University of Southampton

Lot Streaming and Batch Scheduling: splitting and grouping jobs to improve production efficiency

Edgar Possani

submitted for the degree of Doctor of Philosophy in Operational Research

Faculty of Mathematical Studies

December 2001

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF MATHEMATICAL STUDIES
OPERATIONAL RESEARCH

Doctor of Philosophy

LOT STREAMING AND BATCH SCHEDULING: SPLITTING AND GROUPING JOBS TO IMPROVE PRODUCTION EFFICIENCY

Edgar Possani

This thesis deals with issues arising in manufacturing, in particular related to production efficiency. Lot streaming refers to the process of splitting jobs to move production through several stages as quickly as possible, whereas batch scheduling refers to the process of grouping jobs to improve the use of resources and customer satisfaction.

We use a network representation and critical path approach to analyse the lot streaming problem of finding optimal sublot sizes and a job sequence in a two-machine flow shop with transportation and setup times. We introduce a model where the number of sublots for each job is not predetermined, presenting an algorithm to assign a new sublot efficiently, and discuss a heuristic to assign a fixed number of sublots between jobs. A model with several identical jobs in an multiple machine flow shop is analysed through a dominant machine approach to find optimal sublot sizes for jobs.

For batch scheduling, we tackle the NP-hard problem of scheduling jobs on a batching machine with restricted batch size to minimise the maximum lateness. We design a branch and bound algorithm, and develop local search heuristics for the problem. Different neighbourhoods are compared, one of which is an exponential sized neighbourhood that can be searched in polynomial time. We develop dynamic programming algorithms to obtain lower bounds and explore neighbourhoods efficiently. The performance of the branch and bound algorithm and the local search heuristics is assessed and supported by extensive computational tests.

Contents

Acknowledgements

I would first like to thank my supervisor, Prof. Chris N. Potts, for all his help and time; without his support and guidance this work would have never been done or written. His fine intuition, broad knowledge, and work capacity have always been an inspiration to me. I would also like to thank Dr. Celia A. Glass, who was my supervisor for the first half of my PhD. Had it not been for her I would probably not have started my studies at the University of Southampton. It has been an enriching experience to do research with both of them.

I was funded by Consejo Nacional de Ciencia y Tecnología (CONACYT 115925) throughout my studies, and was a grateful recipient of the Postgraduate Overseas Scholarship from the Faculty of Mathematical Studies. I am in debt to both institutions for allowing me the opportunity to study in this country.

If anything, I will take with me very fond memories of all the people I have met throughout these years. I would like to thank: Richard, Jon and Marta, who shared an office and a common research area with me, they kept me company in what is sometimes a very lonely experience; Pablo, Richard, Tom, Catherine and Eva for all the fun time we spent (and spend) together outside the Faculty; special thanks to Carla, Spiros, Paul, and Marta for sharing the same roof with me; and all the others who I have not mentioned, but made my stay in Britain a delightful experience.

I would also like to thank Yvonne Oliver and Pat Taylor for all their help and kindness, they always made a visit to the fifth floor a fun trip; and Peter Hubbard for help with computing facilities.

Thanks to my family for being close and always present: all the e-mails and phone calls kept me going; also to my aunt Lorena for making my first Christmas away from home a bit more bearable.

"La tercera es la vencida", once again, thanks to Marta, I would not have finished this thesis without your love, never-ending patience, help and encouragement. Te quiero cosita.

Chapter 1

Introduction

1.1 Background

Activities that transform resources into goods and services, that take place in all sorts of organizations, are of interest to production and operations management. Efficient decision making at this level is important in increasing productivity as pointed out by Heizer & Render (1996). Scheduling plays an important role in production planning for manufacturing, and in increasing its efficiency, more so with the advent of computers and automated systems. However, its uses are wider, and many applications can be found outside manufacturing and production in the service industries area. In the competitive environment of today, efficient scheduling has become a necessity for survival in the marketplace.

This thesis deals with issues arising in manufacturing, in particular related to production efficiency. The problems we consider are scheduling problems. Scheduling deals with the allocation of scare resources (machines) over time to tasks (jobs). We concentrate in two different areas: *lot streaming* and *batching*. Lot streaming refers to the process of splitting jobs, where as batching refers to the process of grouping them. Both are common processes in manufacturing, and are of considerable interest, as can be seen by an extensive literature.

Lot streaming is motivated by a desire to move the production of a job through several work stations or stages as quickly as possible. It can be considered an extension on the classical models in scheduling theory, where jobs are processed fully on a machine before continuing to another stage in the system, for in this case partially completed parts of a job are passed to downstream machines. Our aim was to show the advantages of network representations for the models, like the ones presented in Glass & Potts (1998), to obtain new results for the flow shop environment. Research on lot streaming was done under the supervision of Dr. Celia A. Glass.

Most batching problems arise from the effort of grouping similar jobs in order to reduce common setup times. However, in this thesis we are interested in modeling a batching machine, one which can process more than one job at a time. Applications can be found in the 'burn-in' operations in the manufacture of circuits boards, and for chemical processes that are performed in tanks or kilns. Our aim was to tackle the NP-hard problem of scheduling jobs on a batching machine with a restricted batch size to minimize the maximum lateness objective function. Research on batching was done under the supervision of Prof. Chris N. Potts.

1.2 Contributions

The contents of this thesis are a result of work wholly carried out by the author while registered in postgraduate candidature at the University of Southampton. We now describe the main contributions of this thesis.

Lot streaming

A job is split into sublots, the question that most lot streaming models try to answer is that of finding optimal sizes for the sublots. Instead of using a linear programming approach we have used a network representation and critical path approach to analyse models in lot streaming, with good results. We have presented an alternative approach to analyse the lot streaming problem of finding optimal sublot sizes and job sequence in a two-machine flow

shop with transportation and setup times. Our aim was to clarify and give more insight into the results given by Vickson (1995). We have also analysed extension of the models for the flow shop environment. In particular we have analysed the case where the number of sublots each job has is not predetermined. We are considering a situation where the number of sublots is a technological constraint for the whole schedule, rather than a particular one for each job, as is common in the literature. We present an algorithm to assign a new sublot efficiently, and discuss a heuristic to assign a fixed number of sublots between jobs. We have also analysed a model with n jobs in an m-machine flow shop with $m > 3$. The jobs have the same processing times and sublots on each machine. We have applied a dominant machine analysis to find optimal sublot sizes for jobs, and proven that the model reduces to a simpler single-job lot streaming problem.

Batching

We have tackled the NP-hard problem of scheduling jobs on a batching machine with restricted batch size to minimise the maximum lateness. Our aim was to develop exact as well as approximation methods for this problem. We have designed a branch and bound algorithm for the problem. The performance of the algorithm has been assessed through extensive computational testing. We give dynamic programming algorithms to find lower bounds on the maximum lateness of a schedule with a restricted maximum batch size. We also developed local search heuristics for the problem. Different neighbourhoods have been designed and compared, one of which is an exponential sized neighbourhood that can be searched in polynomial time. We rely on dynamic programming algorithms to explore the neighbourhoods efficiently. Again, the development work is supported by computational tests. As far as the author knows these are the first algorithms, and heuristics developed for the problem. Potts & Kovalyov (2000) point out that there is little research done on branch and bound and local search for batching machine problems, and our aim is to fill part of this gap.

1.3 Organisation

The thesis is divided into two main parts. Part I, deals with models in lot streaming while Part II is dedicated to the batching machine model. In Chapter 2, we give a brief introduction to the theory of scheduling, and computational complexity. We present the notation we will be using throughout the thesis. Chapter 3 gives an overview of the models studied on the lot streaming literature, and the techniques we are interested in applying. Here we present the alternative approach to schedule jobs in a two-machine flow shop. Chapter 4 looks at the extensions of the known lot streaming models. In Chapter 5, we talk about batching machine scheduling in the context of combinatorial optimisation problems. We explain techniques to find exact solutions (branch and bound, and dynamic programming), as well as approximate solutions (local search heuristics). We also introduce the problem we study in the two following chapters. In Chapter 6, we describe the branch and bound algorithm developed for the batching machine problem, and discuss the computational results obtained. In Chapter 7, we compare several local search heuristics over different neighbourhoods. We present a exponential size neighbourhood that can be searched in polynomial time. Computational tests are used to compare the different methods. Finally Chapter 8 concludes the thesis, outlining potential further work.

Chapter 2

Scheduling

In this chapter we give a brief introduction to the theory of scheduling, and computational complexity. Our aim is to familiarise the reader with some scheduling problems and their models, and explain a general framework to classify the difficulty of solving them. We focus mostly on those concepts that are relevant to subsequent chapters. More elaborate introductions can be found for scheduling in Conway, Maxwell & Miller (1967), Baker (1974) French (1982), and Pinedo (1995).

Scheduling problems go back to the beginning of the industrial era. However, the first samples of scientific analysis of such problems date back to the 1950's. The theory of scheduling is concerned with the efficient allocation of resources to tasks over time. For example, a resource may be a machine in a workshop, surgeons in a hospital, processing units in a computing environment, and so on. The corresponding tasks may be operations in a production process, the surgical procedures to be performed to patients, computer programs to be executed, etc. Each task and resource might have different properties, which need to be taken into account to do the allocation. The value of the allocation is usually expressed as a function of the completion time of the tasks, referred to as an objective function. The problem is then one of finding a minimum value for this objective function. We introduce several models in Section 2.1, discussing popular objective functions

and properties of interest for the tasks and resources. We confine ourselves in this thesis to deterministic scheduling models (problems). That is, we assume the data that define the problem (tasks and resources) are known with certainty in advance.

2.1 Scheduling Machine Models

The standard terminology in use is one set in a manufacturing environment, which reflects the beginnings of the theory. Hence, a task is referred to as a *job*, and a resource is referred to as a *machine*. Each job may consist of several operations. It is common to talk about n jobs to be scheduled on m machines, and usually a subscript j refers to a job, whereas the subscript i refers to a machine. It is sometimes useful to refer to the set of jobs or machines. The notation we use is $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and $\mathcal{M} = \{M_1, \ldots, M_m\}$ for the set of jobs, and machines respectively. Thus, it is the same to refer to job j, or job J_j, likewise to machine i or machine M_i. Several models have been proposed, analysed, and classified in the literature. Classification schemes have been proposed based on different dimensions. There are two major dimensions that specify any model: the *machine environment* and the *job properties*.

Machine environments are determined by the number of machines, how they are organised in the production system, and the way the jobs are processed through them. We can divide the machine environments into two broad classes *single-stage* and *multi-stage* environments. A single-stage environment corresponds to production systems requiring only one operation per job, whereas multi-stage environments correspond to production systems where there are jobs that require operations on different machines.

2.1.1 Single-Stage Environments

Single-stage environments involve either a single machine or m machines operating in parallel. The *single-machine* model is the simplest one, where

jobs require one operation on the (single) machine. It is standard to consider a single machine as being able to processes just one job at at time. However, in this thesis we are interested in modeling a single machine that can process more than one job at a time: we call such a machine a *batching machine* (see Section 5.5). Single-machine models not only come from situations that are in their own right a single machine problems, but also arise as simplifications of more complex models. It is not uncommon that a single machine is a bottleneck in a complex production system, and thus the performance of the entire system depends on it. Single machine models are also important in decomposition approaches, where scheduling problems in complicated environments are broken down into a number of smaller, single machine scheduling problems, as simple rules to solve them are more common due to their simplicity.

A *parallel-machine* system, is a generalisation of the single-machine model. In this case we have several machines that can process a job. It is common to distinguish between three settings: one where the m machines are *identical*, one where machines have different speeds, called *uniform* parallel machines, and one where the machines are *unrelated*. In an identical parallel machine environment each job requires a single operation and it may be processed on any one of the machines. In a uniform parallel machine environment, any machine can process a job, but the machines operate at different speeds, where the speed does not depend on the job, but on the machine. This portrays the fact that some machines in the system might be older, and therefore operate at lower speeds (an obvious example is computers). In the unrelated parallel machine environment the speed of the machines not only differ from machine to machine, but also vary for different jobs on the same machine. For example, when machines represent people, then the processing time may depend on the job as well as on the person (machine). One person may excel in one type of job, whereas another person may specialize in another type of job.

2.1.2 Multi-Stage Environments

Multi-stage environments consist of several machines, where the jobs generally have more than one operation to be performed on the machine system. There are three main types of multi-stage environments: *flow shop*, *open shop*, and *job shop*. In a flow shop environment each job has to be processed on each one of the m machines (stages) in the same order. For convenience, we say the jobs pass through from machine 1 to m. All jobs have the same routing, and after processing on one machine they join a queue for the next. Jobs may be resequenced between machines. If no job changes order while waiting in the queue, then we refer to it as a *permutation flow shop*. In an open shop environment jobs have to be processed once on each of the m machines, but there is no restriction on the order any single job must pass through them. Hence, the routing becomes part of the allocation (decision) process. In a job shop each job has its own route to follow through the machines. The route is prescribed beforehand for each job and it can differ from job to job. It is assumed that each job visits each machine at most once. However, if we allow a job to visit a machine more than once, then the job shop is subject to *recirculation*.

It is usual for a job to be processed completely at a stage before it is sent to the next stage. Nevertheless, there are cases when the operation on a job is stopped to be resumed later, or when partially completed work on a job is passed along to other stages (downstream machines). Specifically, some models allow job *preemptions* where the processing of a job j may be interrupted (preempted), to put a different job in the machine, and resumed at a later time. Preemption is motivated by a desire to implement appropriate priorities among jobs in a situation where two or more jobs compete for limited production resources. In fact we are splitting the operation of the job into smaller parts. Another case where splitting of jobs is allowed is *lot streaming*. However in this case the job processing is not necessarily interrupted by another job, but merely passed along to another machine in the system, while the processing continues on the job. Lot streaming

is motivated by a desire to move the production of the job through several work stations (stages) as quickly as possible. Sometimes jobs have *precedence constraints*, when certain jobs require the completion of some others before work can start on them.

2.1.3 Job properties

The following are popular job properties, and we present the notation we will use for them throughout the thesis.

- Processing time p_{ij} of job j on machine i, is the time it takes for machine i to process job j. The subscript i is omitted if the processing time of the job does not depend on the machine, or if we are working with a single machine. Hence, for identical parallel machines p_j is the processing time on any machine, whereas in a uniform parallel machine environment the processing time of job j may be expressed as p_j/u_i, where u_i is the speed of machine i.

- Due date d_j of job j, is the time by which the job should be completed. In a manufacturing system it represents the committed shipping (or completion) date the job is promised to the customer. The completion of a job after its due date is allowed; however a penalty is incurred (usually expressed in the objective function). If the due date must necessarily be met then we refer to it as the *deadline*.

- Release date r_j of job j, is the time (date) at which the job becomes available for processing in the system.

- Weight w_j of job, which denotes the importance of a job relative to others; it is a priority factor. It might represent the actual cost of keeping the job in the system, such as a holding or inventory cost. Alternatively, it might just be a predefined value (cost) of the job.

- Setup s_{ijk} time between jobs j and k on machine i. This may represent the clean up time after job j before job k starts processing on machine i.

If the setups are not sequence dependent then we say s_{ij} is the setup time on machine i before job j starts. If the setup does not depend on the machine, or we are working with a single machine the subscript i is omitted. Sometimes similar jobs share a setup, we say they belong to the same *family*. If job k is processed immediately after job j on machine i and they belong to the same family, then $s_{ijk} = 0$.

- Transportation time t_{ijh} of job j between machine i and h, to represent the time it takes to transport job j from machine i to machine k. This property is inherent in multi-stage environments. If it is machine independent, or if the transport is always made in the same direction and only between two machines we may omit subscripts i and h. This can be viewed as a setup between machines for the same job, in a similar way to s_{ijk} being one between jobs in the same machine.

Note how some properties are predominantly time related like d_j, and r_j, while some are also dependent on the job sequencing on the machines like s_{ijk} and t_{ijk}. Others are solely dependent on the job like w_j, or are based on the particular environment where they are set such as preemption, lot streaming, or precedence constraints.

2.1.4 Objective Functions

In scheduling terminology, a distinction is often made between a *sequence*, and a schedule. A sequence usually corresponds to a permutation of the jobs, that is, the order in which jobs are to be processed on a given machine. A schedule provides additional information, including the time that the operations occupy the machines and possibly some other features. For example, in an two-machine flow shop with lot streaming a schedule not only specifies the order the jobs go through both machines but the size of the sublots for each job (see Section 3.4).

Once a schedule of the jobs is determined, we can calculate the completion time of each job. We will denote by C_j the completion time of job j, that

is the time the job leaves the system (i.e. its completion time on the last machine on which it requires processing). Sometimes it is useful to be able to refer to the completion time of an operation of job j on machine i, which we denote by C_{ij}. As explained before, the objective function is usually expressed in terms of the completion time of the jobs. Some measures that can be calculated for each job and help define popular objective functions are:

- the lateness of a job $L_j = C_j - d_j$;
- the unit penalty $U_j = 1$ if $C_j > d_j$, otherwise $U_j = 0$;
- the tardiness $T_j = \max\{C_j - d_j,\ 0\}$;
- the earliness $E_j = \max\{d_j - C_j,\ 0\}$;
- and the flow time $F_j = C_j - r_j$.

Based on these measures we are able to calculate and propose the following objective functions.

- Maximum completion time or makespan $C_{\max} = \max_j C_j$. A minimum makespan usually implies a high utilisation of the machine(s). It is also a measure of the output rate of products in a system.
- Total (weighted) completion time $\sum(w_j)C_j$, which gives an indication of the holding costs incurred by the schedule. When $w_j = 1$ for each job j this objective function is equivalent to minimising the average number of jobs in the system, and is a measure of the average throughput time.
- Total (weighted) flow time $\sum(w_j)F_j$ is a similar measure to the previous one, but this one considers the job release date. It allows for jobs entering the system at different times. Note that if $r_j = 0$ for all jobs j then $\sum(w_j)C_j = \sum(w_j)F_j$.

- Maximum lateness, $L_{\max} = \max_j L_j$; this measures the worst violation of the due date. In some sense, minimising $L_{\max}$ is equivalent to minimising the worst performance of the schedule.

- Total (weighted) number of late jobs $\sum U_j$; this is a common measure in practice, and easily recorded. It does not account for how late a job is, but just if it is late or not. However minimising this objective function may result in schedules where jobs are very late, which is often unacceptable in practice.

- Total (weighted) tardiness $\sum(w_j)T_j$; this measures the conformance to due dates, similarly to $\sum(w_j)U_j$, but it is less likely that the wait for any given job will be unacceptably long.

- Maximum tardiness, $T_{\max}$, is similar to $L_{\max}$, but early jobs bring no reward.

In this thesis we are interested in particular in the $C_{\max}$, and $L_{\max}$ objective functions.

2.1.5 Model Notation

The standard representation scheme for scheduling problems (Graham, Lawler, Lenstra & Rinnoy Kan 1979) is a three-field descriptor $\psi_1|\psi_2|\psi_3$, where ψ_1 indicates the machine environment, ψ_2 describes the job properties, and ψ_3 the objective function to be minimised. We let $\psi_1 = \alpha\, m$, where m is the number of machines, and $\alpha \in \{P, Q, R, F, J, O\}$ for identical, uniform, and unrelated parallel machines, flow shop, job shop and open shop environments, respectively. We use the notation suggested by Potts & Kovalyov (2000) where if the machines involved are batching machines we use tilde on top of the environment (for example, $\tilde{P}m$ is an identical parallel machine environment with m batching machines). Under ψ_2 we may have any of the job properties given above, and, or:

- b_i if the maximum batch size on machine i is b_i, and just b if the maximum batch size on all machines is b.
- q_j is number of sublots job j is allowed to have. When all jobs have the same number of sublots, we drop the index j. Models without lot streaming have $q = 1$, and there is no need to make it explicit in ψ_2.
- *prec* if there are precedence constrains,
- *pmtn* if preemption is allowed,
- *recrc* if recirculation is present in the system,

Field ψ_3 may be any of the objective functions explained before. For example, $F2\ |q_j,\ s_{ij},\ t_j|\ C_{max}$ is the two-machine lot streaming flow shop problem, minimising makespan, with sequence independent setup times s_{ij}, transportation time t_j, and q_j sublots for each job. This problem is solved in Section 3.4. Another example is $\tilde{J}2\ |b_1 = 1,\ b_2 = 3|\ T_{\max}$, a two machine job shop where the first machine is a classical machine, and the second is a batching machine that can process up to 3 jobs at the same time, minimising the maximum tardiness. In this thesis we study problem $\tilde{1}\ |b|\ L_{\max}$, a single batching machine with restricted batch size to minimise the maximum lateness.

2.1.6 Dispatching rules

A simple solution procedure for scheduling problems is to priorities jobs, and schedule them according to a *dispatching rule.* A dispatching rule prioritizes all the jobs that are waiting for processing on a machine. The prioritization scheme may take into account the jobs properties and machine attributes, as well as the current time. Pinedo & Chao (1999) distinguish between static and dynamic dispatching rules. These rules are usually constructive in their approach. A static rule prioritizes all jobs before constructing the schedule, and does not change the priority as the schedule is being constructed (i.e. it

is just a function of the data of the problem). Dynamic rules, on the other hand, are time dependent, and prioritize jobs differently with time as the schedule is constructed. The first results on the theory of scheduling were static dispatching rules that gave optimal solutions for some problems. As an example we present the following:

- Shortest processing time (SPT) rule prioritizes jobs according to their processing time. It forms a schedule as a sequence of jobs in non-decreasing order of processing time. A schedule constructed in such a way minimizes the $\sum C_j$ or $\sum w_j C_j$ if $w_j = w \; \forall j$ in a single machine environment. This rule is also optimal for $Pm||\sum C_j$, and $Fm|prmu, p_{ij} = p_j|\sum C_j$ (see Pinedo (1995), Sections 4.3 & 5.1).

- Earliest due date (EDD) rule prioritizes jobs according to their due dates. It takes the jobs and orders them in non-decreasing order of their due dates to form the schedule as a sequence of jobs. It was proven by Jackson (1955) that this rule yields an optimal schedule for $1||L_{\max}$. We use it as well to construct starting solutions for our local search heuristics, see Chapter 7.

- Shortest processing time first, largest processing time second (SPT(1)-LPT(2)) rule, also referred to as Johnson's rule. This rule was originally designed for the $F2||C_{\max}$ problem, (Johnson 1954). It divides jobs into two sets; *set 1* where the processing time on the first machine is smaller than the second (i.e. $p_{1j} < p_{2j}$), and *set 2* where the processing time on the second machine is smaller than on the first machine (i.e $p_{1j} > p_{2j}$) Jobs where $p_{1j} = p_{2j}$ may be in either set. The SPT(1)-LPT(2) rule dispatches jobs in the *set 1* first, and in non-decreasing order of p_{1j}, and dispatches jobs in *set 2* afterwards, in non-increasing order of p_{2j}.

Another famous dispatching rule is the weighted shortest processing time first (WSPT) rule for $1||\sum w_j C_j$, where the jobs are ordered in decreasing order of w_j/p_j, due to Smith (1956). Several other dispatching rules

have been devised for different objective functions and environments (including $O2, Jm, Fm$); for a summary of other dispatching rules consult Pinedo (1995).

Throughout the thesis we focus mainly on the $C_{\max}$ objective function in a flow shop, and the $L_{\max}$ objective functions on a batching machine. Hence, we are particularly interested in the SPT(1)-LPT(2) and the EDD dispatching rules. We now prove why these rules work. In particular Johnson's rule will be used extensively in Chapters 3 and 4.

