k'}s_{k'} + \rho_k s_k$ units must be supplied to machine (k,i), of which $\beta_{jk} = \beta_{jk'} + \rho_{k'}s_{k'}$ units must be put through the bin during the course of processing the batch. In other words,

$$\beta_{jk} = \beta_{jk'} + \rho_{k'}s_{k'}. \tag{4.1}$$

We wish to look at the relationships between batch size, β_{jk}, the physical constraints as depicted by ρ_k, $\rho_{k'}$, and c_b and limits on relative start times of the batch on the two machines.

Theorem 2 *The earliest that an operation may be scheduled to start after its predecessor operation has started producing output is*

$$\delta_{jk'} = \max\left\{\beta_{jk}\left(\frac{\rho_{k'} - \rho_k}{\rho_k \rho_{k'}}\right), 0\right\}. \tag{4.2}$$

Proof: We identify two cases: Machine (k', i') is faster $(\rho_k \leq \rho_{k'})$ and machine (k,i) is faster $(\rho_k \geq \rho_{k'})$.

Case 1: Machine (k', i') is faster

We cannot start machine (k', i') so early that it finishes before machine (k,i), thus the earliest that we can start (k', i') is at time $\delta_{jk'}$, as shown in figure 4.8.

$$\delta_{jk'} = \frac{\beta_{jk}}{\rho_k} - \frac{\beta_{jk'}}{\rho_{k'}} - s_{k'}.$$

From equation 4.1,

$$\delta_{jk'} = \beta_{jk}\left(\frac{\rho_{k'} - \rho_k}{\rho_k \rho_{k'}}\right),$$

and since $\rho_{k'} \geq \rho_k$, equation 4.2 holds as claimed.

Case 2: Machine (k,i) is faster

We can start processing on the second machine as soon as the first has started generating output, since the first machine will finish first (being the faster). Thus,

$$\delta_{jk'} = 0,$$

which gives equation 4.2, since $\rho_{k'} \leq \rho_k$ in this case. □

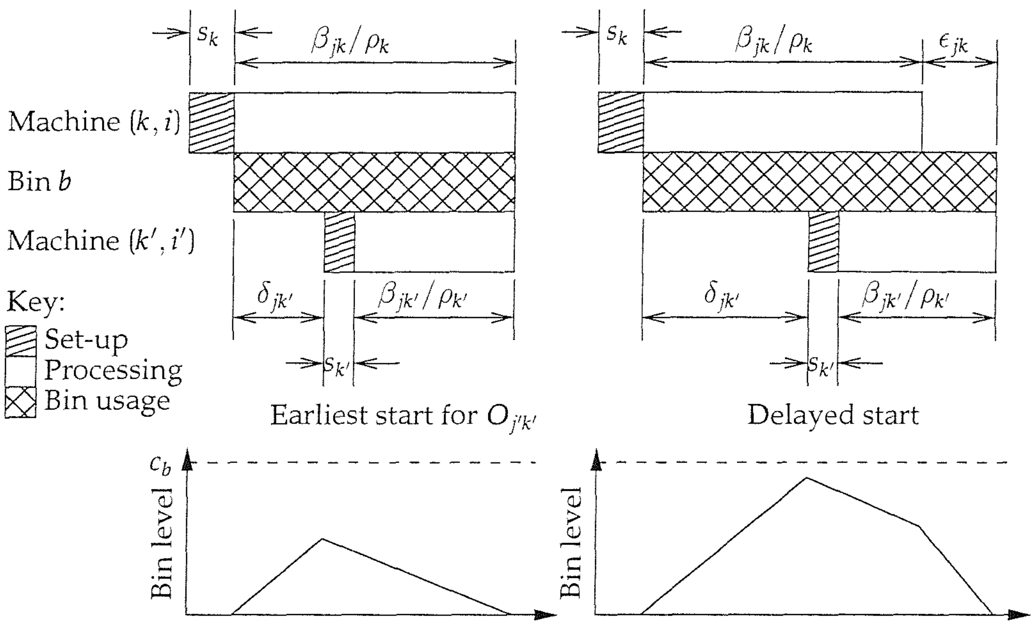


Figure 4.8: Gantt charts where machine (k', i') is faster

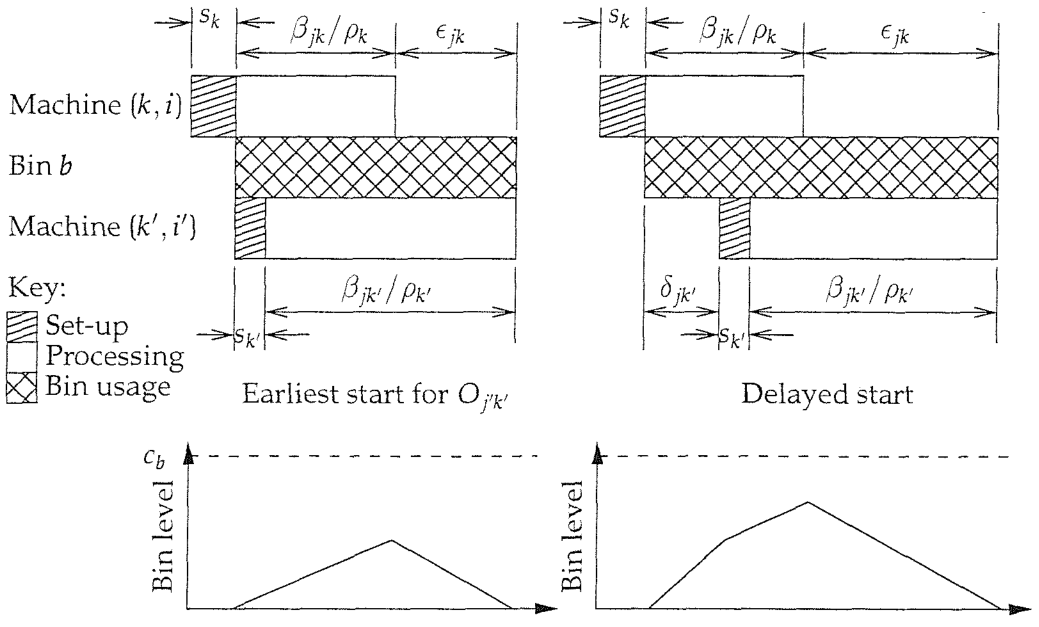


Figure 4.9: Gantt charts where machine (k, i) is faster

Note that in the notation used here, the delay, $\delta_{jk'}$, is indexed by the job j and only one of the stages, k', since the job j will have an end-product type associated with it. From the product type, we can deduce the sequence of stages which the job passes through, and so only one stage of the two involved is required to specify exactly which delay we are referring to.

Theorem 3 *If the batch to be placed in a bin is smaller than the bin, then the consecutive operations do not have to overlap and there is no limit to the possible delay to the second operation.*

Proof: If the batch after processing on the first machine is smaller than the bin size, i.e. $\beta_{jk} \leq c_b$, then we can put the whole batch on (k,i), into the bin, before starting processing on (k',i') and we can have any length of delay before starting the second stage of processing. □

Theorem 4 *The batch size, β_{jk}, that may be processed through a bin b by two machines (k,i) and (k',i') is limited by the bin size c_b in the following manner:*

$$\beta_{jk} \leq c_b \frac{\max\{\rho_k, \rho_{k'}\}}{|\rho_k - \rho_{k'}|} \tag{4.3}$$

Proof: As the overlap between the two stages of processing increases, so does the size of the batch which can be processed. Thus, for the case when machine (k',i') is faster, suppose that both machines stop processing at the same time. The bin is fullest when the later machine starts its set-up, and its level is $\frac{\beta_{jk}}{\rho_{k'}}(\rho_{k'} - \rho_k)$, since its net rate of emptying is $(\rho_{k'} - \rho_k)$, and it takes $\frac{\beta_{jk}}{\rho_{k'}}$ time to empty. When machine (k,i) is faster, an inverse argument holds, and the maximum bin level is $\frac{\beta_{jk}}{\rho_k}(\rho_k - \rho_{k'})$. □

Corollary 1 *If the batch size is larger than the capacity of the bin, then the delay $\delta_{jk'}$ is limited according to the following expression:*

$$\max\left\{\beta_{jk}\left(\frac{\rho_{k'} - \rho_k}{\rho_{k'}\rho_k}\right), 0\right\} \leq \delta_{jk'} \leq \max\left\{\frac{c_b}{\rho_k}, \frac{c_b}{\rho_{k'}}\right\} - \max\left\{\beta_{jk}\frac{\rho_k - \rho_{k'}}{\rho_k\rho_{k'}}, 0\right\}. \tag{4.4}$$

Proof: The left-hand inequality of equation 4.4 is simply derived from equation 4.2. For the right-hand inequality, we consider the two cases of different machine speeds again, as in the previous proof:

Case 1: Machine (k', i') is faster

From figure 4.8, the point at which the bin is most full is $\delta_{jk'}$. We must start processing on (k', i') before the bin becomes filled. The bin fills at rate ρ_k, and has capacity c_b, so

$$\delta_{jk'} \leq \frac{c_b}{\rho_k}. \tag{4.5}$$

Case 2: Machine (k, i) is faster

The bin is at its fullest when the batch finishes processing on the first machine (see figure 4.9). The content of the bin is then emptied at a rate $\rho_{k'}$. The delay between the end of processing on (k, i) and the end of processing on (k', i') is ϵ_{jk}, and is limited to the time taken to empty the bin from full:

$$\epsilon_{jk} \leq \frac{c_b}{\rho_{k'}}. \tag{4.6}$$

Now, the total time taken to process the batch is the time on (k, i) plus the end-delay ϵ_{jk}, which is equal to the start delay on (k', i') plus the processing time on that machine (see figure 4.9):

$$\frac{\beta_{jk}}{\rho_k} + \epsilon_{jk} = \delta_{jk'} + \frac{\beta_{jk}}{\rho_{k'}}. \tag{4.7}$$

Substituting ϵ_{jk} from equation 4.7 into equation 4.6, we obtain

$$\delta_{jk'} \leq \frac{c_b}{\rho_{k'}} + \frac{\beta_{jk}}{\rho_k} - \frac{\beta_{jk}}{\rho_{k'}} = \frac{c_b}{\rho_{k'}} - \beta_{jk}\left(\frac{\rho_k - \rho_{k'}}{\rho_k \rho_{k'}}\right). \tag{4.8}$$

Finally, combining equations 4.5 and 4.8, we obtain the right-hand inequality of equation 4.4. □

4.4 Disjunctive representations with continuous flow

The continuous-flow systems discussed in the previous section have scheduling behaviour which is not the same as the discrete-flow systems for which we developed disjunctive graph representations in §4.2. We now develop graph representations similar to those of §4.2 for continuous-flow systems. We consider two different types of limit on inter-process storage. The first type of limit is of a limited number of bins, which is equivalent to the bin storage limits in the discrete flow case. The second type of limit is of limited size bins, which brings in the timing limits from §4.3.1.

4.4.1 Continuous flow, limited number of bins

Consider a system with a limited number of bins to which operations must be allocated, but where each bin has unlimited capacity. In this case, $c_b = \infty$ in the equations of §4.3, and there is only a single additional limit to be considered – that of the earliest start of an operation.

In the discrete flow system of §4.2.2, there were five constraints on the start times of operations. In the continuous case which we are now considering (with no limit on the bin size), there are also five constraints, but they are slightly different. These are:

- the operation on machine A is of an exact length,
- the operation on machine B is of an exact length,
- the job starts in the bin after the setup on machine A,
- the job finishes in the bin when it finishes on machine B, and
- processing cannot finish on machine B before it does on machine A.

Of these constraints, only the fifth is worthy of further comment. Equation 4.2 gives the minimum time which can be left between starting an operation on machine A and starting the same operation on machine B. This is embodied as a simple minimum-time constraint between the respective starts of two operations. The disjunctive graph representation of this system is shown in figure 4.10.

As before, there are several types of arc on this diagram. The arcs of types 1 and 2 are identical in nature to those in §4.2.2, although there are now two kinds of type 1 arc: type 1a, identical to the type 1 arcs of the earlier diagrams, and type 1b, which have zero length. There are no type 4 arcs (zero length bin sequence constraints). Instead, the type 5 arcs are used and selected in a similar manner. The type 6 arcs represent the time-delay constraint from equation 4.2, and replace the type 3 arcs. Thus:

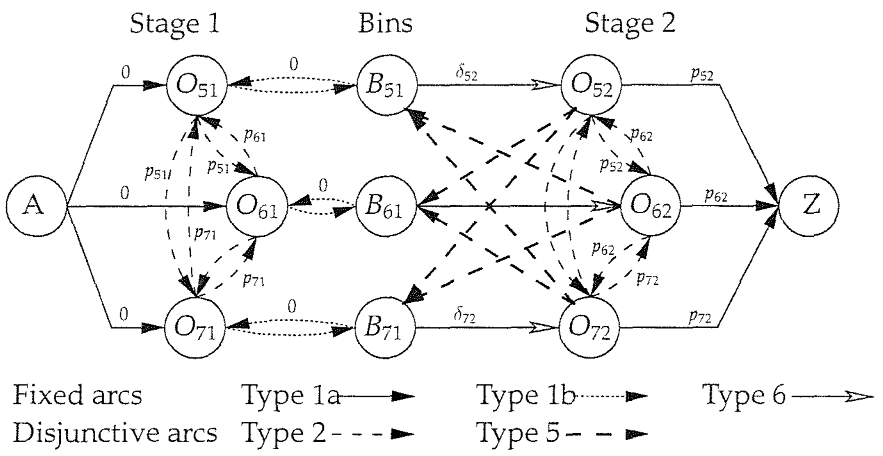


Figure 4.10: Disjunctive graph representation with continuous flow and bins of unlimited size

Type 1a arcs starting at node O_{jk} have length p_{jk}, and represent the length constraint of the operation.

Type 1b arcs have length zero, and represent the fact that the bin is used as soon as the previous operation starts.

Type 2 arcs starting at bin node B_{jk} have length p_{jk}, and are disjunctive arcs representing the finish-start constraints on operation of different jobs performed on the same machine.

Type 5 arcs have length p_{jk}, and form disjunctive pairs. For every pair of nodes B_{jk} and $B_{j'k}$ with subsequent operations $O_{j\ell}$ and $O_{j'\ell'}$ respectively, there is a pair of disjunctive type 5 arcs, one from $O_{j\ell}$ to $B_{j'k}$ and one from $O_{j'\ell'}$ to B_{jk}. These arcs, like the type 4 arcs of §4.2.2, represent the sequence of operations in a given bin.

Type 6 arcs are always present. The arc from B_{jk} to $O_{jk'}$ has length $\delta_{jk'}$ as given in equation 4.2. They represent the minimum delay to the start of processing the operation on the second stage.

4.4.2 Continuous flow, limited number and size of bins

Finally, consider a continuous-flow system with bins which are limited in size. In that case, the limits of theorem 4 come into play. The minimum time, $\delta_{jk'}$, between the start of operation O_{jk} and the start of operation $O_{jk'}$ is defined in equation 4.2. The maximum time, $\lambda_{jk'}$, is defined below.

$$\lambda_{jk'} = \begin{cases} \max\left\{\frac{c_b}{\rho_k}, \frac{c_b}{\rho_{k'}}\right\} - \delta_{jk'} & \text{if } \beta_{jk} \leq c_b \\ \infty & \text{otherwise} \end{cases} \tag{4.9}$$

With this additional constraint present, the appropriate disjunctive network looks like the example in figure 4.11.

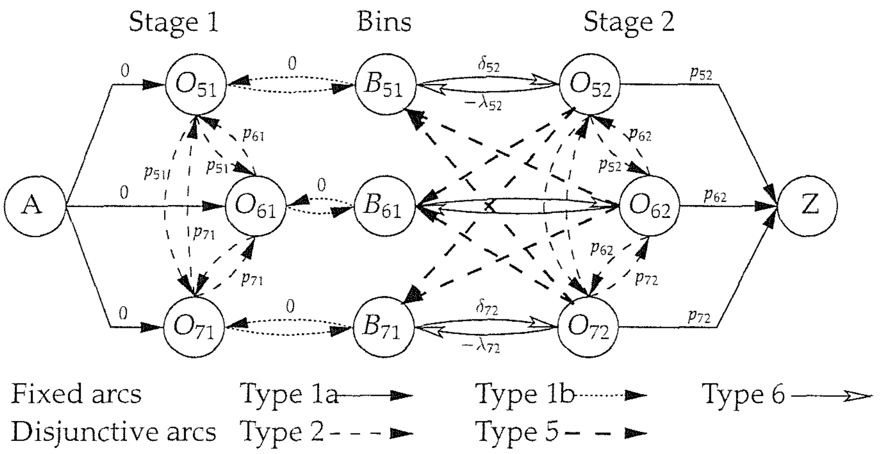


Figure 4.11: Disjunctive graph representation with continuous flow and bins of limited size

Note that there are now type 6 arcs going in both directions, so we have an extended definition:

Type 6 arcs are always present. The arc from B_{jk} to $O_{jk'}$ has length $\delta_{jk'}$, as given in equation 4.2. The arc from $O_{jk'}$ to B_{jk} has length $-\lambda_{jk'}$, as given in equation 4.9.

4.5 Using the disjunctive graphs for scheduling

In this chapter, we started with the disjunctive digraph representation often used in complex scheduling systems for evaluating feasibility and earliest start schedules. We have extended the representation to cover both discrete- and continuous-flow hybrid flow shop systems with inter-process storage. We have also placed limits on the timing of continuous-flow systems with storage bins of limited capacity.

In the graph representations developed in this chapter it is possible to create (positive) cycles by certain selections of job order or allocation. When a positive cycle exists in a directed graph, there is no longest path, since a path of arbitrarily large length can be found by traversing the positive cycle any number of times. When this happens, the suggested schedule which the graph represents is infeasible, and is most likely to be due to a job being too large for the bin in which it is being placed. Where a schedule is found to be infeasible, the only option is to find an alternative schedule, either by changing the processing order of jobs, or by re-allocating jobs to different machines or bins.

The disjunctive graph representations given in §4.2 and §4.4 allow us to model all of the features of the core scheduling model we described in §3.5.

The technique which we will use, and which is set out in detail in chapters 6 and 7, is to use a heuristic algorithm to produce an allocation of operations to machines, and to sequence the operations on the machines. After generating an allocation and sequence, a schedule can be obtained by selecting the appropriate arcs from the disjunctive arc set, and running Dijkstra's algorithm to determine the start time of each operation.

In the algorithm which we develop in the following chapters, we have ignored allocations of operations to bins, since this is outside the scope of this thesis.

Chapter 5

Specialist parallel machines: unit length jobs

5.1 Introduction

There is an extensive literature on scheduling problems on parallel identical machines. Identical machine problems are, generally, the easiest of the parallel machine problems, although the vast majority of the problems of practical interest are **NP**-hard.

In this chapter we consider the situation in which there are restrictions upon which of the parallel machines can process particular jobs. More precisely, if M denotes the set of m identical parallel machines and J a set of n jobs, then each job j has processing time p_j and a set of machines, $M_j \subseteq M$, which may process that job. We refer to the sets M_j as *processing sets*. We wish to allocate jobs to machines and, further, order the jobs on each machine so that some objective function is minimised. Typically, the objective function will depend on the completion times of the jobs, C_j, and sometimes also on the due dates, d_j, and weights, w_j, of the jobs.

It is the presence of processing sets which makes this model distinct from previously studied parallel identical machine problems. We may view this class of problem as a standard unrelated parallel machines scheduling problem with a particular structure imposed upon the processing times of the jobs. A more detailed description of this relationship is given in section 5.2. Brucker, Jurisch and Krämer [8] refer to problems with processing sets as having ***multi-purpose machines***. They examine the computational complexities of a wide range of processing set problems.

We may further modify the parallel machines problem with processing sets by allowing only nested processing sets. A set of sets $\{M_1, \ldots, M_n\}$ is *nested*, as defined in Pinedo [65], if and only if, for all pairs of sets M_j, M_k, one of the following is true:

$$\begin{aligned} M_j &\subseteq M_k, \\ M_k &\subseteq M_j, \text{ or} \\ M_j \cap M_k &= \emptyset. \end{aligned}$$

The model of parallel machines with processing set restrictions arises from the specialist treatment plant at Foster Mills. In particular, the bottleneck production process of steam treatment is carried out by three identical machines which work in parallel. All three machines feed downstream machinery in the main plant, while one of the machines additionally feeds the micro-clean plant. Thus there are some products which can only be made on one of the three machines. The other stages of production exhibit identical parallel processing, each with different configurations of processing sets.

We examine here the problems that may arise from imposing processing set restrictions on jobs to be scheduled on identical parallel machines. We concentrate on the following variants of the problem:

- objective functions: makespan, mean (weighted) completion time, total tardiness, total weighted tardiness, total (weighted) number of late jobs and maximum lateness;
- job lengths: identical ($p_j = 1$);
- processing sets: none, nested and unnested.

The next section is devoted to establishing a few definitions and results about processing sets. In the remainder of this chapter, we examine the computational complexity of a number of variants of the parallel machine scheduling problem with processing set restrictions. Each of the six objective functions listed above is considered. In all cases we consider both nested and general processing sets for unit processing time (i.e. $p_j = 1$ for all jobs j). For each case with nested processing sets and identical jobs we develop a polynomial time optimisation algorithm.

We also discuss solution methods and propose heuristics for some of the problems which are not polynomially solvable. A summary of complexity results is given in table 5.1 at the end of the chapter.

5.2 Preliminary results

An important observation in the classification of parallel identical machine problems with processing sets is that their complexity falls between that of two better-known problems. Firstly, a problem in the class of parallel machine problems with processing sets, $Pm|M_j|\circ$, may be considered to be in the class of processing on unrelated parallel machines, $Rm||\circ$ by putting

$$p_{ij} = \begin{cases} p_j & \text{if } i \in M_j \\ \infty & \text{if } i \notin M_j \end{cases} .$$

Thus $Pm|M_j|\circ$ is no harder than $Rm||\circ$.

Secondly, we may view the problem $Pm||\circ$ as a special case of $Pm|M_j|\circ$, where the M_j are each simply the full set of machines. Thus we have established a clear hierarchy of problems, where $Rm||\circ$ is at least as hard as $Pm|M_j|\circ$, which in turn is at least as hard as $Pm||\circ$. We use both of these results to establish the complexity of the various problems we are considering. A summary of the complexity hierarchy is given in Figure 5.1.

5.3 Makespan

The aim when scheduling to minimise the makespan ($C_{\max}$) is to complete all jobs as soon as possible. Makespan problems on parallel machines have been a source of much research over the last thirty years. One of the earliest papers to look at the problem was that of Graham [30], who developed a simple heuristic

$$\begin{tikzcd}
Pm|p_j = 1|\circ \arrow[r] \arrow[d] & Pm|M_j, p_j = 1|\circ \arrow[r] \arrow[d] & Rm|p_j = 1|\circ \arrow[d] \\
Pm||\circ \arrow[r] & Pm|M_j|\circ \arrow[r] & Rm||\circ
\end{tikzcd}$$

Key: $A \longrightarrow B$ Problem B is at least as hard as problem A

Figure 5.1: Complexity hierarchy of parallel machine scheduling problems

for solving the problem with unit processing times, the *longest processing time first* or *LPT* rule. Jobs are listed in order of decreasing (more precisely, non-increasing) processing times. The schedule is built up by taking one job at a time from the list and placing it in the schedule so as to minimise the resultant makespan. This construction is termed *list scheduling*.

In the first worst-case analysis for any heuristic reported in the literature, Graham demonstrated that the LPT list scheduling heuristic would result in a makespan of at most $\frac{4}{3} - \frac{1}{3m}$ of the optimal makespan. He also showed that any list-processing heuristic would result in a makespan no greater than twice the optimal makespan.

The makespan problem on a fixed number of parallel identical machines is **NP**-hard in the ordinary sense [25]. As a result, most of the attempts at solving the problem have concentrated on finding pseudo-polynomial time solutions or polynomial-time approximation heuristics. In the latter class, the most notable are the MULTIFIT method of Coffman, Garey & Johnson [12], which uses the obvious similarity between this problem and the bin-packing problem, and the 3-PHASE heuristic of França, Gendreau, Laporte and Müller[24]. Another solution method, using the notion of a *dual approximation algorithm* is presented by Hochbaum and Schmoys [38]. Amongst other results, they give an explanation of why it is extremely unlikely (unless **P**=**NP**) that there exists a fully polynomial approximation scheme for this problem with an arbitrary number of machines. However, they do develop $\frac{1}{5}$- and $\frac{1}{6}$- polynomial approximation schemes for finding solutions, and suggest that a $\frac{1}{7}$-approximation scheme could easily be developed.

There has been some work on exact solutions to the makespan problem on parallel machines, most notably Dell'Amico and Martello's proposal of new upper bounds for parallel identical machines, and construction of a branch and bound algorithm [19].

Adding processing set restrictions, we find that the problem remains ordinarily **NP**-hard, even with general processing sets, since there exists a pseudo-polynomial dynamic programming technique for solving the equivalent unrelated machines problem (see Rothkopf [68]). Martello, Soumis and Toth have also worked on the problem with parallel unrelated machines [53], and propose both a branch and bound algorithm and a heuristic which runs in $O(n^2m)$ time. These methods can all be used to solve $Pm|M_j|C_{\max}$.

Observe that in the absence of processing sets, the special case of scheduling jobs with identical processing times on parallel machines, is trivial. Jobs are completely interchangeable, and list scheduling therefore minimises the maximum completion time regardless of the list order.

5.3.1 Nested processing sets

For nested processing sets and unit length processing times, Pinedo [65] gives an $O(n \log n)$ list scheduling algorithm, the LEAST FLEXIBLE JOB (LFJ) algorithm. The *flexibility* of a job in this context is the number of machines on which the job may be processed (i.e. the size of its processing set). The nature of the scheduling algorithm is greedy in that it incorporates one job at a time, optimising the objective function at each iteration. In the case of identical parallel machines, the convention is typically to select a machine with least total processing. The problem of un-nested processing sets is not studied in the literature. We now establish a polynomial time algorithm for the makespan objective when processing times are identical (and hence may all be taken to be 1).

5.3.2 Generalised processing sets

Minimising makespan on parallel machines is more difficult with processing sets. We shall now show how the problem with identical job processing times, $Pm|M_j$, $p_j = 1|C_{\max}$, can be solved in polynomial time by reducing it to a network flow problem.

A network flow diagram, as illustrated in Figure 5.2, is constructed between a single source node and a single sink node as follows. First establish a set of nodes (labelled J_i) corresponding to the individual jobs, each of which has a single arc from the source node with a maximum capacity of 1. Secondly, insert a set of nodes (labelled T_i) corresponding to the machines, each of which has an arc to the sink, with a maximum capacity of C. The arcs between the job nodes and the machine nodes indicate which jobs may be processed on which machines and are uncapacitated. In a feasible solution to the maximum network flow problem, a flow from a job node to a machine node indicates that the job is processed on that machine. Thus, the value of the maximum flow through the network corresponds to the number of jobs allocated in the corresponding solution.

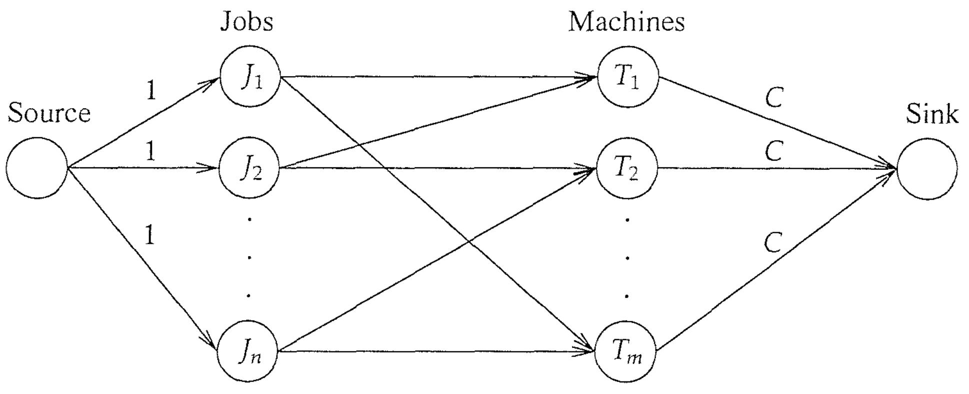


Figure 5.2: An example of the network flow formulation for $Pm|M_j, p_j = 1|C_{\max}$

Now, a solution to the original problem can be obtained by finding the minimum integral value of C for which there exists a solution to the flow problem with flow equal to n. For a given value of C, a maximum integral flow in the network corresponds to a feasible allocation of jobs to machines in the original problem with makespan no greater than C. The minimum value of C can be found by using a binary search on a suitable interval.

Finding a feasible maximum flow in the resulting network problem takes $O((n+m)nm\log(\frac{n}{m}+\frac{m}{n}))$ time using the push-relabel algorithm of Goldberg and Tarjan [29]. Thus, given any upper bound B on the value of $C_{\max}$, we can find a solution to $Pm|M_j, p_j = 1|C_{\max}$ in $O((n+m)nm\log(\frac{n}{m}+\frac{m}{n})\log B)$ time, by binary search on the interval $[0, B]$. Since every job has length 1, n is an upper bound on the value of $C_{\max}$, giving us a complexity of $O((n+m)nm\log(\frac{n}{m}+\frac{m}{n})\log n)$.

5.4 Mean completion time

The classical mean completion time objective ($\sum C_j$) problems on parallel machines are easier to solve than the makespan objective. Mean completion time problems are polynomially solvable for both identical and unrelated machines. Conway, Maxwell and Miller [13] give a modification of the SPT (shortest processing time first) rule with list scheduling, which gives an optimal solution in $O(n\log n)$ time for identical parallel machines. List scheduling with arbitrarily ordered jobs is sufficient for the special case when all jobs have identical processing times, since jobs are then interchangeable with respect to the sum of their completion times.

Horn [40] solves the unrelated parallel machine problem by reduction to an assignment problem of size at most $nm+n$ nodes. The algorithm runs in $O((nm)^3)$ time.

5.4.1 General processing sets

To solve the mean completion time problem on parallel identical machines with general processing set restrictions, we may use the reduction to unrelated parallel machine problem without processing sets, established in §5.2. Horn's graph construction is equivalent to the following bipartite graph for identical parallel machines where jobs have processing sets.

Take n source nodes, J_j, corresponding to individual jobs, j and a destination node, T_{it} for each combination of machine, i (between 1 and m) and position t between 1 and n. For each job j, an arc is inserted between node J_j and each node T_{it}, for $i = 1, \ldots, n$, for machines i in processing set M_j. A cost

$$c_{jit} = (n-t+1)p_{ij}$$

is assigned to the arc between nodes J_j and T_{it}, if it exists. This models the fact that, in the $\sum C_j$ objective function, the first job of, say, r jobs on a machine has its processing time counted r times, the second job is counted $r-1$ times, and so on until the last job makes a single contribution to the total. In a minimum cost matching, jobs are placed in early positions on machines. This construction is illustrated in Figure 5.3. Observe that nodes are labelled as "nth last position", rather than "first position" since a minimum cost matching will, evidently, match up later positions in the graph without a gap, reserving any unmatched positions to be consecutive from nth last position onwards.

The cost of this matching gives the total cost of job completion times, and each job is matched to a place in the schedule on a particular machine. A minimum cost matching between the two partitions of the graph is then found using the so-called Hungarian Algorithm [48], which gives an optimal schedule from the job positioning on each machine, ignoring gaps and its minimum cost. This construction differs from Horn's only in that his graph contains all possible arcs and when $M_j \notin M^{(k)}$, costs $c_{ijk} = \infty$ are assigned. The use of the Hungarian algorithm enables the problem to be solved in $O((nm)^3)$ time.

5.4.2 Nested processing sets

For the nested case, we establish a much simpler optimal algorithm which runs in $O(nm)$ time, the SMALLEST PROCESSING SET FIRST or SPSF list scheduling algorithm. The algorithm simply schedules a currently unscheduled job with the smallest processing set on a machine (in its processing set) with the earliest free time. A formal description of SPSF is given below. Note that the time A_i, used below, is the point at which machine i becomes free to process another job.

Algorithm SMALLEST PROCESSING SET FIRST (SPSF)

```
1. List jobs
      Index the jobs such that $M_j \subset M_l \implies j < l \quad (1 \leq j, l \leq n)$, breaking ties
      arbitrarily
2. Schedule jobs in list order
      For each job $j$ from 1 to $n$
```

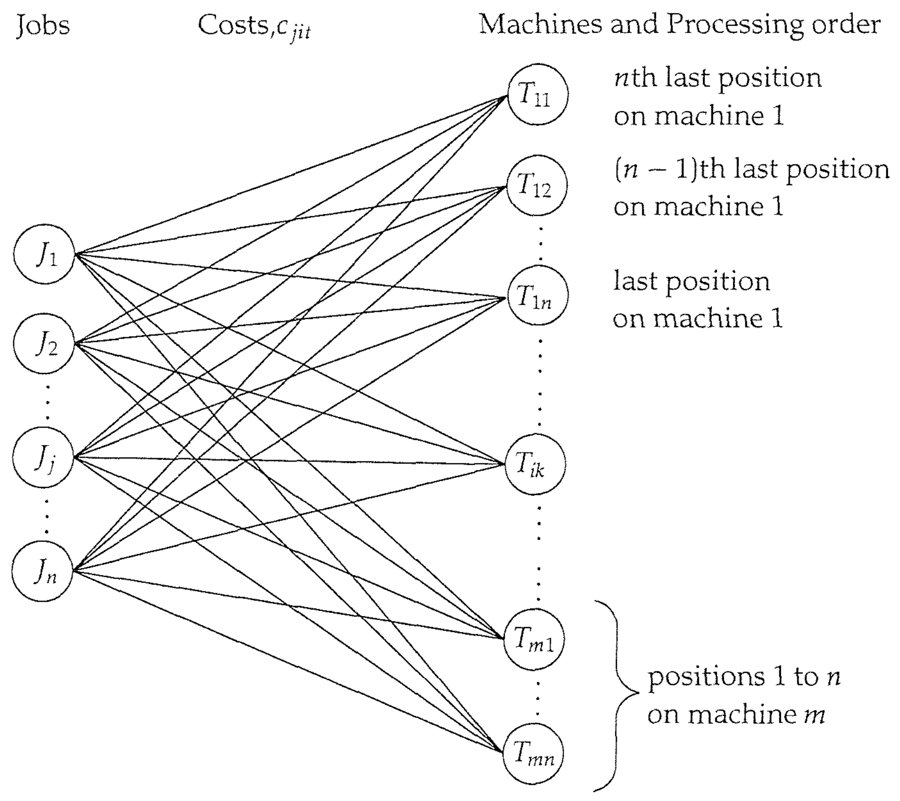

Jobs
Costs, c_{jit}
Machines and Processing order
J_1
J_2
J_j
J_n
T_{11}
T_{12}
T_{1n}
T_{ik}
T_{m1}
T_{mn}
nth last position on machine 1
$(n-1)$th last position on machine 1
last position on machine 1
positions 1 to n on machine m

Figure 5.3: An example of the bipartite graph matching formulation for minimising total completion time on unrelated parallel machines

```
        Let $i^*(j)$ be a machine such that $A_{i^*(j)} = \min_{i \in M_j} A_i$
        Schedule job $j$ on machine $i^*(j)$
    End For
```

Theorem 5 *The* SPSF *algorithm generates optimal schedules for minimising the sum of completion time for jobs with unit processing times on parallel identical machines with nested processing set restrictions in $O(nm)$ time.*

Proof: Assume that there is an optimal schedule S^* which could not have been generated by any implementation of the SPSF algorithm. To prove the theorem, we shall convert it into one produced by the SPSF algorithm without increasing the objective value. Take a schedule, S, produced by SPSF. The jobs in S on each machine are in increasing index order, by construction. We may re-order the jobs in S^*, within each machine in non-decreasing index order without affecting $\sum_j C_j$, since the jobs are all of unit length.

Let J_i^* and J_i denote the set of jobs scheduled on machine i in S^* and S respectively, for $1 \leq i \leq m$. Let j be the job with the smallest index which is not assigned to the same machine in S^* as it is in S, but rather to machines i^* and i respectively, say. If none such exists then we have the desired contradiction. Otherwise, consider the partial schedule S' of S^* (and of S) consisting of all jobs with indices smaller than j. Denote the completion times of machine i and i^* in S' by C_i' and C_{i^*}' respectively. Completion time C_{i^*}' cannot be smaller than C_i', or job j would have been allocated to machine i^* in preference to machine i by SPSF. Thus $C_{i^*}' \geq C_i'$.

Machine i must have some job, ℓ say, with index greater than j in S^*, since otherwise job j could be moved to machine i, which would reduce the $\sum C_j$, giving a contradiction. Swapping jobs j and ℓ in S^* causes no change in $\sum C_j$, as all jobs are the same length and $M_\ell \supseteq M_j \supseteq \{i, i^*\}$, by virtue of the fact the the processing sets are nested. Job j is now on the same machine in both S^* and S.

We can repeat the process with the transformed schedule S^*. With each successive iteration, the new lowest-index misplaced job in S^*, j, increases. Thus, eventually no such job exists and S^* has been transformed into the schedule S produced by the SPSF algorithm.

With regard to the running time of the algorithm, the first part of the algorithm may be accomplished by sorting the jobs according to the size of their processing sets. This takes $O(nm)$ time to calculate the size of the sets (counting at most m machines for each of n jobs), and $O(n)$ time to perform the sort using a radix sort: there are only m possible sizes for the processing sets, and hence the jobs may be sorted into m array-indexed bins (since sorting by size of processing set fulfils the ordering condition of step 1). The second part of the algorithm may be executed in $O(nm)$ operations, since the loop is executed at most n times, and the process of finding the machine with the earliest finish time takes at most $O(m)$ operations, to examine each machine. Scheduling a job on a machine is an $O(1)$ operation. Therefore, the whole process is limited in its computational efficiency by the first step, and takes $O(nm)$ time to execute. $\square$

5.5 Total weighted tardiness

The total weighted tardiness ($\sum w_j T_j$) problem for identical parallel machines with identical jobs lengths is polynomially solvable. Lawler [49] gives a reduction of this problem to a transportation problem, which can be solved in $O(\frac{n^4}{m}\log n)$ time using Orlin's version of the Scale-and-Shrink algorithm [62].

5.5.1 General processing sets

Lawler's formulation for unit length jobs does *not* apply when we impose processing set restrictions, as it takes no account of the allocation of jobs to individual machines. We must look for an alternative formulation. Consider a bipartite matching formulation similar to that in §5.4.1, constructed as follows. Take n nodes, $J_j\,(1 \leq j \leq n)$, on the left-hand side of a bipartite graph, where node J_j corresponds to job j. On the right hand side of the graph, illustrated in Figure 5.4, there are nm nodes, $T_{it}\,(1 \leq i \leq m; 1 \leq t \leq n)$, where T_{it} corresponds to the time slot t on machine i. Each job needs to be placed on precisely one machine in its processing set, in one time slot. Thus, arcs are established between a node J_j and each node T_{it} for $i \in M_j$ and $1 \leq t \leq n$. A bipartite matching therefore provides a feasible schedule for the original problem.

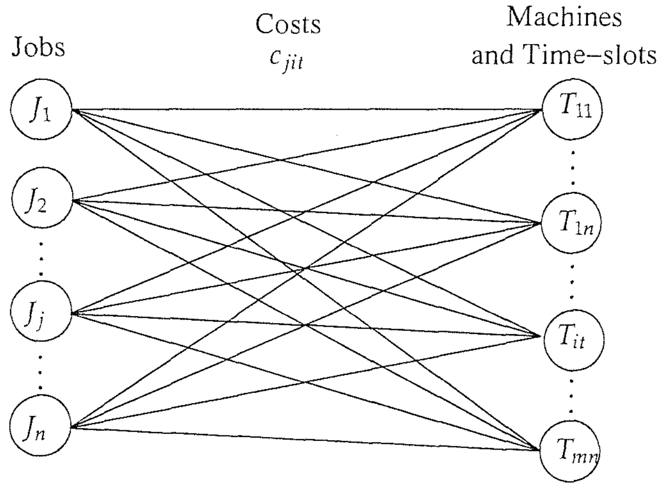


Figure 5.4: An example of a bipartite matching formulation for total (weighted) tardiness with identical length jobs with processing sets

We now show that an optimal schedule for the parallel machine scheduling problem with processing sets is given by a minimum cost matching for the bipartite graph depicted in Figure 5.4 when the arc joining J_j and T_{it} is given weight

$$c_{jit} = w_j \max\{0, t - d_j\} \quad \text{if } i \in M_j.$$

The weight c_{jit} corresponds to the weighted tardiness cost of the job j finishing on machine i at the given time, t. Thus a minimum cost matching for the bipartite graph gives a feasible schedule with minimum possible total weighted tardiness. Such a matching may be found by use of the Hungarian Algorithm [48], which can be implemented to run in $O(n^3m^3)$ time.

An alternative construction using a complete graph with a prohibitively high cost for allocating jobs to a machine upon which it cannot be done, i.e.

$$c_{jit} = \infty \quad i \notin M_j,$$

gives an identical result.

Observe that the above construction can also be used to give optimal solutions for unweighted tardiness problems, with processing sets, by simply setting each $w_j = 1$.

5.6 Weighted mean completion time

As a corollary to the method presented in the previous section, we observe that the weighted mean completion time for unit-length jobs on parallel identical machines with processing sets can be solved with a similar construction. We use a complete bipartite graph with n nodes, J_j, on the left, and nm nodes, T_{it} on the right. The arc between J_j and T_{it} has weight c_{jit}, where

$$c_{jit} = \begin{cases} w_j t & \text{if } i \in M_j \\ \infty & \text{if } i \notin M_j \end{cases}.$$

This allows us to solve our unit-length job problem with weighted mean completion time in $O((nm)^3)$ time.

5.7 Number of late jobs

Minimising the (unweighted) number of late jobs is generally a difficult problem, although easier than the weighted case. With jobs of general length on a single machine, the problem is polynomially solvable using Moore's Algorithm [56]. On parallel machines, the problem becomes **NP**-hard, as it may be reduced to PARTITION [51].

With unit length processing times and no processing sets, minimising the unweighted number of late jobs on parallel identical machines can be solved in $O(n)$ time using an algorithm of Monma[55]. However, Monma's algorithm does not necessarily produce an optimal schedule when applied to problems with nested processing sets. We develop below an efficient optimal algorithm for minimising the unweighted number of late jobs with unit-length jobs and nested processing sets.

5.7.1 Nested processing sets

To solve the problem of minimising the unweighted number of late jobs with nested processing sets and unit-length processing times (in the standard nomenclature, $Pm|p_j = 1, M_j \text{ nested}| \sum U_j$), we now present a list scheduling algorithm, called SPS-List. Algorithm SPS-List fits jobs into time-slots in a similar way to Monma's algorithm.

The algorithm given below handles jobs for one processing set at a time in a sequence preserving the partial order of processing sets. Within a processing set, $M^{(q)}$, jobs are scheduled in free time-slots (i.e. when there are fewer than $|M^{(q)}|$ jobs in the time-slot) on or before their respective due dates, if possible. When all time-slots before the due date of a job are full, the job is discarded to be scheduled late.

Algorithm SPS-LIST

1. *List jobs*

   Reindex the jobs so that if $M_j \subset M_\ell$ then $j < \ell$

2. *Schedule jobs in list order, on-time where possible*

```
For each job, $j$, in index order
    Schedule the job to complete at time $d_j$ on some machine in $M_j$, if possible,
    Otherwise, schedule job $j$ in the latest earlier free space on any machine in $M_j$, if there is one.
End For
```

3. *Schedule the remaining late jobs*

   Schedule the remaining, discarded, jobs in any free time-slot, and count them to find the number of late jobs.

Theorem 6 *For the problem $Pm|p_j = 1, M_j$ nested$|\sum U_j$, Algorithm SPS-List is optimal.*

Proof: Suppose that S is not optimal. Take an optimal schedule, S^*. Then S^* has more on-time jobs than S. Let $S^{*(q)}$ and $S^{(q)}$ denote the sub-schedules of S^* and S, respectively, consisting of on-time jobs in the set $\{j|M_j \subseteq M^{(q)}\}$, for $q = 1, \ldots, Q$. We shall derive a contradiction by proving the stronger property: $S^{*(q)}$ need not have any more jobs on-time than does $S^{(q)}$ in any given time-slot, for any value of q, not just for Q (the whole problem) itself. The proof proceeds by induction on $q = 1, \ldots, Q$, and at each induction step S^* may be transformed, but no additional jobs are made late. Formally, we may be considered to start our induction step at $q = 0$ and consider $M^{(0)}$ to be the empty set which trivially satisfies the above property.

Let q be a smallest processing set index for which there are more on-time jobs in schedule $S^{*(q)}$ than in $S^{(q)}$ in any given time-slot, and let t be the earliest such time-slot. Let j_1 denote a job scheduled at time t in $S^{*(q)}$ and not in $S^{(q)}$. Then, either $M^{(q)}$ has no processing subsets, or else for all processing subsets q', $q' < q$ and hence by induction, $S^{*(q')}$ has no more on-time jobs than $S^{(q')}$ in any time-slot. Thus, there is such a job, j_1, for which $M_{j_1} = M^{(q)}$. Since slot t is not fully occupied in $S^{(q)}$, and t is less than d_{j_1}, when job j_1 is scheduled in Step 2 of algorithm SPS-List, it cannot be made late and therefore must be scheduled on-time in S, at time t_1 say, later than t.

By induction, $S^{*(q)}$ has no more on-time jobs at time t_1 in proper subsets of $M^{(q)}$. Thus there is at least one place at time t_1 in $S^{*(q)}$ for a job in a processing set $M^{(q)}$ or larger. If there is a free space in $S^{*(q)}$ at time t_1, then move job j_1 to t_1 in schedule $S^{*(q)}$. This may require another job, in a larger processing set, moving from time t_1 earlier, to time slot t in S^*. If there is no such free space in $S^{*(q)}$ at time t_1, then there is at least one job j_2, say, scheduled in $S^{*(q)}$ and not in S at time t_1 with $M_{j_2} = M^{(q)}$. Moreover, job j_2 cannot be late in S since there is a time-slot at time t, and $t < t_1 \leq d_{j_2}$, in which it could have been scheduled by algorithm SPS-List. Therefore, j_2 must have been scheduled on-time in $S^{(q)}$ at a later time than t, t_2 say.

By repeating this process (on job j_2 and time-slot t_2 in place of job j_1 and time-slot t_1), as many times as required, $S^{*(q)}$ is transformed into a feasible schedule with the same number of on-time jobs at time t as $S^{(q)}$, and no additional late jobs.

By induction on t, the property we claimed holds for all time slots for the given value of q. The proof is now complete by induction on q, up to $q = Q$. □

Lemma 1 *Algorithm SPS-List may be implemented to run in $O(nm + n\log n)$ time.*

Proof: In part 1 of the algorithm, indexing the jobs in processing set order is an $O(nm)$ process, as described in theorem 5. In part 2, we maintain two data structures: an array of the free space in each individual time-slot (the *status array*), and an ordered, balanced tree of the time-slots which have free space in them (the *free space tree*). The outer loop of part 2 executes $O(n)$ times. Inside this loop, the status of a given time-slot may be determined in $O(1)$ time by accessing the array directly by index. If the job can be scheduled at time d_j, then it is added to the list of jobs in that time-slot in $O(1)$ time. If the job to be scheduled cannot be placed at time d_j, then the latest earlier free space can be found in $O(\log n)$ time by searching the free space tree. (Although a simple binary-tree implementation of the free-space tree has a best case of $O(\log n)$ for both searching and deleting, the worst case is $O(n)$, when the tree becomes completely unbalanced. However, by using slightly different implementation details to make a *red-black* tree, both searching and deleting become $O(\log n)$ in the worst case[72]). When a job is scheduled, the status array is updated, taking $O(1)$ time. If the updated entry in the status array shows the time-slot has become empty, then the relevant entry in the free space tree can be found and removed in $O(\log n)$ time.

In part 3 of the algorithm, late jobs can be placed on any machine in any time-slot. At most $O(n)$ jobs will remain unplaced, and finding a free space in the schedule takes $O(1)$ time (the root entry in the tree can always be found in $O(1)$ time). Placing a job and updating the tree takes $O(\log n)$ time again. Thus, the second phase of the algorithm dominates the first and third phases, and the algorithm as a whole takes $O(nm + n\log n)$ time to run. □

Note that the use of a tree structure in the above implementation of the algorithm parallels the use of similar structures in certain guaranteed $O(n\log n)$ sorting algorithms, such as the heap sort.

5.8 Weighted number of late jobs

When minimising the weighted number of late jobs, we consider three different situations: no processing sets, general processing sets, and nested processing sets. We present here computationally efficient methods of solving each of the above problems. We first present an algorithm to solve the problem of minimizing the weighted number of late jobs on parallel identical machines with unit-length jobs.

5.8.1 Weighted number of late jobs with no processing sets

The algorithm considers jobs in descending order of weight, so that the most important (highest weight) are allocated to time-slots first. Jobs are placed in the latest free time-slot in which they are not late. If a job cannot be fitted before its due date, it is reserved to be scheduled at an arbitrary time after all the jobs have been considered, since the amount of time by which it is late does not affect the objective function.

Algorithm SMALLEST WEIGHT LIST SCHEDULING (SW-LIST)

1. *List jobs*

   Index jobs in order of non-increasing weight

```
2. Schedule jobs early where possible
     For jobs $j$ from 1 to $n$
       If there are fewer than $m$ jobs already scheduled to complete at time $d_j$,
       schedule job $j$ to complete at time $d_j$.
       Otherwise, schedule job $j$ in the latest earlier free space on any machine, if there is one.
     End For

3. Schedule remaining (late) jobs
     Place the remaining, unscheduled, jobs in any free time-slot on any machine, and calculate the weighted sum of late jobs.
```

Theorem 7 *The Smallest Weight List Scheduling Algorithm, SW-List, is optimal for solving $Pm|p_j = 1|\sum w_jU_j$.*

Proof: Observe that steps 2 and 3 of SW-List are just Monma's algorithm[55], and that they are applied to a list specified by step 1. Take a schedule S produced by algorithm SW-List. From Monma's result [55], no other schedule will accommodate more jobs on time than S. Therefore a saving can only be made by scheduling a currently late job in S, j say, on time in place of an on-time job with lesser weight, possibly accompanied by some rescheduling of the other on-time jobs of S.

Consider such a rescheduling, in order to show that no such job exists. In it a late job j dislodges an on-time job which we shall denote k_1. Job k_1 must have been considered before job j by the algorithm, since otherwise job j would have been placed in that time-slot originally. Thus, $k_1 < j$ in Step 1 and $w_{k_1} \geq w_j$. Therefore, this move can only be of advantage if job k_1 is rescheduled on-time but after time d_j, hence $d_j > d_{k_1}$. But there cannot be free time-slot before d_{k_1} later than k_1 was placed in Step 2. job k_1 will have to dislodge Therefore, another on-time job, k_2 say, which was considered by Step 2 before it and therefore has a weight at least as large. Since no more jobs can be made on time, eventually the rescheduling process will result in a job, k_r say, being made late. But from the above argument $w_{k_r} \geq w_{k_{r-1}} \geq \ldots \geq w_{k_1} \geq w_j$ and thus no saving can have been achieved by this rescheduling, providing the required contradiction. □

Lemma 2 *The SW-List algorithm may be implemented to run in $O(n \log n)$ time.*

Proof: The algorithm may be implemented identically to the SPS-LIST algorithm, with the exception that the initial ordering in part 1 of the algorithm is ordered by different criteria. In this case, the sorting may be performed in $O(n \log n)$ time (since there is no overhead for handling processing sets). The remainder of the algorithm is identical to parts 2 and 3 of the SPS-LIST algorithm, thus giving an $O(n \log n)$ implementation of the algorithm. $\square$

5.8.2 General processing sets

The SW-List algorithm does not allow for processing sets. However, when processing set restrictions are imposed and jobs have unit length, we can use a modified version of the bipartite matching solution from §5.5.1. In order to minimise the weighted number of late jobs rather than the weighted tardiness, we modify the weights of the arcs c_{jit} to be

$$c_{jit} = \begin{cases} 0 & \text{if } i \in M_j \text{ and } t \leq d_j \\ w_j & \text{if } i \in M_j \text{ and } t > d_j \\ \infty & \text{if } i \notin M_j \end{cases} .$$

This formulation as a bipartite matching problem allows the minimisation of the weighted number of late jobs with unit-length jobs and general processing sets to be solved in $O((nm)^3)$ time.

5.9 Maximum lateness

For unit length jobs and identical parallel machines, the algorithms for minimising the number of late jobs presented in Section 5.7 may be employed to solve the equivalent maximum lateness problem, as we shall now demonstrate. The construction is standard, and has broader applications.

Consider any problem $\mathcal{P}$, with m parallel identical machines, n jobs with processing times p_j $(1 \leq j \leq n)$, due dates d_j and processing set M_j. The maximum lateness $L_{\max}$ of $\mathcal{P}$ can be minimised by the following procedure.

Construct a related instance $\mathcal{P}'(z)$ for a given value of z with m parallel identical machines and n jobs by setting $p'_j = p_j$, $M'_j = M_j$, and $d'_j = d_j + z$ for which the number late jobs is to be minimised. Then, solve $\mathcal{P}'(z)$ to find the minimum number of late jobs $(\sum U_j)$. The minimum value of z for which $\sum U_j$ is zero gives the minimum value of $L_{\max}$ for $\mathcal{P}$, and a solution to $\mathcal{P}'(z)$ is also an optimal solution to $\mathcal{P}$.

The search for the optimal z in the above procedure can be performed by binary search in $O(\log(U - L))$ operations, where U and L are an upper and lower bound respectively on the value of $L_{\max}$. In the case of unit-length processing times with processing sets, the problems (both with and without nesting) are polynomially solvable, since suitable bounds are $U = n - \min d_j$ and $L = 1 - \max d_j$.

Thus, in particular for minimising $L_{\max}$ on parallel identical machines with unit length jobs and general processing sets, we can construct an $O((nm)^3 \log D)$ algorithm (where $D = U - L = \max d_j - \min d_j + n - 1$) using the reduction to a bipartite-matching solution for $Pm|M_j, p_j = 1|\sum U_j$ from Section 5.7. Similarly, with nested processing sets, we can construct an $O(nm \log n \log D)$ algorithm using the more efficient SPS-LIST algorithm from the same section.

5.10 Conclusion

In conclusion, we have found that the parallel identical machine problem with processing sets, $Pm|M_j|\circ$ in the standard three-field notation of scheduling theory, lies in complexity between the parallel identical machine problem without processing sets $(Pm||\circ)$ and the parallel unrelated machine problem $(Rm||\circ)$. Any $Pm|M_j|\circ$ problem may thus be solved (approximated) by applying a known solution (approximation) method for the equivalent $Rm||\circ$ problem. We give a summary of the complexity of the different problem variants covered in this chapter in Table 5.1.