Theorem 2.1 (Johnson 1954). *$F2||C_{\max}$ is solved by an SPT(1)-LPT(2) schedule.*

Proof: Suppose that no optimal sequence corresponds to an SPT(1)-LPT(2) schedule. In an optimal sequence, there is a pair of adjacent jobs, say h before l such that they satisfy one of the following 3 conditions:

(i) job h belongs to *set 2* and l to *set 1*;

(ii) job h and l belong to *set 1* and $p_{1h} > p_{1l}$;

(iii) job h and l belong to *set 2* and $p_{2h} < p_{2l}$.

We need to show that, under any of the three conditions mentioned above, by interchanging the positions of job h and l in the sequence we get a makespan that is shorter or the same. Let C_{ij} denote the completion time of job j $(1 \leq j \leq n)$ on machine i $(i = 1, 2)$ under the original sequence and $\widehat{C}_{ij}$ the completion time under the sequence where h and l have been interchanged. Suppose that under the original sequence job t immediately precedes job h and that job g immediately follows job l. Interchanging h and l does not affect the starting time on the first machine of job g, which is equal to $C_{1t} + p_{1h} + p_{1l}$. However, we are interested in knowing when the second machine becomes available to process job g. Under the original sequence it is C_{2l}, but with the interchange it is $\widehat{C}_{2h}$. We need only to prove that, under any of the three conditions mentioned above, $\widehat{C}_{2h} \leq C_{2l}$. The completion of job l on the second machine under the original sequence is

$$C_{2l} = \max\{\max\{C_{2t}, C_{1t} + p_{1h}\} + p_{2h}, C_{1t} + p_{1h} + p_{1l}\} + p_{2l} =$$

$$= \max\{C_{2t} + p_{2h} + p_{2l},\ \ C_{1t} + p_{1h} + p_{2h} + p_{2l},\ \ C_{1t} + p_{1h} + p_{1l} + p_{2l}\},$$

and similarly the completion time of job h on the second machine after the interchange is

$$\widehat{C}_{2h} = \max\{C_{2t} + p_{2l} + p_{2h},\ \ C_{1t} + p_{1l} + p_{2l} + p_{2h},\ \ C_{1t} + p_{1l} + p_{1h} + p_{2h}\}.$$

Under condition (i), $p_{1h} \geq p_{2h}$, and $p_{1l} \leq p_{2l}$. Note that the first terms within the max expressions of C_{2l} and $\widehat{C}_{2h}$ are identical. The second term in the expression for $\widehat{C}_{2h}$ is no greater than the third term in the expression for C_{2l}, and the third term in the expression for $\widehat{C}_{2h}$ is no greater than the second term in the expression for C_{2l}. Hence, under condition (i), $\widehat{C}_{2h} \leq C_{2l}$.

Under condition (ii), $p_{1h} \leq p_{2h}$ as well as $p_{1l} \leq p_{2l}$, and $p_{1h} > p_{1l}$. In this case the second and the third terms in the expression for $\widehat{C}_{2h}$ are no greater than the second term in the expression for C_{2l}. Hence, under condition (ii) $\widehat{C}_{2h} \leq C_{2l}$.

Finally condition (iii) implies that $p_{1h} \geq p_{2h}$, and $p_{1l} \geq p_{2l}$, as well as $p_{2h} < p_{2l}$. So that the third term in the expression for C_{2l} is no less than the third and second term in the expression for $\widehat{C}_{2h}$. Hence under condition (iii) $\widehat{C}_{2h} \leq C_{2l}$. We need to repeat the same argument until we get an SPT(1)-LPT(2) sequence.

□

Theorem 2.2 (Jackson 1955). $1||L_{\max}$ *is solved by an EDD schedule.*

Proof: Let σ be an optimal sequence, and $L_{\max}(\sigma)$ its maximum lateness. Suppose that no optimal sequence corresponds to an EDD schedule. Then there are at least two consecutive jobs in σ say j sequenced before $j+1$ such that $d(j) > d(j+1)$. Construct a new sequence σ' where job $j+1$ is swaped with job j. The completion time of job j under σ is $C_{j-1}+p_j$, and its lateness is $L_j = C_{j-1} + p_j - d_j$, while the completion time of job $j+1$ under σ is $C_{j-1}+p_j+p_{j+1}$ and its lateness $L_{j+1} = C_{j-1}+p_j+p_{j+1}-d_{j+1}$. Now consider the maximum lateness of sequence σ'. The lateness of each job before j and

$j+1$ remains the same as no such job is affected by the swap. The lateness of each job after j and $j+1$ also remains the same as the completion time of the job j in σ' is $\widehat{C}_j = C_{j-1} + p_{j+1} + p_j = C_{j+1}$. We only need to look a the lateness of jobs j and $j+1$ under σ'. The lateness of job j in σ' is $\widehat{L}_j = C_{j-1} + p_{j+1} + p_j - d_j = C_{j+1} - d_j < C_{j+1} - d_{j+1} = L_{j+1} \leq L_{\max}(\sigma)$, and the lateness of job $j+1$ in σ' is $\widehat{L}_{j+1} = C_{j-1} + p_{j+1} - d_{j+1} < C_{j-1} + p_j + p_{j+1} - d_{j+1} = L_{j+1} \leq L_{\max}(\sigma)$. Thus, $L_{\max}(\sigma') \leq L_{\max}(\sigma)$. We can repeat the same argument until we get a sequence in EDD order.

□

Dispatching rules, however, tend only to yield optimal solutions for simple models. We consider an *algorithm* to be a step by step solution procedure which yields an optimal solution to a problem. That is, an algorithm will construct an optimal schedule (i.e. one that minimises the objective function). A procedure that does not yield an optimal solution is referred to as a *heuristic*. We analyse a class of heuristics (local search heuristics) in Chapter 5, Section 5.4. General purpose algorithms like branch and bound, and dynamic programming are introduced in Chapter 5, Section 5.3. Many scheduling problems can be formulated as *(mixed integer) linear programming problems* MILP, see Schrijver (1986). Another solution procedure for scheduling problems, therefore, is the methodology for MILP problems. However, for most scheduling problems it is hard to find algorithms that solve them in a reasonable amount of time. It is common to classify the difficulty of scheduling problems by the time it takes to solve them. A frame for such a classification was introduced by Cook (1971) and Karp (1972), and is referred to as the theory of computational complexity. We give an overview of the theory in the next section.

2.2 Computational Complexity

An intuitive way of classifying problems is the effort (time) required to solve them to optimality. The main contribution of the theory of computational complexity is to give a framework for such a classification. In this section we give a non-rigorous overview of the concepts involved in this classification. For a rigorous and detailed treatment we refer the reader to the classical book by Garey & Johnson (1979). We also base our presentation on books by Papadimitriou & Steiglitz (1982), Papadimitriou (1994), and Cook, Cunningham, Pulleyblank & Schrijver (1998).

An *instance* of a problem is obtained by specifying particular values for the parameters (machine environment, job properties) that define it. The size of an instance can be roughly defined as the number of symbols required to represent it. The effort (time) required for an algorithm to obtain a solution grows with the size of the problem instance. It is natural then, to represent this effort as a function of the size of the instance. In fact the *running time* of an algorithm for a given problem is measured by an upper bound on the number of elementary steps the algorithm has to perform. Specifically, if n is the size of an instance, and $f(n)$ is an upper bound on the number of steps the algorithm performs, then the algorithm *runs* in $O(g(n))$ time, or is said to have $O(g(n))$ time complexity, if there exists a constant $c > 0$ such that for large enough n, $f(n) \leq c\ g(n)$. An *polynomial time* algorithm is one where $g(n)$ is a polynomial function. We have an *exponential time* algorithm, when it is not a polynomial time algorithm, this includes functions of the form $n!$ and $n^{\log n}$.

Other standard notation for the asymptotic running time of an algorithm (see Cormen, Leiserson & Rivest (1993)) is as follows. Let $f(n) : \mathcal{Z}^+ \longrightarrow \Re^+$ and $g(n) : \mathcal{Z}^+ \longrightarrow \Re^+$ then $f(n)$ is $\Omega(g(n))$ if there exists a constant $c > 0$ such that, for large enough n, $f(n) \geq c\ g(n)$, and $f(n)$ is $\Theta(g(n))$ if there exist constants $c_l, c_u > 0$ such that, for large enough n, $c_l\ g(n) \leq f(n) \leq c_u\ g(n)$. Note that just as O-notation gives an asymptotic upper bound, the Ω-notation provides an asymptotic lower bound. We can use this same

notation to refer to the space requirement of the algorithm (information an algorithm needs to access while running), or to denote the size of a solution space (see Section 5.4).

As explained before, complexity theory aims to classify problems into 'hard' or 'easy' depending on the time it takes to solve them. The core of the classification lies in grouping problems into two main classes, namely P and NP. The definitions of P (deterministic polynomial) and NP (non-deterministic polynomial) classes are based on the concept of a deterministic and non-deterministic Turing Machine. A Turing Machine is a mathematical model of an algorithm. A problem is in the P class if it is possible to solve it in polynomial time by a deterministic Turing Machine. A non-deterministic Turing machine is a theoretical extension of the deterministic machine which can evaluate an exponential number of solutions in a polynomial bounded number of computations. A problem is in NP if it can be solved in polynomial time by a non-deterministic Turing Machine. It follows that $P \subseteq NP$.

In practice a problem belongs to class P if there exists an polynomial time algorithm that solves it. For example, the ellipsoid algorithm by Khachian (1979) solves any linear programming (LP) problem in polynomial time; hence LP problems belong to P. However, for a long time it was unknown if LP problems belonged to P. That there is no known polynomial algorithm for a problem does not guarantee that it does not belong to P, rather that the problem is *intractable.* However, it is of interest to know if such a problem is NP-complete. We note that the NP-completeness theory refers to *decision problems* (one whose solution is either 'yes' or 'no'). For any given problem there exists an associated decision problem. Membership of the NP-complete class is 'harder' to establish, and is based on the fact that all NP problems can be reduced to any NP-complete problem in polynomial time. Cook (1971) was the first to give an NP-complete problem, namely the *satisfiability* problem. If problem Π can be reduced to problem Π' by a polynomial time procedure, then they belong to the same class. This is significant because if we were able to find a polynomial time algorithm for

an NP-complete problem, then we would prove that P = NP. A list of NP-complete problems can be found in Garey & Johnson (1979), with recent results summarised in (Papadimitriou 1994). Unfortunately there has been no success in finding any polynomial algorithm for an NP-complete problem, nor in proving P $\neq$ NP. In fact, what the theory seems to conclude is that the 'hardest' problems (NP-complete problems) are in some sense equivalent, and it is widely believed that NP $\neq$ P.

The size of an instance is encoding-dependent (i.e. dependent on the code or 'language' utilized to represent the parameters of the problem). NP-complete problems, can be divided into two subclasses depending on the encoding scheme. These are *unary* NP-complete (or strongly NP-complete), and *binary* NP-complete (sometimes referred to as NP-complete in the ordinary sense). These concepts were introduced by Garey & Johnson (1978) and Lageweg, Lawler, Lenstra & Rinnooy Kan (1978). If a problem is unary NP-complete, then it is NP-complete even when the encoding scheme uses unary notation (where a string of n ones represents number n, expressed in base 1). Such a problem differs from a binary NP-complete problem by the fact that a binary NP-complete problem may have a *pseudo-polynomial* algorithm, one that runs in polynomial time if the encoding is unary. Hence, in some sense unary NP-complete problems are 'harder' than binary NP-complete problems. Very informally a problem is termed unary NP-hard when the corresponding decision problem is NP-complete in the strong sense, and binary NP-hard if the corresponding decision problem is NP-complete in the ordinary sense. For a more detailed treatment of these terms (including Turing reducibility and number problems) consult Garey & Johnson (1979).

In scheduling problems there is a thin line dividing NP-hard problems from P problems. For example, in problem $1||L_{\max}$, that belongs to P (as EDD rule solves it), changing to a batching machine with restricted batch sizes (i.e. $\tilde{1}|b|L_{\max}$) produces an NP-hard problem.

Computational complexity is a worst case analysis, and is by no means a measure of the expected running time in practice. An illustrative example

is the *simplex algorithm* of (Dantzig 1949) that solves linear programming problems. It is an exponential time algorithm, but it works very well on average, and for many instances outperforms the ellipsoidal method. However, the theory does give an indication of the complexity of the problem, from which we can deduce the type of methods that are most suitable to find solutions. Heuristic methods, like the ones described in Chapter 5, are used for NP-hard problems if solutions are required using reasonable computational resources.

Part I

Lot streaming

Chapter 3

Lot Streaming Basic Models

3.1 Introduction

As explained in chapter 2, it is common for a job to finish its operation on one machine before proceeding to the next machine in the system. Lot streaming, however, is the process of splitting partially completed jobs into *sublots* to be transferred to downstream machines in multi-stage production systems. It is an extension on the classical machine models where the number of sublots for each job j, denoted by q_j, is $q_j = 1$. By allowing the overlapping between successive operations we may obtain a reduction on the completion time of the jobs in the resulting schedule, and the work-in-process inventory levels. Lot streaming also improves customer services, as partially completed sublots may be delivered before the whole job (order) is completed, as pointed out by Potts & Van Wassenhove (1992).

We illustrate this in the following example. Consider the $F2 \mid\mid C_{\max}$ model with 3 jobs, and the following processing times.

Table 3.1: Processing times for a 3-job example

job	J_1	J_2	J_3
p_{1j}	4	6	6
p_{2j}	5	8	4

Johnson's SPT(1)-LPT(2) dispatching rule will give a schedule where job J_1 is sequenced before J_2, and job J_2 before J_3, with makespan $C_{\max} = 22$. A Gantt chart representing this optimal schedule is shown in Figure 3.1. Note how the operation on machine M_1 for a particular job is finished before its operation on machine M_2 begins. Machine M_2 is idle for 4 time units before the operation on the first job starts, and it is idle for 1 time unit while it awaits for operation on job J_2 to finish on the first machine.

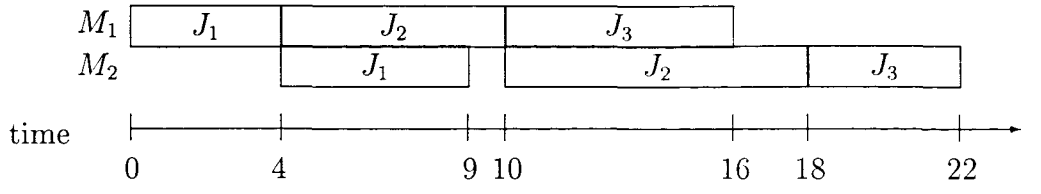

Figure 3.1: Gantt Chart for the 3-job example

Allow lot streaming on J_2, say by dividing it into two equally sized sublots throughout the flow shop. Hence, divide the operation of J_2 on M_1 in two 3 time unit sublots, and the operation on M_2 in two 4 time unit sublots. The schedule with the same job sequence as before is shown in the Gantt chart of Figure 3.2. Note how for job J_2 there is an overlap of the operation on the second sublot on machine M_1, and the operation of the first sublot on machine M_2 (between time 9 and 10). Half of job J_2 (sublot 1) becomes available for dispaching to the customer at time 13, and the whole job is completed at time 17, whereas before its completion time was $C_2 = 18$. Not only that, but the duration of the whole process, the makespan of the schedule, is reduced to 21. This reduction in makespan results from a higher utilization of machine M_2, where the idle time is reduced by 1 unit.

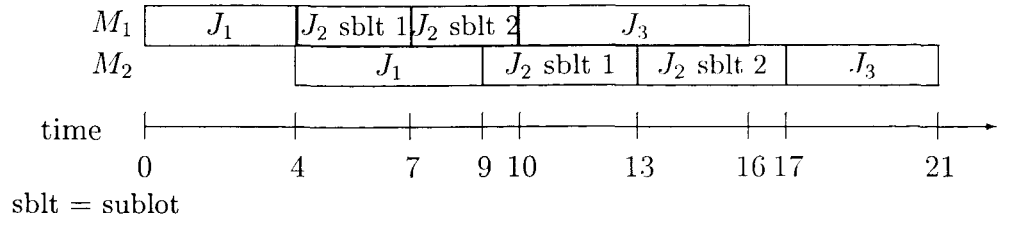

Figure 3.2: Gantt Chart for the 3-job example, where $q_2 = 2$

A job may consist of several items. Hence, after they are completed in the first machine they wait until all of the items are finished before they move to the second machine. Those items waiting for others to be completed are part of the work-in-process inventory. We can appreciate how the inventory levels between time 4 and 10 are reduced when lot streaming is allowed.

The aim of this chapter is to review important results for some models that deal with lot streaming, and give more insight into the $F2\,|q_j,\ s_{ij},\ t_j|\,C_{\max}$ model. In Section 3.2 we give an overview of the results in the literature. In Section 3.3 we give more details on the network representation of the single job flow shop lot streaming model, and the dominant machine analysis by Glass & Potts (1998). In Section 3.4 we present a critical path approach to find an optimal schedule for the $F2\,|q_j,\ s_{ij},\ t_j|\,C_{\max}$ model.

3.2 Literature Review

Lot streaming models date back to work by Szendorvits (1975). His model shows that there are potential savings in the holding, work-in-process, and final inventory costs when lot streaming (overlap of operations) are considered in multi-stage production systems. His analysis focuses on a single job (production lot) with equally sized sublots. Goyal (1976) treats the problem of optimising the sublot sizes in Szendorvits model, both models assume there is no idle time between sublots (but there may be between different jobs). These results are in accordance with the just in time (JIT) philosophy where shorter production runs are preferred.

The first lot streaming models focused in obtaining optimal sublot sizes for a single job. The makespan is a natural cost function, as the aim is to pass the lot (job) through the system as quick as possible. Baker (1988) gives a linear programming formulation for a single job in a flow shop environment, minimising the makespan. He allows idle time between sublots, and expresses the makespan as the sum of the idle times in the last machine plus the processing time of the sublots on that machine. His model assumes that the

number of sublots is known (q fixed beforehand). One possible formulation for this problem focusing on the completion time of sublots, rather than the idle time is as follows.

Lot streaming for a single job minimising the makespan

Let C_{ki} denote the completion time of sublot k ($1 \leq k \leq q$) on machine i ($1 \leq i \leq m$), and let x_k be the proportion of the job belonging to sublot k, then the problem of minimising the makespan of a single job with q sublots through m machines can be formulated as:

$$
\begin{aligned}
\min \quad & C_{qm} \\
\text{subject to} \quad & \\
& C_{11} - p_1 x_1 \geq 0, \\
& C_{ki} - C_{ki-1} - p_i x_k \geq 0, \quad \text{for } 1 \leq k \leq q, \text{ and } 2 \leq i \leq m \qquad (3.1) \\
& C_{ki} - C_{k-1i} - p_i x_k \geq 0, \quad \text{for } 2 \leq k \leq q, \text{ and } 1 \leq i \leq m \qquad (3.2) \\
& \textstyle\sum_{k=1}^{q} x_k = 1, \\
& x_k, C_{ki} \geq 0, \qquad \text{for } 1 \leq k \leq q, \text{ and } 1 \leq i \leq m.
\end{aligned}
$$

Note that C_{qm} is the completion time of the last sublot of the job in the last machine, which is the makespan of the single job. The processing time of sublot k on machine i is $p_i x_k$. Thus, the first inequality guarantees that the completion time of the first sublot on the first machine is not smaller than the processing time of the first sublot on the first machine. The set of inequalities (3.1) ensure that the processing of sublot k on machine $i-1$ is completed before it starts its processing on machine i. This is usually referred to as the *sublot processing constraint.* The second set of inequalities (3.2) makes sure that sublot $k-1$ has completed its processing on machine i before the processing of the next sublot (k) starts. This is referred to as the *machine capacity constraints.* The fourth inequality guarantees that x_k is a proportion of the job. A solution to this problem is fully determined by the values of x_k for $k = 1, \ldots, q$.

Baker (1988) showed explicitly that the sublot sizes for the two machine case ($m = 2$) are geometric. That is, if x_k is the proportion of the job

belonging to sublot k then,

$$x_k = \frac{(\frac{p_2}{p_1})^{k-1}}{\sum_{v=0}^{q-1}(\frac{p_2}{p_1})^v} . \tag{3.3}$$

In general, let us denote with x_{ijk} the proportion of job J_j belonging to sublot k, on machine M_i for $j = 1, \ldots, n$, $i = 1, \ldots m$, and $k = 1, \ldots, q_j$. We refer to x_{ijk} as the *sublot size.*

We say sublots are *consistent* if the sublot size x_{ijk} does not vary from machine to machine (i.e. we may drop index i). In fact, Potts & Baker (1989) showed that for the 2 or 3 machine flow shop ($m = 2, 3$) there is an optimal schedule with consistent sublot sizes. They point out that this property is valid for n different jobs not only the single job case. In their paper they also compared equally sized sublots ($x_k = 1/q$) with optimal ones, finding that the advantage of using optimal sublot sizes can not be more than 1.53 times that of using equally sized sublots. An upper bound on the improvement of an extra sublot on the makespan of the single job is $1/(q+1)^2$. Note that even though Baker's initial model did not consider the possibility of varying sublot sizes on different machines, his result still holds, and formula (3.3) is valid. Consider the 3-job example given in Section 3.1. If we were to schedule only job J_2 without lot streaming the makespan of that single job would be 14 ($=p_{12} + p_{22}$). Under the suggested equally sized sublots, sublots 1 and 2 would overlap for 3 time units, and the makespan would be reduced to 11. However, if we were to use the optimal sublot sizes ($x_1 = 3/7$, and $x_2 = 4/7$), the makespan would be reduced even further to $10\frac{4}{7}$.

Vickson & Alfredsson (1992), Trietsch & Baker (1993), and Cetinkaya (1994) consider models where jobs are composed of several indivisible items. Each item becomes available to be transferred to the next stage immediately after the operation on it has been performed. There is no limit on the number of sublots one can pass along to downstream machines. However, consideration of setup and transportation times, might not make it ideal to divide the job into a large number of sublots. Trietsch & Baker (1993) refer to such a model as a *discrete* model, and to those models where jobs can be

divided into any fraction of the job as *continuous* models. It is a common suggestion to use the solutions to continuous models to approximate those of discrete models. The more items in a job the closer the approximation will be. Discrete models are usually explored with linear integer programming techniques. However they tend to give little insight into the structure of the problem, and are rarely useful in exploring extensions to the model. Throughout this thesis we will focus on continuous models, and the use of network formulations for them.

For continuous models a network representation of the problem, like the one used by Potts & Baker (1989), Glass, Gupta & Potts (1994), Glass & Potts (1998) has proven a useful analysis tool. We give some details in the next section. Glass, Gupta & Potts (1994) also analysed the job shop and open shop environments. They explain how a flow shop relaxation algorithm can be used to generate a schedule with minimum makespan for a job shop. They also provide an $O(q)$-time algorithm for the 3 stage open shop when using consistent sublot sizes. Attach and detached setup times are considered for the three machine flow shop in Chen & Steiner (1996) and Chen & Steiner (1998), effortlessly from the analysis of Glass, Gupta & Potts (1994).

Going back to the 3 job example of section 3.1 if we were to change the sequence so that job J_2 precedes J_1 (i.e. job sequence (J_2, J_1, J_3)) with the suggested equally sized sublots on J_2, the makespan would be reduced even further to $C_{\max} = 20$. Hence, improvements on the schedule can be obtained from the sublots sizes as well as the re-sequencing of jobs. Decisions on the sublot size and sequence of jobs have to be taken into account when minimizing the makespan of a given set of n jobs. Vickson (1995) studies a model with n different jobs in the two-machine flow shop, which includes transportation and setup times. His analysis focuses on the idle time on the machines, relies on linear programming formulation for the model, and is mainly algebraic. Surprisingly he found that the problem collapses to a simple two-machine flow shop, and that sublot size and sequencing decisions can be taken independently. We give more insight into his results using a network

representation for the problem in Section 3.4. Vickson (1995) presents an algorithm for the case where jobs are composed of several indivisible items, but assumes that there is a finite number of sublots per job. Baker (1995) also tackles the setup times with transfer lots of size one, using a time lag model.

Approximation methods for the discrete lot streaming are given in Chen & Steiner (1997), Sen, Topaloglu, & Benli (1998), and Chen & Steiner (1999). Dauzère-Pérès & Lasserre (1997) explore the more general job shop model. They present computational results for their procedure, which solves iteratively a lot sizing and sequencing decision. When setups are not considered their approximation procedures show that a small number of sublots may only be needed. Their results also indicate that ignoring the sequencing of operations in the machine capacity constraints may be a valid model when lot streaming is available. Upper bounds on the benefit of using lot streaming for the makespan, mean flow time and average work-in-process performance measures are presented in Kalir & Sarin (2000), though they use simplifying assumptions for their models and do not consider transfer, or set up times. They also propose a heuristic (Karlir & Sarin 2001) to schedule several jobs with consistent sublot sizes through several machines. However they ignore the work done by (Vickson 1995), and the analysis on dominant machines by Glass & Potts (1998), and focus mainly on improving the heuristic given by Dauzère-Pérès & Lasserre (1997). Their heuristic focuses in finding a bottleneck machine and reducing the idle time on it by sequencing the jobs efficiently on it.

Ramasesh, Fu, Fong, & Hayya (2000) analyse the benefits of lot streaming in the work-in-process inventories focusing on a single job, and present numerical examples to show the improvement in the manufacturing cycle time. A no-wait flow shop is considered by Kumar, Bagchi, & Sriskandarajah (2000), they develop a genetic algorithm for the multiple job case to minimise the makespan. Bogaschewsky, Buscher, & Lindner (2001) review Goyal (1976) model to consider a modification of their two stage model for

manufacturing systems, and present computational results for the heuristic they propose.

We will denote by $\underline{q} = (q_1, \ldots, q_n)$ the vector that holds the information on the number of sublots each job is allowed to have. The 3-job example of Section 3.1 has $\underline{q} = (1, 2, 1)$. Most models in the literature consider $\underline{q}$ as a technological constraint, and regard it as given beforehand, but we explore a new model where this in not the case in Chapter 4.

3.3 Network Representation and Dominant Machines

In this section we briefly introduce a network representation for the single job lot streaming model in a flow shop, and the dominant machine analysis done by Glass & Potts (1998), which we will use extensively in the next chapter.

In the m-machine flow shop environment for a fixed *vector of sublots sizes* $\underline{x} = (x_1, \ldots, x_q)$, the network representation $N(\underline{x})$ of a single job is composed of mq vertices, where each vertex (i, k) represents the processing of sublot k on machine i (for $i = 1, \ldots, m$, and $k = 1, \ldots, q$). Vertex (i, k) has an associated weight of $p_i x_k$ (p_i the processing time on machine i and x_k the size of sublot k). An arc is directed from (i, k) to $(i, k+1)$, (for $1 \leq k \leq q-1$ and $1 \leq i \leq m$) to represent the *sublot constraint* that sublots may not overlap, and an arc from (i, k) to $(i+1, k)$, (for $1 \leq i \leq m-1$ and $1 \leq k \leq q$), to represent the *machine capacity constraint* that a machine is unable to process two sublots at the same time. Taking the length of a path in $N(\underline{x})$ to be the sum of weights of the vertices which lie in the path, then any longest path from $(1, 1)$ to (m, q) will give the makespan of the job. The makespan for $m = 2$ can be expressed as

$$\max_{1 \leq k^* \leq q} \{ \sum_{k=1}^{k^*} p_1 x_k + \sum_{k=k^*}^{q} p_2 x_k \}. \tag{3.4}$$

Glass, Gupta & Potts (1994) define as a *critical path* any longest path from

$(1,1)$ to (m,q) in $N(\underline{x})$; and a subpath of a critical path as a *critical segment.* If $(i,k)-(i+1,k)$ is a critical segment for some machine i $(1 \leq i \leq m-1)$ then k is a *critical sublot*, and if $(i,k)-(i,k+1)$ is a critical segment for some sublot k $(1 \leq k \leq q-1)$ then i is a *critical machine.* They prove that in any network of optimal sublot sizes, every sublot is critical, and that there exists a vector of optimal sublot sizes $\underline{x}$ for which $N(\underline{x})$ contains at least two critical machines from sublot k for $1 \leq k \leq q-1$. Based on this properties they obtain a closed form for the optimal sublots sizes for the 3 machine flow shop.

Glass & Potts (1998) expand this result to environments with more machines, introducing the concept of dominant machines. They define a machine v to be *dominated* by machines u and w $(u < v < w)$ if

$$\left(\sum_{i=u+1}^{v} p_i\right)\left(\sum_{i=v}^{w-1} p_i\right) \leq \left(\sum_{i=u}^{v-1} p_i\right)\left(\sum_{i=v+1}^{w} p_i\right).$$

A machine is dominant if it is not dominated by any other two machines. They prove that the machine capacity constraints of a dominated machine does not come into consideration when finding optimal sublot sizes.

Glass & Potts (1998) give an $O(m)$-time algorithm to find the dominant machines in any m machine flow shop. After applying the algorithm to a lot streaming problem, say $\mathcal{F}$, a new related problem $\mathcal{R}$, with m' machines, is obtained. Problem $\mathcal{R}$ consists of alternating *capacitated* and *lag* machines starting and finishing with a capacitated machine. Capacitated machine u' in $\mathcal{R}$ $(1 \leq u' \leq m')$ is associated with dominant machine $\mu_{u'}$ in $\mathcal{F}$ with the same processing time, where $1 = \mu_1 \leq \mu_{u'} \leq \mu_{m'} = m$. Every dominant machine in $\mathcal{F}$ corresponds to a capacitated machine in $\mathcal{R}$. Every lag machine in $\mathcal{R}$ corresponds to either one or more dominated machines in $\mathcal{F}$, (with processing time, or *lag time*, equal to the sum of the processing times of the dominated machines between the dominant machines) or is a dummy machine (with processing time zero; this happens when $\mu_{u'+1} = \mu_{u'} + 1$). $\mathcal{R}$ is a relaxation of $\mathcal{F}$ since it corresponds to having the machine capacity constraint relaxed. For an optimal solution the machine capacity constraint

of a dominated machine will not be tight.

The network representation of the original problem $\mathcal{F}$ is denoted by $N_{\mathcal{F}}(\underline{x})$, and $N_{\mathcal{R}}(\underline{x})$ denotes the network representation for problem $\mathcal{R}$. There is a vertex (u', k) in $N_{\mathcal{R}}(\underline{x})$ for $1 \leq u' \leq m'$, which corresponds to a capacitated machine vertex $(\mu_{u'}, k)$ in $N_{\mathcal{F}}(\underline{x})$ with weight $p_{\mu_{u'}} x_k$; and a vertex $[u', k]$ for every lag machine in $\mathcal{R}$ with weight $l_{u'} x_k$ where

$$l_{u'} = \sum_{i=\mu_{u'}+1}^{\mu_{u'+1}-1} p_i$$

when $\mu_{u'+1} \neq \mu_{u'} + 1$, and $l_{u'} = 0$ when $\mu_{u'+1} = \mu_{u'} + 1$. There are arcs directed from (u', k) to $[u', k]$ and from $[u', k]$ to $(u' + 1, k)$ in $N_{\mathcal{R}}(\underline{x})$, for $1 \leq u' \leq m' - 1$ and $1 \leq k \leq q$, and an arc from (u', k) to $(u', k + 1)$ for $1 \leq u' \leq m'$ and $1 \leq k \leq q - 1$. As an example consider the case where $m = 5$ and the processing times are given by $p_1 = 5, p_2 = 4, p_3 = 3, p_4 = 4$ and $p_5 = 2$. The corresponding network graphs for $N_{\mathcal{F}}(\underline{x})$, and $N_{\mathcal{R}}(\underline{x})$ are given in Figure 3.3.

A *block* $B(u, i; v, j; \underline{x})$ in $N_{\mathcal{F}}(\underline{x})$ is composed of all paths from vertex (u, i) to vertex (v, j), except those that have a segment on some machine other than u and v. They are of the form $(u, i) - \ldots - (u, k) - \ldots - (v, k) - \ldots - (v, j)$. A block is said to be a *critical block* if all of its segments are critical. The following theorem states the structure of an optimal solution in the network representation of a one job lot streaming problem.

Theorem 3.1 *There exists a vector of optimal sublot sizes $\underline{x}$ for the original problem $\mathcal{F}$ for which $N_{\mathcal{F}}(\underline{x})$ has critical blocks $B(\mu_{u'}, h_{\mu_{u'}}; \mu_{u'+1}, h_{\mu_{u'+1}}; \underline{x})$ for $1 \leq u' \leq m' - 1$ for some integers $h_1, \ldots, h_{m'}$, where $1 = h_1 \leq h_2 \leq \ldots \leq h_{m'} = q$, and*

$$\frac{x_k}{x_{k+1}} = \frac{\sum_{i=\mu_{u'}}^{\mu_{u'+1}-1} p_i}{\sum_{i=\mu_{u'}+1}^{\mu_{u'+1}} p_i} \tag{3.5}$$

for $h_{u'} \leq k \leq h_{u'+1} - 1$ and $1 \leq u' \leq m' - 1$.

Finally to solve the problem they focus on the vector of integers $\underline{h} = (h_1, h_2, \ldots, h_{m'})$ where $h_1 = 1$ and $h_{m'} = q$. Note that equation (3.5)

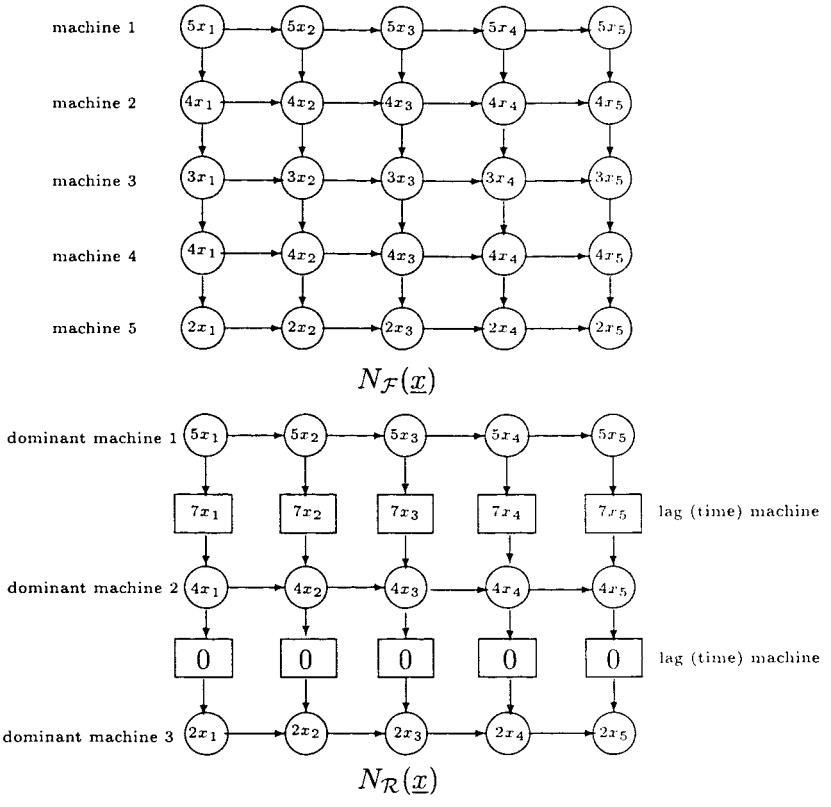

Figure 3.3: Network $N_{\mathcal{F}}(\underline{x})$, and $N_{\mathcal{R}}(\underline{x})$

defines a recursive relationship from which, for a fixed $\underline{h}$, the vector of sublot sizes $\underline{x}$ can be calculated. That is, if we define

$$\rho_{u'} = \frac{\sum_{i=\mu_{u'}}^{\mu_{u'+1}-1} p_i}{\sum_{i=\mu_{u'}+1}^{\mu_{u'+1}} p_i},$$

then under $\underline{h}$,

$$x_j^{(h)} = \rho_{u'} x_{j+1}^{(h)} \text{ for } h_{u'} \leq j \leq h_{u'+1} - 1,$$

and so

$$x_{h_{u'}}^{(h)} = \rho_{u'}^{h_{u'+1}-h_{u'}} x_{h_{u'+1}}^{(h)} \text{ for } 1 \leq u' \leq m' - 1.$$

The makespan in this case is given by

$$M(\underline{h}) = \sum_{u'=2}^{m'} \left(p_{\mu_{u'-1}} \sum_{j=h_{u'-1}}^{h_{u'}} x_j^{(h)} + l_{u'-1} x_{h_{u'}}^{(h)} \right) + p_{m'} x_q^{(h)}.$$

So that the makespan $M(\underline{h})$ can be computed in $O(m')$ time for a fixed $\underline{h}$. There are $O(q^{m'-2})$ feasible vectors $\underline{h}$ which define critical blocks so to obtain the optimal solution requires $O(m + m'q^{m'-2})$ time.

3.4 A two-machine flow shop model

In this section we study a continuous lot streaming model set in a two machine flow shop with multiple jobs, set up times, and transportation times, $F2|q_j,\ s_{ij},\ t_j|C_{\max}$. Vickson (1995) has analysed this model, focusing on the idle time between machines. He shows that the makespan minimisation reduces to a simple two machine flow shop without lot streaming. However he says "we do not currently posses an intuitive explanation of why the setup time, processing time and lot streaming effects collapse into such a simple form". In this section we give more insight into why this happens, using a network representation for the problem.

The data for any given instance of the problem comprise

- q_j the number of sublots for each job j ($j = 1, \ldots n$),
- p_{ij} the processing time of job j on machine i (for $j = 1, \ldots n$, and $i = 1, 2$),
- s_{ij} the setup time of each job j on each machine i (for $j = 1, \ldots n$, and $i = 1, 2$),
- t_j sublot size dependent transportation time, and
- T_j a fixed transportation time for each job $j = 1, \ldots, n$.

The sublot size dependent transportation time portraits the fact that the transportation time may vary depending on the amount of items being transported, and the fixed transportation time represents the time taken every time a sublot of a particular job j is passed along form machine 1 to 2 in the flow shop. Interleaving of sublots (preemption) is not allowed, and the setups are anticipatory (they can start before the job arrives to the machine). If $C_{\sigma(j)}$ is the completion time of the job in the j-th position in sequence σ;

then, $C_{\sigma(n)}$ is the completion of the last job in sequence σ, and the objective is to find

$$\min_{\sigma, x_{kj}} C_{\sigma(n)},$$

the sequence of jobs σ (a permutation of the n jobs) and sublot sizes x_{kj} $(1 \leq k \leq q_j)$ for each job $j = 1, \ldots n$ that minimize the makespan of the n jobs to schedule.

Any solution to this model is determined by two elements, the sublot sizes for each job, and the sequencing of jobs. We show why these two elements can be tackled separately in Theorem 3.2, give a formula for the optimal sublot sizes of each job in Theorem 3.3, and show that there is a simple rule to sequence all jobs in Theorem 3.4. Our aim is to clarify the effect the setups and transportation times have on the solution.

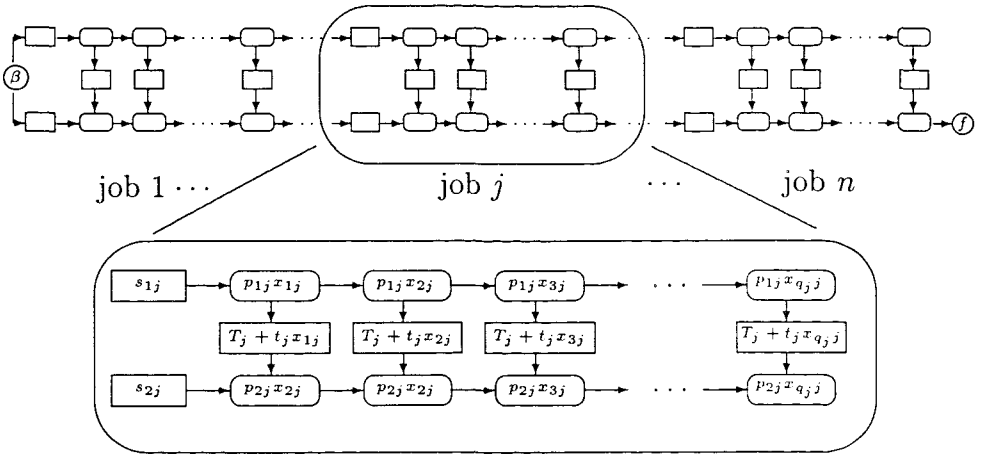

Figure 3.4: Network representation for sequence σ.

For a given sequence of jobs σ, numbered for convenience as $\sigma = (1, \ldots, n)$, we choose to represent the problem with the network shown in Figure 3.4. This network representation follows similar ideas to the network for a single job explained in Section 3.2. We have $3\sum_{j=1}^{n} q_j + 2(n+1)$ nodes. The circular node β represents the start of the schedule, while the circular node f represents the finishing of the schedule. The oval nodes represent the processing time of sublot k of job j on machine i with associated weight $x_{kj}p_{ij}$. The arcs joining them horizontally represent the machine capacity constraints in-

dicating that sublot x_{k-1j} must be completed on machine j before sublot x_{kj} starts its processing ($k = 2, \ldots, q_j$). A setup must be performed before we start processing the first sublot on the first machine which is represented by an arc is directed from the rectangular node with weight s_{1j} to the node with weight $p_{1j}x_{1j}$. For the same reason, the rectangular node with weight s_{2j}, representing the setup on the second machine, precedes work on the first sublot on the second machine. The total transportation time between the first machine and the second for sublot x_{kj} is $T_j + t_j x_{kj}$. This is expressed by the arcs and the rectangular node with weight $T_j + t_j x_{kj}$ that join sublot x_{kj} between machine one and two. The makespan of the n jobs under sequence σ is the length of the longest path joining node β to f in the network. If the schedule starts at time 0, then the weight associated with node β is zero. There is a vector of sublot sizes for each job. We will use the notation $\underline{x}^j$ to refer to the sublot sizes of job j. That is, $\underline{x}^j = (x_{1j}, \ldots, x_{q_j j})$. For a specific job j and its vector of sublot sizes $\underline{x}^j$, let

$$M(j, \underline{x}^j) = s_{1j} + \max_{1 \le k^* \le q_j} \left\{ \sum_{k=1}^{k^*} p_{1j}x_{kj} + T_j + t_j x_{k^*j} + \sum_{k=k^*}^{q_j} p_{2j}x_{kj} \right\}. \tag{3.6}$$

For $j < n$, $M(j, \underline{x}^j)$ is the length of the longest path from the setup of job j in the first machine to the setup of the job $j+1$ on the second machine. The makespan of sequence σ with sublot sizes $X = (x_{11}, \ldots, x_{q_1 1}, \ldots, x_{1n}, \ldots, x_{q_n n})$, denoted by $C_{\max}(\sigma, X)$, can be calculated as

$$\max \left\{ \begin{array}{ll} \sum_{j=1}^{n} (s_{2j} + p_{2j}), & \max_{1 \le j^* \le n} \{ \sum_{j=1}^{j^*-1} (s_{1j} + p_{1j}) + M(j^*, \underline{x}^{j^*}) \\ & \qquad + \sum_{j=j^*+1}^{n} (s_{2j} + p_{2j}) \} \end{array} \right\} \tag{3.7}$$

That is, the longest path from node β to node f (see figure 3.4) goes along machine two all the way form job 1 to job n, or starts on the first machine and goes down on a particular sublot k^* of a job j^* to machine two and continues on machine 2 to job n, or goes along the first machine until job n in a partiuclar sublot k^*. The only part in the expression for $C_{\max}(\sigma, X)$, in equation (3.7), that depends on the sublot sizes of the jobs is $M(j^*, \underline{x}^{j^*})$.

This is true regardless of the sequence σ. Hence, it is always better to use the sublot sizes that minimise $M(j^*, \underline{x}^{j^*})$ for any σ. This yields the following theorem.

Theorem 3.2 *For the $F2|q_j,\ s_{ij},\ t_j|C_{\max}$ model, the sublot size decision is independent of the sequencing decision. The optimal sublot sizes for job j are the ones that minimise $M(j, \underline{x}^j)$ for $j = 1 \ldots n$.*

To find out the size of the optimal sublot, let us focus on the value of $M(j, \underline{x}^j)$. Equation (3.6) can be rewritten as

$$M(j, \underline{x}^j) = (s_{1j} + T_j - t_j) + \max_{1 \le k^* \le q_j} \{\sum_{k=1}^{k^*} (p_{1j} + t_j)x_{kj} + \sum_{k=k^*}^{q_j} (p_{2j} + t_j)x_{kj}\}. \quad (3.8)$$

Observe that the first term $s_{1j} + T_j - t_j$ is constant with respect to $\underline{x}^j$. Not only that, but the second term is the expression for the makespan of an q-sublot single job two-machine flow shop model with $(p_{1j} + t_j)$ processing time on the first machine, and $(p_{2j} + t_j)$ processing time in the second machine as explained in Section 3.3 (equation 3.4). The optimal sublot sizes are given by the geometric progression of equation (3.3) derived by Potts & Baker (1989). Hence, we can state the following theorem.

Theorem 3.3 *The sublot sizes $\underline{x}^j = (x_{1j}, \ldots, x_{q_j})$ that minimise $M(j, \underline{x}^j)$ $(1 \le j \le n)$ are given, for $k = 1, \ldots, q_j$, by*

$$x^*_{kj} = \frac{\left(\frac{p_{2j}+t_j}{p_{1j}+t_j}\right)^{k-1}}{\sum_{v=0}^{q_j-1} \left(\frac{p_{2j}+t_j}{p_{1j}+t_j}\right)^{v}}. \quad (3.9)$$

We now focus on the sequencing decision. Fix the sublot sizes to be the ones given by equation (3.9), which from Theorem 3.3 we know to be optimal. Note that we have reduced the problem to one without transportation times. In the reduced problem (without transportation times and sublot size decision) the setup time are $s'_{1j} = s_{1j} + T_j - t_j$, and $s'_{2j} = s_{2j}$, while the

job processing times are $p'_{1j} = p_{1j} + t_j$, $p'_{2j} = p_{2j} + t_j$ on machine 1 and 2 respectively. From the properties of the critical paths in a two-machine flow shop (Potts & Baker 1989), we can write $p'_{1j} - a_j = p'_{2j} - b_j = g_j$, where a_j is the processing time of the first sublot on the first machine, and b_j is the processing time of last sublot on the second machine. That is $a_j = p'_{1j}x^*_{1j}$, and $b_j = p'_{2j}x^*_{q_j j}$. The expression for the makespan of any sequence σ, $C_{\max}(\sigma, X)$, in equation (3.7), depends upon the job sequence which minimises the second term. It can be rewritten as

$$\max_{1\le j^*\le n}\left\{\sum_{j=1}^{j^*}(s'_{1j}+a_j)+\sum_{j=j^*}^{n}(s'_{2j}+b_j)\right\}+\sum_{j=1}^{n}g_j$$

The last term in the above expression is independent of the job sequence. While the first term is the expression for the makespan of the n job two machine flow shop model with $p^*_{1j} = (s'_{1j} + a_j)$ processing time on the first machine, and $p^*_{2j} = (s'_{2j} + b_j)$ processing time on the second machine. This problem is solved using a Johnson SPT(1)-LPT(2) sequence, as explained in Chapter 2. Hence we have the following theorem.