The case where the processing sets are nested $(Pm|M_j \text{ nested}|\circ)$ appears to be easier to solve than the case where the processing sets are general, as the additional structure of the nesting allows the construction of more efficient algorithms. This is particularly evident in the cases with identical length jobs, where there are simple $O(n \log n)$ algorithms for solving the problem for the makespan and mean completion time objective functions, compared to the $O((n+m)nm \log(\frac{n}{m} + \frac{m}{n} \log n)$ algorithm for makespan, and the $O((nm)^3)$ algorithm for mean completion time.

In general, as with normal parallel machine problems, the only polynomially solvable problems are those with identical job lengths, although the mean completion time objective function has polynomial solutions even for general job lengths. We have presented in this chapter a polynomial algorithm for solving the parallel identical machine problem with processing sets and identical length jobs for the mean completion time, where the processing sets are nested. We have also presented polynomial algorithms for solving the equivalent makespan, total (weighted) tardiness and total (weighted) number of late jobs objective functions for general processing sets and identical length jobs. We give a brief summary of the solution methods for each problem variant in Table 5.2.

Objective	Processing sets		
Function	None	Nested	General
$C_{\max}$	n	$n \log n$	$(n+m)nm \log(\frac{n}{m} + \frac{m}{n}) \log n$
$\sum C_j$	n	nm	$(nm)^3$ *
$\sum w_j C_j$	$n \log n$	$(nm)^3$ *	$(nm)^3$ *
$\sum T_j$	$n \log n$	$(nm)^3$ *	$(nm)^3$
$\sum w_j T_j$	$(nm)^3$ or $\frac{n^4}{m} \log n$	$(nm)^3$ *	$(nm)^3$
$\sum U_j$	n	$nm + n \log n$	$(nm)^3$
$\sum w_j U_j$	$n \log n$	$(nm)^3$	$(nm)^3$
$L_{\max}$	$n \log n$	$nm + n \log n \log D$	$(nm)^3 \log D$

Table 5.1: Computational complexity of parallel machine problems

* – More efficient algorithms may exist

$D = \max d_j - \min d_j + n - 1$

Obj.	Processing sets		
Function	None	Nested	General
$C_{\max}$	Arbitrary	LFJ [65]	Flow formulation §5.3.2
$\sum C_j$	Arbitrary	SPSF §5.4.2	Assignment [40]
$\sum w_j C_j$	Largest Weight First	Bipartite matching §5.7	
$\sum T_j$	EDD	Bipartite matching §5.7	
$\sum w_j T_j$	Transportation [49] or matching §5.7	Bipartite matching §5.7	
$\sum U_j$	Monma [55]	SPS-List §5.7.1	Bipartite matching §5.7
$\sum w_j U_j$	SW-List, §5.8.1	Bipartite matching §5.7	
$L_{\max}$	EDD	As for $\sum U_j$ with binary search	

Table 5.2: Algorithms for parallel machine problems

Chapter 6

Specialist parallel machines: general length jobs

In chapter 5, we developed algorithms for parallel machine problems with unit-length jobs and specialist machines. In this chapter, we examine the problem of scheduling jobs of general length in similar circumstances. We concentrate on minimising the maximum lateness objective function, $L_{\max}$. We first present Jackson's heuristic, an effective scheduling heuristic for the problem of parallel machines without processing set restrictions taken from the literature. We then develop several different modifications of the heuristic which aim to solve the extended problem with processing set restrictions. We perform no direct comparison of the effectiveness of the various parallel-machine heuristics themselves, but we use these algorithms later in chapter 7, and they are tested in the experiments of chapters 8 and 9.

6.1 Heuristics

Minimising maximum lateness on parallel machines with release dates, but with no processing set restrictions is a strongly **NP**-hard problem, since the single-machine problems is strongly **NP**-hard[52]. To solve the problem, three main algorithms have prominence in the literature: Jackson's scheduling heuristic, Carlier's branch-and-bound method[10], and the improved branch-and-bound algorithm of Gharbi and Haouari[27]. Although the two branch-and bound methods are exact, and are highly efficient in most cases, we do not consider them here. This is primarily for pragmatic reasons: we were developing these algorithms for use in the hierarchical decomposition framework in chapter 7, and development, implementation and testing of a branch-and-bound solution scheme was considered to be too complex and time-consuming given the already complex infrastructure required by our hierarchical decomposition framework.

6.1.1 Earliest due date heuristic

We first present a simple earliest due date algorithm. The algorithm simply schedules the unscheduled job with the earliest due date on the next available machine. If the job must be delayed in order to meet its release date constraint, then it is. We present the algorithm below:

Algorithm EARLIEST DUE DATE (EDD)

```
1. Initialise
      Let $J$ be the set of unscheduled jobs. Initialise $J$ to contain all jobs.

2. Process the jobs
      While $J$ is not empty
        Select the machine, $i$, which finishes earliest.
        Let $A$ be the set of jobs in $J$ which can be processed on machine $i$.
        If $A$ is empty, then set the available time of machine $i$ to infinity, and
        restart the loop.
        Schedule the job in $A$ with the earliest due date on machine $i$, delaying
        the job to meet its release date constraint if necessary.
      End While
```

The above algorithm may be implemented to run in $O(n^2m)$ time, and has the advantage of being extremely simple to implement. However, there is a heuristic for the problem without processing sets (Jackson's heuristic) which is much more effective than a simple EDD rule.

6.1.2 Jackson's scheduling heuristic

Jackson's scheduling heuristic[44] is the earlier-developed of the two main methods for solving $Pm|r_j|L_{\max}$. The heuristic has a peculiar property with respect to maximum lateness problems on single or parallel machines, in that it appears to be fundamentally "right", in some sense, for solving the problem[34]. Despite being an approximate heuristic for a strongly **NP**-hard problem, it performs very well in general when solving the problem. It forms the basis of several other scheduling methods (including Carlier's method, below, and Hall and Shmoys' approximation schemes [34] for the single-machine problem).

The heuristic is a simple list scheduling rule, which can be summed up in one sentence: *Schedule the available job with the earliest due date.* More formally, we present the algorithm below. Note that the *available time* of a machine is the completion time of the (current) last job on the machine.

Algorithm JACKSON (WITHOUT PROCESSING SETS)

```
1. Initialise
      Let $A$ be a queue of the available jobs, ordered by due date. Initialise $A$ to
      be empty.
      Let $J$ be the set of unscheduled unavailable jobs. Initialise $J$ to contain all
      jobs.

2. Process the jobs
      While $J$ is not empty
        Select the machine, $i$, which finishes earliest.
        If $A$ is empty, set the available time of machine $i$ to that of the job in $J$
        with the earliest release date.
        If there are jobs in $J$ with release dates equal to or earlier than the avail-
        able time of machine $i$, move those jobs to the available queue, $A$, en-
        suring that the queue remains sorted by due date.
        Schedule the first job in $A$ on machine $i$.
      End While
```

The above heuristic may be implemented to run in $O(n\log(n+m))$ time. At each iteration, the algorithm schedules a job, so the outer loop of the algorithm is $O(n)$. Inside the loop, the earliest available machine is determined. By keeping a balanced tree of machines in available-time order, this may be found in $O(1)$ time. Maintaining the tree takes $O(\log m)$ time at each iteration, as a machine must be removed and re-inserted into the tree in each iteration. Finally, the queue A may be maintained and used in $O(\log n)$ time. Insertion into and removal from the queue may be performed in $O(\log n)$ time by using an appropriate balanced-tree implementation (see Sedgwick[72]).

6.1.3 Jackson's heuristic for processing sets

Jackson's heuristic does not take account of processing set restrictions – therefore it may attempt to schedule a job on a machine which cannot process it. We now present a modification of Jackson's method to schedule jobs with processing set restrictions which ensures that the additional constraints of processing sets are honoured. We consider at each iteration only those jobs which can be scheduled on the earliest available machine. If there are no jobs remaining to go on the machine, the machine is removed from the problem (or its "earliest available time" is set to infinity):

Algorithm JACKSON FOR PROCESSING SETS (JPS)

```
1. Initialise
      Let $J$ be the set of unscheduled unavailable jobs. Initialise $J$ to contain all
      jobs.

2. Process the jobs
      While $J$ is not empty
        Select the machine, $i$, which finishes earliest.
        If there are no available jobs, set the available time of machine $i$ to that
        of the job in $J$ that may be scheduled on $i$ with the earliest release date.
        Let $A$ be the set of unscheduled jobs with release dates earlier than the
        earliest available machine, and which can be scheduled on machine $i$.
        If $A$ is empty, set the available time of machine $i$ to infinity, and start
        the loop again.
        Schedule the job in $A$ with the earliest due date on machine $i$.
    End While
```

The algorithm for processing sets described can be implemented to run in $O(n^2 + m \log m)$ time. The outer loop of the algorithm executes $O(n + m)$ times, since it either schedules a job or sets a machine start time to infinity. By holding a balanced tree of the machines in available-time order, the inside of the loop takes $O(\log m)$ time to find the earliest available machine, and $O(n)$ time to construct the set A and find the job in it. Thus, the overall running time of the algorithm is $O((n + m)(n + \log m)) = O(n^2 + m \log m)$.

6.1.4 Nested Jackson's Heuristic

The JPS algorithm described above attempts to retain the behaviour of Jackson's heuristic, whilst taking account of the limitations in processing sets. However, it does not perform well the "load balancing" function which is seen in other algorithms for nested processing sets (see e.g. the SPS-LIST algorithm of §5.7.1). We therefore seek an algorithm which uses the load-balancing of a nested structure, but also uses the algorithm-specific power of Jackson's heuristic.

To accomplish the above goal, we employ a nested structure, at each iteration fixing the machine allocations (but not the scheduling times) of the jobs which have already been placed at an earlier stage. This "nested" version of the algorithm is presented below:

```
Algorithm NESTED JACKSON (NJ)

  1. Initialise

      Index the processing sets to preserve the partial order of nested sets. Thus,
      if $M^{(q)} \subset M^{(t)}$ then $q < t$.

  2. Process the jobs

      For $q = 1, \dots, Q$
        Let $J$ be the set of jobs which have processing sets contained in $M^{(q)}$.
        Run JPS on the set of jobs $J$, and the machines in $M^{(q)}$
        For every job in $J$, set its processing set to the machine on which it was
        scheduled
    End For
```

Since JPS runs in $O(n^2 + m \log m)$ time, and it is used $O(m)$ times, this algorithm will run in $O(n^2 m + m^2 \log m)$ time. The processing set ordering at the beginning of the algorithm is dominated by the main loop of the algorithm.

Unfortunately, both the JPS and the NJ algorithm fall foul of the following simple instance. Take a problem with two parallel machines, and two jobs. Job 1 has processing time $p_1 = t$, due date $d_1 = t + \epsilon$, release date $r_1 = \epsilon$, and processing set $M_1 = \{1\}$. Job 2 has processing time $p_2 = t$, due date $d_2 = t$, release date $r_2 = 0$ and processing set $M_2 = \{1, 2\}$. Both JPS and NJ will schedule both jobs on machine 1, job 2 starting at time 0, and job 1 starting at time t. The maximum lateness of that schedule is $L_{\max} = t - \epsilon$. The optimal schedule has $L_{\max} = 0$, with job 1 scheduled to start at time ϵ on machine 1, and job 2 starting at time 0 on machine 2. See figures 6.1 and 6.2.

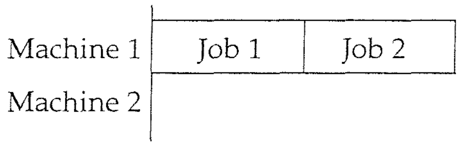


Figure 6.1: Poor performance of JPS and NJ algorithms

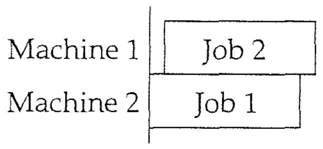


Figure 6.2: Optimal schedule

In this chapter, we have proposed three heuristic algorithms (EDD, JPS and NJ) for minimising maximum lateness on parallel identical machines with nested processing set restrictions. In the next chapter, we will develop a framework for solving the wider hybrid flow shop problem, and we will use the algorithms from this chapter as a part of that framework.

Chapter 7

Algorithms for the hybrid flow shop

In this chapter, we will present a method for the scheduling of hybrid flow shops. The method is specifically adapted to the simplified model of the industrial plant which we produced in chapter 3. We develop a framework for a general hierarchical decomposition method which has four main components: decomposition into sub-problems; criticality measure or ordering; sub-problem solution procedure; and backtracking. This framework can be used for a wide range of problems beyond the hybrid flow shop problem on which we concentrate our efforts. Each of the main components of the method may be performed in several ways. We propose in this chapter alternative methods for each component of the overall algorithm. In chapters 8 and 9 we test the behaviour of the components to identify which ones perform best.

7.1 Literature

The hybrid flow shop problem has been approached in a number of ways in the past. In this section, we give a review of the literature of hybrid flow shop scheduling.

7.1.1 Two-stage hybrid flow shops

There is a large body of literature on the two-stage hybrid flow shop, most of which concentrates on minimising the makespan of the schedule. As early as 1972, Shen and Chen[73], present a simple heuristic (the More Early heuristic, or ME), and prove its worst-case performance bound of $\frac{3}{2} - 1/\left(2\max\{m_1, m_2\}\right)$. Later research on the two-stage problem tends to concentrate on systems where one or other stage has only a single machine. Examples include Sriskandarajah and Sethi's[75] heuristic algorithms and performance bounds for problems with a single machine at the first stage, and the branch-and-bound approach of Gupta, Hariri and Potts[33], who develop five lower bounds for problems with a single machine at the second stage.

Narasimhan and Panwalkar[58] develop a heuristic (CMD or cumulative minimum deviation) for a two-stage hybrid flow shop with a single machine at the first stage, and two machines at the second, where the production rate on the first stage is fast enough to feed both processes at the second. The CMD heuristic minimises the sum of the idle time and inter-process waiting time for each job as it is added to the schedule for the second stage. Later, Narasimhan and Mangiameli[57] extended the CMD rule to handle multiple parallel machines at both stages. The extended heuristic, the generalised cumulative minimum deviation (GCMD) rule out-performs SPT, LPT (shortest and longest processing time first respectively) and minimum deviation (MD) heuristics. The GCMD rule appears to be one of the better heuristics for minimising makespan on two-machine hybrid flow shops.

On a similar system to that studied by Narasimhan and Panwalkar above, Riane, Artiba and Elmaghraby[67] attempt to solve the makespan problem with a dynamic program, and compare it to a sequence-and-merge approach and a simple greedy heuristic. Their dynamic program formulation has the disadvantage that it cannot solve problems of larger than 15 jobs on current computers.

7.1.2 Multi-stage hybrid flow shops

Moving on to multi-stage hybrid flow shop problems, there have been a number of different methods applied. The largest class of methods in the literature is that of rule-based heuristic algorithms. However, branch and bound algorithms and local search methods have also been used with some success to solve these problems. Also, recently hierarchical decomposition methods such as the shifting bottleneck method have been studied as one more approach to hybrid flow shop problems.

Rule-based heuristics

Brah and Loo[6] modified five heuristics for the standard flow shop to solve hybrid flow shop problems. They used the solution from each flow-shop algorithm to generate a list at each stage of the hybrid flow-shop problem from which list scheduling was used to generate a no-delay, non-preemptive schedule on the parallel machines of each stage. They performed a comparison of the different heuristics, for both the makespan and total completion time objectives. They conclude that the NEH heuristic of Nawaz, Enscore and Ham[59] and that of Ho [37] are both consistently good, and generally better than the other three (two versions of the CDS heuristic, and Hundal and Rajgopal's modified Palmer heuristic[41]).

Guinet and Solomon[32] use a similar technique to minimise the maximum tardiness in a multi-stage hybrid flow shop. They use, in three different variations of their algorithm, CDS, NEH and a branch-and-bound method due to Townsend[76] to give a sequence for the jobs. They then allocate operations to machines using two different methods: either to minimise the job's overall completion time; or to minimise the completion time of the next operation in the job. They conclude, as did Brah and Loo, that the NEH heuristic out-performs the others in this application.

Another big comparison of heuristics was made by Kadipasaoglu, Xiang and Khumawala[46]. They compare nine heuristics using ten objective functions, concluding that the GCMD rule of Narasimhan and Mangiameli[57] is superior to the other rules for off-line scheduling. The other rules studied include SPT, LPT (shortest and longest processing time first, respectively), and a minimum cost-over-time (COVERT) heuristic.

Hunsucker and Shah[42] examine several heuristics for hybrid flow shops with limited inter-process storage, where the storage is a global shared resource, rather than being in specific locations. They compare six simple scheduling heuristics – SPT, LPT, FIFO and LIFO (respectively, first- and last-in-first-out), and MWR and LWR (respectively, most and least work remaining). They examine the behaviour of the heuristics under several different types of problems, and show that, for the total tardiness objective function, FIFO is the best heuristic to use. To minimise the number of late jobs, LIFO and MWR have the best performance, except for problems which are not heavily constrained by the inter-process storage (i.e. where there are lots of storage bins), where SPT shows the best solutions.

Some attention has also been paid to scheduling hybrid flow shops with missing operations (sometimes called *flexible flow lines*). Probably the best-known algorithm for this problem is the WLA heuristic of Wittrock[78], which uses LPT list scheduling to allocate jobs to machines at each stage before sequencing the jobs on machines according to his own workload approximation (WLA) heuristic. Also of interest is Sawik's RITM_NS heuristic for hybrid flow shops with missing operations and no intermediate storage. Unfortunately, there do not appear to be comparisons of Sawik's RITM_NS with any other algorithms for the same problem, so its relative performance is unknown.

Kyparisis and Koulamas examine the hybrid flow shop problem, minimising weighted completion time. They look at a weighted shortest processing time first (WSPT) heuristic, and show a worst-case bound of $\left(\frac{\sqrt{2}+1}{2}\right)\lceil\sum m_k/\min m_k\rceil$ for it. They also show better bounds of $\lceil\sum m_k/\min m_k\rceil$ for unweighted problems and for problems with a single machine at one or more stages.

Finally, the use of a multi-pass heuristic known as EH was proposed by Santos, Hunsucker and Deal[71]. The EH heuristic has been used previously for job shop scheduling, but can be modified to solve hybrid flow shop problems as well. They test the heuristic with problems up to 20 jobs and 5 stages, and show that it performs well – finding optimal solutions to two of their 20-job 5-stage problems.

Local search methodology

It is possible to solve hybrid flow shop problems using local search methods. Most popular appears to be the tabu search, exemplified by the method of Nowicki and Smutnicki[61]. They use an insert neighbourhood, where an operation may be moved to any position on any machine at the relevant stage. Their tabu list is a list of attributes of solutions (specifically, of adjacent operations in the solution) rather than of solutions themselves, and they retain the best solutions to date rather than using random kicks to restart the search.

Nebenman[60] makes a comparison study of several local search methods for solving the hybrid flow shop. They examine methods from Nowicki and Smutnicki [61], Dauzère-Pérès and Paulli [18], Brandimarte [7], and Hurink, Jurisch and Thole [43]. The latter three approaches were all designed to solve the flexible job shop problem, of which the hybrid flow shop is a special case. They conclude that Nowicki and Smutnicki's method performs better than the others for a broad selection of problems.

It is worth also noting the work of Mastrolilli [54], who studies suitable local search neighbourhoods for the flexible job shop. The flexible job shop is effectively a superset of hybrid flow shop problems, as all HFS problems, even those with processing sets and/or missing operations, may be formulated as a flexible job shop problem.

Branch and bound

The heuristics and local search methods described in the previous sections, while generally fast to calculate, do not guarantee exact solutions. With any general hybrid flow shop, the only approach currently available (and, provided that $\mathbf{P} \neq \mathbf{NP}$, ever likely to be available) to give exact solutions is some form of full enumeration. It is this category that branch-and-bound methods falls into. The use of branch-and-bound methods does not appear to have been investigated deeply in the literature. Brah and Hunsucker[5] give one of the few implementations for the hybrid flow shop problem in the literature. They observe that their upper and lower bounds on the makespan are not particularly good. As a result, the procedure is unable to prune the search tree aggressively, and thus it was unable to solve problems to optimality greater than about 6 jobs on 5 production stages (on 1991 computers).

Although not a full branch and bound scheme, Santos, Hunsucker and Deal [70] develop a new global lower bound on the makespan for hybrid flow shops. Their method takes the maximum of a naïve lower bound (the job with the largest total processing time), and a more sophisticated lower bound constructed from makespan bounds for each stage.

7.1.3 Shifting bottleneck procedures

The shifting bottleneck procedure was originally developed by Adams, Balas and Zawack[2], for the solution of the job shop scheduling problem. The concept of the shifting bottleneck procedure is to break the original problem down into individual sub-problems (in the case of the job shop problem, single machines), and to solve the sub-problems one at a time. There then follows a second step, of re-optimising, where one sub-problem is selected and re-solved whilst keeping the other sub-problems static. This re-optimisation procedure is performed many times until a suitable solution is found. We will go into more detail on the structure of shifting bottleneck procedures (or hierarchical decomposition methods in general) later in this chapter. Adams, Balas and Zawack's paper uses a branch-and-bound solution procedure for the single-machine problems developed by Carlier[9].

In the simplest case of such a solution method, the subproblems used are the parallel machine problems of each stage. Simple scheduling rules are used at each stage to schedule the operations on that stage. Examples of such methods include the GCMD rule of Narasimhan and Mangiameli[57], and the various sequencing rules examined by Hunsucker and Shah[42]. These methods have the advantage of low computational complexity when running, allowing them to handle large problems. However, the solution quality is not necessarily particularly high.

Dauzere-Peres and Lasserre[17] update the method of Adams *et al.*, modifying Carlier's algorithm to handle job precedence constraints. This modification allows the release dates of operations to be updated as they change, and guarantees a monotonic decrease in makespan when re-optimising.

Holtsclaw and Uszoy[39] also use the shifting bottleneck procedure to solve job shop problems, minimising $L_{\max}$. They use several different variants on the solution method, concluding that for random routings between machines (a feature which appears in the job shop but not the hybrid flow shop) a simple criticality measure will suffice. They also conclude that inexact heuristic methods for solving the individual sub-problems work as well as the more costly exact branch-and-bound methods. Finally they observe that the shifting bottleneck procedure appears to work best when there is more structure in the job routings between machines.

Pinedo and Singer[66] concentrate on solving total weighted tardiness on a job shop. Their main contribution is in using a partial-depth tree search to find the best sequence in which to solve the sub-problems.

Yang, Kreipl and Pinedo[79] apply a shifting bottleneck methodology to a hybrid flow shop problem. Their paper looks at the problem of minimising the total weighted tardiness for a multi-stage parallel machine problem. The two major contributions of their work are related to the tardiness objective function. The first is a development on Vepsalainen and Morton's ATC[77] heuristic for minimising weighted tardiness on a single machine. The modified version extends the original ATC heuristic to work for (identical) parallel machines, and also takes account of release dates on the individual jobs. The second advance offered by the paper is in the manner in which operations are handled as jobs in the context of the sub-problems. In particular, methods for calculating "job" weights and due dates are provided. The primary difficulty in using the shifting bottleneck method with the (weighted) total tardiness objective function is that, unlike the makespan or maximum lateness objectives, there is no simple and direct method of calculating suitable due dates. If modelled precisely, the objective function for the sub-problems should be a piecewise linear function with a non-decreasing derivative. Yang, Kreipl and Pinedo instead use an approach where the piecewise linear function is approximated by a weighted tardiness function. As well as the shifting bottleneck approach, they test a local search method and a combined method utilising both the shifting bottleneck and local search methods. The hybrid algorithm was shown to out-perform the others.

Most recently, Cheng, Karuno and Kise [11] develop a shifting bottleneck procedure to minimise the maximum lateness on a hybrid flow shop. They use an approximate solution method for the parallel-machine sub-problems generated in their decomposition, rather than the more heavyweight exact branch-and-bound method favoured by others. They compare their method to that of Wittrock[78], and also to the lower bounds of Santos, Hunsucker and Deal[70], and show that their heuristic generally produces near-optimal solutions to the problem.

7.2 Hybrid flow shop with processing sets

The hybrid flow shop problem for which we will develop solutions in this chapter is inspired by the problem described in chapter 3. We will develop algorithms to solve an extended version of the hybrid flow shop problem. Specifically, we will be concerned with a hybrid flow shop (§3.5.1) with processing set restrictions at each stage (§3.5.3). We are not developing here solution methods to handle the other aspects of the industrial process which were identified in §3.5, to whit the continuous flow, product hierarchy or storage bins. Note, however, that the algorithm may be modified easily to handle continuous flow processes, as we showed in chapter 4.

To recap on the problem, consider a multi-stage production process with s stages in a fixed processing order. There are n jobs to be processed on these stages. Each job passes through the stages in order, although any given job may only be processed on a [job-dependent] subset of the stages. A job may not start processing on a stage until it has completed processing on the previous stage (i.e. the job flow is discrete, not continuous). There is assumed to be unlimited inter-process storage. At stage k, there are m_k parallel identical machines. Each job to be processed at that stage is processed on one of the machines, taking time p_{jk}. Each job, j, has a set of machines, M_{jk} (the processing set) at each stage on which it may be processed. The processing sets for any given stage are nested, according to the definition in §3.5.3. Each job has a due date, d_j.

We had originally intended to develop a general multi-stage decomposition framework and test it out on the weighted tardiness objective function. While the company are most interested in the number of late jobs and secondarily maximum lateness, those objective functions were poorly behaved. We therefore had reserved the option of using the better behaved total tardiness objective function for detailed computational experiments. A robust general framework could then be used for any objective function. However, shortly after we developed our initial framework, including the use of the hierarchical decomposition technique, the paper by Yang, Kreipl and Pinedo (YKP) was published. We felt that the YKP paper was too close to our original intended line of research. This indicated that our general approach was sound, but that we needed to change the objective function to maximum lateness rather than weighted tardiness. Therefore we wish to minimise the maximum lateness across all jobs, $L_{\max}$,

$$L_{\max} = \max_{j=1}^{n} C_j - d_j,$$

where C_j is the completion time of job j.

We go beyond the scope of Yang, Kreipl and Pinedo's work in several key areas, however. Firstly, and most obviously, we develop algorithms which take account of processing set restrictions on each stage of parallel machines. Secondly, we look in more detail at the overall concept of the hierarchical decomposition method, examining individually the different components required to produce such an algorithm.

7.3 The algorithm

The hybrid flow shop problem which we are investigating is complex. We cannot expect to be able to solve realistic problems (up to 100 jobs on 20 machines) using exact methods, so we must resort to some form of heuristic or approximation method. We might attempt to use a simple rule-based heuristic, or a local search method, but we instead use a hierarchical decomposition method similar to the shifting bottleneck procedure, particularly in the light of Holtsclaw and Uszoy's observation that the shifting bottleneck works best where the problem is highly structured.

The general idea of the method is first to divide the overall problem into multiple (related) sub-problems, and then to solve those sub-problems separately. The structure of the algorithm is as follows:

1. Separate the original problem into sub-problems.

2. Order the sub-problems according to some criterion.

3. For each sub-problem in order, incorporate it into the partial solution by:

   (a) Solving the selected unsolved sub-problem.

   (b) Scheduling/sequencing the current (partial) schedule, if improvement is possible.

4. Optionally, improve the schedule using a local search procedure.

Phases 1 and 2 involve identifying and calculating a suitable *criticality measure* for each possible sub-problem, and ordering the sub-problems according to which is the most *critical*. We describe the sets of machines which define the sub-problems as *execution sets*. The criticality measures and division into sub-problems which we use are developed in §7.5. An efficient problem sub-division method is described in §7.8.1.

Phase 3 uses a heuristic algorithm to solve each sub-problem in turn (in order of their criticality measure), and to add the solved sub-problems into a partial solution of the whole problem. As more sub-problems are solved, and more information becomes available, some of the sub-problems are re-solved so that they may fit better with the rest of the partial solution. The method of deciding which sub-problems to re-solve, and in what order, is discussed in §7.6.