Theorem 3.4 *For the $F2|q_j,\ s_{ij},\ t_j|C_{\max}$ model, an optimal job order is an SPT(1)-LPT(2) sequence with respect to p^*_{1j}, and p^*_{2j} processing times in the first and second machine respectively where $p^*_{1j} = s_{1j} + T_j - t_j + (p_{1j} + t_j)x^*_{1j}$, $p^*_{2j} = s_{2j} + (p_{2j} + t_j)x^*_{q_j j}$, and $x^*_{kj} = \frac{(\frac{p_{2j}+t_j}{p_{1j}+t_j})^{k-1}}{\sum_{v=0}^{q_j-1}\left(\frac{p_{2j}+t_j}{p_{1j}+t_j}\right)^v}$.*

We have reduced the problem to one without lot streaming, where the optimal sublot sizes are given by equation (3.9), and the jobs are ordered according to Johnson's SPT(1)-LPT(2) rule over p^*_{1j}, and p^*_{2j}, as given above. We have shown that the sublot sizes of the jobs are not influenced by the job sequence or the setup times, and can be determined solely on the relationship between the processing times on the machines and the transportation times. On the other hand, the sequencing decision involve both the setup and transportation times as well as the relative order of the processing times of the different jobs. We have also made clear the reduction of this extended model

to one without lot streaming, setup or transportation times. We should also note, from equation (3.7), that if the sum of the setup and processing time on the second machine are big enough, then sequencing and lot streaming have no effect on the makespan. A Johnson sequence is found in $O(n \log n)$ time. Thus, as the sublot sizes can be calculated in $O(n)$ time, a solution to the model considered in this section can be found in $O(n \log n)$ time. There is no need to solve the linear program given in Vickson (1995) to find the optimal solutions for this model.

3.5 Concluding Remarks

We presented in this chapter a network representation and path approach for the two machine flow shop with transportation and setup times to minimise the makespan of n jobs. We show that the sublot size decision can be taken independently of the job sequencing decision, and give a closed formula for the optimal sublot sizes. Furthermore, the model with setup times and transportation times can be reduced to simpler models. We believe this gives more insight into the result given by (Vickson 1995).

Note that when no transportation times and setups are involved the dispatching rule will be to fix the sublot size to the ones given by equation (3.9) and to schedule jobs in SPT(1)-LPT(2) order with respect to the processing times of the first sublot on the first machine and last sublot on the second machine (Theorem 3.4 with $T_j = t_j = s_{1j} = s_{2j} = 0$). We will use this property extensively in the next chapter to analyse a model where the number of sublots for each job (q_j) is not predetermined beforehand. A possible further extension of these results, would be to apply dominant machine analysis. We believe the results can be extended to two dominant machines. We explore the case of identical jobs in an m-machine flow shop in the Chapter 4.

Chapter 4

Extended Models

4.1 Introduction

In this chapter we present two new models for lot streaming that are extensions on those analysed in the literature. Throughout this chapter we focus our attention on the flow shop environment, with the makespan objective function.

The first model relaxes the assumption that numbers of sublots are predetermined. We analyse the two-machine flow shop with lot streaming to minimise the makespan when sublots are not predetermined. We use the strong results of Chapter 3 for the two machine flow shop. In Section 4.2 we present a heuristic to allocate a fixed number of sublots between jobs. First we explain how to add one extra sublot to a job in a schedule in an optimal way. We then focus on a critical path analysis to show that when adding optimaly an extra sublot there is no need to calculate the entire schedule (sublot sizes and sequence of jobs) when starting with a Johnson sequence. We use the results of this analysis to devise an $O(Qn^2)$ heuristic to allocate Q sublots between n jobs.

The second model is a lot streaming m-machine flow shop with n identical jobs to minimise the makespan. In Section 4.3 we use a dominant machine analysis similar to Glass & Potts (1998) to find optimal sublot sizes for this

model. We prove that the model reduces to a simpler single job lot streaming model.

4.2 Lot streaming with sublot allocation

The model we work with in this section is set in a two-machine flow shop environment, where the objective is to minimise the makespan of n jobs with consistent sublots sizes. We will use the same notation introduced in Chapter 3, where q_j is the number of sublots for job j, and $\underline{q} = (q_1, \ldots, q_n)$ is the vector of sublot sizes that holds the information on the number of sublots each job has.

An instance of this problem is determined by

- p_{ij} the processing times of the job j $(1 \leq j \leq n)$ on machine i, $(i = 1, 2)$
- and by Q the sublots to allocate between the n jobs.

Hence, in this case q_j is not specified as in the other models presented in this thesis. Most models in the literature consider $\underline{q}$ as a technological constraint particular to each job, and hence q_j is given beforehand for $j = 1, \ldots, n$. We relax this assumption, so $\underline{q}$ is not predetermined, q_j is not fixed for each job, rather we have a fixed amount Q of sublots to allocate between the jobs. We are thinking of a situation where $\underline{q}$ is a technological constraint for the whole run, rather than a particular one for each job. A solution to this problem not only requires deciding the sequencing of jobs and the sublot sizes, but also the number of sublots each jobs should have.

If C_{ikj} is the completion time on machine i $(i = 1, 2)$ of the k-th sublot, $1 \leq k \leq q_j$, of job j, p_{ij} the processing time of job j on machine i, x_{kj} the size of sublot k of job j $(1 \leq j \leq n)$, and σ a sequence of the n jobs to schedule. Then, $C_{2q_{\sigma(n)}\sigma(n)}$ is the completion on the last machine of the last sublot of the last job in sequence σ, and the objective is to

$$\begin{aligned} \min_{\sigma} \quad & C_{2q_{\sigma(n)}\sigma(n)} \\ \text{subject to} \quad & \\ & C_{11\sigma(1)} - p_{1\sigma(1)} x_{1\sigma(1)} \geq 0, \end{aligned}$$

$$
\begin{aligned}
C_{ik\sigma(j)} - C_{i-1k\sigma(j)} - p_{i\sigma(j)}x_{k\sigma(j)} &\geq 0, \quad 1 \leq k \leq q_j, 2 \leq i \leq m, 1 \leq j \leq n \\
C_{ik\sigma(j)} - C_{ik-1\sigma(j)} - p_{i\sigma(j)}x_{k\sigma(j)} &\geq 0, \quad 2 \leq k \leq q_j, 1 \leq i \leq m, 1 \leq j \leq n \\
\textstyle\sum_{k=1}^{q_j} x_{k\sigma(j)} &= 1, \quad 1 \leq j \leq n \\
q_j &\geq 1 \quad 1 \leq j \leq n \\
\textstyle\sum_{j=1}^{n} q_j &= Q \\
\sigma \text{ a permutation} &\quad \text{of the } n \text{ jobs to schedule} \\
x_{kj}, C_{ikj} &\geq 0, \quad \text{for } 1 \leq i \leq m,\ 1 \leq k \leq q_j,\ 1 \leq j \leq n.
\end{aligned}
$$

That is, we wish to find the number of sublots q_j each job j should have such that $\sum_{j=1}^{n} q_j = Q$, together with a sequence σ (a permutation of the n jobs) and sublot sizes x_{kj} $(1 \leq k \leq q_j)$ for each job $j = 1, \ldots n$, in order to minimize the makespan of the n jobs to schedule. The first three constraints are the sublot processing time and machine capacity constraints (similar to the ones explained in section 3.2 for the single job problem). The fourth constraint ensures that x_{kj} is the proportion of the job belonging to sublot k of job j. As $q_j \geq 1$, and $\sum_{j=1}^{n} q_j = Q$ we have that $Q = n$ corresponds to a model without lot streaming (i.e. $q_j = 1$ for $j = 1, \ldots, n$), the simple $F2 \mid\mid C_{\max}$ model.

In this section we present a heuristic to allocate the Q sublots between the n jobs, which gives the optimal sequencing of jobs and sublot sizes for that given allocation. The main idea is to add one sublot at a time efficiently until the Q sublots are assigned. The procedure will become clear in the following subsections.

4.2.1 Preliminary results

From Theorem 3.2 we know that the sublot size decision can be tackle separately from the sequencing decision for a given sublot allocation. This is true regardless of the sublot allocation. Thus, it is always better to use the optimal sublot sizes for any given sequence. Hence, throughout this section we always use the optimal sublot sizes. As this model does not include setups or transportation times the optimal sublot sizes for $j = 1, \ldots, n$ and

$k = 1, \ldots, q_j$ are given by

$$x_{jk} = \frac{(r_j)^{k-1}}{1 + r_j + \ldots + (r_j)^{q_j - 1}}, \text{ where } r_j = \frac{p_{2j}}{p_{1j}}. \tag{4.1}$$

Then, the problem of minimising the makespan reduces to one of how to sequence jobs, and to which jobs to assign the sublots.

From Theorem 3.4 we know that for a given sublot allocation, an SPT(1)-LPT(2) sequence with respect to the processing times of the first sublot on the first machine, and the last sublot on the second machine minimises the makespan. We will denote the processing time of the first sublot of job j on the first machine by a_j, and the processing time of the last sublot of job j on the second machine by b_j. Their values vary according to the number of sublot for job j. Hence, when we need to make this explicit we will write $a_j(q_j)$, and $b_j(q_j)$. We have that,

$$a_j(q_j) = \frac{p_{1j}}{1 + r_j + \ldots + (r_j)^{q_j - 1}} \text{ and,} \tag{4.2}$$

$$b_j(q_j) = \frac{p_{2j}\,(r_j)^{q_j - 1}}{1 + r_j + \ldots + (r_j)^{q_j - 1}}, \text{ where } r_j = \frac{p_{2j}}{p_{1j}}. \tag{4.3}$$

That is, once an allocation of sublots $\underline{q} = (q_1, \ldots, q_n)$ is fixed, the optimal sequence can be obtained in $O(n \log n)$ time by ordering the n jobs according to the SPT(1)-LPT(2) rule over $a_j(q_j)$ and $b_j(q_j)$. To order any sequence of jobs we need only to focus on $a_j(q_j)$ and $b_j(q_j)$.

As we focus on the job sequence and sublot allocation alone, the following notation is also useful:

- $\sigma(\underline{q})$ a sequence σ with sublot allocation $\underline{q}$.
- $C_{\max}(\sigma(\underline{q}))$ the makespan of a sequence σ with sublot allocation $\underline{q}$.
- $C^*_{\max}(j, q_j)$ the makespan of job j with q_j sublots, using the optimal sublot sizes given in (4.1).
- $P(h; \sigma(\underline{q}))$ the length of the path that goes down to machine 2 on job in the h-th place under sequence σ in the network representing the sequencing problem.

Note that $C^*_{\max}(j, q_j) = a_j(q_j) + p_{2j} = p_{1j} + b_j(q_j)$. The use of $P(h; \sigma(\underline{q}))$ will become apparent in the following sections, as our analysis will focus on the path length to determine job sequences.

4.2.2 Assigning one sublot efficiently

First we analyse which job gives the best reduction in makespan if we add an extra sublot. Let e_j be an q dimensional vector with zeros as all its entries except the j-th which is equal to 1. Then, $\underline{q} + e_j$ is the operation of adding one extra sublot to job j under sublot allocation $\underline{q}$. We want to find $j \in \{1, \ldots, n\}$ such that $\underline{q} + e_j$ yields the greatest decrease in makespan.

The 3 job example of section 3.1 without lot streaming with $\underline{q} = (1, 1, 1)$ has an optimal schedule with $C_{\max} = 22$. Suppose $Q = 4$, that we have on extra sublot to add. The sublot allocation given in that example was $\underline{q}' = \underline{q} + e_2 = (1, 2, 1)$, and the makespan for the job sequence $\sigma = (2, 1, 3)$ (job J_2 sequenced before J_1, and J_1 before J_3) is 20. The same makespan is obtained when using the optimal sublot sizes given by equation(4.1) for J_2. Any other sublot allocation $\underline{q} + e_1$, or $\underline{q} + e_3$ will not have reduced the makespan.

An inefficient way of verify this is to try the 3 possible allocations, and for each allocation the 6 different sequences. Alternatively as explained before we could calculate the optimal sequence, SPT(1)-LPT(2) over a_j and b_j, and compare then the makespan under each of the n different allocations ($\underline{q} + q_j$, $j = 1, \ldots, n$), choosing the one that yields the best value.

Finding the best job to which to add a new sublot takes $O(n^2 \log n)$ time, as there are n jobs, and it takes $O(n \log n)$ time to find the optimal sequence and sublot sizes for a given sublot allocation. In this section we show how we can actually reduce this to $O(n^2)$ time by focusing on the critical jobs and updating and storing the information about the critical paths. We use the ideas of (Glass et al. 1994) for critical paths. Recall we introduced this concepts briefly in Section 3.3. We assume that we start with an SPT(1)-LPT(2) sequence before we add the extra sublot. From Chapter 2 we know

that in an SPT(1)-LPT(2) order a job j belongs to *set 1* if $p_{1j} < p_{2j}$, and belongs to *set 2* if $p_{1j} \geq p_{2j}$.

The following result states the changes in the job ordering an SPT(1)-LPT(2) sequence must suffer to remain optimal if we add an extra sublot to a job.

Lemma 4.1 *Consider an SPT(1)-LPT(2) sequence σ, indexed for convenience so that jobs go from 1 to n (i.e. $\sigma = (1, \ldots, n)$), optimal for the predetermined sublot sizes $\underline{q} = (q_1, \ldots, q_n)$. Sequence $\sigma' = (\pi(1), \ldots, \pi(n))$ will be optimal for the new sublot allocation $\underline{q} + e_t$ if σ' has the following properties:*

1) If job t belongs to set 1 under σ (i.e. $a_t < b_t$), then

σ' satisfies:

$$\pi(h) = \begin{cases} h & \text{for } h \leq l-1, \\ t & \text{when } h = l, \\ h-1 & \text{for } l+1 \leq h \leq t, \\ h & \text{for } h \geq t+1, \end{cases} \tag{4.4}$$

where job l is a job in σ such that $a_l = \min_{1 \leq j \leq t}\{a_j \mid a_j > a_t(q_t + 1)\}$.

2) If job t belongs to set 2 under σ (i.e. $a_t \geq b_t$), then

σ' satisfies:

$$\pi(h) = \begin{cases} h & \text{for } h \leq t-1, \\ h+1 & \text{for } t \leq h \leq l-1, \\ t & \text{when } h = l, \\ h & \text{for } h \geq l+1, \end{cases} \tag{4.5}$$

where job l is a job in σ such that $b_l = \min_{t \leq j \leq n}\{b_j \mid b_j > b_t(q_t + 1)\}$.

Proof: We know that $\sigma(\underline{q})$ is an optimal schedule, and from equations (4.2) and (4.3))we have

$$\frac{b_j(q_j)}{a_j(q_j)} = (r_j)^{q_j}.$$

Let t be the job to which another sublot is to be added. If job t belongs to *set 1* in sequence σ then $r_t > 1$. Hence, $1 < \frac{b_t(q_t)}{a_t(q_t)} = r_t^{q_t} < r_t^{q_t+1} = \frac{b_t(q_t+1)}{a_t(q_t+1)}$, which

means that job t also belongs to *set 1* in σ'. No other job has been altered and they therefore remain in the same set in σ' as they were in σ. Moreover, the order of the jobs in *set 2* remains unchanged. Note that $a_t(q_t + 1)$ is the new processing time of the first sublot of job t after operation $\underline{q} + e_t$. The SPT(1)-LPT(2) rule dictates that jobs in *set 1* are to be scheduled in increasing order of a_j. As $a_t(q_t + 1) < a_t(q_t)$ it follows that the order of jobs after job t in σ remain in the same position under σ'. Hence $\pi(h) = h$ for $h > t$. From $a_l = \min_{1 \leq j \leq t}\{a_j \mid a_t(q_t + 1) < a_j\}$ we conclude that job t must be in the l-th position in σ', and hence jobs l to $t - 1$ move one position up in σ' relative to σ. This is precisely what is described by equation (4.4), and roughly illustrated in Figure 4.1, *(a)*.

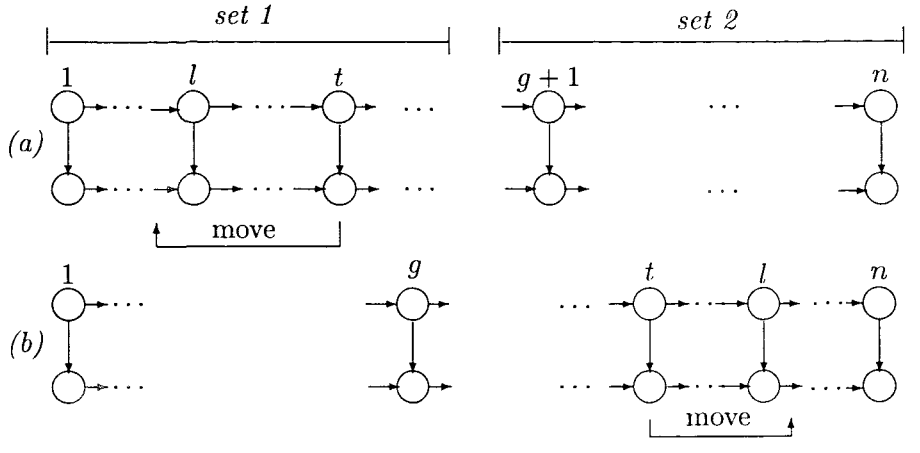

Figure 4.1: Change from sequence σ to sequence σ' in a network.

Conversely if $r_t \leq 1$, then job t belongs to *set 2*. The result described in equation (4.5), illustrated in Figure 4.1 *(b)*, follows from the above analysis, due to the reversibility of the problem (standard property of flow shops, see Pinedo (1995)).

□

This result shows that there is no need to recalculate the position of all jobs in sequence σ when we add a new sublot to a job. We merely have to move the jobs as indicated by equation (4.4) or (4.5) to obtain a new optimal

sequence σ'. Note, as well, that jobs that belong to a particular set before allocating any sublot (i.e. when $q_j = 1$ for $1 \leq j \leq n$) will remain in the same set, regardless of how many sublots are allocated to them.

Observe that the problem is reversible, that is, interchanging machines one and two, and taking the sequence in reverse order will yield the same makespan. So the following lemmas will concentrate on the cases where the critical job is in *set 2*.

Recall that we are using optimal sublot sizes for each job individually and $C^*_{\max}(j, q_j)$ is the (minimum) makespan of a job j with q_j sublots. Consider the network representation of the problem for a given schedule $\sigma(\underline{q})$. $\mathcal{P}(h; \sigma(\underline{q}))$ denotes the length of the path going down on the job positioned in the h-th place under sequence σ with $\underline{q}$ sublot allocation. If $\sigma = (1, \ldots, n)$, the length of the path going down on job h is easily calculated as:

$$\mathcal{P}(h; \sigma(\underline{q})) = \sum_{j=1}^{h-1} p_{1j} + C^*_{\max}(h, q_h) + \sum_{j=h+1}^{n} p_{2j}.$$

From Section 3.3 a path is critical if its length is equal to the makespan of the schedule, and the job on which this path goes down is considered a critical job. That is, if $C_{\max}(\sigma(\underline{q}))$ is the makespan of schedule $\sigma(\underline{q})$, and k is a critical job, then

$$C_{\max}(\sigma(\underline{q})) = \mathcal{P}(k; \sigma(\underline{q})).$$

In the next lemma we state which jobs will not reduce the total makespan if we allocate an extra sublot to any of them.

Lemma 4.2 *Consider a Johnson sequence σ, indexed for convenience so that jobs go from 1 to n (i.e. $\sigma = (1, \ldots, n)$), optimal for the predetermined sublot sizes $\underline{q} = (q_1, \ldots, q_n)$. Suppose this schedule has a critical job, say k, belonging to* set 2 *under σ. The makespan of schedule $\sigma(\underline{q})$, $C_{\max}(\sigma(\underline{q}))$, remains unchanged by adding one additional sublot to any job in* set 1 *or any job t in* set 2 *for which $b_t < b_k$.*

Proof: We know that $\sigma(\underline{q})$ is an optimal schedule. As k is a critical job we have that

$$C_{\max}(\sigma(\underline{q})) = \max_{1 \le h \le n} \mathcal{P}(h; \sigma(\underline{q})) = P(k; \sigma(\underline{q})).$$

Add a new sublot to job t (i.e. let q_t be now $q_t + 1$), and calculate the new optimal sequence, call it $\sigma' = (\pi(1), \pi(2), \ldots, \pi(n))$. Its makespan is $C_{\max}(\sigma'(\underline{q} + e_t))$.

Let us now evaluate the path length going down on job k under σ'. Suppose t is in *set 1*. Since k is in *set 2*, $k > t$, from Lemma 4.1 we have that $\mathcal{P}(k; \sigma'(\underline{q} + e_t)) = \sum_{j=1}^{t} p_{1\pi(j)} + \sum_{j=t+1}^{k-1} p_{1j} + C^*_{\max}(k, q_k) + \sum_{j=k+1}^{n} p_{2j}$, but $\{\pi(j) \mid j = 1, \ldots, t\} = \{j \mid j = 1, \ldots, t\}$. Thus

$$\mathcal{P}(k; \sigma'(\underline{q} + e_t)) = \mathcal{P}(k; \sigma(\underline{q})) = C_{\max}(\sigma(\underline{q})). \tag{4.6}$$

If, on the other hand, t is in *set 2*, and $b_t < b_k$, that is if $t > k$ under σ, then from (4.5) of Lemma 4.1, $\mathcal{P}(k; \sigma'(\underline{q} + e_t)) = \sum_{j=1}^{k-1} p_{1j} + C^*_{\max}(k, q_k) + \sum_{j=k+1}^{t-1} p_{2j} + \sum_{j=t}^{n} p_{2\pi(j)}$, but $\{\pi(j) \mid j = t, \ldots, n\} = \{j \mid j = t, \ldots, n\}$, and thus equation (4.6) holds in this case as well.

Since allowing one extra sublot cannot increase the optimal makespan, (as even in the case where $\sigma' = \sigma$ the length of all the paths remain the same and the path going down in t is reduced), the optimal makespan in both of the cases considered in this lemma, from equation (4.6) satisfies

$$C_{\max}(\sigma(\underline{q})) \ge C_{\max}(\sigma'(\underline{q} + e_t)) \ge \mathcal{P}(k; \sigma'(\underline{q} + e_t)) = \mathcal{P}(k, \sigma'(\underline{q})) = C_{\max}(\sigma(\underline{q})).$$

Thus equality holds and the makespan remains unchanged as claimed.

□

In the following Lemma we continue focusing our attention on Johnson sequences, this time with only one critical job, but we now give the necessary conditions a job must satisfy in order to reduce the total makespan of the sequence if we were to add one more sublot to the job.

Lemma 4.3 *Consider a Johnson sequence σ, indexed for convenience so that jobs go from 1 to n (i.e. $\sigma = (1, \ldots, n)$), optimal for the predetermined sublot sizes $\underline{q} = (q_1, \ldots, q_n)$. Suppose this schedule has only one critical job, say k, belonging to* set 2 *under σ. The makespan of schedule σ, $C_{\max}(\sigma)$, will be reduced if we add an extra sublot to job t belonging to set 2 and $b_t(q_t) \geq b_k(q_k) > b_t(q_t + 1)$.*

Proof: Let $\sigma(\underline{q})$ be an optimal schedule. As k is a critical job we have that

$$C_{\max}(\sigma(\underline{q})) = \max_{1 \leq h \leq n} \mathcal{P}(h; \sigma(\underline{q})), = P(k; \sigma(\underline{q})).$$

Moreover $\mathcal{P}(k; \sigma(\underline{q})) > \mathcal{P}(h; \sigma(\underline{q}))$ for $h \neq k$ because k is the only critical job.

Add a new sublot to a job t satisfying the conditions stated in the lemma. Calculate the new optimal sequence under this new sublot allocation $\underline{q} + e_t$, and call it $\sigma' = (\pi(1), \pi(2), \ldots, \pi(n))$. Its makespan is $C_{\max}(\sigma'(\underline{q} + e_t))$.

Let job l in σ be such that $b_l = \min_{t \leq j \leq n}\{b_j(q_j) \mid b_j(q_j) > b_t(q_t + 1)\}$. From Lemma 4.1 we know that $\pi(j)$ the job in position j in sequence σ' is given by

$$\pi(h) = \begin{cases} h & \text{for } h \leq t-1, \\ h+1 & \text{for } t \leq h \leq l-1, \\ t & \text{when } h = l, \\ h & \text{for } h \geq l+1, \end{cases}$$

See Figure 4.2 for a diagramatic illustration of how σ' is constructed.

For $h \leq t-1$ and $h \geq l+1$ we have that

$$\mathcal{P}(h; \sigma'(\underline{q} + e_t)) = \sum_{j=1}^{h-1} p_{1\pi(j)} + C^*_{\max}(\pi(h), q_{\pi(h)}) + \sum_{j=h+1}^{n} p_{2\pi(j)}$$

$$= \sum_{j=1}^{h-1} p_{1j} + C^*_{\max}(h, q_h) + \sum_{j=h+1}^{n} p_{2j} = \mathcal{P}(h; \sigma(\underline{q})),$$

so that $C_{\max}(\sigma(\underline{q})) > \mathcal{P}(h; s'(\underline{q} + e_t))$ for $h \leq t-1$ and $h \geq l+1$.

For $t \leq h \leq l-1$

$$\mathcal{P}(h; \sigma'(\underline{q}+e_t)) = \sum_{j=1}^{t-1} p_{1\pi(j)} + \sum_{j=t}^{h-1} p_{1\pi(j)} + C^*_{\max}(\pi(h), q_{\pi(h)}) + \sum_{j=h+1}^{l} p_{2\pi(j)} + \sum_{j=l+1}^{n} p_{2\pi(j)}$$

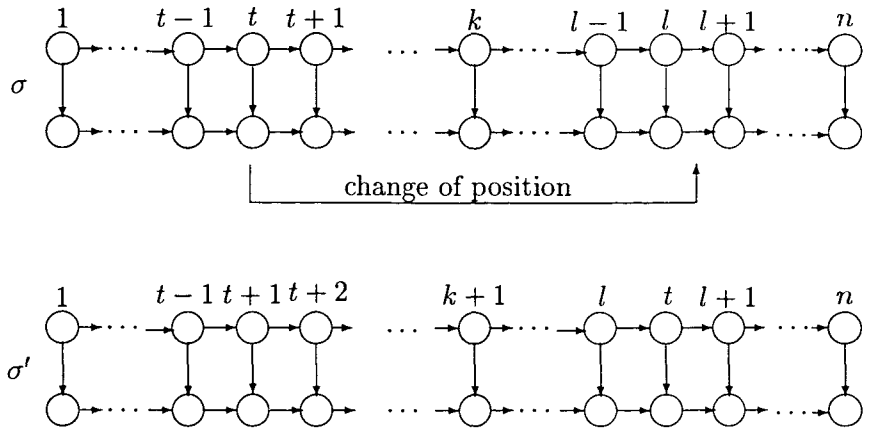

Figure 4.2: σ and σ'

$$= \sum_{j=1}^{t-1} p_{1j} + \sum_{j=t}^{h-1} p_{1j+1} + C^*_{\max}(h+1, q_{h+1}) + \sum_{j=h+1}^{l-1} p_{2j+1} + p_{2t} + \sum_{j=l+1}^{n} p_{2j}.$$

Thus,

$$\mathcal{P}(h; \sigma'(\underline{q} + e_t)) = \sum_{j=1}^{h} p_{1j} - p_{1t} + C^*_{\max}(h+1, q_{h+1}) + \sum_{j=h+2}^{n} p_{2j} + p_{2t} \quad (4.7)$$

$$= \mathcal{P}(h+1; \sigma(\underline{q})) - p_{1t} + p_{2t}.$$

Since t is in *set 2*, $p_{1t} \geq p_{2t}$, and hence, $\mathcal{P}(h+1; \sigma(\underline{q})) > \mathcal{P}(h; \sigma'(\underline{q} + e_t))$. However, $C_{\max}(\sigma(\underline{q})) \geq \mathcal{P}(h+1; \sigma(\underline{q}))$, with equality holding only for $h = k-1$. Hence, $C_{\max}(\sigma(\underline{q})) > \mathcal{P}(h; \sigma'(\underline{q} + e_t))$ for $t \leq h \leq l-1$.

Now for $h = l$ we have that

$$\mathcal{P}(l; \sigma'(\underline{q} + e_t)) = \sum_{j=1}^{t-1} p_{1\pi(j)} + \sum_{j=t}^{l-1} p_{1\pi(j)} + C^*_{\max}(\pi(l), q_{\pi(l)}) + \sum_{j=l+1}^{n} p_{2\pi(j)}$$

$$= \sum_{j=1}^{t-1} p_{1j} + C^*_{\max}(t, q_t + 1) + \sum_{j=t+1}^{l} p_{1j} + \sum_{j=l+1}^{n} p_{2j}$$

$$= \mathcal{P}(l; \sigma(\underline{q})) - C^*_{\max}(l, q_l) + C^*_{\max}(t, q_t + 1) - p_{1t} + p_{1l}$$

$$= \mathcal{P}(l; \sigma(\underline{q})) - b_l(q_l) + b_t(q_t + 1).$$

But from the choice of l; $b_l(q_l) > b_t(q_t+1)$, so that $\mathcal{P}(l;\sigma(\underline{q})) > \mathcal{P}(l;\sigma'(\underline{q}+e_t))$ and $C_{\max}(\sigma(\underline{q})) > \mathcal{P}(l;\sigma'(\underline{q}+e_t))$. Hence $C_{\max}(\sigma(\underline{q})) > \mathcal{P}(h;\sigma'(\underline{q}+e_t))$ for $1 \leq h \leq n$ (all h). Thus $C_{\max}(\sigma(\underline{q})) > C_{\max}(\sigma'(\underline{q}+e_t))$.

□

We conclude from this Lemma that for those optimal sequences with only one critical job, in order for the makespan to be reduced when adding a new sublot to a job, either the critical job needs to change its position in the sequence, or the new sublot must be added to the critical job. There are situations where the reduction in makespan is better by adding a sublot to a non-critical job which makes the position of the critical job change within the sequence, rather than to a critical job.

Corollary 4.1 *Under the conditions of Lemma 4.3, if $\sigma(\underline{q})$ is the original schedule and σ' is the optimal sequence for the new sublot allocation $\underline{q}+e_t$, the path lengths in the two schedules are related as follows:*

$$\mathcal{P}(h;\sigma'(\underline{q}+e_t)) = \begin{cases} \mathcal{P}(h;\sigma(\underline{q})) & \text{for } h \leq t-1 \\ \mathcal{P}(h+1;\sigma(\underline{q})) - p_{1t} + p_{2t} & \text{for } t \leq h \leq l-1, \\ \mathcal{P}(l;\sigma(\underline{q})) - b_l(q_l) + b_t(q_t+1) & \text{for } h = l, \\ \mathcal{P}(h;\sigma(\underline{q})) & \text{for } h \geq l+1, \end{cases} \tag{4.8}$$

where job l is a job in σ such that $b_l = \min_{t\leq j\leq n}\{b_j \mid b_j > b_t(q_t+1)\}$.

Note that $l+1-t$ paths have changed length, namely those corresponding to jobs that have changed their position from sequence σ to sequence σ'. Another way of calculating $\mathcal{P}(h;\sigma'(\underline{q}+e_t))$, for $t \leq h \leq l-1$, from equation 4.7 in the proof of Lemma 4.3, is $\mathcal{P}(h,\sigma'(\underline{q}+e_t)) = \sum_{j=1}^{t-1} p_{1t} + C^*_{\max}(t, q_t + 1) + \sum_{j=t+1}^{l} p_{1j} + \sum_{j=l+1}^{n} p_{2j}$.

We will denote by $\mid \underline{q} \mid$ the sum of the entries of vector $\underline{q}$. Thus, $\mid \underline{q} \mid = \sum_{j=1}^{n} q_j$ is the total amount of sublots the n jobs have. The next theorem integrates the previous lemmas to state the condition for the decrease in the total makespan to be maximum when adding a sublot to a job. This forms the basis for the heuristic we present later in the next section.

Theorem 4.1 *Consider the case where we have assigned* $|\ \underline{q}\ |$ *sublots between* n *jobs, with* $\underline{q} = (q_1, \ldots q_n)$ *the sublot allocation, and* σ *the optimal Johnson sequence. Suppose the schedule* $\sigma(\underline{q})$ *has only one critical job, say* k*, belonging to set 2. Let* σ' *be the optimal Johnson sequence for the sublot allocation* $\underline{q}+e_t$*. Adding a sublot to job* t^* *will give the maximum decrease in the makespan if*

$$C_{\max}(\sigma'(\underline{q}+e_{t^*})) = \min_t \left\{ C_{\max}(\sigma'(\underline{q}+e_t)) \mid t \text{ in set } 2\ , b_t(q_t) \geq b_k > b_t(q_t+1) \right\}.$$

Proof: From Lemma 4.2 we know that if k is the only critical job in the sequence and it belongs to *set 2*, then a necessary condition for the makespan to be reduced is that t is a job in *set 2* under σ, and $t \leq k$. From Lemma 4.3 we know that a sufficient condition is for $b_t(q_t) \geq b_k > b_t(q_t + 1)$. So that we are only considering those jobs which give a decrease in the makespan and we are choosing the one that gives the greatest decrease.

□

Observe that the problem is reversible, that is, interchanging machines one and two, and taking the sequence in reverse order will yield the same makespan. Hence the properties stated in Lemmas 4.2 and 4.3 have their counterpart proposition of *set 2* in the reversed problem which is *set 1* in the original problem.

Thus, we have that if we have a Johnson sequence σ, optimal for a given sublot allocation $\underline{q}$, with only one critical job, say k belonging to *set 1*. If we choose a job t in σ belonging to *set 1* such that $a_t(q_t + 1) \leq a_k < a_t(q_t)$ then, the makespan will be reduced if we add a new sublot to t. Let σ' be the new ordering of the jobs as given in Lemma 4.1 when adding a new sublot to job t. The length of the path going down on the job in the h-th position $(1 \leq h \leq n)$ in sequence σ' under allocation $\underline{q} + e_t$ can be calculated from

the length of the path in sequence σ under allocation $\underline{q}$ as follows:

$$\mathcal{P}(h;\sigma'(\underline{q}+e_t)) = \begin{cases} \mathcal{P}(h;\sigma(\underline{q})) & \text{for } h \leq l-1, \\ \mathcal{P}(h-1;\sigma(\underline{q})) + p_{1t} - p_{2t} & \text{for } l+1 \leq h \leq t, \\ \mathcal{P}(l;\sigma(\underline{q})) + a_l(q_l) - a_t(q_t+1) & \text{for } h = l, \\ \mathcal{P}(h;\sigma(\underline{q})) & \text{for } h \geq t+1, \end{cases} \tag{4.9}$$

where job l is a job in σ such that $a_l = \min_{1\leq j\leq t}\{a_j \mid a_t(q_t+1) < a_j\}$

Summarising, we have found that jobs in a Johnson sequence do not change from *set 1* to *set 2*, or from *set 2* to *set 1*. We have stated how the job sequence changes when adding an extra sublot to a job (Lemma 4.1). We have shown that a critical job needs to change place for the makespan to be reduced, and give conditions for this to happen (Lemma 4.3). We have also stated how the path lengths change when re-sequencing, (Corollary 4.1 equation (4.8), and equation (4.9)). The heuristic we propose to assign Q sublots uses all of the above results, and is given in the following section.

4.2.3 Sublot Allocation Heuristic

In this section we propose a heuristic to allocate Q sublots between n jobs. The idea of this heuristic is to add one sublot at a time, until we have assigned the total amount of sublots Q between the n jobs. At each step we will choose the job which yields the maximum decrease in the makespan. That is, if σ' is the optimal sequence for sublot allocation $\underline{q}+e_t$, we will add an extra sublot to a job t^* such that

$$C_{\max}(\sigma'(\underline{q}+e_{t^*})) = \min_t \left\{ C_{\max}(\sigma'(\underline{q}+e_t)) \mid \right\}.$$

We are looking for a maximum decrease in makespan at each step.

In fact, we do not need to look at all jobs t, but rather at a reduced set, as presented in Theorem 4.1 in the previous section. If there is a critical job k in *set 2* then we need to look at those jobs t belonging to *set 2* such that $b_t(q_t) \geq b_k > b_t(q_t+1)$, and if there is a critical job k in *set 1* then we focus on those jobs t belonging to *set 1* such that $a_t(q_t+1) \leq a_k < a_t(q_t)$.

We will use three sequences, one to store the optimal sequence under allocation $\underline{q}$, and two more to find the job which yields the maximum decrease in makespan. The makespan will be calculated from the maximum path length. Path lengths will be updated using equation (4.8) of Corollary 4.1, and equation (4.9) stated in the previous section.

For a given sublot allocation $\underline{q}$, we know that a Johnson sequence, say σ, is optimal. Its makespan, $C_{\max}(\sigma(\underline{q}))$, is calculated as the maximum path length. If

$$\mathcal{P}(1;\sigma(\underline{q})) = C_1 + \sum_{j=2}^{n} p_{2j}, \tag{4.10}$$

where $C_h = p_{1h} + p_{2h}$ $(1 \leq h \leq n)$, the length of the paths can be recursively calculated for $h = 2, \ldots n$, by

$$\mathcal{P}(h;\sigma(\underline{q})) = \mathcal{P}(h-1;\sigma(\underline{q})) - C_{h-1} + C_h + p_{1h-1} - p_{2h}. \tag{4.11}$$

Using all of the above we can state our heuristic as follows.

Maximum Marginal Decrease Heuristic

Input Data:

Processing times p_{1j}, p_{2j} of each job j $(1 \leq j \leq n)$, and Q the total number of sublots to be allocated between the jobs.

Variables:

σ, σ', σ^* (*each stores an ordering of n jobs*). $\mathcal{P}(h;\sigma)$, $\mathcal{P}(h;\sigma')$, and $\mathcal{P}(h;\sigma^*)$ for $h = 1, \ldots, n$ (*each stores the path length going down on the job in the h-th position in the sequence*). Set $\mathcal{K}$ (*of critical jobs*). g (*the number of jobs in set 1*). sb (*extra sublots allocated*). $C_{\max}(\sigma)$, $C_{\max}(\sigma')$, and $C_{\max}(\sigma^*)$, temporary variables to store makespan of schedules. $\underline{q}$ (*vector of sublot sizes*).

Step 1: (Initialisation)

Initial allocation: $\underline{q} = (1, 1, \ldots, 1)$.

Let σ^* be a Johnson SPT(1)-LPT(2) sequence over the processing times $a_j(1)$ and $b_j(1)$).

Calculate $\mathcal{P}(1;\sigma^*)$ following equation 4.10.

Calculate recursively $\mathcal{P}(h;\sigma^*)$ from $h = 2$ following equation (4.11).

$C_{\max}(\sigma^*) = \max_{1\le h\le n}\left\{\mathcal{P}(h;\sigma^*)\right\}$; $\mathcal{K} = \{k \mid \mathcal{P}(k;\sigma^*) = C_{\max}(\sigma^*)\}$.

$g =\mid \{j \mid p_{1j} < p_{2j}\} \mid$ (number of jobs in the first set).

Step 2: (Allocating Sublots One at a Time)

DO For sb = 0 until sb = Q − n (while there are sublots to allocate)

Begin

If there is a job $j \in \mathcal{K}$ such that $j \le g$ (*a critical job in set 1*)

then

$k = \min\{j \mid j \in \mathcal{K}\}$ and go to step 3,

else (*the critical job(s) is(are) in set 2*)

$k = \max\{j \mid j \in \mathcal{K}\}$, and go to step 4.

Step 3:

(Calculating t^* in *set 1* which gives the best saving in makespan)

$\sigma = \sigma^*$, $C_{\max}(\sigma) = C_{\max}(\sigma^*)$.

For $t = g$ down to k do

Begin

If $a_t(q_t + 1) \le a_k(q_k) < a_t(q_t)\}$ then { Calculate $C_{\max}(\underline{q} + e_t)$ }

Begin

Let $\sigma' = \sigma^*$

Find l such that $a_l = \min_{1\le j\le t}\{a_j \mid a_t(q_t + 1) < a_j\}$

Reorder the job in σ' according to 4.4.

Calculate $\mathcal{P}(h;\sigma')$ from $\mathcal{P}(h;\sigma)$ following 4.9.

Calculate $C_{\max}(\sigma') = \max\left\{\mathcal{P}(h{:}\,\sigma')\right\}$

End

If $C_{\max}(\sigma) > C_{\max}(\sigma')$ then

Begin

$t^* = t$; $\sigma = \sigma'$, $C_{\max}(\sigma) = C_{\max}(\sigma')$

$\mathcal{P}(h;\sigma) = \mathcal{P}(h;\sigma')$ for $h = 1, \ldots . n$

End

End

Go to Step 5.

Step 4: (Calculating t^* in *set 2*)

$\sigma = \sigma^*$, $C_{\max}(\sigma) = C_{\max}(\sigma^*)$

For $t = g + 1$ to k do

Begin

If $b_t(q_t + 1) < b_k(q_k) \leq b_t(q_t)\}$ then { Calculate $C_{\max}(\underline{q} + e_t)$ }

Begin

Let $\sigma' = \sigma^*$

Find l such that $b_l = \min_{t \leq j \leq n}\{b_j \mid b_j > b_t(q_t + 1)\}$.

Reorder the job in σ' according to 4.5.

Calculate $\mathcal{P}(h; \sigma')$ from $\mathcal{P}(h; \sigma)$ following 4.8.

Calculate $C_{\max}(\sigma') = \max\left\{\mathcal{P}(h; \sigma')\right\}$

End

If $C_{\max}(\sigma) > C_{\max}(\sigma')$ then

Begin

$t^* = t$; $\sigma = \sigma'$, $C_{\max}(\sigma) = C_{\max}(\sigma')$

$\mathcal{P}(h; \sigma) = \mathcal{P}(h; \sigma')$ for $h = 1, \ldots, n$

End

End (For $t = g + 1$ to k)

Follow to step 5.

Step 5: (Updating sequence, and path lengths)

$sb = sb + 1$ (one more sublot added)

$\underline{q} = \underline{q} + e_{t^*}$ (updated sublot allocation)

$\sigma^* = \sigma$; $C_{\max}(\sigma^*) = C_{\max}(\sigma)$

$\mathcal{P}(h; \sigma^*) = \mathcal{P}(h; \sigma)$ for $h = 1, \ldots, n$

Find set $\mathcal{K}$ of critical jobs, $\mathcal{K} = \{k' \mid \mathcal{P}(k; \sigma^*) = C_{\max}(\sigma^*)\}$.

End of Do loop (back to beginning of step 2).

Allocated the Q sublots

4.2.4 Complexity of Heuristic

If n is the number of jobs to be scheduled and Q is the total amount of sublots to allocate, then this heuristic runs in $O(Q\ n^2)$ time, as is shown by the following arguments. Let us first analyse Step 1. A Johnson sequence can be calculated in $O(n\ log\ n)$ time. $\mathcal{P}(\sigma; h)$ can be obtained by equation 4.11 in $O(n)$ time. The set of critical jobs $\mathcal{K}$ and g are both found in $O(n)$ time. Thus we have that Step 1 has a complexity of $O(n \log n)$. Let us now analyse the remaining steps. Note that Step 2 is visited $Q - n$ times (until all the sublots have been allocated). k is found in $O(n)$ time. Step 3 and 4 find job t^*, to which a sublot should be added. We calculate $C_{\max}(q_t + e_t)$ in the worst case n times (if all jobs belong to *set 1* and k is the first job, or if all jobs are in *set 2* and k is the last job). Finding l can be done in $O(n)$ time. Reordering the job in sequence σ' according to equation (4.4) or (4.5) can be done in $O(n)$ time. Calculating $\mathcal{P}(h; \sigma')$ from $\mathcal{P}(h; \sigma)$ by equations (4.8) or (4.9) takes $O(n)$ time. Since $C_{\max}(\sigma')$ is computed in $O(n)$ time, we obtain $C_{\max}(q_t + e_t)$ in $O(n)$ time. Hence calculating t^* requires $O(n^2)$ time, either in Step 3 or 4. We find job t^* a total of $Q - n$ times (Step 2), but $Q > n$, so that $O((Q-n)n^2)$ is $O(Q\ n^2)$. This yields the desired complexity of $O(Q\ n^2)$ for the heuristic.

4.2.5 Counterexample of optimality

We now show why this heuristic is not an exact algorithm (one that does not always obtain a global optima, see Chapter 5). We give a counterexample that it does not find the optimal allocation of sublots.

Consider the instance with 26 jobs and processing times given in the following table.

Instance Data						
Job j	1	2	3	...	25	26
p_{1j}	48	8	8	...	8	8
p_{2j}	24	7	7	...	7	7

For an initial situation where no extra sublot has been assigned to any job, $\underline{q} = \underline{q}_0 = (1, 1, \ldots, 1)$, as shown in the following table. $\sigma(j)$ is the j-th job in sequence σ, where σ is the SPT(1)-LPT(2) sequence that minimises the makespan for the given sublot allocation $\underline{q}$. $\mathcal{P}(j)$ is the length of the path going down on job $\sigma(j)$ under sublot allocation $\underline{q}$. The critical path goes down in job 26, and the makespan for this sequence and sublot allocation is $C_{\max}(\sigma(\underline{q}_0)) = 255$.

$\underline{q} = \underline{q}_0$								
$\sigma(j)$	1	2	3	4	5	...	25	26
j	1	2	3	4	5	...	25	26
$\underline{q}$	1	1	1	1	1	...	1	1
p_{1j}	48	8	8	8	8	...	8	8
p_{2j}	24	7	7	7	7	...	7	7
$\mathcal{P}(j)$	247	231	232	233	234	...	254	**255**

If we were to add an extra sublot to job 1 then $b_1(q_1) = b_1(2) = x_{12}p_{21} = 8$. The makespan of the n jobs with this sublot allocation $\underline{q}_0 + e_1$ would remain the same as job 1 would not move from its position. No improvement will be attained by this assignment. On the other hand, if we were to add a new sublot to any job j with $j = 2, \ldots, 25$ the processing time of the second (last) sublot in the second machine would be $b_j(q_j) = b_j(2) = x_{j2}p_{2j} = \frac{49}{15}$. The heuristic adds an extra sublot to job 26; with Step 4, reordering according to equation (4.5), this job will go at the end of schedule, as $7 > \frac{49}{15}$. There is a reduction of $3\frac{11}{15}$ to the makespan of the n jobs for this new allocation of sublots $\underline{q}_0 + e_j$. In fact $C_{\max}(\sigma'(\underline{q}_0 + e_j)) = 254$ for $j = 2, \ldots, 25$. The following table shows the situation after we added a sublot to job $j = 26$. When the job has more than one sublot, we have made explicit the processing time of the sublots, and in what order they are processed in the first and second machine.