After solving each sub-problem (whether for the first time, or whilst performing a backtracking procedure), the schedule of the sub-problem is effectively discarded, retaining the allocation of operations to machines and the sequence of operations on those machines. The current partial schedule can be represented using a disjunctive graph representation, like those presented in chapter 4. Where allocations and sequences of operations have been made, at most one arc from each disjunctive pair is selected to reflect the sequencing; where no allocation or sequencing has been made, no arcs are used, and the relevant operations are assumed to be unconstrained. After constructing the graph representation of the partial schedule, an actual schedule may be calculated by means of critical path analysis. One of the by-products of the critical-path algorithm is information for easily calculating release dates and due dates for each operation (see §7.7.1).

Phase 4 is an optional local search procedure which is applied to the solution after all of the sub-problems have been solved and a whole solution is available. This phase aims to improve the schedule quickly where possible. We shall not implement such a procedure in our algorithm.

Note that although we describe the framework assuming a discrete process flow, it requires only a simple modification to solve problems which have continuous flow of material through the machines. We need only alter the disjunctive graphs so that they represent a continuous flow process using an appropriate model from those developed in §4.4.

7.4 Problem decomposition

The first component of our hierarchical decomposition method is the decomposition of the problem itself. This step breaks down the original problem into distinct and disjoint sub-problems which are to be solved one at a time. The most obvious and natural method of decomposition for a hybrid flow shop structure is a stage by stage decomposition, where each stage is treated as an independent sub-problem. This is the method used by Yang, Kreipl and Pinedo. Our problem is typified by processing set restrictions at each stage. This feature affects the allocation of jobs to machines, and also has some implications for the balance of workload across machines, as will be seen in §7.5.3. It is therefore sensible for us to consider alternative decomposition methods. Our alternative method of decomposition involves breaking the problem into smaller parallel machine problems, where each stage is broken up into one or more sub-problems. The precise method for doing this decomposition is described later, in §7.5.3, as it relies on some concepts developed in the next section. We term this latter decomposition method *decomposition by execution sets.*

7.5 Criticality measures

For operating any progressive heuristic such as the shifting bottleneck algorithm, we need measures of the *criticality* of the different components of the system. Such criticality measures (often called bottleneck measures) give a measure of how important the scheduling of a given component of the overall system is. Criticality may be thought of as a measure of difficulty, congestion, degree of "bottleneck", or importance.

We distinguish between two different types of criticality measure which we wish to consider: static and dynamic. A *static* criticality measure depends solely upon the problem instance, and will highlight the machines (or possibly jobs or operations) which are likely to be most critical to finding a good schedule. A *dynamic* criticality measure is dependent upon not only the problem instance, but also upon a proposed schedule or partial schedule for that instance. The dynamic measure will indicate which operations (or possibly machines or jobs) in the schedule are contributing most to the objective function value. Dynamic criticality measures would be of most use in local search or incremental-improvement algorithms, as they could be used to guide the algorithm to where its effort will do the most good.

When using shifting bottleneck methods to solve makespan ($C_{\max}$) problems on the hybrid flow shop, the bottleneck measure used is often simply the average workload that passes through a stage. For other, more general, problems, this is a poor measure, as it does not take account the effect which some features may have on it. For example, allowing processing sets at each stage means that a stage may have a low mean workload, but have one highly critical machine on it. In this section, we develop criticality measures for some variants of the basic hybrid flow shop problem.

7.5.1 Mean stage workload and no processing sets

The most obvious criticality measure is the *mean stage workload* (MSW), W_k,

$$W_k = \frac{\sum_{j=1}^{n} p_{jk}}{m_k}.$$

where k is the stage under consideration, n is the number of jobs, and p_{jk} is the processing time of job j at stage k. This is simply a measure of the mean amount of processing time per machine required to finish all of the jobs. This measure can, of course, only be computed if the processing time for each job is the same on any allowable machine. However, the mean stage workload does not take account of such things as processing sets.

7.5.2 Critical processing sets

When processing sets are introduced, the workload across all machines in a stage may not be the same and the mean stage workload measure therefore can be misleading. Consider a hybrid flow shop with two stages, and two machines at each stage. At the second stage, there are two processing sets, $M^{(1,2)} = \{1, 2\}$ and $M^{(2,2)} = \{2\}$. There are n jobs, with $p_{j1} = 3$ and $p_{j2} = 2$ for all jobs j, and, at stage 2, job 1 having processing set $M^{(1,2)}$, and all other jobs having processing set $M^{(2,2)}$.

The mean stage workload measure will identify the workloads of the two stages as $W_1 = \frac{3n}{2}$ and $W_2 = n$, thus identifying the first stage as the more important. However, it is clear that almost all of the jobs at the second stage must go through machine 2, making that the most critical part of the process. We need to consider measures which identify the criticality of individual processing sets and not simply whole stages of machines.

Total mean processing set workload

One possible measure to use is the *total mean processing set workload* (or TMPSW). Firstly, for each processing set q at stage k, define the mean processing set workload,

$$\overline{w}_{qk} = \frac{\sum_{j:M_j=M^{(qk)}} p_{jk}}{|M^{(qk)}|}.$$

Note that the processing sets overlap; each machine may belong to several processing sets. Then, for a machine i at stage k, the total mean processing set workload may be taken to be the total of the mean processing sets which the machine belongs to:

$$w_{ik} = \sum_{q:i \in M^{(qk)}} \overline{w}_{qk}.$$

The TMPSW measure misses out on one important aspect of the operation of parallel machines with processing sets namely load balancing. If one machine belongs to many processing sets, some of its workload will (if possible) be moved to other machines belonging to fewer processing sets.

Load balancing with general processing sets

It is possible to calculate the *mean* effects of load balancing by use of the max-flow/min-cut theorem [23]. We construct the flow network shown in figure 7.1. The arcs on the left of the diagram are capacitated to the processing time of each job. The arcs in the middle of the diagram connect each job to the set of machines which may process it, and are uncapacitated. The arcs on the right of the diagram represent the workload of each machine, and are capacitated to the same value, C. The value C represents a limit on the amount of time available for processing. We denote by $\mathcal{F}(C)$ the maximum-flow problem for any given value of C.

To find the minimum amount of time required to process all of the tasks, and the bottleneck processing set, take some upper bound on the time required to complete all processing (trivially, $U = \sum_j p_{jk}$). Using a binary search on the interval $[0, U]$, find the minimum value of C for which all of the "job" arcs (those on the left of the diagram in figure 7.1) in the problem $\mathcal{F}(C)$ limit the maximum flow through the network. In other words, find the minimum value of C for which none of the job arcs have spare capacity. The process may be performed in $O((n^3 + m^3) \log U)$ time, using the push-relabel algorithm of Goldberg and Tarjan [29].

At this point, a number of the "machine" arcs (those on the right of the diagram) will be at full capacity. Those machines are the "bottleneck" machines; the minimum value of C which has been found is the workload on those machines.

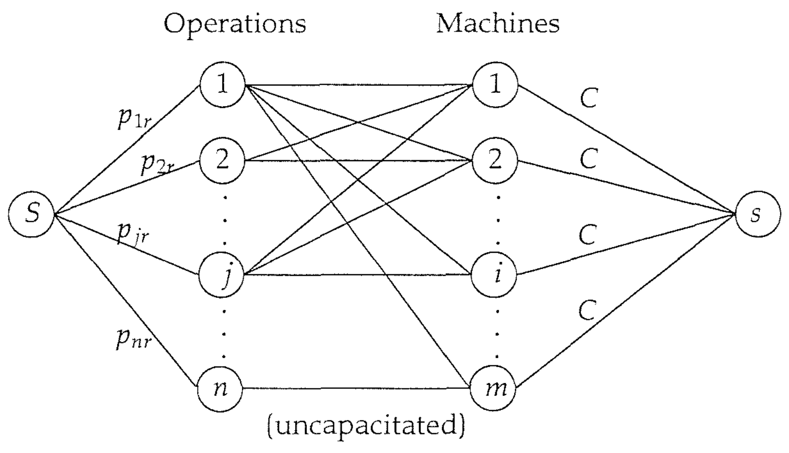


Figure 7.1: Example of the max-flow network for fully load-balanced parallel machines with processing set restrictions

In the case of nested processing sets, the bottleneck machines will form a single processing set, although it may contain some smaller processing sets as well.

7.5.3 Workload criticality measures for nested processing sets

Where processing sets are nested, the network formulation given above is overly complicated. The nesting of processing sets allows a simple algorithm to be used to sort the processing sets of a stage in decreasing order of workload. As with most algorithms for nested processing sets, we start work at the smallest processing sets, and work up to the largest sets.

At stage k, for a processing set $M^{(qk)}$, we define the workload as the mean time to completion of all jobs with that processing set over the machines in the set:

$$L_{qk} = \sum_{j:M_{jk} \subseteq M^{(qk)}} p_{jk}/|M^{(qk)}|.$$

We shall refer to this as the *Nested Load-Balanced Workload* measure, or NLBW.

Now reconsider the example of §7.5.2, for which the mean stage workload measure proves to be misleading. Calculating the NLBW workloads for each of the two processing sets at the second stage: $L_{12} = 2$ for the larger processing set and $L_{22} = 2n - 2$ for the smaller processing set. Thus, with this measure, the smaller processing set at stage 2 (i.e. $\{2\}$) is given the highest workload value, as it should be.

To understand what the NLBW (nested load-balanced workload) criticality measure is doing, consider a system with two nested processing sets and a large number of small operations performed over the machines encompassed by those processing sets. There are then two different cases which can arise: either the smaller processing set is busier than the larger, or the larger is busier than the smaller. These two cases can be illustrated by the simple block diagrams in figures 7.2 and 7.3, respectively. In both diagrams, the smaller processing set is marked $M^{(1)}$, and the larger $M^{(2)}$.

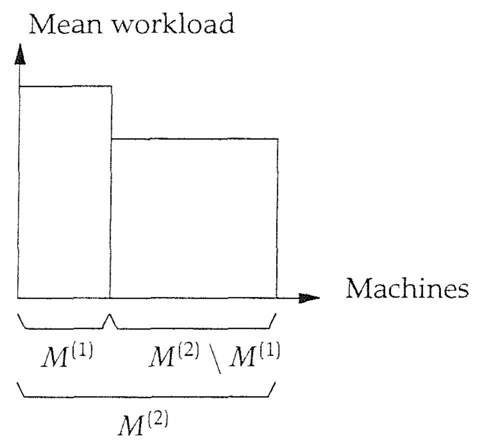


Figure 7.2: Smaller processing set busier

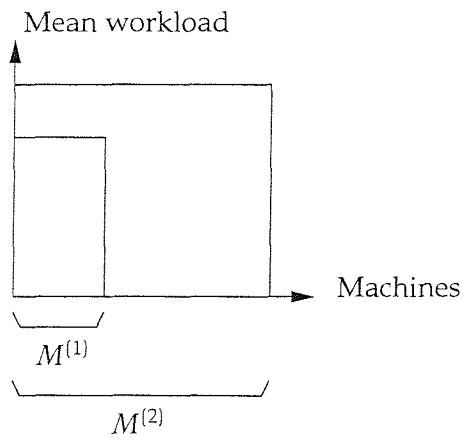


Figure 7.3: Larger processing set busier

When the smaller processing set, $M^{(1)}$, has the greater mean workload (figure 7.2), the machines in $M^{(1)}$ should, in a balanced system, process only those operations with that processing set. When the larger processing set, $M^{(2)}$, has the greater mean workload (figure 7.3), then the machines in the smaller processing set may process some of the operations from the larger processing set as well. The consequence of this is that in the latter case, the two nested processing sets should be treated as a single unit, and scheduled together. In the former case, however, we can schedule the two processing sets as two disjoint groups of machines: $M^{(1)}$ and $M^{(2)} \setminus M^{(1)}$. Observe that the mean workload of the processing sets gives the relative order of workload. Moreover, in practice one would expect $M^{(1)}$ to be scheduled first and then the remaining operations with processing set

$M^{(2)}$ afterwards.

The NLBW criticality measure calculates the mean workload for each processing set at each stage, and produces a measure which is comparable between stages. By examining the relative workload of the processing sets at a given stage, the machines of that stage may be placed in a number of disjoint groups (which we shall call execution sets), according to their relative workloads. The first execution set comprises the machines belonging to the processing set with the highest workload. The nth execution set is the machines belonging to the processing set with the nth highest workload, except those which have already been assigned to other execution sets.

Mathematically,

1. Order the processing sets $M^{(1k)}, \ldots, M^{(qk)}, \ldots, M^{(Q_k k)}$, from the most to the least critical set, using the NLBW criticality measure.
2. Define the qth execution set at stage k, $E_{qk} = M^{(qk)} \setminus \bigcup_{i=1}^{q-1} M^{(qk)}$.
3. Associate the workload measure of the qth processing set, w_{qk}, with the qth execution set.

Performing this process for each stage of the problem, we are left with a collection of disjoint execution sets, from all stages, each of which has an NLBW measure of criticality. This method forms the basis of the execution set decomposition referred to in the previous section. Note also that when this method is used to perform decomposition by execution set, the NLBW criticality measure returns exactly the same values as the MSW criticality measure, since the execution sets are, in that case, by definition well-balanced.

7.6 Backtracking schemes

One of the main features of the shifting bottleneck methodology for the job shop is *backtracking*. By backtracking, we mean that after a sub-problem has been solved, it is re-visited later by the algorithm (often many times), and solved again. This re-solving of a sub-problem allows the sub-problem to take account of its relations to other sub-problems which are solved later in the operation of the algorithm. There many different ways of performing the backtracking. In this section, we propose several different methods.

The sequence of solution of the sub-problems referred to in this section is that determined by the order of criticality of the components of the problem (see §7.5).

7.6.1 Classification of backtracking schemes

We may classify backtracking schemes according to three criteria: the *backtracking path* defines the order in which the sub-problems are revisited after each new sub-problem solution is added to the partial schedule. The *depth* of the backtracking specifies the maximum number of sub-problem solutions which may be performed in any one backtracking run. The depth may be unlimited, in which case the whole backtracking path is used, regardless of how many sub-problems there are to re-schedule. The *looping method* governs the backtracking path and how many times it is executed. Examples of different possible looping methods are given in §§7.6.4-7.6.5, below.

7.6.2 Fixed order, single-pass, no backtracking

Simple rule-based scheduling algorithms such as the GCMD rule (see [57]) use a single pass method, where the operations at each stage are scheduled once. Thus, a problem with s stages requires s sub-problem solutions. This is $O(s)$ behaviour. We describe this as a *single-pass* heuristic with *no* backtracking. The backtracking path definition is simply to do no backtracking.

7.6.3 Fixed order, single-pass backtracking

Single-pass backtracking selects an ordering for the stages (most heavily-loaded stage first), and, for each stage in that order, solves the stage as a sub-problem. The algorithm then revisits all of the previous stages, scheduling each in the original order, starting at the first.

The backtracking path is to revisit each previously scheduled sub-problem in the order in which they were originally solved. The simplest form of this type of backtracking, without further repetition, is said to be *single-looping*. If there are s stages, the algorithm will perform at least $\frac{1}{2}s(s+1)-1$ subproblem scheduling steps. This is $O(s^2)$ subproblem solutions, compared to the $O(s)$ solutions which the basic single-pass scheduling rules use. An illustration of the order of re-scheduling in such a backtracking scheme is shown in figure 7.4.

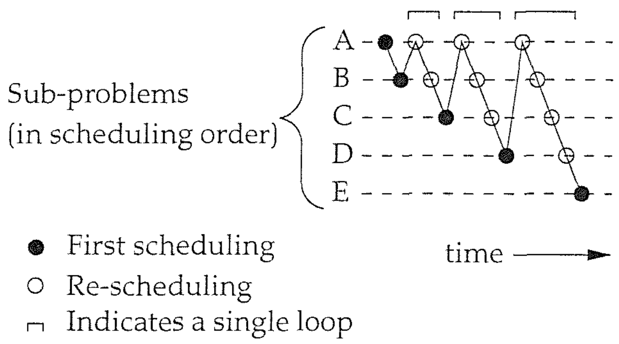


Figure 7.4: Linear backtracking with single-looping (Single-pass)

7.6.4 Fixed order, multiple-pass backtracking

In the single-pass backtracking method given above, one pass of backtracking may not suffice to give a good solution. Some of the later sub-problems in the partial schedule may be affected by the re-scheduling of earlier sub-problems. One simple and obvious solution to this difficulty is simply to re-schedule the subproblems many times, either to a maximum number of loops, or until no more beneficial changes are made. We describe this form of backtracking as *multiple-looping*. An illustration of the re-scheduling order of such an exhaustive multiple-looping scheme is shown in figure 7.5.

7.6.5 Other fixed-order schemes

Other fixed-order schemes may also be considered, in either single- or multiple-loop forms. For example, we may use a "pyramidal" scheme, which reprises all of the previous backtracking at each stage, as illustrated in figure 7.6. This scheme ensures that when a sub-problem is solved *for any reason*, the previous problems are re-solved in a backtracking phase. The disadvantage of the pyramidal scheme

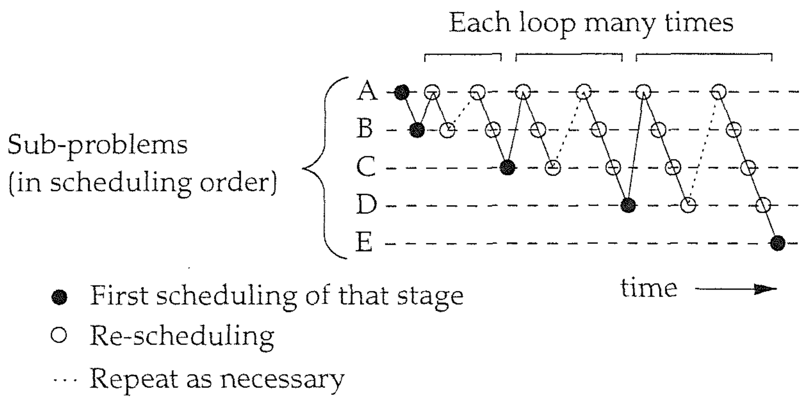


Figure 7.5: Linear backtracking with multiple-looping (Multiple-pass)

is that the amount of backtracking grows as the cube of the number of stages.

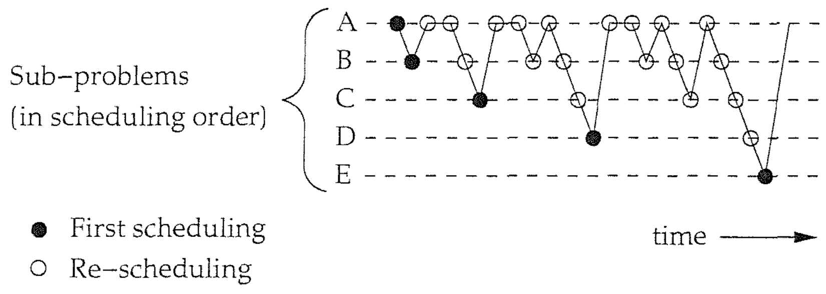


Figure 7.6: Pyramidal backtracking with single-looping

Another alternative is a fixed-depth scheme, where only the last p subproblems are re-solved in the backtracking stage (illustrated in figure 7.7). This scheme has the advantage of using some backtracking, but it is still linear time in the number of sub-problems. It has a distinct disadvantage when, as proposed, the sub-problems are taken in order of likely importance – it does not revisit the most important stages after the first p sub-problems have been added to the partial solution.

7.6.6 Advantages and disadvantages of backtracking

One of the problems of backtracking schemes such as those given above is that they may not necessarily concentrate on the right parts of the problem. It is not clear that extensive and comprehensive backtracking, originally developed for

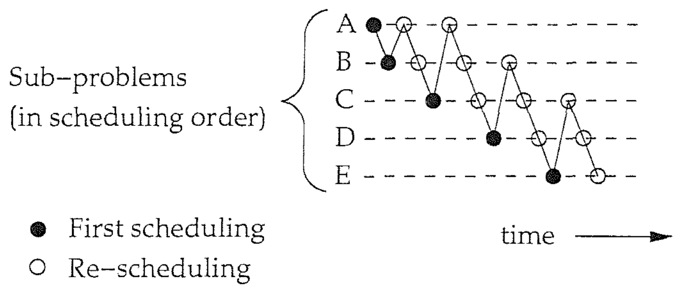


Figure 7.7: Fixed-depth backtracking, single-loop. Depth=3

use in job-shop problems, is necessary for the well-structured hybrid flow-shop problems. We can only establish the latter point by experimentation.

The main objections to the schemes detailed in the above sections, in the context of scheduling our hybrid flow shop problem, are that:

1. they do not take advantage of the additional structure inherent in a flow-shop problem,

2. they do not take account of some of the peculiarities of processing sets – for example, that there may be several effectively unrelated processing sets at each stage, and

3. they do not necessarily achieve the best overall improvement at each re-scheduling.

The backtracking schemes which we have described in previous sections all rely upon a fixed ordering of the sub-problems, and re-schedule the sub-problems according to some rule based on that fixed ordering. In other words, they all assume a static criticality measure for the sub-problems, and they persist in using the same ordering even when more information about the problem becomes available in the process of scheduling it. We would like the order of the sub-problem rescheduling to depend upon the state of the current partial schedule.

In the next sections we propose two methods of backtracking (one of which is original) which are not solely dependent on the initial problem, but are also dependent on the state of the partial schedule as it is built up.

7.6.7 Steepest descent

The backtracking process of the shifting bottleneck algorithm may be thought of as a form of local search, where the neighbourhood is a choice of which sub-problem to re-schedule, and a move is a re-scheduling of a complete sub-problem. Seen in this light, the multiple-loop backtracking scheme of §7.6.4 above is a *first-improve* "local search" method – a move (re-solution) is made whenever one is found. The other commonly-used local search descent method is to take the best of the available moves rather than the first. In this form of backtracking, each sub-problem is re-solved in turn, and the one which shows the best improvement in the overall objective function is the one which is actually adopted in the current partial solution. This is the method which was used by Adams, Balas and Zawack in the original shifting bottleneck algorithm, and subsequently in most other work on shifting bottleneck methods. Our first-improve method provides a shallower descent path, but requires less computation time than the steepest descent method.

7.6.8 Other dynamic measures

The main disadvantage of the best-improve method given above is that many (up to s) sub-problems must be tested in order to re-schedule one sub-problem. This is computationally expensive. When performing backtracking, we would like to concentrate on re-scheduling those sub-problems which are most affected by the changes to the partial solution. We may do so by utilising a priority rule, calculated after each new sub-problem is solved and added to the partial schedule. There are many different priority rules which we might use:

1. workload,
2. maximum workload over the machines in the sub-problem,
3. adjacency of stage in the flow of work,
4. number of adjacent operations of jobs which have been affected by interim re-scheduling,

5. number of (any) operations of jobs which have been affected by interim re-scheduling,
6. total processing time of the operations which have been affected by the previous re-scheduling,
7. total slack,
8. minimum slack over the machines in the sub-problem,

The first criterion listed above is in fact simply the fixed-order backtracking we presented in §7.6.3, and is effectively based on the mean workload criticality measure of §7.5.1. The second is also a fixed order method, but based on the NLBW criticality measure of §7.5.3 instead (which, as we stated in that section is equivalent to the simple workload measure when execution set decomposition is being used). In the following sections, we present a method of selecting the next stage to re-schedule using the third criterion in the above list.

7.6.9 Flow-adjacency backtracking

Consider the relationships between schedules of the sub-problems (execution sets) in the hybrid flow shop problem: some sub-problems will be related to each other by the jobs which are common to both sets; others will be entirely independent of each other (for example, all those at a single stage). We wish to create a backtracking scheme which uses the relationships between sub-problems as the basis for deciding which sub-problem to re-solve next. We also wish this scheme to be capable of being selective in its re-scheduling of the currently scheduled sub-problems.

Note that, in general, adding a sub-problem to the partial solution is a much more disruptive process than simply re-scheduling a sub-problem which is already part of the partial solution being built up. The reason for this is that adding a sub-problem solution adds a (large) number of additional constraints to the partial solution, whereas re-scheduling an existing solution simply rearranges inter-operation constraints which are already present.

We may construct a graph of the relationships between the sub-problems, where the nodes of the graph are the execution sets. The arc joining two execution sets may be weighted by some measure of interdependence between the two sets. One such measure is the number of jobs which the respective sub-problems have in common. Alternatively, the total processing time of operations of the common jobs may be used, or the ratio of common jobs to unrelated jobs in one (or both) sub-problems.

Having constructed a graph of the relationships between nodes, a backtracking scheme may be thought of as a way of traversing the graph. Each time a node is visited, the corresponding sub-problem is re-scheduled. Various schemes for traversing this graph can be envisaged, one of which is presented below.

The relationship graph is constructed piece-meal. As each execution set is scheduled for the first time, its node and all of the corresponding arcs are added to the graph (see figure 7.8). Now, a tree is a much easier structure to traverse in a logical manner than an arbitrary graph. Therefore, after each node is added, the maximal spanning tree of the graph is found in $O(m \log m)$ time, where m is in this case the number of arcs – see figure 7.9. The backtracking procedure then traverses the tree, starting at the newly-added node, and rescheduling the sub-problem for each node it visits.

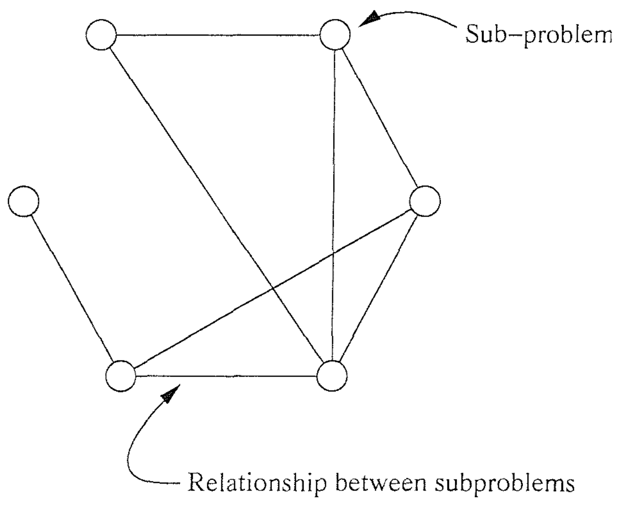


Figure 7.8: Relationship graph for a partial solution to a hybrid flow shop problem

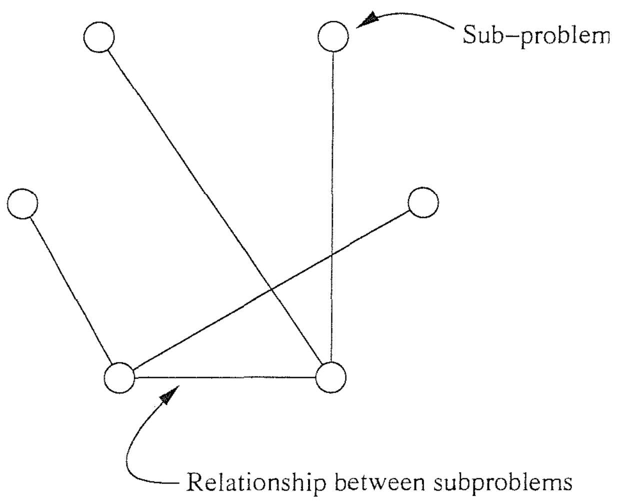


Figure 7.9: Maximal spanning tree generated from relationship graph

Several methods of traversing the tree might be tried, including:

- a breadth-first search of the nodes of the tree,
- a depth-first search of the nodes of the tree,
- following only the n highest-weight arcs from each node.