$\underline{q} = \underline{q}_0 + e_{26}$							
$\sigma(j)$	1	2	3	...	24	25	26
j	1	2	3	...	24	25	26
$\underline{q}$	1	1	1	...	1	1	2
p_{1j}	48	8	8	...	8	8	$4\frac{4}{15}\|3\frac{11}{15}\|$
p_{2j}	24	7	7	...	7	7	$\|3\frac{11}{15}\|3\frac{4}{15}$
$\mathcal{P}(j)$	247	231	232	...	253	**254**	$251\ \frac{4}{15}$

Let us now add another extra sublot. Following the same argument as before adding a new sublot to job 1 will yield no improvement in the makespan of n jobs ($b_1(q_1 = 2) = 8$). However, $C_{\max}(\sigma(\underline{q}_0 + e_{26} + e_j)) = 253$ for $j = 2, \ldots, 25$. As illustrated in the following table, where the heuristic, will choose to add a new sublot to job 25.

$\underline{q} = \underline{q}_0 + e_{26} + e_{25}$							
$\sigma(j)$	1	2	3	...	24	25	26
j	1	2	3	...	24	25	26
$\underline{q}$	1	1	1	...	1	2	2
p_{1j}	48	8	8	...	8	$4\frac{4}{15}\|3\frac{11}{15}\|$	$4\frac{4}{15}\|3\frac{11}{15}\|$
p_{2j}	24	7	7	...	7	$\|3\frac{11}{15}\|3\frac{4}{15}$	$\|3\frac{11}{15}\|3\frac{4}{15}$
$\mathcal{P}(j)$	247	231	232	...	**253**	$250\ \frac{4}{15}$	$251\ \frac{4}{15}$

So far two extra sublots have been added, one to job 25, and another to job 26. If we had added this two extra sublot to job 1 instead, we would have the situation illustrated in the following table.

$\underline{q} = \underline{q}_0 + 2e_1$							
$\sigma(j)$	1	2	3	4	...	25	26
j	2	3	4	5	...	26	1
$\underline{q}$	1	1	1	1	...	1	3
p_{1j}	8	8	8	8	...	8	$\frac{384}{14}\|\frac{192}{14}\|\frac{96}{14}$
p_{2j}	7	7	7	7	...	7	$\|\frac{192}{14}\|\frac{96}{14}\|\frac{48}{14}$
$\mathcal{P}(j)$	190	191	192	193	...	214	$\mathbf{251\ \frac{3}{7}}$

The makespan of the n jobs with this sublot allocation $\underline{q}_0 + 2e_1$ will be $251\frac{3}{7}$. Summarising $C^{\sigma}_{\max}(\underline{q}_0 + 2e_1) < C^{\sigma}_{\max}(\underline{q}_0 + e_{25} + e_{26})$, we have found a different sublot allocation from the one found by the heuristic, which yields a better makespan. In fact, $C^{\sigma}_{\max}(\underline{q}_0 + 2e_1) < C^{\sigma}_{\max}(\underline{q}_0 + e_j + e_k)$ for $2 \leq j, k \leq 26$. Hence, our heuristic does not guarantee finding optimal solutions.

4.3 Identical Jobs Multiple Machines

The model we work with in this section is set in an m-machine flow shop with n "identical" jobs to minimise the makespan of the n jobs. The jobs are identical in the sense that the processing time of any job, on each machine, is the same, and all jobs have the same sublots. Our aim is to find optimal sublot sizes for the jobs when $m > 3$.

An instance of the problem requires

- p_i the processing time of the job on machine i, and
- q the number of sublots a job has.

Let x_{kj} be the k-th sublot of job j $(1 \leq k \leq q,\ 1 \leq j \leq n)$, and C_{ikj} the completion time on machine i $(1 \leq i \leq m)$ of sublot k of job j. Then, we want to find

$$
\begin{aligned}
\min \quad & C_{mqn} \\
\text{subject to} \quad & \\
& C_{111} - p_1 x_{11} \geq 0, \\
& C_{ikj} - C_{i-1kj} - p_i x_{kj} \geq 0, \quad 1 \leq k \leq q, 2 \leq i \leq m, 1 \leq j \leq n \\
& C_{ikj} - C_{ik-1j} - p_i x_{kj} \geq 0, \quad 2 \leq k \leq q, 1 \leq i \leq m, 1 \leq j \leq n \\
& \textstyle\sum_{k=1}^{q} x_{kj} = 1, \\
& x_{kj}, C_{ikj} \geq 0, \qquad \text{for } 1 \leq k \leq q, \text{ and } 1 \leq i \leq m.
\end{aligned}
$$

Any solution to this problem can be viewed as a solution to the corresponding, less restricted, single-job nq-sublot lotstreaming problem, by scaling all the sublot sizes down by a factor of $\frac{1}{n}$. It is know from Glass & Potts (1998) that, for a single job no critical path goes through a dominated machine. Moreover, (Glass & Potts 1998) provide a Relaxation Algorithm for

reducing the problem to one with only dominant machines and intermediate lag times. The resultant machine processing times $\rho_1, \ldots, \rho_{m'}$, and the intermediary lag times $l_1, \ldots, l_{m'-1}$, corresponding to processing on intermediate non-dominant machines, satisfy the dominance condition:

$$\frac{\rho_1 + l_1}{l_1 + \rho_2} < \frac{\rho_2 + l_2}{l_2 + \rho_3} < \ldots < \frac{\rho_{m'-1} + l_{m'-1}}{l_{m'-1} + \rho_{m'}}. \tag{4.12}$$

The network representing the reduced problem is illustrated in Figure 4.3. Each oval node (i, k) for $k = 1, \ldots, q$ represents the processing of sublot x_k on dominant machine i $(1 \leq i \leq m')$, and each rectangular node $[i, k]$ represents the lag time that elapses between the end of the completion time of sublot x_k on dominant machine i and the start of the processing of sublot x_k on dominant machine $i+1$ $(i = 1, \ldots, m'-1)$. The makespan of a schedule is given by the length of a longest path joining node $(1, 1)$ to (m', nq).

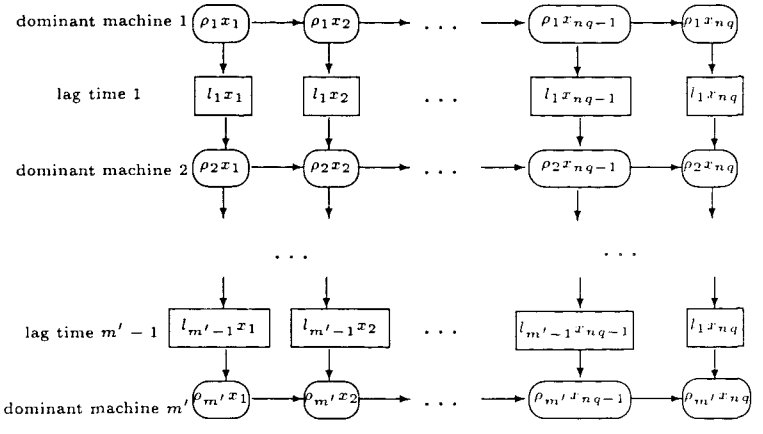

Figure 4.3: Network after applying the Relaxation Algorithm

As the sublots are the same for each job, $x_{(j-1)q+h} = x_{jq+h}$ for $j = 1, \ldots, n-1$ and $h = 1, \ldots, q$. Let $\mathcal{P}(j_1, h_1; j_2, h_2; \ldots; j_{m'-1}, h_{m'-1})$ denote a path that joins node $(1, 1)$ to (m', nq), going down from dominant machine i to machine $i+1$ in job j_i sublot h_i, for $i = 1, \ldots, m'$. Note that $1 \leq h_i \leq q$, for $i = 1, \ldots, m'$, and that $1 \leq j_1 \leq j_2 \leq \ldots \leq j_{m'-1} \leq m'$. Further, let $L(j_1, h_1; j_2, h_2; \ldots; j_{m'-1}, h_{m'-1})$ denote the length of this path. We call a path *critical* if its length is the longest of all paths.

Lemma 4.4 *A feasible solution* $\underline{x} = (x_1, \ldots, x_{nq})$ *has a critical path* $\mathcal{P}(j_1, h_1; j_2, h_2; \ldots; j_{m'-1}, h_{m'-1})$ *with* $h_1 \leq h_2 \leq h_3 \leq \ldots \leq h_{m'-1}$.

Proof: Suppose that the lemma does not hold. Then, there is a feasible solution $\underline{x} = (x_1, \ldots, x_{nq})$ with a critical path $P = \mathcal{P}(j_1, h_1; j_2, h_2; \ldots; j_{m'-1}, h_{m'-1})$ for which $h_a > h_b$ for some pair of machines $a < b$. Since P is a critical path, and $a < b$, then P cannot go down to machine b before a, and thus $j_a \leq j_b$. Moreover, since $h_a > h_b$ using the same argument we have that jobs j_a and j_b cannot be the same. Hence $j_a < j_b$.

We can always choose a and b with $a < b$ such that $h_k \leq h_b$ for $1 \leq k \leq a-1$ and $h_a \leq h_k$ for $b+1 \leq k \leq q$. Thus, $1 \leq h_{a-1} \leq h_b < h_a \leq h_{b+1} \leq q$, and therefore the segments $\sigma_a = (a, q(j_a-1)+h_b)-(a, q(j_a-1)+h_a)-(a+1, q(j_a-1)+h_a)$, and $\sigma_b = (b, q(j_b-1)+h_b)-(b+1, q(j_b-1)+h_b)-(b+1, q(j_b-1)+h_a)$ are on the critical path P.

Now consider the set of paths of the form $P'(h'_a, h'_b) = \mathcal{P}(j_1, h_1; j_2, h_2; \ldots, j_a, h'_a; \ldots; j_b, h'_b; \ldots; j_{m'-1}, h_{m'-1})$ which differ from the given path P only in values h'_a and h'_b, with $h_b \leq h'_a \leq h_a$ and $h_b \leq h'_b \leq h_a$. Suppose that no path $P'(h'_a, h'_b)$ with $h'_a \leq h'_b$ is critical. Consider paths $P'(h_b, h_b)$ and $P'(h_a, h_a)$. They are not critical and must therefore be shorter than path $P = P'(h_a, h_b)$. Thus segment σ_a must be longer than segment $(a, (j_a - 1)q + h_b) - (a+1, (j_a - 1)q + h_b) - (a+1, (j_a - 1)h_a)$, and similarly σ_b is longer than $(b, (j_b - 1)q + h_b) - (b, (j_b - 1)q + h_a) - (b+1, (j_b - 1)q + h_a)$. Thus, if

$$X = \sum_{k=h_{b+1}+1}^{h_{a-1}-1} x_k,$$

then we have the following inequalities:

$$x_{h_{b+1}}(l_a + \rho_{a+1}) + X\rho_{a+1} < x_{h_a}(l_a + \rho_a) + X\rho_a, \text{ and,}$$

$$x_{h_{b+1}}(l_b + \rho_{b+1}) + X\rho_{b+1} > x_{h_a}(l_b + \rho_b) + X\rho_b.$$

Adding Xl_a on both sides of the first equation, Xl_b to both sides of the second equation, yields the following inequality

$$\frac{\rho_b + l_b}{l_b + \rho_{b+1}} < \frac{x_{h_{b+1}} + X}{x_{h_a} + X} < \frac{\rho_a + l_a}{l_a + \rho_{a+1}},$$

contradicting Equation (4.12).

Thus, some path $P'(h'_a, h'_b)$ with $h'_a \leq h'_b$ is critical. Take h'_a the smallest for which $P'(h'_a, h_b)$ is critical, and h'_b the largest for which $P'(h_a, h'_b)$ is critical. Then $P'(h'_a, h'_b)$ is a critical path. Call it P'. Since $h'_a \geq h'_b$ and $h_b \geq h_k$ for $k = 1, \ldots, a-1$, $h_k \leq h'_a$ for $k = 1, \ldots, a-1$. Similarly $h'_b \leq h_k$ for $k = b+1, \ldots, q$. As $h'_a < h_a$ we cannot have additional inequalities of the form $h'_a > h_k$ for $k \geq a+1$. In the same way, since $h'_b < h_b$, there are no additional inequalities of the form $h_k < h'_b$ in P' that were not in P. Moreover, the inequality $h_a > h_b$ has now been replaced by $h'_a \leq h'_b$. We conclude that P' has at least one less inequality, between the pairs of h_k's $(k = 1, \ldots, q)$ than the original path P. By showing that such a pair h'_a, h'_b always exists the proof follows by induction. Hence, $h_1 \leq h_2 \leq \ldots \leq h_{m'-1}$.

□

Before proceeding we need to make a small observation.

Lemma 4.5 *At least one machine with largest processing time is a dominant machine.*

Proof: Dominant machines are those which are not reduced to a lag by the Relaxation Algorithm. Now observe that a machine is reduced to a lag by the Relaxation Algorithm only if at some stage of reduction its processing time, say ρ_r, satisfies an inequality of the form $(l_{r-1}+\rho_r)(\rho_r+l_r) \leq (\rho_{r-1}+l_{r-1})(l_r+\rho_{r+1})$, where ρ_{r-1} and ρ_{r+1} are also machine processing times. Since ρ_r is the largest machine processing time, the inequality implies that $\rho_{r-1} = \rho_r = \rho_{r-1}$, and hence there remains a dominant machine with maximum processing time after the reduction.

□

Lemma 4.6 *A feasible solution $\underline{x} = (x_1, \ldots, x_{qn})$ has a critical path, of the form described in Lemma 4.4, which passes through jobs $2, \ldots, n-1$ on a machine which has the largest processing time among all machines.*

Proof: Given a feasible solution $\underline{x} = (x_1, \ldots, x_{qn})$, for which there is a critical path $P = \mathcal{P}(j_1, h_1; j_2, h_2; \ldots; j_m, h_{m'-1})$ where $j_1 \leq j_2 \leq \ldots \leq j_{m'-1}$, and $h_1 \leq h_2 \leq \ldots \leq h_{m'-1}$, from Lemma 4.4. Let L denote the length of path P.

$$L = \{(j_1 - 1)\rho_1 + \sum_{k=1}^{h_1} x_k\rho_1 + l_1 x_{h_1}\} + \{(j_2 - j_1)\rho_2 + \sum_{k=h_1}^{h_2} x_k\rho_2 + l_2 x_{h_2}\} + \ldots$$

$$+\{(j_r - j_{r-1})\rho_r + \sum_{k=h_{r-1}}^{h_r} x_k\rho_r + l_r x_{h_r}\} + \ldots + \{(j_{m'-1} - j_{m'-2})\rho_{m'-1} +$$

$$\sum_{k=h_{m'-2}}^{h_{m'-1}} x_k\rho_{m'-1} + l_{m'-1}x_{h_{m'-1}}\} + \{(n - j_{m'-1})\rho_{m'} + \sum_{k=h_{m'-1}}^{n} x_k\rho_{m'}\}.$$

Now take a dominant machine r with the largest processing time of any dominant machine, that is,

$$\rho_r = \max_{1 \leq i \leq m'} \rho_i.$$

Consider path $P' = \mathcal{P}(1, h_1; 1, h_2; \ldots; 1, h_{r-1}; n, h_r; \ldots; n, h_{m'-1})$. It passes through jobs 2 to $n-1$ on dominant machine r. Let L' denote the length of path P'.

$$L' = \{\sum_{k=1}^{h_1} x_k\rho_1 + l_1 x_{h_1} + \ldots + \sum_{k=h_{r-1}}^{h_r} x_k\rho_r + l_r x_{h_r}\} + (n-1)\rho_r + \{\sum_{k=h_r}^{h_{r+1}} x_k\rho_{r+1} +$$

$$l_{r+1}x_{h_{r+1}} + \ldots + \sum_{k=h_{m'-2}}^{h_{m'-1}} x_k\rho_{m'-1} + l_{m'-1}x_{h_{m'-1}} + \sum_{k=h_{m'-1}}^{n} x_k\rho_{m'}\}.$$

Thus, $L' - L = (n-1)\rho_r - [(j_1 - 1)\rho_1 + (j_2 - j_1)\rho_2 + \ldots (j_r - j_{r-1})\rho_r + \ldots + (j_{m'-1} - j_{m'-2})\rho_{m'-1} - (n - j_{m'-1})\rho_{m'}]$. Extending the notation by setting $j_0 = 1$, and $j_{m'} = n$, this difference becomes

$$(j_{m'} - j_0)\rho_r - \sum_{i=1}^{m'} (j_i - j_{i-1})\rho_i.$$

Thus by substituting $\sum_{i=1}^{m'}(j_i - j_{i-1})$ for $(j_{m'} - j_0)$ we have

$$L' - L = \sum_{i=1}^{m'}(j_i - j_{i-1})(\rho_r - \rho_i).$$

From the choice of r this is non negative, and hence P' is a critical path of the form described in Lemma 4.4. Note that, from Lemma 4.5, ρ_r is the largest processing time for any machine, not just the dominant ones.

□

Theorem 4.2 *For the multiple identical job problem with the same sublot sizes for each job, the makespan is minimised by the sublot sizes which minimise the makespan of a single job alone. A critical path is given by* $L(1, h_1; 1, h_2; \ldots;\ 1, h_{r-1}; n, h_r; \ldots; n, h_{m'-1})$. *where* $\rho_r = \max_{1\le i\le m} p_i$, *and its makespan is* $C^*_{\max} + (n-1)\max_{1\le i\le m} p_i$. *where* $C^*_{\max}$ *is the makespan of a single job using the optimal sublot sizes.*

Proof: From Lemma 4.6 we know that the critical path length is of the form

$$(n-1)\rho_r + \sum_{i=0}^{m'-1}\sum_{k=h_i}^{h_{i+1}} x_k \rho_{i-1} + \sum_{i=1}^{m'-1} l_i x_{h_i}.$$

where $h_0 = 1$, $h'_m = n$ and $\rho_r = p_{r'}$ is the largest processing time of a single job on any one of the machines. The last two terms of this expression define the makespan of a single job with sublot siz $\underline{x} = (x_1, \ldots, x_q)$. Thus, if for every job in the schedule we use the sublot sizes which minimise $C_{\max}$ for a single job, the makespan of the schedule is minimised. The makespan of such a schedule is $(n-1)\max p_i + C^*_{\max}$.

□

We have thus reduced the problem of lot sizing n identical jobs on m machines to that of a single job on m' machines.

4.4 Concluding remarks

In this chapter we have presented two models that are extensions on those analysed in the literature for lot streaming on the flow shop environment.

First we worked with a model where the numbers of sublots are not predetermined beforehand. We used the results on the two machine flow shop model given in Chapter 3 to focus our attention on the job sequence and sublot allocation. We used an analysis on the critical paths to develop a $O(Qn^2)$ heuristic to allocate sublots. We gave a counterexample of optimality for the heuristic. As further work it is desirable to evaluate the heuristic with computational tests, and analyse the worst case performance.

Finally we have also studied a model with n identical jobs, and applied a dominant machine analysis to find optimal sublot sizes. Not only that, but we have reduced the sublot size decision to that of a single-job. We hope to have illustrated the advantages of using network representation, critical path and dominant machine analysis for lot streaming problems.

Part II

Batching

Chapter 5

Combinatorial Optimisation and Batching Machine Scheduling

5.1 Introduction

Batching machine problems can be considered an extension on classical scheduling models where jobs are not processed simultaneously. As explained in Chapter 2 a batching machine is able to process several jobs at a time. Our analysis approach for these problems is different from the one followed in Chapters 3 and 4. In this chapter we explain the standard methodology to tackle combinatorial optimisation problems, including scheduling problems; presenting methods that look for exact, as well as, approximate solutions. Section 5.2 explains the basis of a combinatorial optimisation problem, section 5.3 presents methods that aim at giving exact solutions to the problems, whereas section 5.4 deals with approximate solutions. Finally, we discuss in section 5.5 the batching machine model in the context of scheduling, and introduce the specific model (subection 5.5.1) we work with in the remaining chapters (Chapters 6, and 7).

5.2 Combinatorial Optimisation

A combinatorial optimisation problem arises when we have a discrete number of choices to make; the problem lies in choosing an optimal one. The optimality of this choice relates to a quantitative measure for each possible choice. The 'combinatorial' term refers to the fact that the choice is made over a set which is at worst a countably infinite set. We formalise this idea.

Definition 5.1 *An instance of a combinatorial optimisation problem is a pair* $(\mathcal{I}, f)$ *where* $\mathcal{I}$ *is the set of feasible solutions and the cost function* f *is a mapping* $f : \mathcal{I} \rightarrow \Re$. *The problem is to find a globally optimal solution i.e.* $i^* \in \mathcal{I}$ *such that* $f(i^*) \leq f(i)$ *for all* $i \in \mathcal{I}$.

Note that the optimal cost is $f^* = f(i^*)$, and a set $I^* = \{i \in \mathcal{I} | f(i) = f^*\}$ will be a set of optimal solutions. If the problem is one of minimisation, then the optimal cost is $\min_{i \in \mathcal{I}} f(i)$; likewise if it is a maximisation problem, then the optimal cost is $\max_{i \in \mathcal{I}} f(i)$. The instance $(\mathcal{I}, f)$ is usually not given explicitly, that is, by a listing of all solutions, and their cost. Instead, it is common to have a compact data representation of an instance and a polynomial-time algorithm to verify the feasibility of the solution (i.e. if i belongs to $\mathcal{I}$), and its cost (i.e. $f(i)$). The size of the data representation, that is, the number of bits needed to store it in a computer, is taken to be the size of the problem instance (as explained in section 2.2). For detailed introductions to the area of combinatorial optimisation we refer to Papadimitriou & Steiglitz (1982) or Schrijver (1986).

Any scheduling problem can be expressed as a combinatorial optimisation problem. The choices are all feasible schedules (the feasibility may rely, for example, on precedence constrains or release dates). The quantitative measure is any of the objective functions (dependent on the completion time of the jobs) introduced in Chapter 2. For example, the two-machine flow shop minimising the makespan is a combinatorial optimisation problem where $f = C_{\max}$, and $\mathcal{I}$ is composed of all possible schedules (a sequence of the jobs that defines the processing order on the two machines). Each solution can be

expressed as a permutation π of the jobs to schedule. A particular instance to the problem is the following:

job	J_1	J_2	J_3	J_4
p_{1j}	3	1	3	3
p_{2j}	3	2	1	3

There are 24 different solutions (i.e. 24 different permutations of 4 jobs), and therefore the size of $\mathcal{I}$ is 24. The optimal solutions are: $i_1^* : J_2, J_1, J_4, J_3$, and $i_2^* : J_2, J_4, J_1, J_3$. Hence, $\mathcal{I}^* = \{i_1^*, i_2^*\}$, with $f^* = f(i_1^*) = f(i_2^*) = 11$.

A simple, but inefficient way of solving combinatorial optimisation problems would be to look at each element in $\mathcal{I}$, and choose the one with optimal cost. This method is referred to as *complete enumeration.* In our example, a complete enumeration method would need to evaluate $n!$ solutions for an instance with n jobs. As this is not a practical method, other implicit enumeration methods have been devised. There are two widely used such methods to solve combinatorial optimisation problems, branch and bound, and dynamic programming, which we will explain below.