Each of the above methods may be limited, for example by a maximum number of sub-problems to re-schedule, by a maximum depth in the tree to search, or by a minimum degree of relationship to follow in the spanning tree. Alternatively, the traversal of the tree may be unlimited, in which case all sub-problems in the partial schedule are re-scheduled. The backtracking scheme which we have described above has a dynamic, partial-schedule-dependent, backtracking path. In addition, it may be implemented with or without a depth limit, and with or without multiple looping.

7.7 Sub-problem construction and solution

The sub-problems generated in the hierarchical decomposition method described in §7.3 are parallel identical machine scheduling problems with processing set restrictions, release dates and due dates. The release dates and due dates for each sub-problem are surrogates, representing the schedule of the operations in other sub-problems in other stages. We require a solution procedure, either exact or approximate, for solving such problems. In this section, we discuss the construction of the sub-problems, and some methods which might be used to solve them. We consider primarily the maximum lateness criteria. It is worth noting that makespan can be minimised by minimising maximum lateness where all due dates are zero.

7.7.1 Construction of sub-problems

Each sub-problem which is solved in the hierarchical decomposition is a problem with parallel identical machines with processing set restrictions. Each job in the sub-problem represents an operation in the overall problem. Now, we wish the sub-problem's objective function to reflect its effect on the overall problem. To this end, we must select the objective function to use for the sub-problem, and we also require additional input data for the sub-problem.

Firstly, observe that the range of possible scheduling times for each operation is restricted: an operation O_{jk} cannot start before its predecessor $O_{j,k-1}$ has completed (if such a predecessor exists). Using one of the disjunctive graph representations from chapter 4, the earliest start time for O_{jk} is the longest distance from the start node to the node corresponding to O_{jk}. Therefore, each operation O_{jk} can be given a release date r_{jk} which is the earliest time at which it can be scheduled.

Secondly, note that we may view the problem of minimising $L_{\max}$ as a problem of minimising $C_{\max}$ where the jobs have "tails". A job with a *tail*, q_j, is processed as normal, taking up p_{jk} time on the machine, but is not considered to be complete until q_j time has elapsed after processing on the last machine. We may turn an $L_{\max}$ problem into a $C_{\max}$ problem with job tails by adding a tail $q_j = \max\{d_j\} - d_j$ to each job. The disjunctive graph representation simply adds an extra arc of length q_j to the end of each job, and we then aim to minimise the total network length. In this view of the problem, although each operation may be scheduled to start at any time after its calculated release date, there is a point beyond which any further delay in its start time will cause some or all of the jobs in the overall problem to finish later than they otherwise would. The amount by which the operation may be moved without changing the objective function is commonly known as its *slack* or *float*. Writing the float for O_{jk} as f_{jk}, we can set a due date for the operation,

$$d_{jk} = r_{jk} + p_{jk} + f_{jk}.$$

Note that this takes account of the due date of the job, since the tail of the job acts as a surrogate for the due date.

7.7.2 Solution of sub-problems

In order to solve the individual sub-problems in this method, we may use any applicable solution procedure. Three such procedures for the $Pm|r_j, M_j|L_{\max}$ problem have been presented in §6.1. These are the Earliest Due Date (EDD), Jackson's for Processing Sets (JPS), and Modified Jackson (MJ) heuristics.

7.8 Implementation

We discuss below some details regarding the implementation of the algorithms and methods described earlier in this chapter.

7.8.1 Efficient calculation of execution sets

The nested load-balanced workload (NLBW) criticality measure for processing sets, and hence for execution sets, is defined in §7.5.3. However, it may be calculated more efficiently by storing the total workload for each processing set as it is calculated, using the total in calculating totals for larger processing sets which contain it. To do this, define the *total processing set workload*, ℓ_{qk} as:

$$\ell_{qk} = \sum_{j:M_{jk}=M^{(qk)}} p_{jk}.$$

The nested load-balanced processing set workload, NLBW, then becomes:

$$L_{qk} = \sum_{q':M^{(q'k)}\subseteq M^{(qk)}} \ell_{q'k}/|M^{(qk)}|.$$

Having defined our decomposition of the problem, we now schedule each execution set in turn, performing backtracking where appropriate. Note that since we schedule by execution sets, the algorithm will move around between stages, possibly visiting each stage several times as the disjoint execution sets of the stage are scheduled one by one.

7.8.2 Ordering nested processing sets

One of the common features of the algorithms for problems with nested processing sets is that they require ordering jobs, operations, or processing sets in such a way as to preserve the partial order implied by the nestedness of the processing set. In most cases, this is to ensure that no processing set has its jobs scheduled before the jobs of any of its subsets.

In a computer implementation of such an algorithm, the obvious method of representing a processing set, particularly with small numbers of machines, is to use an integer bitmap representation, where the $(n-1)^{\text{th}}$ bit of an integer variable is set to be 1 if machine n is a member of the set, and 0 if it is not. For example, the processing set $\{1, 3, 5, 6\}$ would be represented by the number 53 (110101_2 in binary), where bits 0, 2, 4 and 5, reading from the right, are set to 1. (Note that bits are customarily numbered with the least-significant bit labelled as bit 0. Bit n then corresponds to a value of 2^n). In the algorithm implementations developed for this thesis, up to 128 machines are supported in the processing set data type, which occupies 16 bytes (4 32-bit integers) of space.

There is an interesting observation to make on using this representation: Ordering processing sets by the numerical value of their bitmap preserves the partial order defined by the subset operator on the processing sets. In other words, given two processing sets A and B from a set of nested processing sets, with bitmap values a and b respectively, $a < b$ implies either $A \subset B$ or $A \cap B = \emptyset$. This method of ordering also has the property that all items with the same processing set are grouped together in the re-ordered list.

There is therefore generally no need to store or calculate the size of a processing set, and processing sets can be partially ordered efficiently using a simple standard sorting algorithm (although more efficient ordering can be obtained by performing a radix sort on the sizes of the processing sets). Hence this property may be used to implement algorithms for problems with nested processing sets which are efficient in both storage space and CPU time.

In this chapter, we have developed a framework for a hierarchical decomposition method for the hybrid flow shop problem with processing sets at each stage. This framework generalises specific hierarchical decomposition methods, and consists of a number of components:

- a decomposition scheme to break the overall problem into smaller subproblems,
- a scoring scheme ("criticality measure") to indicate which of the subproblems to work on next,
- a method of solving the individual subproblems,
- a method of backtracking through already solved subproblems to improve the solution, and
- a method of evaluating the (partial) schedules obtained at each stage of the process.

We have proposed several methods for each of these components of the overall algorithm, with the exception of the evaluation method, for which we use the digraph representations presented in chapter 4.

Although we have presented at least two possible methods for each of the four main components listed above, we do not know which of them are most effective in solving the problem. To this end, we will test the algorithm variants on a wide variety of problem instances. In the next chapter, we will discuss the set-up of the experiments, including the different factors which we can vary, both algorithm variants and parameters of the test problems. We also discuss in some detail the methods we use for generating test problem instances.

Chapter 8

Experimental framework

In this chapter, we describe the many aspects of our hybrid flow shop problem which may be varied, both of the problem itself and of the algorithms which we have developed in chapter 7 to solve it. We describe how we use these variable aspects to generate (random) hybrid flow shop problems for use in computational experiments. The experiments themselves, including the statistical framework in which they are performed and an analysis of the results, are detailed in chapter 9.

8.1 Overview

Our primary objective in performing these experiments is to identify the best set-up for our algorithm and to analyse the differences between performance of the algorithms to find the best. We also wish to identify any significant differences in the performance of an algorithm variant for any particular type of problem. For example, one of the algorithms we investigate may perform particularly badly for problems with high workload, but well for problems with low workload. The four areas which concern us in making this analysis are the following:

1. Performance measures
2. Algorithm options
3. Problem configuration (size, shape and behaviour)
4. Instance parameters

In this chapter, we will look at the above four aspects of the problem one at a time, and detail how each will be used in the experiments to follow. The largest part of this chapter (§§8.4 and 8.5) concentrates on the latter two aspects, where we show how we create test data sets on which to perform experiments.

8.2 Performance measures

The measure by which the performance of the algorithm is measured is important. In our experiments, we will be using methods (such as the sub-problem solution procedure) which are specifically designed for the maximum lateness objective function, $L_{\max}$. However, this does not prevent us from calculating the values of other objective functions for the completed schedules, and examining the performance of our methods for other objective functions.

It should be noted that the $L_{\max}$ objective function has a peculiar property. The value of $L_{\max}$ can effectively be made arbitrarily large for any given instance, by altering the due dates on the jobs, each by the same amount. The objective function will change in value, but the same job will be the "latest" job, and the jobs will be scheduled in the same way by all algorithms. Thus, it is impossible to compare $L_{\max}$ values and algorithm performance differences by taking ratios of $L_{\max}$ values. Instead, we must examine the additive difference between different methods for the same [type of] instance to find performance differences between algorithms. The most suitable statistical method for analysing the results, and the one which we have chosen to use, does in fact operate on additive differences (see §9.3).

Although the hierarchical decomposition algorithms which we are testing are designed to minimise maximum lateness, we will measure performance on two criteria:

- Maximum Lateness ($L_{\max}$)
- Execution time

We are interested in the maximum lateness because it is what our algorithm was designed to minimise, and the execution time so that we can observe any trade-offs between solution quality and time taken to find that solution.

8.3 Algorithm options

There are four orthogonal sets of algorithm factors from which to choose: decomposition, initial ordering (criticality measure), sub-problem solution, and backtracking. Each factor has a number of possible levels, summarised below:

1. Decomposition method
   (a) By stages
   (b) By execution sets
2. Order of adding sub-problems to partial solution (criticality measure)
   (a) Simple criticality measure (stage workload)
   (b) Stage order
   (c) Reverse stage order
   (d) Nested load-balanced workload measure
   (e) Arbitrary
3. Sub-problem solution procedure
   (a) Earliest Due Date (EDD)
   (b) Jackson for processing sets (JPS)
   (c) Nested Jackson (NJ)
   (d) Look-ahead heuristic (LAH)
4. Backtracking
   (a) No backtracking
   (b) Linear single loop
   (c) Linear multiple loop

(d) Tree-based (see §7.6.9)

(e) Original shifting-bottleneck scheme

Each factor is effectively independent of each of the other factors, and its levels can be selected independently. Thus, there are 200 possible versions of the algorithm to test (2 for decomposition; 5 for initial order; 4 sub-problem solution procedures; 5 backtracking methods: $2 \cdot 5 \cdot 4 \cdot 5 = 200$).

8.4 Problem configuration

The parameters for behaviour of instances may be selected in a number of different ways. We hope to choose them in a way which reflects both a wide range of possible uses, and which encompasses the possible qualitative differences in instance types. There are eight orthogonal parameters which may be varied to construct test instances with (potentially) different behaviour. To whit,

- number of jobs,
- number of stages,
- expected number of machines at each stage,
- processing sets available at each stage,
- balance of workload,
- likelihood of missing operations,
- operation length, and
- distribution of due dates.

These are discussed below and then summarised in table 8.1. We may divide the parameters into three general groups: the problem shape, bottleneck behaviour, and other parameters.

8.4.1 Shape

The shape of an instance of the hybrid flow shop problem is related to three main parameters: the number of stages, s, the number of jobs, n, and the number of machines at each stage, m_k at stage k. We hypothesise that the ratio of jobs to machines is more important than the absolute number of machines, and so instead of the number of machines, we use the ratio of the expected (i.e. mean) number of machines at each stage to the number of jobs, ρ.

8.4.2 Bottleneck behaviour

We may classify hybrid flow-shop problems by the type and location of bottlenecks in the system. For these purposes, we describe as a *bottleneck* any machine or set of machines which has a higher mean workload than those in the rest of the problem (see §7.5 for some methods of estimating machine workloads). We can distinguish between many types of system in terms of their bottlenecks. A brief list is given below:

1. fully balanced: equal mean workloads for machines at all stages,
2. vertically unbalanced:
   (a) all machines at one stage have a higher mean workload,
   (b) all machines have a higher mean workload at several stages,
3. horizontally unbalanced: at each stage, there is a processing set of machines with a higher mean workload than the rest of the stage
   (a) with the same jobs in the bottleneck processing set at each stage,
   (b) with different jobs in the bottleneck processing set at each stage,
4. mixed: a single processing set at one stage with a higher mean workload than the rest of the problem.

It should be noted that some of the types listed above are likely to resemble simpler systems: a single-stage, vertically unbalanced system is likely to resemble a single stage of parallel machines in its scheduling difficulty; while a horizontally unbalanced system with the same jobs unbalanced at each stage is likely to resemble a smaller hybrid flow shop (comprising the bottleneck machines at each stage).

The generation of a problem instance with any of the properties given above can be achieved by manipulating the methods of generating operation lengths, processing sets and missing operations for the problem. We generate vertically-unbalanced problems by changing the mean operation length appropriately (see §8.5.3), and horizontally-unbalanced problems by changing the method of assigning processing sets to operations (see §8.5.4).

Initially, we shall only investigate the simpler systems (types 1, 2a and 3a), and we may go on to investigate the more complex types of system after our first sets of experiments.

8.4.3 Other parameters

The other parameters which may be varied to alter the behaviour of the problem are the selection of processing sets used in the instance, the incidence of missing operations in the problem, and the mean operation length. The selection of processing sets at each stage is performed so as to guarantee that they are nested. The details of generating the processing sets are given in §8.5.4, below.

Although in practice the proportion of missing operations may be higher at some stages than at others, we have chosen to make the proportion the same at all stages of a problem, in the interests of reducing the number of factors to investigate. We will, however, investigate problems with few and with many missing operation. For similar reasons, we will only use one value for the mean operation length. The details of generating operation lengths are given in §8.5.3.

8.4.4 Factors and levels

As discussed above, and summarised in table 8.1, there are many different parameters which can be varied to generate different types of hybrid flow shop problem. Each of these parameters we term a *factor*. Each factor may be set at a number of different *levels* in any given experimental test set. The choice of factors and levels used for a particular test set is discussed for each experimental series individually in chapter 9.

Factor	Notation	Suggested levels
Number of jobs	n	10, 50, 200, 1000
Number of stages	s	3, 5, 10, 20
Expected jobs per machine ratio	ρ	2, 3, 6, 10, 20
Balance of workload	see 8.4.2	3-6 types
Selection of processing sets, $\{M^{(1k)}, \dots, M^{(2m_k-1,k)}\}$:		
Skew of proc. set tree	κ	0.15, 0.5, 0.85
Selection of sets from tree	σ	0.2, 0.5, 0.8
Incidence of missing operations	π	0.0, 0.2, 0.6
Expected length of operation	μ	10

Table 8.1: Parameters for generating instances

8.5 Generation of instances

In this section we describe the details of generating instances. For each combination of the test factors described in table 8.1, the parameters for generating an instance take different values. The data to be generated for each instance is summarised in table 8.2. A detailed description of the method of generating each aspect of an instance from the factors used in table 8.1 is given following the table.

Parameter	Description
n	Number of jobs
s	Number of stages
m_k	Number of machines at stage k
p_{jk}	Operation length of job j at stage k (may be missing)
M_{jk}	Processing set of job j at stage k
d_j	Due date of job j

Table 8.2: Parameters for generating an instance

Firstly, the number of jobs, n, and number of stages, s, in each instance is specified by the experimental design. Secondly, for each choice of n, s and ρ, two arrangements of machines are selected. These two arrangements of machines are used for all instances with the given n, s and ρ – half of the instances for each combination of factors with each arrangement. Thirdly the processing sets to be used at each stage are selected. Fourthly, the missing operations are marked. Then, for each non-missing operation, an operation length and a processing set are chosen. Finally, due dates are chosen for each job.

8.5.1 Machine arrangements

For each value of n, s and ρ, two arrangements of machines are selected. For each arrangement, the number of machines at each stage is chosen from a discrete uniform distribution with mean n/ρ. Any discrete uniform distribution of the form $\mathbf{R}(\frac{n}{\rho} - x, \frac{n}{\rho} + x)$ of length $2x$ will suffice. We choose the value x to be $\lceil \frac{1}{3} n/\rho \rceil$. Thus, we shall use

$$m_k \sim \mathbf{R}\left(\left\lfloor \frac{2n}{3\rho} \right\rfloor, \left\lceil \frac{4n}{3\rho} \right\rceil\right)$$

8.5.2 Missing operations

For each job and each stage, the corresponding operation is marked as missing (i.e. not processed at a given stage) with a given probability, π, as listed in §8.4. If a job is given no operations as a result of this procedure, it is assigned one operation at a randomly generated stage.

8.5.3 Operation length

The lengths of operations are chosen so that the expected mean operation length over the whole problem is μ. In practice, this means that the operation length is chosen in different ways for different behaviours of instances and for each stage in a problem. However, all operation lengths at a stage k are chosen from a discrete uniform distribution of length $2\mu_k - 1$ with mean μ_k (to be determined):

$$p_{jk} \sim \mathbf{R}(1, 2\mu_k - 1). \tag{8.1}$$

Balanced workloads

The simplest case is to give a balanced workload profile across every stage of a problem (see §8.4, item 1). In this case, we wish the mean workload at each stage to be the same:

$$n\mu_k\pi/m_k = w \quad \text{for } 1 \leq k \leq s, \tag{8.2}$$

for some workload w to be determined. We also want the expected mean operation length over all stages, μ_k, to be μ:

$$\sum_{k=1}^{s} n\mu_k/ns = \mu. \tag{8.3}$$

Equations 8.2 and 8.3 together form a set of $k+1$ linear simultaneous equations in $k+1$ variables (the μ_k, and w, which is also unknown). Substituting μ_k from the equations 8.2 into equation 8.3, we get

$$w = \mu s n / \sum_{k=1}^{s} m_k. \tag{8.4}$$

Putting this back into equations 8.3, we find that at stage k, the expected mean operation length, μ_k, is

$$\mu_k = \frac{m_k s \mu}{\sum_{l=1}^{s} m_l}.$$

The actual processing times of operations, p_{jk}, is then taken from the distribution in equation 8.1.

Vertically unbalanced workloads

For vertically-imbalanced instances, some stages may have a bottleneck measure β_k times higher than the rest. For a single "bottleneck" stage (e.g. type 2a from §8.4), one stage, b, is chosen at random, $\beta_k = 1$ for all $k \neq b$, and $\beta_b = 2$, say, to give a single stage with a workload twice as high as all other stages. For systems with multiple bottleneck stages (type 2b), several stages, $b_1, b_2, \ldots$, are chosen to have high workload ($\beta_{b_1} = \beta_{b_2} = \ldots = 2$).

We use equation 8.3 as before, and select the operation lengths from the discrete uniform distribution, as before (equation 8.1). The conditions on workload (equations 8.2, above) become

$$n\mu_k/m_k = \beta_k w \quad \forall k, \tag{8.5}$$

for some w to be determined. Substituting 8.5 into 8.3, we obtain

$$w = \mu s n / \sum \beta_k m_k, \tag{8.6}$$

and hence (from equations 8.5)

$$\mu_k = \frac{\beta_k m_k s \mu}{\sum_{j=1}^{s} \beta_j m_j}.$$

8.5.4 Processing sets

Assigning processing sets to operations is done in two steps. First, for each stage, the available processing sets are generated. Secondly, each operation is given a processing set from those available at the relevant stage.

To generate a collection of processing sets to be made available at a stage, a complete set, $\mathcal{M}_k$, of $2m_k - 1$ processing sets is constructed using a recursive process. The parameter κ governs the *skew* of the generated set of processing sets, $\mathcal{M}_k$. A κ of 0 or 1 will produce a highly skewed collection of processing sets (see figure 8.1), whereas a κ value of 0.5 will produce a generally evenly-balanced collection of processing sets (see figure 8.2). Note that since this a random process, setting $\kappa = 0.5$ does not guarantee a perfect balance, but will on average produce more well-balanced trees than any other value of κ.

```
Procedure MAKE_PROCESSING_SETS
begin
  Initialise $P = \{1, \ldots, m_k\}$ to be the set of all machines at stage $k$.
  Initialise the set $\mathcal{M}_k$ to $\{P\}$.
  Call make_subsets($P$)
end

Procedure MAKE_SUBSETS($M$)
begin
```

```
    Given a processing set $M$, construct two disjoint subsets, $M_1$ and $M_2$ by the following
    process:
    Initialise $M_1$ and $M_2$ to be both empty.
    For each machine in $M$, place it with probability $\kappa$ in set $M_1$. If not placed in
    $M_1$, place it in $M_2$.
    If $|M_1| = 0$, choose a single machine from $M_2$ and move it to $M_1$.
    If $|M_2| = 0$, choose a single machine from $M_1$ and move it to $M_2$.
    Add $M_1$ and $M_2$ to $\mathcal{M}_k$.
    If $|M_1| > 1$, recurse make_subsets($M_1$).
    If $|M_2| > 1$, recurse make_subsets($M_2$).
end
```

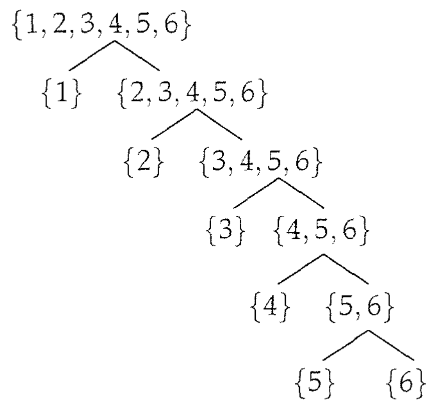


Figure 8.1: Skewed collection of processing sets ($\kappa = 1.0$)

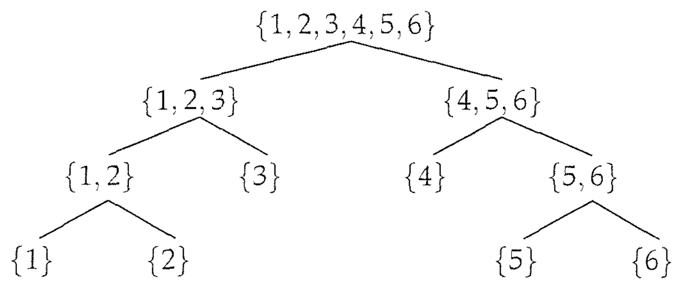


Figure 8.2: Example of a (perfectly-) balanced collection of processing sets ($\kappa = 0.5$)

After generating a complete set of $2m_k - 1$ processing sets, only some are selected for allocation to operations. The full set (the original P, above) is always selected. The other processing sets are each selected with probability σ. Finally, processing sets are allocated to operations at stage k at random from the available ones at that stage.

8.5.5 Due dates

Due dates are selected to give a "reasonable" range for completing jobs. The earliest that a job can be completed is the sum of the processing times of its operations, $\sum_k p_{jk}$. Since we do not know the operation processing times at the point when we choose the due date distribution, we must work on a statistical basis – the expected mean operation processing time is μ, and each job has at most k operations, with a proportion, π, of those being missing operations. Therefore, the expected value of the earliest possible completion time for a job is $k\mu(1-\pi)$, and we use this as the minimum value for the due date distribution.

The latest that we might reasonably expect a job to be completed is after being the last job to be scheduled at each stage, and where the stages have no overlap. Again, we must work on a statistical basis. There are k stages, each of which will have an expected ρ jobs per machine, if there are no missing operations. With operations taking a mean of μ time-units to complete, one would expect each stage's processing to take $\mu\rho(1-\pi)$ time-units, giving an expected value of the latest possible completion time for a job to be $k\mu\rho(1-\pi)$. We use this as the maximum value for the due date distribution. Therefore, we select due dates from the discrete uniform distribution

$$d_{jk} \sim \mathrm{R}(k\mu(1-\pi), k\mu\rho(1-\pi)).$$

In the next chapter, we describe the analysis method and results obtained from computational experiments. The experiments cover much of the range of possible solution methods described in this chapter, and use the problem-generating methods of this chapter to test the algorithm variants over a wide range of possible input problems.

Chapter 9

Experimental results

In this chapter, we describe the computational experiments that we have performed. The aim of these experiments is to answer the questions posed at the end of chapter 7. The techniques which we have used to generate the experimental sets were described in the previous chapter. We first introduce the statistical framework – analysis of variance – which we will use to structure the experiments and results. Then we describe each of four sets of experiments in turn, and analyse their results both individually and taken as a complete corpus of data. Finally, we give a decision tree which can be used to select the optimal algorithm for any given type of input problem.

9.1 Industrial tests

One important experiment we performed was a test with live data taken from Foster Mills. This was to test the applicability of our hierarchical decomposition procedure to the original industrial system which originally inspired the problem. We took a week's worth of production data from the specialist treatment plant and converted it to a mathematical expression of the problem. We used the three-stage hybrid flow shop model of the plant which we developed in §3.5 as the machine model. The complexities of extracting suitable information from the available data meant that it was impractical to attempt to run this experiment with more than one data set. However, the data set used was typical of the plant usage.

After running the algorithm, we drew Gantt charts of the resulting schedule and showed them to the Mill management, including those currently responsible for its scheduling. They concluded that the machine-generated schedule was comparable in quality to the manually-generated schedules which were drawn up daily.

Since we used the decomposition algorithm developed in chapter 7, our solution only dealt with scheduling on the machines, although it took account of the relevant processing sets at each stage. Our solution did not account for the additional constraints from the inter-process storage bins. However, the solution was sufficiently good that it would have required little effort to modify it (even by hand) to fulfil the bin constraints as well.

The major benefit that would have been seen in using our algorithm (or a version of the algorithm which accounted for bin constraints) is that the time taken to generate a schedule drops from several hours to a fraction of a second. This drop in time-scale would make it vastly easier to modify a schedule as and when new orders come into the system – currently a major problem. It also means that jobs will not get omitted or lost during the scheduling process, as our automated computer process is considerably more reliable than the current manual one in that respect.

9.2 Fractional factorial design

In the description of algorithm options and problem configurations given in §8.1, there are eight variable factors governing the type of problem, and four factors governing the type of algorithm to use. Each of these factors has at least two levels. We may select the conditions and algorithm for a single experiment by choosing one level from each factor; this selection may be performed for each factor independently.

If every level of every factor were to be compared against each other, there would be $4 \cdot 4 \cdot 4 \cdot 3 \cdot 2 \cdot 3 \cdot 3 \cdot 2 \cdot 6 \cdot 3 \cdot 6 = 746,496$ possible combinations of the different levels. With a need for several instances of experimentation for each combination (at least 10 is recommended[26]), the number of experiments could be well into the millions.

To reduce the number of experiments required for results, we take two approaches. Our first approach is to perform a set of experiments covering some but not all of the possible levels within each factor. This will enable us to examine the behaviour of some of the possible algorithms, and to discard particularly badly-performing variants, reducing the total number of combinations needed for a second, more complete, run of experiments.

Our second approach is to perform only some of the full set of experiments, omitting experiments in a controlled manner which allows the analysis of variance still to extract the maximum amount of information from the data set. This can be accomplished by the use of a so-called *fractional factorial experimental design*. Such a design uses only a proportion of the combinations in the full grid. To generate a $1/n$ fractional factorial design, we use the following method:

If we denote the factors by $a, b, \ldots$, and the levels of, say, factor a by $a_0, a_1, \ldots$, then a combination of levels $(a_i, b_j, \ldots)$ is used for experiments only if

$$i + j + \ldots \equiv 0 \mod n.$$

(Note that the levels of each factor are indexed from zero.)

This selects $1/n$ of the possible combinations of levels, in an evenly-spread manner such that no combination of factors has more experiments with one set of factors than another. If n is chosen to be greater than the largest number of levels of any factor, then this property does not necessarily hold, so care must be taken to ensure a suitable selection of n.

In the following sections, we set out in detail the factors and levels affecting the problem configuration which we shall use, and how these parameters are used to generate sample problem instances for experimentation.

9.3 Statistical analysis

Given the large number of algorithm options and variation in problem type, some care must be taken to ensure that the results we obtain can be analysed and conclusions drawn as to the efficacy of our algorithms. In particular, there will be some effects in our results which are easily predictable – for example, the time taken to run the algorithm will increase as the number of jobs increases. We must design our experiments and analyse the results so as to be able to account for the known and expected effects, and to reveal the effects which we are trying to investigate.

To this end, we will use analysis of variance (or ANOVA) to inform our experimental design, and to study our results. *ANOVA* [22] is a statistical technique which can be used to separate and identify the effects of several (orthogonal) choices made when performing experiments. As a simple example, consider an experiment with three algorithms, A, B and C, each of which is tested on two types of problem, Y and Z. The example above has two factors, with two and three levels respectively. Thus, six experiments are performed, with the results filling the six squares in the grid shown in table 9.1.

	Alg A	Alg B	Alg C
Problem Y	x_{AY}	x_{BY}	x_{CY}
Problem Z	x_{AZ}	x_{BZ}	x_{CZ}

Table 9.1: Example ANOVA grid

The analysis of variance in this case treats each result as being composed of four components: a constant value, k, a contribution dependent on the algorithm type, a_i ($i \in \{A, B, C\}$), a contribution dependent on the problem, b_j ($j \in \{Y, Z\}$), and a contribution dependent on both algorithm and problem, ε_{ij}. The ANOVA thus breaks down each result x_{ij} as

$$x_{ij} = k + a_i + b_j + \varepsilon_{ij}.$$

Most computer implementations of ANOVA ensure that the mean of the a_is, the mean of the b_js and the mean of the ε_{ij}s are all zero.

With three factors, ANOVA would break down each result as

$$x_{ijk} = k + a_i + b_j + c_k + \{ab\}_{ij} + \{ac\}_{ik} + \{bc\}_{jk} + \varepsilon_{ijk},$$

using not only the first-order effects of the three main factors separately, a_i, b_j and

c_k, but also the second-order effects of each pair of main factors, $\{ab\}_{ij}$, $\{ac\}_{ik}$ and $\{bc\}_{jk}$, and the third-order effects, ε_{ijk}.

Analysis of variance can operate not only with single measurements in each "cell" of the experimental design, but also with samples consisting of several results from similar experiments. In this case, the analysis is performed to identify the effects on the sample means. It is not necessary to ensure that each cell has the same number of readings in its sample; it is not even necessary to have a sample in every cell. However, ANOVA does make the assumption that the samples in all cells come from populations with identical variances.

When given a sufficiently large number of values in each sample, ANOVA can yield confidence intervals for the effect of each level of each factor and for each of the higher-order effects. The confidence interval is an estimate of whether the effect under consideration is indistinguishable from zero, and can be used to identify factors, individual levels of factors or higher-order effects which have no apparent effect on the sample mean. There are a number of computer programs available which perform analysis of variance. We use SPSS for this task.

9.4 Preliminary investigations

Due to the impossibly large size of a complete set of experiments covering all levels of all factors (see §9.2), we have only taken a reduced set of factors and levels in each of two sets of experiments. Neither set covers all of the possible ground, but between them we will obtain enough information to develop and properly target the main investigative experiments.

In order to identify which factors are worth investigating, we first make a broad but incomplete investigation of the whole space of possible combinations. From this broad investigation, we can identify which factors and combinations of factors should be used for more detailed and complete experiments. We present the set-up and results for each of the two sets of preliminary experiments below.

9.5 Preliminary investigation – first set

We present the results for each set of experiments in three parts: firstly, the set-up of the experiments, showing which levels and factors were used to generate the test problems and which algorithm variants were tested. Secondly, we present the main effects and second-order effects affecting the objective function, as found by the ANOVA analysis. Finally, we give a short discussion about the amount of time it takes the algorithm to run and therefore the size of the problems it can solve.

The first set of experiments uses the reduced set of factors and levels given in table 9.2. Some factors are omitted from the design (processing set skew, for example), and hence have only a single level. Some factors, such as the sub-problem solution procedure, do not cover all of the possible levels of the factor. These initial experiments will indicate if the factors are relevant to the performance of the algorithm, and hence whether it is worth investigating in greater depth. Despite the reduction in size, this design has 1944 cells in the ANOVA grid. We selected a 1/3 fractional design, with 20 experiments in each cell, for a total of 12960 individual experimental runs.

Factor	Levels
Number of jobs, n	10, 50, 200
Number of stages, s	3, 5, 10
Jobs/machine ratio, ρ	10, 6, 3
Processing set skew, κ	0.5
Processing set selection, σ	0.2
Balance of workload	Type 1, Type 2a (see 8.4.2)
Missing operations, π	0.0
Mean operation length, μ	10
Decomposition method	Stage, Execution set
Criticality measure	Stage order, Reverse stage order, NLBW
Sub-problem solution procedure	JPS, NJ
Backtracking method	None, Single-pass, Multi-pass

Table 9.2: Preliminary experiments, first set: experimental design

For each experimental run, we recorded the maximum lateness of the resulting solution, and the processor time taken by the solution process. The results were processed through the ANOVA component of SPSS, and the main effects and second-order interactions were examined. It proved impossible to investigate third order or higher effects in SPSS due to memory restrictions.

We can show the main and second-order effects graphically using thumbnail graphs such as that in figure 9.1. Each graph shows the levels of one main effect across the horizontal axis. For those graphs showing second-order effects, the levels of the second factor are shown with different point shapes. A significant second-order effect will show up as non-parallel data series in the graph. On some graphs, "whiskers" can be seen extending above and below the data points, indicating the extent of the 95% confidence interval for each data point. Where there are no visible whiskers, they are smaller than the size of the data point, and are hidden by it. The means plotted on the graphs are not exact means of the notional underlying population from which the samples have been taken. Rather, they are estimates based on the available data. The 95% confidence intervals give a range inside which the actual mean is expected to lie (with 95% probability).

If two samples in the analysis have differing means, it is useful to know whether they are significantly different from each other, or whether they are within the bounds of experimental error. This function is also performed by the confidence intervals: if one sample mean is outside the confidence interval of the other, then the samples are from different populations. This type of comparison is performed automatically by the ANOVA procedure, and is given as a *significance value* for each comparison. The significance value is the probability that the samples being compared come from the same population. In our analysis, where we are working at with 95% confidence intervals, a significance value lower than 5% (0.05) indicates that the samples being compared are significantly different.

In a first-order comparison, a statistically significant comparison indicates that at least two of the levels being compared have sample means which differ by more than their confidence intervals. In a second-order comparison, where the interaction of two different factors is tested, a significant comparison indicates that at least one combination of levels of the two factors has an effect which cannot be explained by an addition of the two relevant first-order effects. Thus, in first-order comparisons, we are looking at the primary differences between levels of one factor. Whereas in a second-order comparison, we are looking for synergistic (or dysergistic) effects from particular levels of two factors operating together.

9.5.1 Main effects

We start by examining the effects on the objective function of each factor separately. The results are summarised in table 9.3. In the table, we see that every factor has significant variation between its levels except for the sub-problem solution procedure. However, we would expect to see many of these significant effects in any set of experiments. In particular those factors above the double line in the table affect the size and type of the problem. These factors are certain to produce variations in the objective function. Briefly,

Jobs: As the number of jobs increases, so does the maximum lateness. With only 10 jobs, the mean $L_{\max}$ is -6.9, rising to 31.6 for 50 jobs, and 52.9 for 200 jobs.

Stages: As the number of stages increases, so does the maximum lateness. Our 3-stage problems have a mean objective function value of 24.4, rising to 30.1 for 10-stage problems. This apparently small change (particularly with reference to the large change as a result of the number of jobs) is probably due to the way in which we calculate job due dates when generating problem instances, which aims to minimise this particular effect.

Jobs/machine: As the number of jobs per machine increases, so does the maximum lateness, since each individual machine is more heavily loaded, and operations become more delayed.

Workload balance: Balanced problems have a lower $L_{\max}$ value (mean of 21.6) than problems with bottleneck stages (with a mean of 30.2). This is to be expected, since the bottleneck problems have greater job congestion at the bottleneck stages.

Of more interest are the effects of the algorithm options, individually as main effects and in combination with each other and with the instance parameters.

Each of these effects is shown in the thumbnail graphs on the following pages.

Decomposition: Execution set decomposition, with an $L_{\max}$ of 28.5, is less effective than stage decomposition, with an $L_{\max}$ of 23.3. See figure 9.1.

Criticality measure: Adding the sub-problem solutions to the partial solution in stage order (mean objective function of 11.2) is better than doing it in reverse stage order ($L_{\max} = 25.4$), which is considerably better than doing it in NLBW order ($L_{\max} = 41.1$) (see figure 9.2).

The difference between stage order and reverse stage order is probably due to the fact that due dates and release dates in the sub-problems are not dealt with in the same way: release dates are hard constraints, whereas due dates may be exceeded. As a result, the problem is not reversible, so an asymmetry would be expected. It makes more sense to solve sub-problems in stage order, as then the release dates in each sub-problem are a realistic surrogate for the earlier operations. On the other hand, solving sub-problems in reverse order does not use representative values for release dates until all sub-problems have a solution.

Sub-problem solution procedure: There is a small, but statistically significant, difference between the Jackson for Processing Sets (JPS) and the Nested Jackson (NJ) sub-problem solution method (figure 9.3). The NJ algorithm appears to have a small benefit over the JPS algorithm, by 26.5 to 25.3.

Backtracking: No backtracking at all appears to be better than either of the backtracking methods we have tried (figure 9.4). No backtracking gives a mean $L_{\max}$ of 24.1, whereas single-pass backtracking gives a mean of 26.5, and multiple-pass backtracking a mean of 27.0. This may be due to the sub-problems becoming too tightly-coupled

9.5.2 Second-order effects

When we examine second-order effects, we are looking for results which are better (or worse) than can be explained by a sum of the first-order effects. In other

Factor	Significance
Jobs	**0.000**
Stages	**0.000**
Jobs/machine	**0.000**
Workload balance	**0.000**
Decomposition	**0.000**
Criticality measure	**0.000**
Sub-problem solution procedure	**0.014**
Backtracking	**0.000**

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant effect of the factor.

Table 9.3: Preliminary experiments, first set: main effects affecting $L_{\max}$

words, are there particular combinations of levels of particular factors which perform significantly better or worse than expected? As in the previous section, we analyse the results using the significance tests from the analysis of variance. A summary of the results can be found in table 9.4.

As before, we are not particularly interested in effects involving the factors governing instance size or shape. Although they all have highly significant interactions, these are entirely expected results. We are more interested in interactions where one or both factors are an algorithm variant. These are shown in table 9.4 in the right-hand four columns, and the bottom three rows for interactions involving two algorithm variants.

The most obvious point about these results is that the sub-problem solution procedure has no significant interactions with any other factor. This is not an unexpected result, given that the choice of sub-problem solution procedure does not have a strongly significant effect on the solution quality (although it is significant). Also expected are the second-order effects of decomposition, criticality measure, workload balance and backtracking against jobs, stages and jobs per machine. These all simply follow further the trends established by the underlying main effects.

More interesting are the interactions between the different algorithm variants (in the bottom right-hand corner of table 9.4). There are three significant interactions between the different levels of these factors:

Criticality measure and decomposition method: The NLBW criticality measure is much better (by 13.1 time units) when used with the stage decomposition method compared to using it the execution set decomposition (figure 9.5). The other two criticality measures we examined are not affected by the decomposition used to anywhere near as great a degree, although the effect of decomposition with the reverse stage criticality measure is also significant (but only by 2.8 time units).

Decomposition method and backtracking: Working with execution set decomposition rather than stage decomposition gives a greater variability in performance across other algorithm options, usually for the worse (see figure 9.6). Using stage decomposition, the different backtracking methods show a total variation of 0.4 time units, whereas execution set decomposition shows a much greater variation of 5.5 time units.

Backtracking and criticality measure: Although this effect is not strictly significant by the letter of the statistical test, it is only barely outside the cut-off value, and is worth mentioning: We find that backtracking has little effect if the subproblems are solved in stage order, and has a worsening effect if the subproblems are solved in any other order (see figure 9.7).

9.5.3 Execution time

All of our test runs were performed on a uni-processor AMD K6-2/500 – roughly equivalent to a Pentium II/500. The actual CPU usage was measured using the timing functions built into the standard POSIX C library. These functions are accurate to 0.01 CPU seconds (1 cs) on the architecture we used.

The running time of the algorithm is generally very small – most of the problems were solved in under 0.1 s. However, some of the problems took over 5 minutes to solve with some algorithm variants. Specifically, the longest running times

Factor	Stages	Jobs/machine	Workload balance	Decomposition	Crit. measure	Sub-prob. sol. proc.	Backtracking
Jobs	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.122	**0.000**
Stages		**0.000**	**0.000**	**0.000**	**0.000**	0.165	**0.000**
Jobs/machine			**0.002**	**0.000**	**0.000**	0.664	0.398
Workload balance				**0.015**	**0.000**	0.271	0.743
Decomposition					**0.000**	0.739	**0.000**
Criticality measure						0.417	0.052
Sub-problem solution							0.241

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant interaction between the factors.

Table 9.4: Preliminary experiments, first set: significance of second-order effects affecting $L_{\max}$

all came from the largest size problems (200 jobs, 10 stages), using the multiple-looping backtracking method and decomposition by execution set. The effect of decomposition by execution set of increasing the running time of the algorithm is quite marked: on average, the execution set decomposition variants took 17.7 s, whereas the stage decomposition variants took 0.29 s. This is probably an effect of there being many more sub-problems, causing many more expensive backtracking phases in the algorithm.

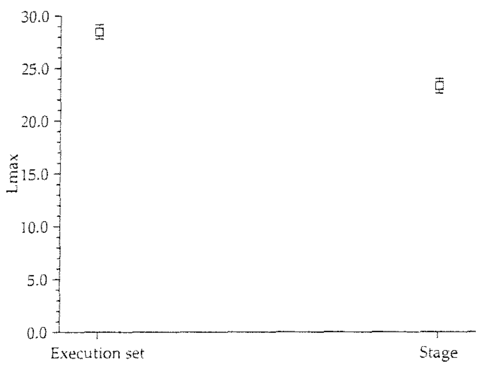


Figure 9.1: Decomposition method: $L_{\max}$

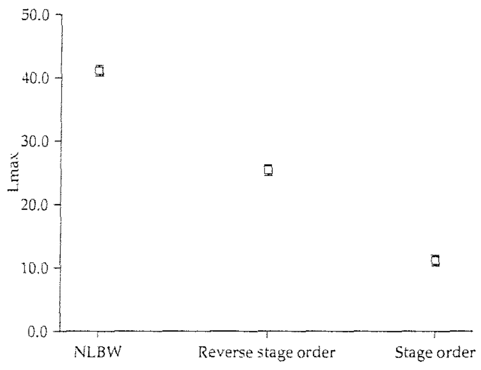


Figure 9.2: Criticality measure: $L_{\max}$

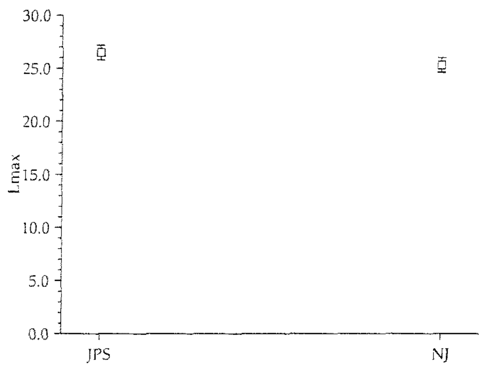


Figure 9.3: Sub-problem solution procedure: $L_{\max}$

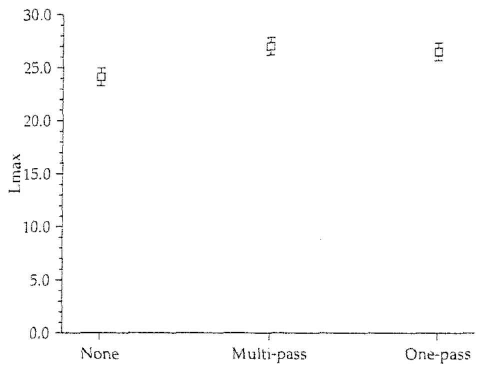


Figure 9.4: Backtracking method: $L_{\max}$

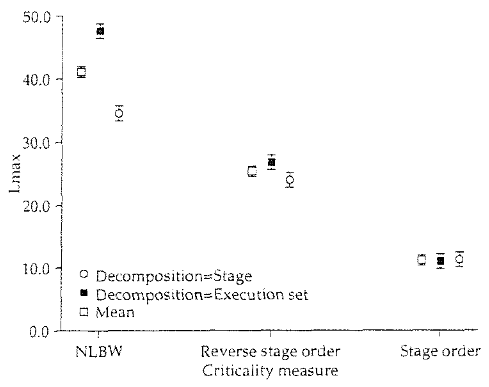


Figure 9.5: Criticality measure and decomposition: $L_{\max}$

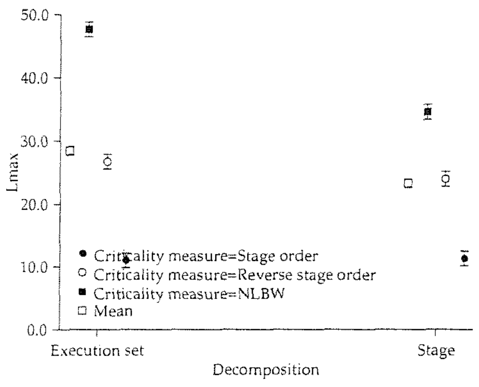


Figure 9.6: Decomposition and criticality measure: $L_{\max}$

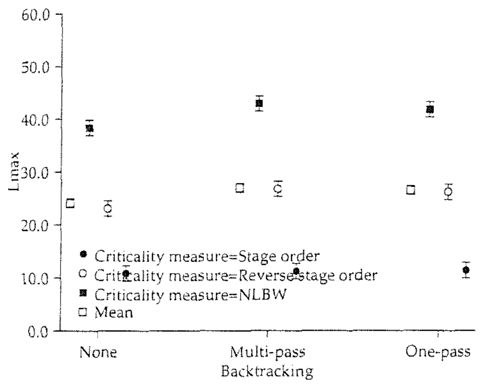


Figure 9.7: Backtracking and criticality measure: $L_{\max}$

9.6 Preliminary experiments – second set

The second set of experiments includes all of the factors not investigated in the first set, and omits some factors and levels of factors. In particular, we omit the very large problems of 200 jobs, since the algorithm variants behaved in the same way for the large problems as for the small, and the omission of the large problems will significantly reduce the amount of time to run the tests. We also omit the $\rho = 5$ jobs per machine level since the effects across the values of ρ appeared to be continuous from the previous experiments. We omit one of the sub-problem procedures, the NJ solution method. The performance of the two sub-problem solution methods used in the last set of experiments was sufficiently close that we need only test with a single method in this set of experiments.

A summary of the factors and levels used in these experiments is given in table 9.5. There is a total of 7776 cells in the ANOVA grid from this design. With a 1/3 fractional factorial design and 20 instances per cell, this gives 51840 experiments in total.

Factor	Levels
Number of jobs, n	10, 50
Number of stages, s	3, 10
Jobs/machine ratio, ρ	10, 3
Processing set skew, κ	0.15, 0.5, 0.85
Processing set selection, σ	0.4, 0.8, 1.0
Balance of workload	Type 1, Type 2a (see 8.4.2)
Missing operations, π	0.0, 0.4, 0.8
Mean operation length, μ	10
Decomposition method	Stage, Execution set
Criticality measure	Stage order, Reverse stage order, NLBW
Sub-problem solution procedure	JPS
Backtracking method	None, Single-pass, Multi-pass

Table 9.5: Preliminary experiments: experimental design, second set

9.6.1 Main effects

The mean objective function values in the analysis of this second run of experiments is much smaller than in the previous set, since the larger problems (with 200 jobs) have been omitted in the interests of reducing the size of the experimental data set and computation time required to perform the tests. Most of the main effects used in the current set of experiments show a statistically significant effect on the objective function value. See table 9.6 for a summary of the significant effects. Many of these – primarily those governing the instance shape – are, as in the first set of experiments, expected. Those results worthy of more detailed comment are:

Density of processing sets: The inclusion of a small number of processing sets makes $L_{\max}$ higher than choosing large numbers of processing sets or none at all (figure 9.8). Picking 40% of the available processing sets makes the mean $L_{\max}$ value 5.4 time units higher than using all of the available processing sets in our tests, compared to a standard error of 0.22 at the 5% level. This is an artefact of the way that problems are generated: with only a few processing sets available at a stage, some of only a few machines in size, the "even" assignment of processing sets to operations ensures that some machines are more heavily overloaded than others. This overloading causes operations on those machines to be delayed, increasing the maximum lateness.

Processing set skew: There is a statistically significant difference between problems with different processing set skews. The data are surprising, showing an unbalanced response – problems with processing sets skewed to the "left" ($\kappa = 0.15$) show a higher mean than well-balanced problems ($\kappa = 0.5$), which in turn have a higher mean objective function value than problems skewed to the "right" ($\kappa = 0.85$). The "left-skewed" problems have a mean of 18.6, falling to 17.4 for the right-skewed problems. We would expect the left- and right-skewed problems to have similar mean values. There is no obvious reason for this difference in either the data generation or solution methods.

Balance of workload: As we saw in the first set of experiments, balanced problems have a lower workload – with a mean $L_{\max}$ of 15.0 compared to problems with bottleneck stages, where the mean is 21.2.

Proportion of missing operations: The proportion of missing operations affects the objective function, as might be expected. With no missing operations, the mean objective function is 14.9. As the number of missing operations increases, so does the objective function, rising to 21.6 with 80% missing operations. One might expect fewer operations (i.e. more missing operations) to cause a drop in $L_{\max}$, but this effect appears to be over-compensated for by the correction factor of $1 - \pi$ in the calculation of due date when generating problems, where π is the proportion of missing operations (see §8.5.5 for more details).

Criticality measure: The results for the effect of criticality measure are almost identical to those from the first set of experiments, with the NLBW criticality measure being significantly worse ($L_{\max} = 24.5$) compared to either reverse stage order ($L_{\max} = 18.3$) or stage order ($L_{\max} = 11.5$) (figure 9.10).

Backtracking: The effect of the backtracking method is not statistically significant at the 5% level. However, it shows the same type of shape as the previous results, even if they are not significant (figure 9.11).

9.6.2 Second-order effects

Looking in more detail at the second-order interactions between pairs of factors summarised in table 9.7, we see that most of the results are similar to those of the previous set of experiments. Those results of most interest are examined in more detail below.

Processing set skew: The left-right skew of processing sets has no significant second-order interactions with any other factor in the experiment, barring

Factor	Significance
Jobs	**0.000**
Stages	**0.000**
Jobs/machine ratio	**0.000**
Processing set skew	**0.001**
Processing set density	**0.000**
Balance of workload	**0.000**
Missing operations	**0.000**
Decomposition	0.302
Criticality measure	**0.000**
Backtracking	0.420

Table 9.6: Preliminary experiments, second set: main effects affecting objective function

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant effect of the factor.

the number of jobs and the number of stages which have such a large effect on their own that we would expect to see some kind of interaction.

Decomposition method and number of stages: Decomposition by execution set is worse than by stage for large numbers of stages, but better for fewer stages. The difference is small – only a matter of 0.7 units better for 3 stages, and 1.3 units worse for 10 stages. See figure 9.12.

Missing operations and decomposition method: As the proportion of missing operations increases, decomposition by execution set improves relative to decomposition by stage (figure 9.13). At 80% missing operations, decomposition by execution set is better by 0.8 units on average, compared to being 1.9 units worse with no missing operations. The effect is probably due to the decreasing linkage between problems as the number of missing operations increases. Each job directly affects those sub-problems in which it has operations. With more missing operations, each job directly affects fewer sub-problems, thus in general decreasing the effect that any one sub-problem may have on any other.

Decomposition and criticality measure: With the stage and reverse-stage orderings for the criticality measure, there are very small (although statistically significant) deviations from the mean for each of the two decomposition methods. Decomposition by execution set is slightly preferable by 0.5–0.6 units. However, when using the NLBW criticality measure, decomposition by stage is clearly better than decomposition by execution set, by 1.0 units (figure 9.14). This is the effect seen in the previous set of experiments.

9.6.3 Execution time

This second set of experiments behaves in general in exactly the same way as the first set of preliminary experiments. It is worth making the comment that although there were four times as many experiments in this set as the previous set, in practice it took less than a quarter of the time to run compared to the first set of experiments (12 hours compared to 2.5 days elapsed time). This appears to be almost exclusively due to the omission of the 200-job instances, which take a very long time to solve – particularly with multiple-looping backtracking and decomposition by execution set.

Factor	Stages	Jobs/machine	Proc. set skew	Proc. set sel.	Workload balance	Missing ops.	Decomposition	Crit. measure	Backtracking
Jobs	**0.000**	**0.000**	**0.021**	**0.000**	**0.000**	**0.000**	0.055	**0.000**	0.487
Stages		**0.000**	**0.000**	**0.000**	**0.001**	**0.000**	**0.000**	**0.000**	0.314
Jobs/machine			0.472	0.461	**0.003**	**0.000**	**0.020**	**0.000**	0.863
Proc. set skew				0.270	0.731	0.391	0.974	0.876	0.772
Proc. set sel.					**0.000**	**0.000**	0.781	**0.000**	0.693
Workload balance						**0.000**	0.395	**0.000**	0.920
Missing ops.							**0.000**	**0.000**	0.712
Decomposition								**0.000**	0.400
Crit. measure									0.638

Table 9.7: Preliminary experiments, second set: significance of second-order effects affecting objective function

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant interaction between the factors.