5.3 Exact Solutions

5.3.1 Branch and Bound

The main idea of a branch and bound approach (Land & Doig 1960) is to partition the feasible solutions $\mathcal{I}$ into disjoint sets, each belonging to a *branch* of a search tree, and then *bound* the cost of each set so as to guide the search to as optimal solution. This can be represented as a tree graph, see Figure 5.1, where the root node is $\mathcal{I}$, and the disjoint sets are at a lower level, each one represented by a node joined to the root by edges. At each node, a decision is made either to partition it further (keep branching form the node) and/or calculate a bound with the aim of discarding it. For example, a lower bound on the cost of the solutions in a set of a minimisation problem can be calculated, and if it is bigger than the best objective value found so far

it can be discarded, as no solution emanating it could be optimal. At each successive level of the tree, the sets of solutions represented by the nodes become smaller, until at the final level each node represents a single solution. Eventually the method terminates with all nodes discarded.

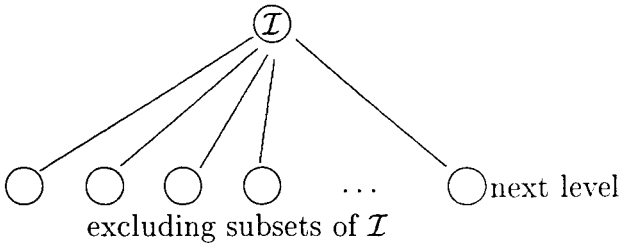

Figure 5.1: Search tree in a branch and bound procedure

We avoid evaluating every single feasible solution by discarding sets of them corresponding to a node of the search tree. Sometimes an upper bound is also used which can be used in conjunction with the best objective value found so far to prune the tree. Thus, the speed of the algorithm relies on how efficient, and effective the lower bound is, in conjunction with the branching and upper bounding schemes. It is also popular to use other elimination criteria in the form of problem-specific dominance rules to prune the tree even further.

A possible branching scheme for the flow shop example we presented previously is to decide, at level i, the job at position i in the job sequence. Hence, we would have 4 branches emanating form the root node (i.e. any of the 4 jobs could be at the beginning of the sequence). For each node in the first level we would have 3 branches (i.e. we have 3 choices for the second position, as one job is already in the first position). In the same way, for each node in the second level we would have 2 branches, and finally just one branch from each node in the third level. At the end (if no node was eliminated, by the bounding schemes or dominance rules) we would have 24 end-nodes (the 24 solutions). It is easy to see the importance of the lower bound calculations. The stronger the lower bound, the fewer nodes that will need examination. However, strong lower bounds usually require more

computing time than weaker bounds. The size of the tree is also affected by the quality of the upper bound, the other elimination criteria (domimance rules), and of course by the branching scheme.

Techniques to obtain lower bounds are generally based on solving some relaxation of the problem, for example a Lagrangian relaxation method. This method was first introduced by [Held & Karp (1970), Held & Karp (1971)]; a good introduction to the subject can be found in Fisher (1981), and Fisher (1985). It is important to note that the application of this method in scheduling does not need to rely on integer linear programming formulations as pointed out by Van de Velde (1991). Probably the most famous branch and bound algorithm in this domain, capable of solving problems with as many as ten thousand jobs, uses bounds obtained by relaxing a specific feature of the problem (Carlier 1982). In batching machine scheduling, we can also relax a specific feature of the problem, the assumption on the restricted batch size, and develop an the algorithm that solves the unrestricted case to obtain lower bounds, see Chapter 6.

5.3.2 Dynamic Programming

Dynamic programming is a general optimisation technique particularly well suited to problems requiring a sequence of inter-related decisions. At each stage in the decision sequence, sub-problems of increasing complexity are solved recursively using knowledge obtained from sub-problems (decisions) solved at previous stages. Solving these sub-problems usually involves minimizing or maximising some measure of value. Each new sub-problem (decision) solved determines more features of the optimal solution, until, at the end, the problem is solved, and the optimal cost of the final decision (problem solution) is obtained.

The efficiency of the method relies on the ability of breaking a problem into stages or sub-problems that could be solved efficently. This entails formulating the solution of the problems in terms of dynamic programming recurrence relations (recursive formula for the sub-problems), identifying the

appropriate state and stage variables, initialization conditions, and characterizing the optimal value function as explained by Dreyfus & Law (1977). We present a dynamic program to solve a scheduling problem in section 5.5.2.

Similarly to branch and bound, dynamic programming does not evaluate all possible solutions, as it discards some by solving them as part of a sub-problem. It is generally much more efficient than complete enumeration, but still not in general a polynomial time algorithm. Hence, as neither branch and bound nor dynamic programming can be applied efficiently (within polynomial time) to most combinatorial optimisation problems, it is of importance to use methods that would yield approximate solutions in a reasonable amount of time. We explain a class of such methods below.

5.4 Local Search Heuristics

Most combinatorial problems, and practical scheduling problems are NP-hard (see list given by Garey & Johnson (1979), pg. 236-244, and (Papadimitriou 1994)). As explained in section 2.2, it is generally believed that these problems cannot be solved optimaly within polynomially bounded computational times. Thus, it is of great interest to be able to give (near-optimal) approximate solutions in a reasonable amount of time. To achieve these, some sort of heuristic is usually employed.

A heuristic is an algorithm (step by step set of instructions) which aims to provide near-optimal solutions. This however does not imply that there is any guarantee of feasibility, or optimality. In fact, there is usually no way of finding how close to optimality a particular feasible solution is. A distinction can be made between two broad classes of approximation algorithms: constructive and local search methods. In this thesis, we are interested in the latter, even though a simple constructive heuristic is usually used as a starting solution (usually an ordering of the jobs, though it might be more complicated). Constructive heuristics are used as starting solutions in chapters 6 and 7.

The main idea that distinguishes a local search heuristic, is that of a *neighbourhood.* For the solutions of a combinatorial optimisation problem, a neighbourhood structure is imposed which determines which solutions are close to each other (neighbours). Once the neighbourhood structure is designed, it is a matter of searching for better neighbours. Ideally this search will take a small amount of time (polynomially bounded), as to make the local search efficient. It is a *local* search as we are just searching locally within the neighbourhood. We now formalise the idea.

Definition 5.2 *Let $(\mathcal{I}, f)$, where $\mathcal{I}$ is the set of feasible solutions and f is the cost function, be an instance of a combinatorial optimisation problem. A neighbourhood function is a mapping $\mathcal{N} : \mathcal{I} \rightarrow 2^{\mathcal{I}}$, which defines for each solution $i \in \mathcal{I}$ a set $\mathcal{N}(i) \subset \mathcal{I}$ of solutions that are close to i.*

The set $\mathcal{N}(i)$ is the neighbourhood of solution i, and each $j \in \mathcal{N}(i)$ is a neighbour of i. For a minimisation problem, a solution $\hat{i}$ is a local optimal if $f(\hat{i}) \leq f(j)$ for all $j \in \mathcal{N}(\hat{i})$. If $\mathcal{I}'$ is the set of locally optimal solutions, then $\mathcal{N}$ is *exact* if $\mathcal{I}' \subset \mathcal{I}^*$, where $\mathcal{I}^*$ is the set of optimal solutions (as defined previously).

The simplest local search heuristic is *descent*, also know as iterative local improvement. However, as not to confuse it with iterated local search, which is explained in subsection 5.4.1, throughout this thesis we will refer to it as descent. This heuristic starts with some initial solution (obtained form a constructive heuristic, or chosen randomly), and then searches its neighbourhood for a solution with a better value. If such a better solution is found, it replaces the current solution and the search continues. Otherwise, the descent heuristic returns the current solution, which is locally optimum. The descent heuristic can apply either *first improve*, in which the current solution is replaced by the first cost-improving solution found in the neighbourhood search, or *best improve*, in which the current solution is replaced by the best solution in its neighbourhood (i.e. usually after examining all the neighbours).

The are some important comments to make about employing local search heuristics to combinatorial optimisation problems:
1) *Related Areas:* Most heuristics have been developed on analogies with processes in nature, relating it to other disciplines such as statistical physics, biological evolution and neurophysiology. Well known examples are simulated annealing, genetic algorithms, and neural networks, which we explain in more detail below.
2) *Theoretical:* Recently some local search algorithms have been mathematically modeled, yielding theoretical results on their performance, and the development of a complexity theory of local search, see (Johnson, Papadimitriou & Yannakakis 1988).
3) *Applicability:* The increased power of computational resources and data structures have enable local search heuristics to be competitive, solving big instances of problems. The flexibility and ease of implementation is also an important factor for employing this type of heuristics.

Any local search heuristic requires a specification of neighbouring solutions, a cost function, and an efficient method for exploring the neighbourhood. However, specifying these, is no guarantee of good quality solutions. In fact, experience shows that applying a simple local search usually does not yield high quality solutions. A simple procedure to improve on the quality is to start at different (usually randomly generated) starting solutions and run the local search on each, taking the best one as the final solution. This is referred to as a *multi-start* local search. Another idea that uses several runs and combines different neighbourhoods is to take the local optimum from one neighbourhood and apply a search in a different neighbourhood, repeating several times this procedure across different neighbourhoods. This is often called *multilevel* local search. Another idea which capitalizes on multiple runs of local search, but modifying the starting solutions is Iterated Local Search.

5.4.1 Iterated Local Search

Recall that in a multi-start approach several runs using the same neighbourhood on different (randomly generated) starting solutions are applied, whereas in a multi-level approach a previous local optimum is taken as starting solution for a new search in a different neighbourhood. The idea behind Iterated Local search is to restart the search near a local minimum, rather than from a randomly generated one, or the last local minimum.

From an initial solution, we perform a local search (usually a traditional descent) to get a current solution. This current solution (local optimum) is then modified, and another local search is performed. We continue modifying the local optimum and applying another local search repeatively until some stopping criteria is met (usually time based, or depending on the number of iterations performed). Modifying the local optimum from a previous iteration is usually referred to as performing a *kick*. This kick should keep enough information of the goodness of local optimum, but be big enough as to dislodge the search from the region of attention of the previous local optimum. If we stay in the same neighbourhood, this can be viewed as performing a local search on the local optima. We can state a general Iterated Local Search as follows.

Iterated Local Search Heuristic

1) **First Local Optimum** Start form an initial solution, and apply a local search to obtain the current solution (local optimum) s_c
2) **Kick** Modify s_c to get s'.
3) **Local Search** From s' perform a local search and get s_o a local optimum.
4) **Stopping Criteria** Decide whether to:
   (a) take s_o as the current solution s_c and repeat steps 2 and 3; or
   (b) take a previous solution (usually the best solution so far) as the current solution s_c and repeat 2 and 3; or
   (c) stop (as time limit or number of iterations is met) and return the best solution found so far.

Step 4(b) is usually referred to as *backtracking*, as we allow the search to go back to a previous solution. This can be also viewed as deciding whether to leave the current solution unchanged or to replace it with the new local optimum. In Chapter 7 we explore iterated local search heuristics for a batching machine model with restricted batch size. We now give a overview of other popular local search heuristics.

5.4.2 Simulated Annealing

Simulated annealing was introduced by Kirkpatrick, Jr. & Vecchi (1983), and Černý (1985). Its name originates form the physical process of *annealing*, where the evolution of a solid as it is slowly cooled from a liquid is simulated. If this is done slowly enough, it eventually settles in a ground state arranged in a highly structured lattice which minimises the energy of the system. The algorithm that simulates the physical process (Metropolis, Rosenbluth, Rosenbluth, Teller & Teller 1953) can also be used to solve combinatorial optimisation problems, where the system states are the feasible solutions, the energy is the cost (objective function), the change of state are the neighbourhood solutions, the temperature is a control parameter, and the frozen state is the final solution. This algorithm is of interest because it has a stochastic component, which facilitates a theoretical analysis of its asymptotic convergence, and it has been applied successfully to a broad range of practical problems. We can state a simulated annealing local search as follows.

Simulated Annealing
begin
Get initial solution: s_0
Current solution $s = s_0$
Iteration $k = 0$
repeat
get a neighbour $s_j \in \mathcal{N}(s)$
calculate the difference is cost $C = f(s_j) - f(s)$

with probability p_k let the new new current solution be $s = s_j$, where

$$p_k = \begin{cases} 1 & \text{if } C \leq 0 \\ \exp(-C/t_k) & \text{if } C > 0 \end{cases}$$

$k = k + 1$

until stopping criterion is satisfied

end

Note that the probability function used to accept or reject a neighbouring solution is the negative exponential distribution with parameter $1/t_k$. The parameter t_k is called the control parameter, and plays an important role in the convergence analysis of the algorithm. For example, using the theory of finite Markov chains, the algorithm finds an optimal solution asymptotically as $t_k \to 0$ and $k \to \infty$. Other distribution functions might be chosen; the idea is that solutions corresponding to a large increase in cost have a small probability of being accepted, whereas solutions corresponding to small increases in cost have a larger probability of being accepted.

Threshold accepting algorithms can be thought of as a simplification of simulated annealing, where moves to a worst solution are accepted (similar to the way a kick allows to move to a worst solution) if its value is smaller than a threshold parameter τ_k. The value of τ_k usually decreases as the number of iterations increases, eventually becoming 0 at which point only improving moves are allowed (becoming more like a simple first improve descent algorithm). Note that in this case the acceptance criterion is deterministic, as no probability function is employed. Aarts, Korst & Van Laarhoven (1997) review threshold algorithms, and give an overview of the type of convergence results that can be stated for simulated annealing.

5.4.3 Tabu Search

Tabu search lacks the convergence properties of simulated annealing. However it makes use of memory of previously analysed solutions or moves. It

was first presented in its current form by Glover (1986). Recall that both simulated annealing and iterated local search allow non-improving moves, so there is a risk of visiting a solution again, and more generally of cycling. Tabu search aims at avoiding cycling by storing information on recently visited solutions which defines forbidden or *tabu* moves. Tabu moves effectively reduce the neighbourhood to be searched. However, memory is used at a deeper level to guide the local search, using *tabu conditions* on forbidden moves, and *aspiration criteria* allow for attractive solutions to be accepted. This is linked with the process of intensifying and diversifying the search. We can state a tabu search as follows.

Tabu Search

begin

Get an initial solution s_0.

Current solution $s = s_0$, iteration $k = 0$.

repeat

Let $\mathcal{N}(s, k)$ be the neighbourhood of s that meets the tabu conditions, or the aspiration criteria.

Choose the best neighbour s' of s in $\mathcal{N}(s, k)$.

Update tabu conditions (list) and the aspiration criteria.

$s = s'$, $k = k + 1$.

until stopping criteria is satisfied.

end

Possible stopping criteria may be: k is larger than a specified number; the number of iterations since the last improvement of the best overall solution is larger than a specified number; and $\mathcal{N}(s, k)$ is empty. The efficiency of the method seems to rely mostly on the modeling of the particular combinatorial optimisation problem. However, tabu search remains a popular local search method, and many problems have been tackled effectively with it.

5.4.4 Genetic Algorithms

Genetic algorithms are inspired by the theory of evolution in nature and population genetics, and were first introduced by Holland (1975). Genetic algorithms differ from evolution strategies (Rechenberg 1975, Schwefel 1981) which use mainly mutation and selection processes by the fact that sexual reproduction is allowed, which is often called recombination. There are many variations known in the literature of algorithms that follow these concepts. We present, as an example, *genetic local search* introduced by Mühlenbein, Gorges-Schleuter & Krämer (1988). Each individual in a population represents a solution of the combinatorial optimisation problem. The idea is to apply the *survival of the fittest principle* to the population, where a monotonic transformation of the cost function corresponds to the measure of fitness of the individual, also referred to as the *fitness score* (a term from theoretical biology). Recombination is used to combine two or more solutions to produce an offspring. Mutation is used to produce variation. In nature, it is normally caused by randomly occurring copying errors in the replication of sections of chromosomes. In a genetic algorithm it may be produced by randomly selecting a neighbour of the offspring solution. A selection procedure is used to reduce the population back to its original size. This corresponds to the survival of the fittest principle, where only a proportion of the population which best adapts survives for the next reproduction cycle. We can state a genetic algorithm in the following way.

Gentetic local search algorithm

begin

Initialise: Construct an initial population of N solutions.

Improve Use local search to replace the N solutions by N local optima.

repeat

Recombine & Mutate: Increase the population by adding n offspring solutions via recombination, and allow some mutation; the new population is now $N + n$.

Improve: Use local search to replace the n offspring solution in the

population by n local optima.

Select: Reduce the population to its original size by selecting N solutions from the current population.

until stopping criterion satisfied.

Returning the best solution.

end

A version of this algorithm is also described as a *parallel genetic algorithm* by Mühlenbein (1997). To implement a genetic algorithm, decisions must be made including the initial population, the recombination process, size and frequency of mutation as well as the selection and stopping criteria. Note that mutation is similar to the kick in iterated local search. The general class of genetic algorithms contains many other approaches, which may differ substantially form the approach presented above. Some of them have been applied successfully to various problems.

There exists other local search heuristics based in nature. For example, ant systems, which simulate the ability of ants to find the shortest path, even under changing conditions, from a food source to the nest, imitating pheromone trails left by ants.

5.4.5 Neural Networks

As its name suggest, this heuristic tries to mimic the biological neural networks inside brains. Such a network consists of linked neurons, which are able to work in parallel to find solutions. The computational counterpart for the neurons are processing units, and thanks to recent development in parallel computing a growing interested in this area has developed. In particular, its application to combinatorial optimisation problems goes back to Hopfield & Tank (1985) who analysed a neural network for the traveling salesman problem. Again, as in most local search methods based on other scientific areas, the efficiency of the method lies on the modeling of the problem through the characteristics, in this case, of a neural network.

5.4.6 Complexity of Local Search Heuristics

Johnson et al. (1988) introduce the complexity class PLS of polynomial-time local search problem, addressing the question of the worst-case behaviour of local search heuristics. Recall that any local search heuristic requires the definition of a neighbourhood. The more powerful (bigger) the neighbourhood the harder it is to explore (find a local optima), but the better quality the optima are expected to have. The most powerful neighbourhood is the exact neighbourhood (recall section 5.4), where local optima are also global optima.

The class PLS contains the problems whose neighbourhood can be searched in polynomial time, by the standard local search algorithm (introduced before as descent). That is, we have polynomial-time algorithms to (a) generate solutions in the neighbourhood, (b) evaluate the cost of the solution, and (c) determine whether it is a local optimum. Unfortunately, the complexity of finding locally optimal solutions for many interesting problems remains open. Similarly to theory that defined NP-complete problems (section 2.2), the class of PLS-complete problems characterizes the complexity of local search problems. A problem is said to be PLS-complete if every problem in PLS can be reduced to it. Hence, if a PLS-complete problem can be solved in polynomial time, it will enable all PLS-complete problems to be solved in polynomial time.

In recent years important local search problems have been found to be PLS-complete (for example, graph partitioning with Kerninghan-Lin neighbourhood). Still, the complexity of many famous local search problems (like the traveling salesman problem under standard Lin-Kernighan or 2-opt neighbourhoods) remains unknown. Even so, for practical purposes the average running time may be a more significant measure on the running time of a local search method. We can mention as an example the famous Simplex method, developed by Dantzig (1949); it works on an exact neighbourhood (the polytope of a linear programming problem). The method looks for local optima (vertex of the polytope), and it has been shown to take exponential

time. However it behaves on average very efficiently, even outperforming non-local search methods which are polynomial time algorithms (like the ellipsoid method of Khachian (1979) and interior point method of Karmarkar (1984) and Grötschel, Lovasz & Schrijver (1988)). This also serves to emphasis the difference between local search complexity and P-NP complexity, PLS lies somewhere between the two. An up to date review on computational complexity for local search heuristics can be found in Yannakakis (1997).

5.5 The Batching Machine Model

There has been a lot of interest in the last decades in scheduling problems with a batching element. There are basically two models where batching arises naturally. One is the *family scheduling model*, where jobs are grouped together as to take advantage of shared set-up times on a machine. The other is when we are actually modeling a batching machine, one that can processes several jobs simultaneously. We refer to such as model as a *batching machine model*. In this thesis we are interested in the batching machine model.

A recent and thorough review of these models can be found in Potts & Kovalyov (2000). They review both family scheduling and batching machine models, and focus on dynamic programming techniques to solve them. Batching machine models generalise models in scheduling theory, as explained in Chapter 2, by considering a machine to be a batching machine and allowing it to process more than one job at a time. Batching machines are common in the metalworking, chemical and microelectronics industries.

Lee, Uzsoy & Martin-Vega (1992), explore the 'burn in' operations in the manufacture of circuit boards with a batching machine model. Their batching machine can process up to a certain limit of jobs simultaneously, and the processing time of a batch is equal to the largest processing time among all jobs in the batch. They are concerned with customer service, quantified by on-time delivery, and thus analyse the maximum tardiness, $T_{\max}$, number of late jobs, $\sum U_j$, and maximum lateness $L_{\max}$ objective functions. Some of

the problems they consider are NP-hard, and heuristics for this are presented, as well as dynamic programming based algorithms to solve others.

Albers, S. & Brucker, P. (1993) explore the complexity of several batching problems with the flow time objective function. Webster & Baker (1995) review family of scheduling models, and present some results for some batching machine models. The results of both papers are summaries and extended in (Potts & Kovalyov 2000); parallel batching machines are also considered in this paper. Hurink (1999) explores a batching machine model where the processing time of a batch is the sum of the processing times of jobs belonging to the batch. He presents a tabu search heuristic for the problem.

Recently, Dupont & Dhaenens-Flipo (2002) proposed a branch and bound algorithm to minimise the makespan of a single batching machine with restricted batch size, where the processing time of a batch is the largest processing time among jobs in the batch. They rely heavily on dominance properties to reduce the enumeration scheme, and find good results for similarily sized jobs. The case where jobs have different release dates is explore by Sung et al. (2002). They give some computational results comparing dynamic programming algorithms for the case where jobs belong to different families.

A seminal article for the batching machine models is Brucker et al. (1998). They provide complexity results for various objectives functions. If the batch size b is bigger than the total number of jobs n to be processed $b \geq n$ then we say the model has unrestricted batch size; likewise it is restricted if $b < n$. For the case where the batch size is not restricted they provide a characterization of a class of optimal schedules, which leads to a generic dynamic programming DP algorithm that solves the problem of minimizing an arbitrary regular cost function in pseudopolynomial time.

We are interested in this result because it leads to an $O(n^2)$ DP algorithm for $\tilde{1} \mid\mid L_{\max}$ which we explain in subsection 5.5.2. An improvement to $O(n \log n)$ time is presented in Wagelmans & Gerodimos (2000). In this thesis we study the restricted batch size version of this model $\tilde{1} \mid b \mid L_{\max}$ which we will abbreviate as BMRS, and in the next subsection we explain it

in detail. We give the first branch and bound algorithm developed for it in Chapter 6. We also compare various local search heuristics for the problem in Chapter 7, with a novel neighbourhood structure.