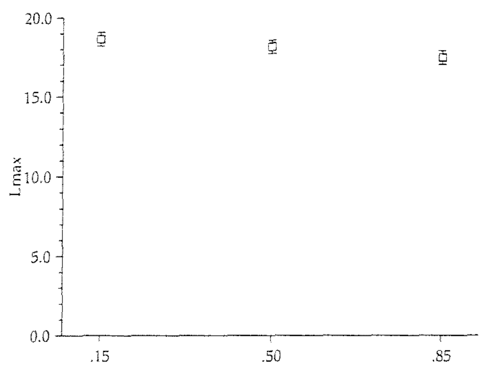


Figure 9.8: Numbers of processing sets: $L_{\max}$

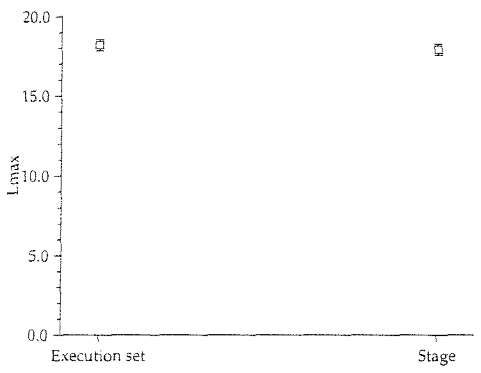


Figure 9.9: Decomposition method: $L_{\max}$

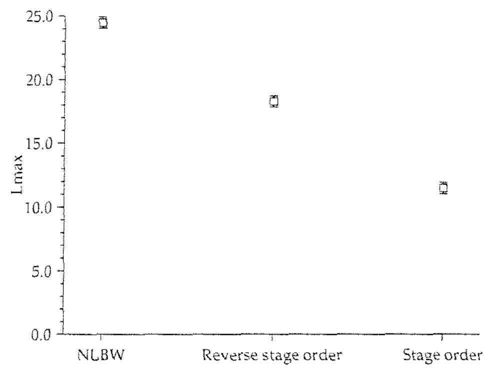


Figure 9.10: Criticality measure: $L_{\max}$

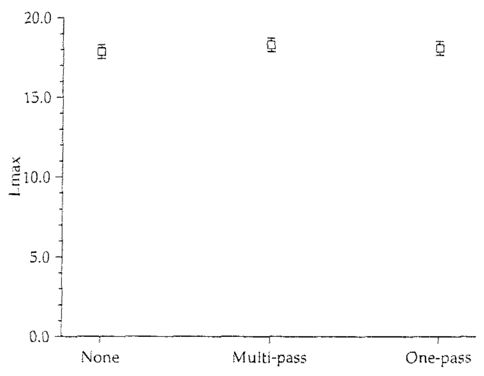


Figure 9.11: Backtracking method: $L_{\max}$

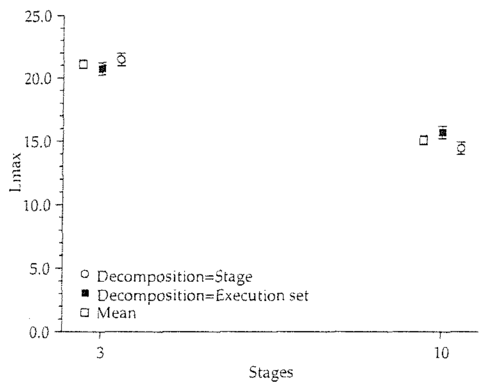


Figure 9.12: Stages and decomposition: L_{max}

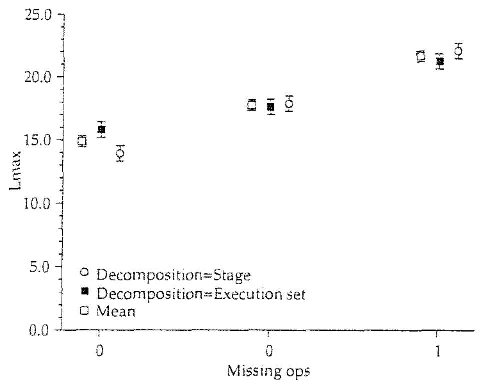


Figure 9.13: Missing operations and decomposition: L_{max}

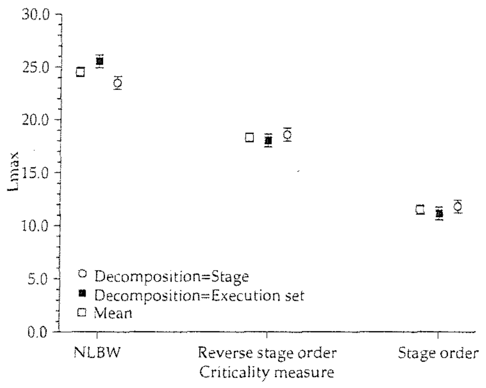


Figure 9.14: Decomposition and criticality measure: L_{max}

9.7 Targeted experiments

We complete the results of the computational experiments with a set of slightly more targeted experiments. We omit some of the less interesting factors, such as balance of workload, which does not show any useful interactions with the algorithm variants, and attempt to concentrate more on those factors which have shown significant first- and second-order effects. We also broaden some of the factors to include more levels – in particular, we examine five different criticality measures, rather than the three which we have used previously. The factors used in the experiments are summarised in table 9.8.

The design consists of 6480 cells in the ANOVA grid – much larger than the first set of experiments, and slightly smaller than the second. We used a 1/4 fractional design with 20 instances per cell. The design was slightly unbalanced, providing 1618 different combinations (as opposed to the 1620 expected), and hence, at 20 experiments per cell, 32360 experiments in total. Although it may theoretically have an impact, we do not believe that the small imbalance in the design will materially affect the results.

Factor	Levels
Number of jobs, n	10, 100
Number of stages, s	3, 10, 20
Jobs/machine ratio, ρ	10, 3
Processing set skew, κ	0.5
Processing set selection, σ	0.1, 0.3, 0.9
Balance of workload	Type 1 (see 8.4.2)
Missing operations, π	0.0, 0.5, 0.9
Mean operation length, μ	10
Decomposition method	Stage, Execution set
Criticality measure	Arbitrary, Stage, Rev. stage, NLBW, Workload
Sub-problem solution proc.	JPS, EDD
Backtracking method	None, Single-pass, Multi-pass

Table 9.8: Targeted experiments: experimental design

9.7.1 Main effects

The significance values of the main effects observed in these experiments are summarised in table 9.9. All of the main effects tested in this set of experiments are statistically significant. The effects of the instance parameters (the first five factors in the table) have been discussed previously. The four factors of the algorithm variant are discussed below:

Decomposition method: As we saw in the previous sets of experiments, decomposition by stage is in general better than decomposition by execution set. In this experimental run, it is better by 24.3 time units to 26.7 units. See figure 9.15.

Criticality measure: The criticality measure is a factor on which we have used more levels than in previous experiments. Specifically, we have run these experiments with the addition of an arbitrary criticality measure, and a workload-based criticality measure. The results (figure 9.16) show that arbitrary order is at least as good as reverse stage order (26.1 time units on average, compared to 26.6 time units for reverse stage order, with a confidence interval of 1.3 units). The workload measure does not perform well, being as bad as the worst-performing NLBW measure (workload $L_{\max}$ = 29.3 time units; NLBW $L_{\max}$ = 30.2 units).

Sub-problem solution procedure: The EDD method is considerably more effective than the JPS algorithm, by 13.7 to 37.4 time units (see figure 9.17). This is a surprising result, as we would expect the JPS method to be better, since it is descended from Jackson's method, which outperforms the equivalent (non-nested) earliest-due-date method for solving parallel machine problems without processing set restrictions.

Backtracking: In these experiments we again see that no backtracking performs better ($L_{\max} = 24.4$) than the other two forms of backtracking we tested. However, in these experiments, multiple-pass backtracking works better ($L_{\max} = 25.6$) than single-pass ($L_{\max} = 26.6$). See figure 9.18.

Factor	Significance
Jobs	0.000
Stages	0.000
Jobs/machine	0.000
Processing set selection	0.000
Missing operations	0.000
Decomposition	0.000
Crit. measure	0.000
Sub-problem solution proc.	0.000
Backtracking	0.012

Table 9.9: Targeted experiments: main effects

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant effect of the factor.

9.7.2 Second-order effects

Here we see in table 9.10 that most of the second-order effects are also statistically significant at the 5% level. Only the backtracking has some non-significant second-order interactions. Those interactions of particular interest are described below:

Number of stages and decomposition method: With a small number of stages, there is no significant difference between the decomposition methods we compared. However, as the number of stages increases, stage decomposition becomes increasingly more effective, with stage decomposition giving an $L_{\max}$ of 12.7, compared to an $L_{\max}$ of 16.5 with execution set decomposition (see figure 9.19).

Number of stages and criticality measure: Increasing the number of stages, we see that the performance differences between the various criticality measures also increase. With three stages, there is no significant difference (or only a small difference) between the criticality measures – a total spread of 2.8 time units on average. Going up to 20 stages, however, increases the total range to 27.9 time units (see figure 9.20). With a small number of stages, the best criticality measure to use is (marginally) the reverse stage order;

with larger numbers of stages, the best measure is clearly stage order, with reverse order and NLBW coming last.

Jobs per machine and backtracking: With few jobs per machine, there is no significant difference between the different backtracking methods. However, with larger numbers of jobs per machine, no backtracking appears to be the better option (see figure 9.21), giving a mean objective function value of 11.0 time units, compared to 14.7 for the next best (multiple pass backtracking), with a confidence interval of 1.4 units.

Processing set density and algorithm variants: We see from these results that as the density of processing sets increases, the algorithm variants generally become closer together in performance terms. See figures 9.22 and 9.23 for examples.

Missing operations and algorithm variants: As with the processing set density, as the number of missing operations increases, the performance of the different algorithm variants becomes closer. Examples are shown in figures 9.24 and 9.25.

Sub-problem solution procedure and decomposition method: The earliest due date algorithm (EDD) shows no significant difference in performance between the two decomposition methods ($L_{\max} = 13.7$). However, with the JPS algorithm, stage decomposition is better than execution set decomposition, by 35.0 to 39.8 time units. See figure 9.26.

Criticality measure and backtracking: Using the most effective stage-order criticality, multiple-pass backtracking has the best effect. However, this effect is reversed when using reverse stage order criticality. For the other criticality measures, there is little or no significant effect on the quality of results between the different backtracking methods. See figure 9.27.

9.7.3 Execution time

Again, the results regarding execution time are little different from the previous results in the two preliminary sets of experiments. This set of experiments also took much less time to run than the first set of preliminary experiments due to the smaller problem sizes (up 100 jobs only).

Factor	Stages	Jobs/machine	Proc. set sel.	Missing ops.	Decomposition	Crit. measure	Sub-prob. sol. proc.	Backtracking
Jobs	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.090
Stages		**0.000**	**0.000**	**0.000**	**0.014**	**0.000**	**0.000**	0.231
Jobs/machine			**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
Proc. set sel.				**0.000**	**0.001**	**0.000**	**0.000**	**0.034**
Missing ops.					**0.000**	**0.000**	**0.000**	**0.006**
Decomposition						**0.000**	**0.000**	0.171
Crit. measure							**0.000**	**0.000**
S-P.S.P								**0.001**

Table 9.10: Targeted experiments: significance of second-order effects

Analysis performed at 5% significance level. Values under 0.05 (in bold) indicate a significant interaction between the factors.

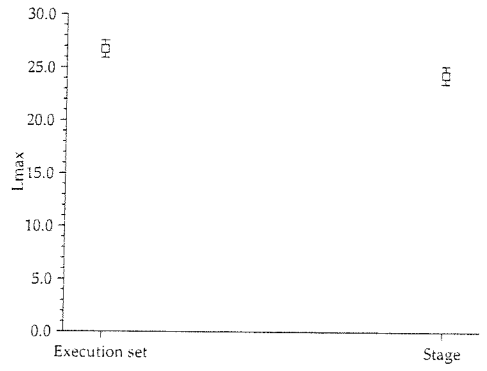


Figure 9.15: Decomposition method: $L_{\max}$

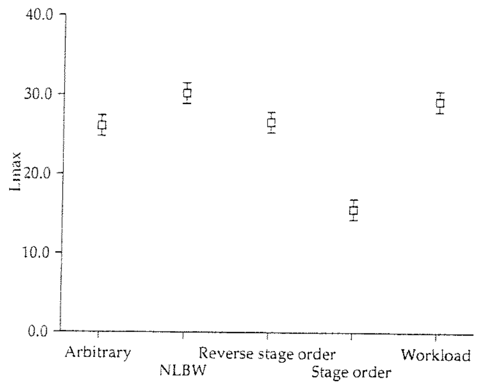


Figure 9.16: Criticality measure: $L_{\max}$

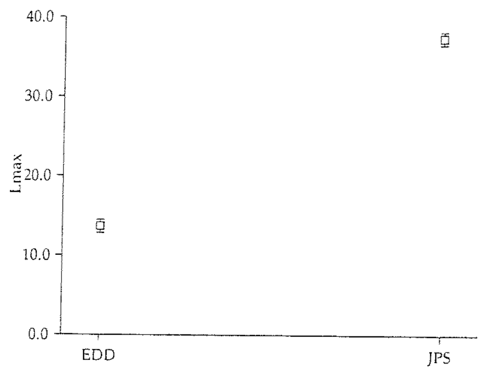


Figure 9.17: Sub-problem solution procedure: $L_{\max}$

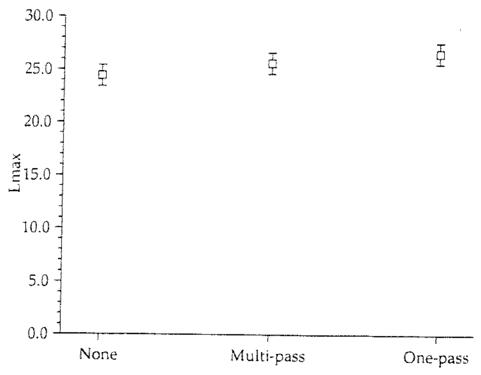


Figure 9.18: Backtracking: $L_{\max}$

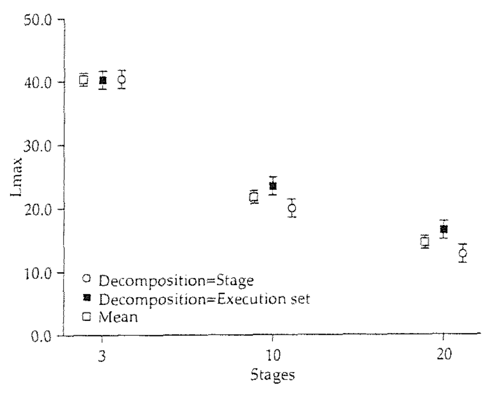


Figure 9.19: Decomposition method and number of stages: $L_{\max}$

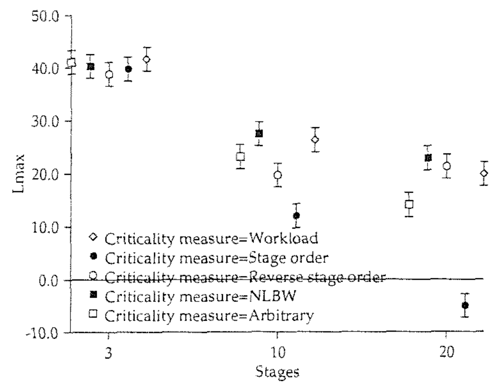


Figure 9.20: Criticality measure and number of stages: $L_{\max}$

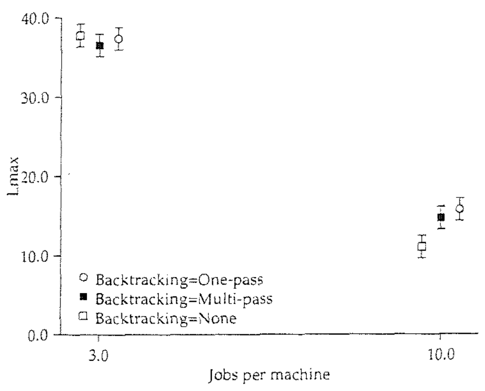


Figure 9.21: Jobs per machine and backtracking method: $L_{\max}$

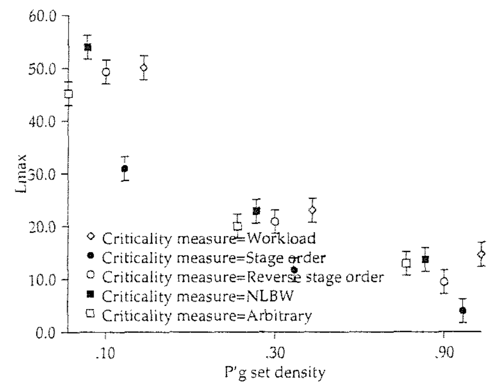


Figure 9.22: Processing set density and criticality measure: $L_{\max}$

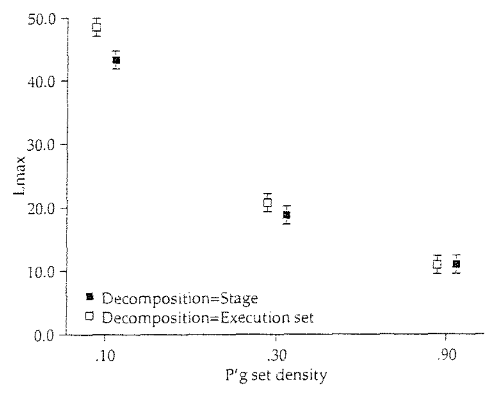


Figure 9.23: Processing set density and decomposition: $L_{\max}$

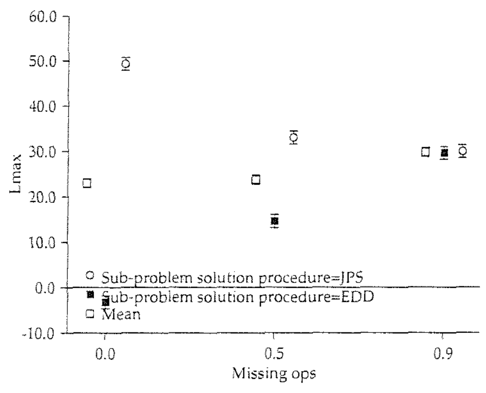


Figure 9.24: Missing operations and sub-problem solution procedure: $L_{\max}$

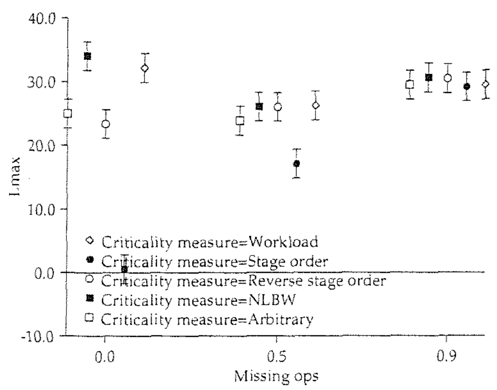


Figure 9.25: Missing operations and criticality measure: $L_{\max}$

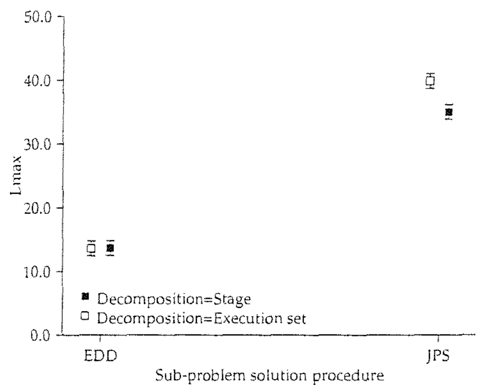


Figure 9.26: Sub-problems solution procedure and decomposition: $L_{\max}$

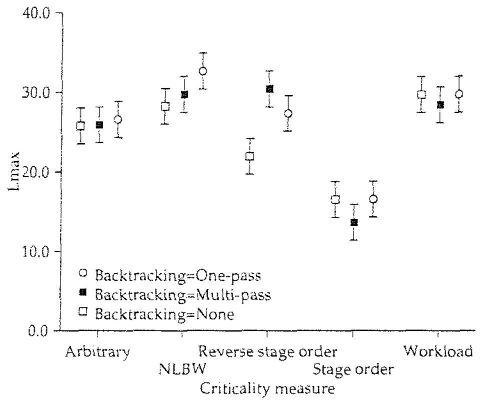


Figure 9.27: Criticality measure and backtracking: $L_{\max}$

9.8 Conclusions

In this chapter we have described three sets of tests of the hybrid flow shop solution method, and analysed the results of those tests. The analysis of each individual set is given separately in the preceding sections of this chapter. In this section, we aim to give a description of the overall results from our experiments, and to give advice on which algorithm variants should be used under different circumstances.

It should be noted that the scale of the testing performed in this chapter is considerably greater than that performed for most work in the mathematical scheduling literature: for example, the YKP paper tests three algorithms, on 11 different problem classes, where we have effectively tested 90 algorithms on over 500 classes of problem. We have performed just over 97,000 individual experimental runs, generating a database of experiments and results several hundred megabytes in size.

9.8.1 Observations

The first important observation to make about our method is that it produces effective solutions which could be used for the real-life scheduling problems at Foster Mills. Despite the fact that we omit several features of the specialist treatment plant – most particularly the inter-process storage bin constraints – our algorithm generates schedules which are usable in practice with only minor modifications, if any. The quality of our solutions is comparable to that of the manual scheduling process currently employed, and our method is several orders of magnitude faster.

Secondly we look at the theoretical performance of the algorithm obtained from our comprehensive computational tests. Examining the main (first-order) effects on the objective function, we can divide the results into two parts. Firstly, there are certain obvious and expected results caused by variation in the instance parameters:

- As the number of jobs increases, so does the objective function value.

- As the number of jobs per machine increases, so does the objective function value.
- As balance of the problem changes from well-balanced to possessing bottlenecks, the objective function is affected.

Secondly, we can make some overall observations about the separate variants of each component of our hierarchical decomposition algorithm:

- The decomposition method has a small effect on the objective function, with decomposition by stage being better than decomposition by execution set.
- The initial order of solution of the sub-problems (the criticality measure) is very important to the solution quality. The best quality solutions are produced when the sub-problems are solved in stage order, and the worst when they are solved using the NLBW- or simple workload-based methods.

  We believe that this effect is caused by the temporal asymmetry of the problem: release dates are absolute constraints which may not be broken, and due dates may be broken. When solving a problem in stage order, the release dates can be calculated exactly, and make a good surrogate for the preceding problems. Solving the problem in any other order means that the release dates used while the algorithm is in progress do not necessarily reflect the preceding problems.
- The simple earliest-due-date solution procedure appears to be better than either the Jackson for processing sets (JPS) or nested Jackson (NJ) methods. NJ has a slight advantage over JPS.

  Solving maximum lateness on parallel identical machines with release dates is a strongly **NP**-hard problem. Jackson's method appears to behave very well in solving the problem. However, it is likely that when we generalise the problem slightly to impose processing set restrictions (even if they are only nested), we force the problem away from the "easy" state where Jackson's method, or some similar method, such as our NJ or JPS, can solve it easily.

- Backtracking appears to make the solutions worse in general – multiple backtracking more than single-pass backtracking. This comes from the interaction of the backtracking method with the non-stage-order criticality measures. These criticality measures require more flexibility in the interim partial solutions so that when sub-problems are incorporated earlier in the solution it can be re-shuffled to account for the new information. By performing backtracking to improve the partial solutions, that flexibility is reduced, and the overall solution quality suffers.

From the above observations, the best variants overall to use for each of the four algorithm components are: decomposition by stage; stage order solution; EDD sub-problem solution; and no backtracking. This is not the end of the matter, however. Some of the factors in the experiments interact with each other, and thus there are certain situations for which the best-performing algorithm is different. These are summarised below, with references to the relevant sections and figures in the results:

- With few stages (2-3) in the problem, execution set decomposition should be used in preference to decomposition by stage. See §9.6, figure 9.12 and §9.7, figure 9.19.
- Stage-order criticality measure should be paired with decomposition by execution set, and the NLBW criticality measure should be paired with stage-based decomposition. Neither of these pairings is consistent with the main effects in the list above; instead, the former pairing should be used in preference, as it has the best performance (i.e. the criticality measure has the dominant effect on the solution quality). See §9.4, figure 9.5 and 9.6 and §9.6, figure 9.14.
- When using the stage-order criticality measure, multi-pass backtracking is the best option. When using other criticality measures (particularly reverse stage order), no backtracking is generally better. See §9.7, figure 9.27.

9.8.2 Advice on algorithm variants

From the above results, it is clear that in general the following algorithm variants should be used:

- Decomposition: by execution set.
- Criticality measure: stage order.
- Sub-problem solution procedure: earliest due date.
- Backtracking: multi-pass.

The algorithm specified by these particular variants will solve problems better than any other combination of components. However, it is also the most computationally intensive method which can be used (using execution set decomposition and multi-pass backtracking). In our experience, problems with 200 jobs and 10 stages can take up to 5 minutes to run. If a faster solution is required, then we recommend the following combination of variants:

- Decomposition: by stage.
- Criticality measure: stage order.
- Sub-problem solution procedure: earliest due date.
- Backtracking: single-pass.

This will reduce the running time to solve the aforementioned 200 job, 10 stage problem to a few seconds.

Chapter 10

Conclusion

We started this thesis by presenting an industrial system – a flour mill and its associated specialist treatment plant for making biscuit, batter and cake flours. Our first task was to make a detailed analysis of the plant concentrating on those aspects of the plant which were relevant to its scheduling. From that detailed description, which was presented in chapter 2, we proceeded to develop a model by identifying a simple, sequential-stage layout for the machines in the plant, and then making progressive simplifications to the model. The final core scheduling model which we developed in chapter 3 resembled a three-stage hybrid flow shop with a number of interesting non-standard features:

- continuous flow of material between the machines,
- restrictions on which machines may process some jobs at certain stages,
- increasing diversity of products through the system, and
- a highly complex set of inter-process storage bins.

In chapter 4 we started to develop methods for modelling the process. Specifically, we took the method of disjunctive graph modelling, which is frequently used in job shop modelling in the mathematical scheduling literature, and we extended the technique to show how it can be used to model our hybrid flow shop. We also developed a way to incorporate the inter-process storage and continuous flow aspects of our problem into the disjunctive graph. This extension allows an algorithm for the discrete-flow hybrid flow shop which uses disjunctive graphs to be used for continuous-flow problems as well, simply by changing the construction of the digraph used to evaluate potential solutions.

We then concentrated on another aspect of our production system – restrictions on which machines can process certain jobs. These *processing set restrictions* form an interesting extension to the scheduling literature. The simplest scheduling model which can exhibit processing set restrictions is the parallel identical machine model. In chapter 5, we look at the behaviour of a wide range of parallel identical machine scheduling problems where the jobs all have unit length and processing sets. We developed efficient exact algorithms for these problems, both with general processing sets, and for the case where the processing sets have a nested property, for which more efficient algorithms can be constructed. We developed algorithms for all of the common regular objective functions used in the scheduling literature.

In the following chapter, we proposed several heuristic algorithms for parallel identical machine problems with processing set restrictions where the jobs lengths are general. We concentrated on algorithms for minimising the maximum lateness, presenting a simple earliest-due-date algorithm (EDD), and two methods based on Jackson's highly effective heuristic for the problem without processing set restrictions, namely the Nested Jackson's method (NJ), and Jackson's for Processing Sets (JPS). These algorithms, although not optimal, are polynomial time heuristics, and can run extremely quickly even for very large problems.

Returning to the hybrid flow shop model in chapter 7, we examined a generalisation of the hybrid flow shop with processing set restrictions. We first developed a general framework for a hierarchical decomposition solution method. The framework breaks the whole problem into a number of smaller sub-problems, and then attempts to solve the sub-problems individually, tying the separate solutions together as they are produced. We used that general framework to construct a solution method for our extended hybrid flow shop problem, utilising the disjunctive graph representations from chapter 4 and the heuristics for parallel machines from chapter 6. We also developed components for the other three parts of our decomposition framework – problem decomposition schemes, criticality measures to set an order for solving the sub-problems, and backtracking schemes to guide the essential step of re-solving the sub-problems. We developed several alternatives for each of the main components of the decomposition framework. Other algorithms for these replacable components could be placed within the framework, as more efficient methods are developed. Although the hierarchical decomposition method which we had developed did not handle either continuous flow or the complex inter-process storage which we had highlighted at the beginning of the thesis, both features can be incorporated into the algorithm without difficulty, using slightly modified parts in the overall framework.

Having proposed a framework solution method and a set of different components to fit in that framework, we then tested it on a wide variety of problems. Chapter 8 listed which algorithm variants we would test, and gave details of how we constructed the problem instances to use in the tests. In chapter 9, we performed three separate sets of experiments, coming to a total of just over 97,000 individual experimental runs. The results of these experiments were processed using analysis of variance. We analysed and interpreted the results in some detail, and drew conclusions as to which algorithm variants were most appropriate to use across our wide range of problems. We suggest two different algorithm variants: one for obtaining the best results (lowest objective function) from the algorithm, and one for obtaining fast and effective results for solving particularly large problems – we can obtain results in under 10 seconds processor time on a 500MHz K6/2 for problems with 200 jobs and 10 stages. We also performed a small test using a week's worth of live data (approximately 200 jobs) from the specialist treatment plant at Foster Mills. After showing the resulting schedule to the management and staff at the plant, it was concluded that our algorithm could be used in practice, being considerably faster and at least as effective than the manual process in place at the time.

To summarise, we have started with a specialist treatment plant for making flour. We identified the scheduling problems in the plant, and generalised them to a mathematical scheduling model, which exhibited a number of interesting features. We then developed efficient algorithms for solving one stage of this model under a restricted set of assumptions. We also developed a general framework for a hierarchical decomposition method, and used it to make a solution method with many variants for an extended hybrid flow shop. Finally, we tested the solution methods which we produced on a wide range of problems, and identified which of the algorithm variants performed best.

10.1 Further Research

There are several areas highlighted by this work which would probably benefit from further research. Probably the greatest benefit to the hierarchical decomposition algorithm would come from the use of local search to refine the solution. Improved sub-problem solution procedures might also be beneficial – for example, developing the branch-and-bound methods of Carlier[10] to cope with processing set restrictions would yield optimal solutions for the sub-problems during the operation of the algorithm, and possibly improve the solution quality (although at the cost of higher CPU usage and hence time to find a solution).

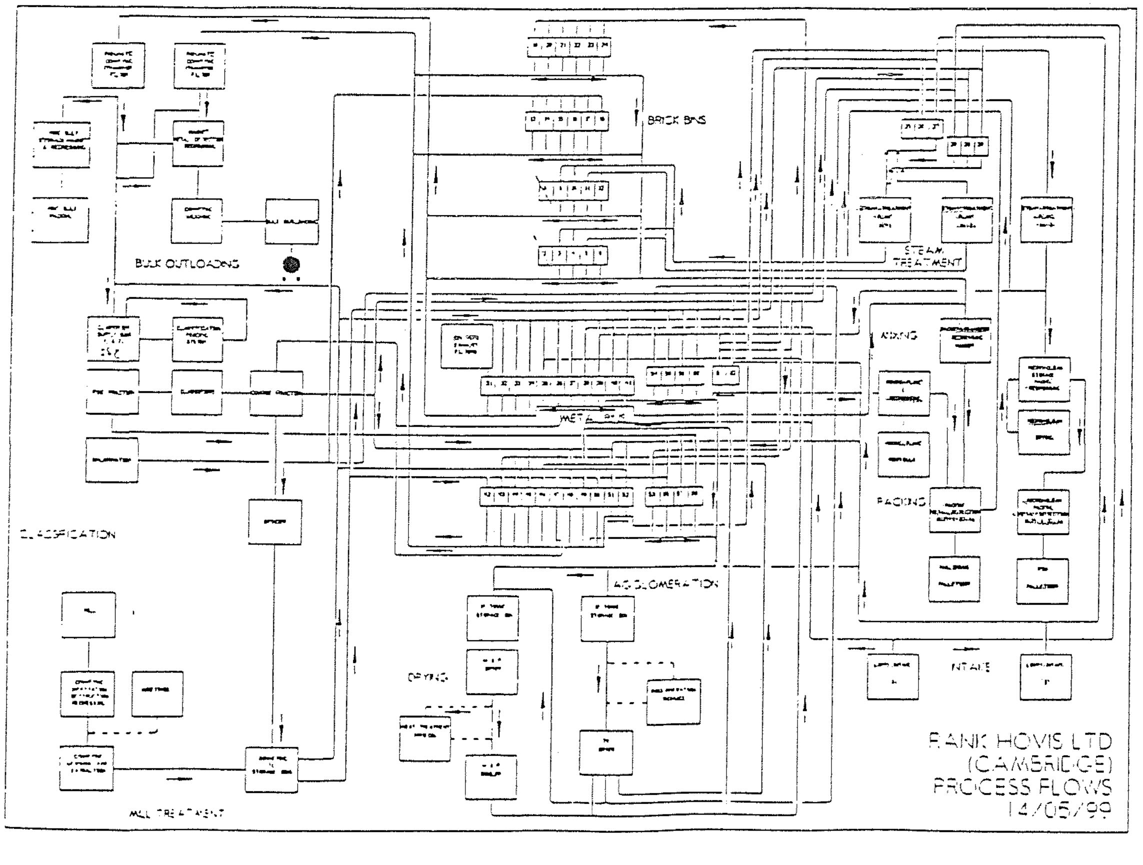
There are two major problems left open by this thesis which we did not have time to investigate in any depth. Firstly there is the problem of inter-process storage allocation and scheduling with the complex storage model identified in chapters 2 and 3. To develop a method for allocating jobs to bins to achieve the required routing through machines, and to sequence the jobs on the bins is a combinatorial problem at least as large and as difficult as the original hybrid flow shop problem we tackled in this thesis.

Secondly, there is the unresolved problem involving the batching of operations which are making similar products at a given stage. The out-tree nature of the product hierarchy makes this a particularly interesting problem, as jobs which may be run together at one stage (and, in our model, stored in the same bin after manufacture) may not necessarily be amenable to similar treatment at a later stage.

Appendix A

Original process diagram

The details on this diagram are deliberately unreadable for reasons of commercial confidentiality. We include it here merely to give an impression of the complexity of the full process model at Foster Mills.



Appendix B

Dijkstra's Algorithm

Dijkstra's algorithm[20] is an $O(n+m)$ algorithm for finding the longest (or shortest) path from a given node in an acyclic directed graph (or *digraph*) to every other reachable node in the graph. The algorithm relies upon the fact that every acyclic digraph has at least one topological sort of the nodes. A *topological sort* of a digraph is an ordering of the nodes such that every arc joins an earlier node in the topological sort order to a later one, and never a later node to an earlier.

For example, consider the digraph shown in figure B.1. The graph is acyclic, so it possesses at least one topological sort. In this case, the sequence a, d, c, e, b, f, g, h will suffice. A topological sort can be found by the following procedure:

Algorithm TOPOLOGICAL SORT

```
1. Initialise
      Count the incoming arcs at each node (for each arc, increment a counter
      at its destination node)
      Place all nodes with no incoming arcs in the set of available nodes

2. Process nodes
      While there are nodes available,
        Pick an available node ($N$)
        For each node, $I$, connected to $N$
          Reduce the incoming arcs count of node $I$ by one
        If node $I$ has no remaining incoming arcs, add it to the set of available
        nodes
      End For
      Place node $N$ next in the topological sort order, and remove it from the
      set of available nodes
    End While
```

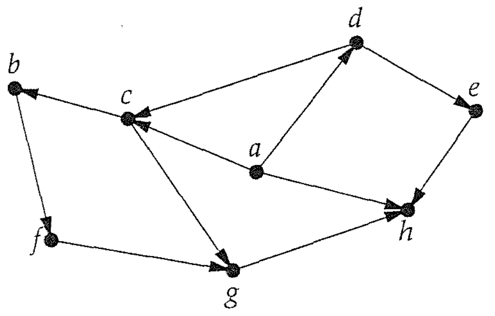


Figure B.1: Example directed graph

Dijkstra's algorithm works by performing a topological sort of its input graph, and then processing each node of the graph in topological sort order, thus ensuring that it does not have to deal with backward arcs. In most applications of Dijkstra's algorithm (such as critical-path analysis), it is usual to have a single "start node" with no predecessors – and to know from the construction of the graph which node is that start node. The existence of such a node simplifies the topological sort process, since then step 1 of the algorithm given above becomes "Initialise the list of available jobs to contain the start node".

As the algorithm processes each node N of the graph, it updates every successor, I, of that node to reflect the longest path currently known from the start node to node I. We shall denote the start node by A. The longest path from node A to node I we shall denote by D_I, and the length of the arc from node I to node J we shall denote by P_{IJ}.

At each node N, the algorithm performs the following process:

1. For each successor, I, of node N:

2. Set the longest known path from the start node to node I to be the maximum

of the current value and the path through node N:

$$D_I = \max(D_I, D_N + P_{NI})$$

The above description separates the topological sort process and the longest-path calculation. In practice, it is possible to combine the two parts into a single pass through the nodes. As an example, consider the graph in figure B.2. The algorithm proceeds as follows:

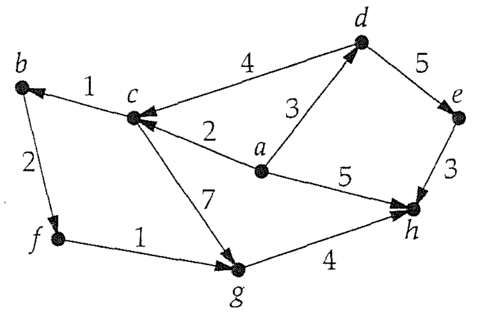


Figure B.2: Directed graph with distances

Stage 0: Initialise

The longest path to each node in the graph is initialised to $-\infty$, and the reference counts are initialised to zero.

Stage 1: Reference counts

The number of incoming arcs for each node is calculated:

Node a: Add one to reference counts of nodes c, d and h
Node b: Add one to reference counts of node f
Node c: Add one to reference counts of nodes b and g
Node d: Add one to reference counts of nodes c and e
Node e: Add one to reference counts of node h
Node f: Add one to reference counts of node g
Node g: Add one to reference counts of node h
Node h: Nothing to do
(See table B.1).

Node	a	b	c	d	e	f	g	h
References	0	1	2	1	1	1	2	3
Distance	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Table B.1: Example: Reference counts calculated

Stage 2: Topological sort and longest path calculation

Now start with node a in the list of available nodes. Process each of its successor nodes:

Node c: decrease reference count; set distance to $\max(-\infty, 0+2)$

Node d: decrease reference count; set distance to $\max(-\infty, 0+3)$

Node d has reference count of zero: add to list of available nodes

Node h: decrease reference count; set distance to $\max(-\infty, 0+5)$

(See table B.2)

Node	a	b	c	d	e	f	g	h
References	0	1	1	0	1	1	2	2
Distance	0	$-\infty$	2	3	$-\infty$	$-\infty$	$-\infty$	5

Nodes available: d

Table B.2: Example: After processing node a

Next, process node d, as the only available node:

Node c: decrease reference count; set distance to $\max(2, 3+4)$

Node c has reference count of zero: add to list of available nodes

Node e: decrease reference count; set distance to $\max(-\infty, 3+5)$

Node e has reference count of zero: add to list of available nodes

(See table B.3)

We can now process either node c or node e next – the order is irrelevant. Continuing this process, we obtain the contents of table B.4 as the final state of the algorithm. From this table, we can read, in the distance row, the longest path from node a to any other node in the graph.

Node	a	b	c	d	e	f	g	h
References	0	1	0	0	0	1	2	2
Distance	0	$-\infty$	7	3	8	$-\infty$	$-\infty$	5

Nodes available: c, e

Table B.3: Example: After processing node d

Node	a	b	c	d	e	f	g	h
References	0	0	0	0	0	0	0	0
Distance	0	1	7	3	8	3	7	11

Nodes available: none

Table B.4: Example: After processing node h

B.1 Efficient implementation

The algorithm described above may be implemented to run in $O(n + m)$ time, where n is the number of nodes, and m the number of arcs. In this section, we give the implementation details to achieve this efficiency.

The graph is stored as an *adjacency array*. Each element of the array corresponds to a node of the graph. Each element also contains the head of a linked list of the nodes to which it is connected – corresponding to the (directed) arcs of the graph, a reference count, and the value of the longest distance. In C++,

```
class Arc
{
   Node* destination;
   int length;
};

class Node
{
    list <Arc> arcs;
    int  references ;
    int  distance;
};

vector<Node> graph;
```

Thus, for the example graph shown in figure B.2, the graph would be stored in a data structure similar to the one in figure B.3.

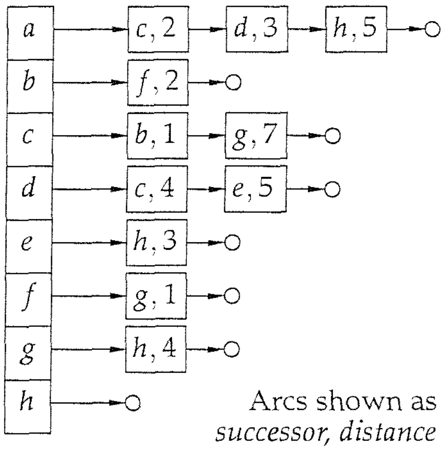


Figure B.3: Data structure for graph

The list of available nodes should be kept in some container with $O(1)$ insertion and removal from a given point. Since it does not matter what order the available nodes are processed in, this list could be implemented in any number of ways, such as a FIFO queue based on a linked list (inserting at the end, removing from the beginning), or as a LIFO queue based on an array (inserting and removing at the end).

When the algorithm runs, it visits each node once to initialise them ($O(n)$ operations). When setting the reference counts, the algorithm visits each node ($O(n)$ operations), and for each outgoing arc increments the reference count of the target node. Thus, every arc is also processed once, requiring $O(m)$ operations. Finally, when performing the topological sort, again every node is processed once and every arc is processed once. Thus, the overall running time of this algorithm is

$O(n + m)$. If there is a fixed upper bound (say, k) on the number of outgoing arcs that each node possesses, then we observe that $m \leq kn$, and hence the algorithm runs in $O(n)$ time.

It might be considered easier to implement the adjacency array as a two-dimensional array, rather than as an array of linked lists. However, in this case, it should be noted that the array allocated for each node should be precisely the size required and no larger (thus requiring the storage of a successor count at each node as well), for if a larger array is used – for example, an m-element array for each node, the algorithm may become $O(nm)$ due to the time taken to allocate the larger array.

Appendix C

Notation

Firstly, we give the basic problem parameters:

n	Number of jobs
j	Job index, $1 \leq j \leq n$
s	Number of processing stages
k	Stage index, $1 \leq k \leq s$
m_k	Number of machines (at stage k)
i	Machine index, $1 \leq i \leq m_k$
(k, i)	Machine i at stage k
$\rho_{k\mathrm{F}}$	Processing rate of machines at stage k making product **F**
$s_{k\mathrm{F}}$	Set-up time of machines at stage k making product **F**
O_{jk}	Operation at stage k of job j
d_j	Due date of job j
r_j	Release date of job j
F_j	Product type of job j
p_{jk}	Length of operation O_{jk}
β_{jk}	Storage capacity required after processing for operation O_{jk}
Q_k	Total number of processing sets available at stage k
	Note: for nested processing sets, $Q_k \leq 2m_k - 1$
	for unnested processing sets, $Q_k \leq 2^{m_k} - 1$
$M^{(qk)}$	Processing sets available at stage k
M_{jk}	Processing set for operation O_{jk}

	Note: $M_{jk} \in \{M^{(1k)}, \dots, M^{(Q_k k)}\}$
$\mathbf{F}$	Index for product types, $\mathbf{F} = (F_0, F_1, \dots, F_s)$
$\mathcal{B}$	The set of all bins
b	Bin index, $1 \leq b \leq \lvert\mathcal{B}\rvert$
$B_{(k,i),(k',i')}$	The set of bins between machine (k,i) and (k',i')
c_b	Capacity of bin b

Derived measures:

W_k	Mean stage workload for stage k
$\overline{w}_{qk}$	Mean processing set workload for processing set $M^{(qk)}$
w_{qk}	Total mean processing set workload (TMPSW) for processing set $M^{(qk)}$
L_{qk}	Nested load-balanced workload (NLBW) for processing set $M^{(qk)}$
d_{jk}	"due date" of operation O_{jk}
r_{jk}	"release date" of operation O_{jk}
f_{jk}	Available float of operation O_{jk}

$\chi(\mathbf{F})$	Last stage at which product type $\mathbf{F}$ was processed
$P_k(\mathbf{F})$	Ancestor of product $\mathbf{F}$ at stage k
$P(\mathbf{F})$	Immediate ancestor (parent) of product $\mathbf{F}$

Other notation:

$\mathbb{E}(x)$	The expected value of x
$\mathbf{R}(a,b)$	A (discrete) uniform distribution, generating integer values between a and b inclusive

Appendix D

Glossary of abbreviations

ANOVA Analysis of Variance[22].

ATC Apparent Tardiness Cost heuristic[77], for $1||\sum w_j T_j$.

CDS Cambell, Dudek and Smith heuristic, for $Fs(P)||C_{\max}$ and $F2(P)||\sum C_j$.

CF Coarse Fraction: flour with large particle size from classifier.

CMD Cumulative Minimum Deviation algorithm[58], for $F2(P1, Pm)||C_{\max}$.

EDD Earliest Due Date algorithm, for several problems.

EH [71] Heuristic for $Js(P)||C_{\max}$.

FF Fine Fraction: flour with small particle size from classifier.

FIBC Flexible Intermediate Bulk Container: a bag holding approximately 750 kg-1 tonne of flour.

FIFO First In First Out heuristic, see e.g. [42].

GCMD General Cumulative Minimum Deviation[57], for $F2(Pm_1, Pm_2)||C_{\max}$.

HO Ho heuristic[37], for $Fs(P)||C_{\max}$ and $F2(P)||\sum C_j$.

JPS Jackson's algorithm for Processing Sets, for $Pm|M_j$ nested$|L_{\max}$, see §6.1.3.

LFJ Least Flexible Job algorithm[65], for $Pm|M_j$ nested, $p_j = 1|C_{\max}$.

LIFO Last In First Out heuristic, see e.g. [42].

LPT Longest Processing Time first algorithm[30], for $Pm||C_{\max}$ and others.

LWR Least Work Remaining heuristic[42], for $F2(P)|\text{bins}|\cdot$.

MCP Micro-Clean Plant: a section of the mill kept under positive pressure and low microbiological activity.

ME More Early algorithm[73], for $F2(P)||C_{\max}$.

MSW Mean Stage Workload criticality measure, see §7.5.1.

MWR Most Work Remaining heuristic[42], for $F2(P)|\text{bins}|\cdot$.

NEH Nawaz, Enscore and Ham heuristic[59], for $Fs(P)||C_{\max}$ and $F2(P)||\sum C_j$.

NJ Nested Jackson's algorithm, for $Pm|M_j$ nested$|L_{\max}$, see §6.1.4.

NLBW Nested Load-Balanced Workload criticality measure, see §7.5.3.

SPSF Smallest Processing Set First algorithm, for $Pm|M_j$ nested, $p_j = 1|\sum C_j$, see §5.4.2.

SPS-List Smallest Processing Set List algorithm, for $Pm|M_j$ nested, $p_j = 1|\sum U_j$, see §5.7.1.

SPT Shortest Processing Time first algorithm[13], for $Pm||\sum C_j$ and others.

ST Steam Treated: type of flour usually used for making batters.

SW-List Smallest Weight List algorithm, for $Pm|M_j$ nested, $p_j = 1|\sum w_j U_j$, see §5.8.1.

TMPSW Total Mean Processing Set Workload criticality measure, see §7.5.2.

TV Thermo-Venturi: one type of flour dryer used in the mill.

WLA WorkLoad Approximation heuristic[78], for $Fs(P)|\text{missing}|\cdot$

WP Werner-Pflederer: one type of flour dryer used in the mill,

YKP Yang, Kreipl and Pinedo algorithm[79], for $Fs(P)||\sum w_j T_j$.

Bibliography

[1] The P vs NP problem. http://www.claymath.org/prizeproblems/pvsnp.htm. Visited 25 June 2002.

[2] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.

[3] E. Balas. Machine sequencing via disjunctive graphs: An implicit enumberation algorithm. *Operations Research*, 17:941–957, 1969.

[4] E. Balas and A. Vazacopolous. Guided local search with shifting bottleneck for job scheduling. *Management Science*, 44(2):262–275, February 1998.

[5] S. A. Brah and J. L. Hunsucker. Branch and bound algorithm for the flow shop with multiple processors. *European Journal of Operational Research*, 51:88–99, 1991.

[6] S. A. Brah and L. L. Loo. Heuristics for scheduling in a flow shop with multiple processors. *European Journal of Operational Research*, 113:113–122, 1999.

[7] P. J. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993.

[8] P. Brucker, B. Jurisch, and A. Krämer. Complexity of scheduling problems with multi-purpose machines. *Annals of Operations Research*, 70:57–73, 1997.

[9] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[10] J. Carlier. Scheduling jobs with release dates and tails on identical machines to minimize the makespan. *European Journal of Operational Research*, 29:298–306, 1987.

[11] J. L. Cheng, Y. Karuno, and H. Kise. A shifting bottleneck approach for a parallel-machine flowshop scheduling problem. *Journal of the Operations Research Society of Japan*, 44(2):140–156, 2001.

[12] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. An application of bin-packing to multi-processor scheduling. *SIAM Journal of Computing*, 7(1):1–17, 1978.

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

[14] S. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of computing*, pages 151–158, 1971.

[15] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimisation*. Wiley-Interscience, New York, 1998.

[16] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[17] S. Dauzere-Peres and J.-B. Lasserre. A modified shifting bottleneck procedure for job-shop scheduling. *International Journal of Production Research*, 31(4):923–932, 1993.

[18] S. Dauzère-Pérès and J. Paulli. An integrated approach for modelling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70:281–306, 1997.

[19] M. Dell'Amico and S. Martello. Optimal scheduling of tasks on identical parallel machines. *ORSA Journal on Computing*, 7(2):181–200, 1995.

[20] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[21] R. A. Dudek, S. S. Panwalkar, and M. L. Smith. The lessons of flowshop scheduling research. *Operations Research*, 40(1):7–13, 1992.

[22] R. A. Fisher. *The Design of Experiments*. Oliver and Boyd, 1935.

[23] L. R. Ford, Jr and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[24] P. M. França, M. Gendreau, G. Laporte, and F. M. Müller. A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Computers and Operations Research*, 21(2):205–210, 1994.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Co., San Francisco, 1979.

[26] R. Gerrard. Private communication, June 2001.

[27] A. Gharbi and M. Haouari. Minimizing makespan on parallel machines subject to release dates and delivery times. *Journal of Scheduling*, 5:329–355, 2002.

[28] P. C. Gilmore and R. E. Gomory. Sequencing a one state-variable machine: a solvable case of the travelling salesman problem. *Operations Research*, 12:655–679, 1964.

[29] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the Association of Computing Machinery*, 35:921–940, 1988.

[30] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[31] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Operational Research*, 5:287–326, 1979.

[32] A. G. P. Guinet and M. M. Solomon. Scheduling hybrid flowshops to minimize maximum tardiness or maximum completion time. *International Journal of Production Research*, 34(6):1643–1654, 1996.

[33] J. N. D. Gupta, A. M. A. Hariri, and C. N. Potts. Scheduling a two-stage hybrid flow shop with parallel machines at the first stage. *Annals of Operations Research*, 69:171–191, 1997.

[34] L. A. Hall and D. B. Shmoys. Jackson's rule for single-machine scheduling: making a good heuristic better. *Mathematics of Operations Research*, 17(1):22–35, Feb 1992.

[35] N. G. Hall and C. Sriskandarajah. A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research*, 44(3):10–525, 1996.

[36] A. Hertz, Y. Mottet, and Y. Rochat. On a scheduling problem in a robotized analytical system. *Discrete Applied Mathematics*, 65:285–318, 1996.

[37] J. C. Ho. Flowshop sequencing with mean flowtime objective. *European Journal of Operational Research*, 81:571–578, 1995.

[38] D. S. Hochbaum and D. B. Schmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the Association of Computing Machinery*, 34:144–162, 1987.

[39] H. H. Holtsclaw and R. Uzsoy. Machine criticality measures and subproblem solution procedures in shifting bottleneck methods: A computational study. *Journal Of The Operational Research Society*, 47(5):666–677, 1996.

[40] W. A. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.

[41] T. S. Hundal and J. Ragopal. An extension of palmer's heuristic for the flow-shop scheduling problem. *International Journal of Production Research*, 26(6):1119–1124, 1988.

[42] J. L. Hunsucker and J. R. Shah. Performance of priority rules in a due date flow shop. *OMEGA International Journal of Management Science*, 20(1):73–89, 1992.

[43] J. Hurink, B. Jurisch, and M. Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spektrum*, 15:205–215, 1994.

[44] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical Report Research Report 43, Management Science Research Project, UCLA, 1955.

[45] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[46] S. N. Kadipasaoglu, W. Xiang, and B. M. Khumawala. A comparison of sequencing rules in static and dynamic hybrid flow systems. *International Journal of Production Research*, 35(5):1359–1384, 1997.

[47] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[48] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[49] E. J. Lawler. On scheduling problems with deferral costs. *Management Science*, 11:280–288, 1964.

[50] E. L. Lawler. A 'pseudopolynomial' algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.

[51] J. K. Lenstra. *Sequencing by Enumerative Methods*. Mathematisch Centrum, Amsterdam, 1977.

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

[53] S. Martello, F. Soumis, and P. Toth. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75:169–188, 1997.

[54] M. Mastrolilli and L. M. Gambardella. Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, 3:3–20, 2000.

[55] C. L. Monma. Linear-time algorithms for scheduling on parallel processors. *Operations Research*, 30(1):116–124, January-February 1982.

[56] J. M. Moore. An *n* job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968.

[57] S. L. Narasimhan and P. M. Mangiameli. A comparison of sequencing rules for a two-stage hybrid flow shop. *Decision Sciences*, 18:251–265, 1987.

[58] S. L. Narasimhan and S. S. Panwalkar. Scheduling in a two-stage manufacturing process. *International Journal of Production Research*, 22(4):555–564, 1984.

[59] M. Nawaz, E. Enscore, and I. Ham. A heuristic algorithm for the *m* machine, *n* job flow shop sequence problem. *OMEGA*, 11(1):91–95, 1983.

[60] E. G. Nebenman. Local search algorithms for the multiprocessor flow shop scheduling problem. *European Journal of Operational Research*, 128(1):147–158, 2001.

[61] E. Nowicki and C. Smutnicki. The flow shop with parallel machines: A tabu search approach. *European Journal of Operational Research*, 106:226–253, 1998.

[62] J. B. Orlin. A faster strongly polynominal minimum cost flow algorithm. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 377–387, Chicago, Illinois, 2-4 1988.

[63] C. H. Papadimitriou and P. C. Kannelakis. Flowshop scheduling with limited temporary storage. *Journal of the Association for Computing Machinery*, 27(3):533–549, 1980.

[64] F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for the job shop problem. *European Journal of Operational Research*, 120:297–310, 2000.

[65] M. Pinedo. *Scheduling: theory, algorithms and systems*. Prentice-Hall, 1995.

[66] M. Pinedo and M. Singer. A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Naval Research Logistics*, 46:1–17, 1999.

[67] F. Riane, A. Artiba, and S. E. Elmaghraby. Sequencing a hybrid two-stage flowshop with dedicated machines. *International Journal of Production Research*, 40(17):4353–4380, 2002.

[68] M. H. Rothkopf. Scheduling independent tasks on parallel processors. *Management Science*, 12(6):437–447, January 1966.

[69] B. Roy and B. Sussmann. Les problèmes d'ordonnancement avec contraintes disjonctives. Technical report, SEMA, Paris, 1964.

[70] D. L. Santos, J. L. Hunsucker, and D. E. Deal. Global lower bounds for flow shops with multiple processors. *European Journal of Operational Research*, 80:112–120, 1995.

[71] D. L. Santos, J. L. Hunsucker, and D. E. Deal. On makespan improvement in flow shops with multiple processors. *Production planning and control*, 12(3):283–295, 2001.

[72] R. Sedgewick. *Algorithms in C*, volume 1, chapter 13. Addison-Wesley, Reading, Massachusetts, third edition, 1998.

[73] V. Y. Shen and Y. E. Chen. A scheduling strategy for the flowshop problem in a system with two classes of processors. In *Conference on Information and Systems Science*, pages 645–649, 1972.

[74] W. E. Smith. Various optimizers for single-stage problems. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[75] C. Sriskandarajah and S. P. Sethi. Scheduling algorithms for flexible flowshops: Worst and average case performance. *European Journal of Operational Research*, 43:143–160, 1989.

[76] D. W. Townsend. Sequencing n jobs on m machines to minimize tardiness: a branch and bound solution. *Management Science*, 23:1016–1019, 1977.

[77] A. Vepslainen and T. E. Morton. Priority rules and lead time estimation for job shop scheduling with weighted tardiness costs. *Management Science*, 33:1036–1047, 1987.

[78] R. J. Wittrock. An adaptable scheduling algorithm for flexible flow lines. *Operations Research*, 36(3):445–453, 1988.

[79] Y. Yang, S. Kreipl, and M. Pinedo. Heuristics for minimising total weighted tardiness in flexible flow shops. *Journal of Scheduling*, 3(2):71–88, March-April 2000.