5.5.1 A batching machine model with restricted batch size to minimise the maximum lateness BMRS

As explained before, a batching machine is one that can process several jobs simultaneously. The BMRS problem is one where n jobs are to be schedule in a single batching machine, with restricted batch size $b < n$, to minimise the maximum lateness. Let

- $J_1, \ldots, J_n$ denote the n jobs,
- p_j the processing time of job J_j,
- d_j the due date of job J_j,
- b the restricted batch size ($b < n$),
- all jobs become available at time zero ($r_j = 0$ for $j = 1, \ldots, n$),

In this case, a schedule σ is a sequence of batches $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$, where each batch $\mathcal{B}_h$ is a set of jobs. Let $|\mathcal{B}_h|$ denote the number of jobs belonging to batch $\mathcal{B}_h$ for $h = 1, \ldots, r$, and $L_{\max}(\sigma)$ the maximum lateness of schedule σ. A possible formulation for the BRMS problem is as follows.

$$\min_{\sigma} L_{\max}(\sigma) = \min_{\sigma} \max_{1 \leq j \leq n} \{C_j(\sigma) - d_j\} \qquad (5.1)$$

subject to

$$\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r) \quad \text{a sequence of batches}$$

$$\mathcal{B}_h \text{ a set of jobs} \quad h = 1, \ldots, r$$

$$\mathcal{B}_h \bigcap \mathcal{B}_l \quad h \neq l \qquad (5.2)$$

$$\bigcup_{h=1}^{r} \mathcal{B}_h = \{J_1, \ldots J_n\} \qquad (5.3)$$

$$|\mathcal{B}_h| \leq b \quad h = 1, \ldots, r \qquad (5.4)$$

$$p(\mathcal{B}_h) = \max_{J_j \in \mathcal{B}_h} \{p_j\} \quad h = 1, \ldots, r \qquad (5.5)$$

$$C(\mathcal{B}_h) = \sum_{j=1}^{h} p(\mathcal{B}_j) \quad h = 1, \ldots, r \qquad (5.6)$$

$$C_j(\sigma) = C(\mathcal{B}_h) \quad \text{if } J_j \in \mathcal{B}_h \text{ under } \sigma \qquad (5.7)$$

Note that a schedule is a sequence of batches, where each batch contains a different set of jobs, equation (5.2), and all jobs are schedule, equation (5.3). Jobs that are processed together form a batch, and constraint (5.4) ensures that there are no more than b jobs in any batch (a restricted batch size). The processing time of a batch is equal to the longest processing time of jobs in the batch, equation (5.5). All jobs in a batch begin processing at the same time, and have a common completion time, constraints (5.6) & (5.7). Note that the completion time of job J_j in σ, for each $J_j \in \mathcal{B}_h$ and $h = 1, \ldots, r$, is $C_j(\sigma) = C(\mathcal{B}_h)$. If $d(\mathcal{B}) = \min_{J_j \in \mathcal{B}} d_j$ is the due date of batch $\mathcal{B}$ then, there are two ways of calculating the maximum lateness of a given schedule σ; the one stated in equation (5.1)

$$L_{\max}(\sigma) = \max_{1 \leq j \leq n} \{C_j(\sigma) - d_j\},$$

where we calculate it over the lateness of each job, and another

$$L_{\max}(\sigma) = \max_{1 \leq h \leq r} \{C(\mathcal{B}_h) - d(\mathcal{B}_h)\},$$

where we calculate it over the lateness of each batch ($h = 1, \ldots, r$). The goal is to find an optimal schedule σ^* which minimises the maximum lateness $L_{\max}(\sigma^*) = \min_{\sigma} L_{\max}(\sigma)$.

This problem was proven to be unary NP-hard by Brucker et al. (1998). Potts & Kovalyov (2000) point out the importance of designing algorithms for NP-hard problems in view of the interest in scheduling with batching. We now explain how the unrestricted batch model $\tilde{1}||L_{\max}$ is solved, in this case constraint (5.2) can be omitted from the formulation.

5.5.2 Solution for the unrestricted batch size model

As mentioned before, Brucker et al. (1998) gave a characterization of optimal schedules for the unrestricted batch version of the batching machine model with regular scheduling criterion. If we assume that jobs are indexed according to the SPT (shortest processing time) rule ($p_1 \leq \ldots \leq p_n$), an *SPT-batch schedule* is one in which adjacent jobs in the sequence $J_1 \ldots J_n$

may be grouped to form batches. For example if there were 6 jobs, a possible batching might look like $\{J_1J_2J_3\}\{J_4\}\{J_5J_6\}$. Brucker et al. (1998) showed that there exists a SPT-batch schedule that is optimal. They propose the following backward dynamic programming algorithm with batch insertion to solve the problem.

Let F_j be the minimum value of the maximum lateness of a schedule that contains jobs $J_j, \ldots, J_n$ where the jobs are indexed in SPT order, and the processing of the first batch in the schedule starts at time zero . The initialisation is $F_{n+1} = -\infty$, and the recursion for $j = n, n-1, \ldots, 1$ is

$$F_j = \min_{j<k\leq n+1}\{\max\{F_k + p_{k-1}, \max_{j\leq i\leq k-1}\{p_{k-1} - d_i\}\}\}.$$

The optimal solution value is given by F_1, and the corresponding schedule can be found by backtracking. Note that p_{k-1} is the cost of adding a batch with jobs $J_j, \ldots J_{k-1}$ at the beginning of a schedule with jobs $J_k, \ldots, J_n$; whereas the maximum lateness of jobs in that batch can be calculated as $\max_{j\leq i\leq k-1}\{p_{k-1} - d_i\}$. A graphic representation of this calculations is shown in Figure 5.2.

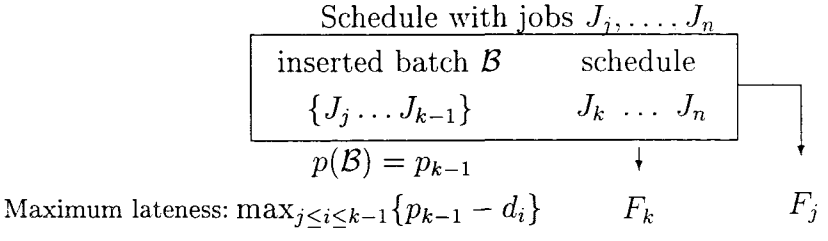

Figure 5.2: Illustration of dynamic program

A straight forward implementation of the algorithm requires $O(n^2)$ time in $O(n)$ space. Wagelmans & Gerodimos (2000) give an alternative calculation in $O(n \log n)$ time, through the efficient use of data structures.

5.6 Concluding Remarks

In this chapter we have reviewed solution procedures for combinatorial optimisation problems, and presented the BMRS problem. In particular we were

interested clarifying the elements of a branch and bound algorithm, as we develop one in the next chapter, and in explaining the main ideas of a local search heuristic, as we propose several ones in Chapter 7.

We have also presented an example of a dynamic program, in particular one that solves the unrestricted version of the problem (BMRS) we focus on the rest of the thesis. Modifications on this formulation will be used extensively throughout the next chapters.

Chapter 6

Branch and Bound for BMRS problem

6.1 Introduction

In this chapter we present a branch and bound algorithm developed for scheduling a batching machine, with restricted batch size, to minimise the maximum lateness. In the previous chapter we referred to this problem as the BMRS problem. As far as the author knows this is the first branch and bound algorithm developed for the problem. Applications can be found in the 'burn in' operations in manufacturing of circuit boards, and chemical processes that occur in tanks or kilns.

A batching machine is one that can process several jobs at the same time in a batch. In the BMRS problem there is an upper limit on the number of jobs in any batch. That is, there is a restricted (or maximum) batch size. The processing time of a batch is given by the biggest processing time of any job in the batch. We will use the same notation introduced in Chapters 2 and 5. There are n jobs to schedule $\mathcal{J} = \{J_1, \ldots, J_n\}$, and b denotes the maximum batch size. A schedule σ is a sequence of batches $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$, where each batch $\mathcal{B}_h$ is a set of no more than b jobs taken from $\mathcal{J}$. The processing time of any batch $\mathcal{B}$ is $p(\mathcal{B}) = \max_{J_j \in \mathcal{B}}\{p_j\}$. The goal is to find an optimal

schedule which will minimise the maximum lateness $L_{\max}$.

There is the decision of which jobs to assign to each batch, and how to order them. Note that there can be no more than $r = n$ batches, corresponding to a schedule with one job per batch (equivalent to $1 \mid\mid L_{\max}$), and no less than $r = \lceil n/b \rceil$, where we force as many batches as possible to have b jobs. The completion time of batch h is $C(\mathcal{B}_h) = \sum_{j=1}^{h} p(\mathcal{B}_j)$. The completion time of job J_j ($j = 1, \ldots, n$) in schedule σ, for each $J_j \in \mathcal{B}_h$ and $h = 1, \ldots, r$, is $C_j(\sigma) = C(\mathcal{B}_h)$. When there is no ambiguity, we abbreviate $C_j(\sigma)$ simply by C_j. Recall that d_j is the date by which job J_j should be completed, then

$$L_{\max} = L_{\max}(\sigma) = \max_{1 \le j \le n} \{C_j(\sigma) - d_j\} = \max_{1 \le j \le n} \{C_j - d_j\}. \tag{6.1}$$

The due date of a batch $\mathcal{B}$ is $d(\mathcal{B}) = \min_{J_j \in \mathcal{B}} d_j$, the most restrictive due date of any job in the batch. Another way of calculating the maximum lateness is

$$L_{\max}(\sigma) = \max_{1 \le l \le r} \{C(\mathcal{B}_l) - d(\mathcal{B}_l)\}. \tag{6.2}$$

The BMRS problem is a unary NP-hard problem as shown by Brucker, Gladky, Hoogeveen, Kovalyov, Potts, Tautenhahn & Van de Velde (1998). Hence, as explained in Chapter 2 Section 2.2 it is unlikely to find solutions in polynomial time, unless $NP = P$. Therefore we do not expect an implicit enumeration scheme, like the branch and bound algorithm we present, to yield optimal solution for big instances. One of our aims in this chapter is to explore what are the biggest instances solvable in reasonable time. The results we obtained are later used as basis of comparisons for the local search heuristics of Chapter 7.

As explained in Chapter 5 Section 5.3.1, to specify a branch and bound algorithm we need to choose a branching scheme, lower and upper bounding schemes, and possibly some elimination criteria. The branching scheme specifies how sets of solutions are partitioned and searched, and the lower bound is used to evaluate the minimum cost of this sets. An upper bound on the optimal solution can be easily calculated with the value of an initial feasible solution, and updated with the best value of solutions found as the

search progresses. This upper bound is used together with the lower bound to reduce the set of solutions that need inspection. To familiarise the reader with the type of properties and proofs we will be working with, we present in Section 6.2 some characteristics of optimal schedules. An initial feasible solution is obtained from a constructive heuristic developed in Section 6.3. In Section 6.4 we explain our proposed branching scheme, while the lower bound calculation is derived in Section 6.5. In Section 6.6 we present some dominance rules which further reduce the search in our tree. Finally in Section 6.7 we present and discuss the computational results.

6.2 Preliminary Results

As mentioned in Section 5.5.1, the unrestricted version of the problem, where $b \geq n$, is solved by Brucker et al. (1998). They prove that the search can be reduced to SPT-batch schedules. Based on this fact they propose a backward dynamic programming algorithm with batch insertion to find optimal schedules, as explained in Section 5.5.2.

In this section we introduce the concept of an EDD-batch schedule for the case of a restricted batch size. We show that there is always an EDD-batch schedule that is optimal. This property allows us to reduce our search for an optimal schedule to EDD-batch schedules. In particular, we use this result to justify an elimination criterion (Dominance Rule A, Section 6.6) when branching. We will aim for EDD-batch schedules in the constructive heuristics we use to obtain an upper bound, and later in chapter 7, to obtain initial solutions for our local search algorithms. We start by defining an EDD-batch schedule.

Definition 6.1 *An EDD-batch schedule with r batches $\mathcal{B}_1, \ldots, \mathcal{B}_r$ is one where $d(\mathcal{B}_l) \leq d(\mathcal{B}_{l+1})$ for $l = 1, \ldots, r-1$.*

Note that the batches in an EDD-batch schedule are ordered in non-decreasing order of their due date times.

Theorem 6.1 *There exists an EDD-batch schedule which minimizes the maximum lateness.*

Proof: Suppose there does not exist any EDD-batch schedule which is optimal. Take an optimal schedule σ. As σ is not an EDD-batch schedule, there is at least two consecutive batches in σ, say $\mathcal{B}_l$ and $\mathcal{B}_{l+1}$ $(1 \leq l \leq r-1)$, where $d(\mathcal{B}_l) > d(\mathcal{B}_{l+1})$. Construct a new schedule σ' where $\mathcal{B}_{l+1}$ is swaped with $\mathcal{B}_l$ in σ' (see Figure 6.1). Let C be the completion time of batch $\mathcal{B}_{l+1}$ under σ. Note that the completion time of batch $\mathcal{B}_l$ under σ' is also C, as we have the same $l+1$ scheduled batches before time C (see Figure 6.1).

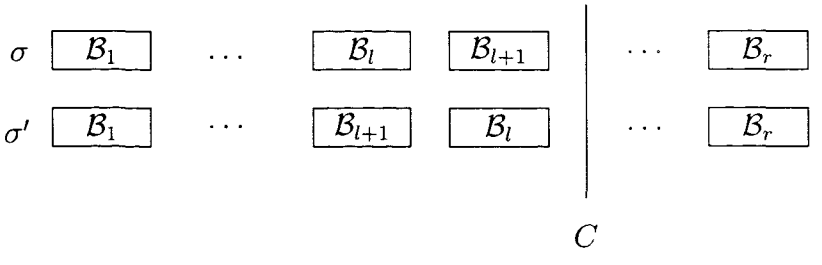

Figure 6.1: Schedules σ and σ'.

Let $L_{\max}(\sigma)$ be the maximum lateness of σ. We now analyse the maximum lateness of jobs in the new schedule σ'. From equation (6.2) we have that $L_{\max}(\sigma) \geq C - d(\mathcal{B}_{l+1})$ (i.e. the maximum lateness is an upper bound of the lateness of any job in batch B_{l+1}). On the other hand any job $J_h \in \mathcal{B}_{l+1}$ under σ' has a smaller lateness than in σ as the completion time of batch $\mathcal{B}_{l+1}$ is smaller under σ'. An upper bound on the lateness of any job in $\mathcal{B}_l$ under σ' is $\max_{j \in \mathcal{B}_l(\sigma')} L_j = C - d(B_l)$. Note that $C - d(\mathcal{B}_l) < C - d(\mathcal{B}_{l+1}) \leq L_{\max}(\sigma)$. Thus, the lateness of any job j in batch $\mathcal{B}_l$ under σ' is smaller than the maximum lateness of schedule σ. The lateness of all other jobs not belonging to B_l or B_{l+1} remain the same. Hence, σ' is also optimal. We repeat this argument for all batches $\mathcal{B}_l$ $(1 \leq l \leq r-1)$, with $d(\mathcal{B}_l) > d(\mathcal{B}_{l+1})$ in order to get an EDD-batch schedule which is optimal, contradicting our assumption. Therefore, we conclude that there always exists an EDD-batch schedule which is optimal, one that minimises the maximum lateness.

□

Another way of proving this result is to treat each batch as a job, and then use Jackson (1955) earliest due date (EDD) rule. This rule, as explained in Chapter 2, prioritizes jobs in non decreasing order of their due dates. The rule is equivalent to ordering batches in non-decreasing order of their due date. Thus, the result follows from Theorem 2.2.

Any constructive heuristic will address two questions when building a solution to the problem 1) how many batches to construct, and 2) which jobs should go in each batch. There are obvious limits to the amount of batches we can have; an upper bound of n (maximum number of batches given that there are n jobs), and a lower bound of $\lceil n/b \rceil$ (minimum number of batches we can construct with the n jobs, given that the maximum batch size is b). If the constructive heuristic builds one batch at a time filling it with jobs, and it is at a stage where a new batch $\mathcal{B}_l$ will start to be filled with jobs, we know because of Theorem 6.1 that the un-assigned job with the earliest due date (among all un-assigned job) is a good candidate for batch $\mathcal{B}_l$. Furthermore, any schedule constructed at this stage where the earliest due date job is not in batch $\mathcal{B}_l$ has a worse, or the same maximum lateness.

6.3 Upper Bound

To calculate an initial upper bound, we devise a constructive heuristic which gives a feasible solution to the problem. The heuristic constructs a schedule by adding a single job, one at a time, to a partial schedule until all of the jobs have been assigned and a complete schedule achieved. We work with EDD indexing for the n jobs ($d_1 \leq \ldots \leq d_n$). Note that assigning job J_1 (job with the smallest due date) to any batch but the first will not be consistent with Theorem 6.1. Thus, job J_1 should belong to batch $\mathcal{B}_1$. Hence, our first step is to add job J_1 to batch $\mathcal{B}_1$. Thus, initially $p(\mathcal{B}_1) = p_1$, and $d(\mathcal{B}_1) = d_1$. Let $\mathcal{B}_r$ be the last batch in the partially constructed schedule. Initially $r = 1$. Let $\delta \geq 0$, be a parameter, that specifies by how much we allow a batch to increase its processing time when adding a new job. If $\delta = 0$ we will not

add a job to a given batch if it increases its processing time. The heuristic constructs a schedule by adding job J_j for $j = 2, \ldots, n$ as follows:

- Find the smallest k with $k \leq r$, for which batch $\mathcal{B}_k$ has less than b jobs, and $p_j \leq p(\mathcal{B}_k) + \delta$. If such a batch exists then add job J_j to batch $\mathcal{B}_k$. Update the processing time of batch $\mathcal{B}_k$ if necessary.

- Otherwise, create a new batch $\mathcal{B}_{r+1}$. Add job J_j to $\mathcal{B}_{r+1}$, the processing time of this batch is p_j, and the due date is d_j. Increase r by one, as the last batch in the partially constructed schedule is now batch $\mathcal{B}_{r+1}$.

This heuristic yields an EDD-batch schedule. Note that each new job added to a batch will not increase the processing time of that batch more than δ. We use $\delta = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(p_j - \overline{p})^2}$, where $\overline{p}$ is the mean of the processing times of the n jobs. δ is the standard deviation of the processing times of the n jobs. Thus, in the worst case, the difference between the smallest processing time in a batch and the biggest is not more than $\delta(b-1)$. The heuristic has a time complexity of $O(n \log n)$. The maximum lateness of the final schedule obtained from this heuristic will be used as the upper bound of the branch and bound algorithm. The upper bound will be updated every time a new schedule with better value (smaller maximum lateness) is found when searching the tree.

6.4 Branching Scheme

The branching scheme specifies how sets of solutions are partitioned and searched. At each node, our partition excludes sets of solutions by assigning certain jobs to specific batches. Each node is associated with a partial schedule, where some jobs have already been fixed. In our branching scheme, at each node in our search tree, we focus in one particular batch; call it the *current batch.* A branch coming out of any node will either add another job to the current batch (branch denoted by the job added), or consider the current batch as complete with the jobs it has (branch denoted by node ϕ).

Our jobs will be indexed in EDD (earliest due date) order ($d_1 \leq \ldots \leq d_n$). We use a depth first policy to explore the tree. Thus, if job J_k is assigned (added) to the current batch and the current batch has less than b jobs, a branch with job J_j for $j < k$ (representing the operation of adding job J_j to the current batch) is not feasible (as this option is analysed previously). At each node, a lower bound on that situation is calculated and compared with the current upper bound on the optimal solutions. We eliminate the node if the lower bound is greater than or equal to the upper bound. Once job J_j is included in the partial schedule it is excluded for the remaining nodes. If the current batch $\mathcal{B}_h$ has b jobs, or if we evaluate the branch where this batch is complete, the feasible branches are those corresponding to jobs which are not yet assigned. For this branches, $\mathcal{B}_{h+1}$, becomes our new current batch. Hence, there will be no more than $n-1$ branches coming out of each node, and an end node will be reached in no more than $2n-1$ branching levels. Note that an end node in our tree represents a complete schedule. As mentioned before, we continue branching from any node if the lower bound is smaller than the current upper bound. Initially, this upper bound is calculated from the approximate solution found by the heuristic explained in Section 6.3. The upper bound is updated every time we reach an end node representing a better schedule. Pruning of nodes is done by comparing the lower bound evaluation at each node with the current upper bound on the optimal schedule, and by applying dominance rules. We explain both the lower bound calculations and dominance rules in the following sections.

To illustrate the branching scheme consider an instance with 6 jobs, numbered in EDD (earliest due date) order (i.e. $d_1 \leq \ldots \leq d_6$), and restricted batch size $b = 3$. Suppose job J_1 is the only job scheduled in batch $\mathcal{B}_1$, that batch $\mathcal{B}_2$ is the current batch, and that it only contains job J_2. The last job added is J_2. The branches coming out of that node are: (a) job J_3 belongs to $\mathcal{B}_2$, (b) job J_4 belongs to $\mathcal{B}_2$, and J_3 does not, (c) job J_5 belongs to $\mathcal{B}_2$, but J_3, and J_4 do not (d) job J_6 belongs to $\mathcal{B}_2$, but J_3, J_4, and J_5 do not, finally (e) no more jobs belong to batch $\mathcal{B}_2$. Depth first means we explore

branch (a) before exploring branch (b), and (b) before (c), and so on. In the same example, if instead of job J_2 belonging to batch $\mathcal{B}_2$ we had that job J_5 belonged to batch $\mathcal{B}_2$, the only feasible branches would be (d) and (e). On the other hand, if we went down branch ϕ, the new branches emanating from it, would be (a) $J_3 \in \mathcal{B}_3$, (b) $J_4 \in \mathcal{B}_3$, (c) $J_5 \in \mathcal{B}_3$, (d) $J_6 \in \mathcal{B}_3$, excluding branch (e) and the new current batch changes to $\mathcal{B}_3$.

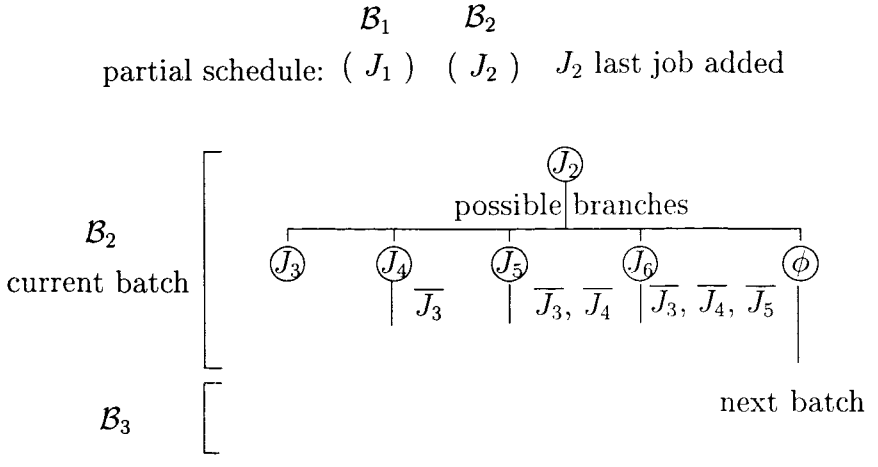

Figure 6.2: Branching from node J_2

6.5 Lower Bound

In this section we explain how to calculate a lower bound at each node. There are two characteristics which a lower bound should aim to have, one it should be quick to compute, and the second, that it should be as tight as possible. In our case, tight means close to the minimum maximum lateness of the schedules represented by the node. Recall that for unrestricted batch sizes the minimum $L_{\max}$ is straightforward to compute, as we only had to order jobs in SPT and then apply the dynamic program explained in Section 5.5.2. This was possible because we could reduce our search to SPT-batch schedules. In the restricted case there is no guarantee that an SPT-batch schedule would be optimal. In fact, there are several instances where the optimal schedule is

not an SPT-batch schedule. The lower bound we propose considers schedules where batches instead of being composed of jobs are composed of a set of processing times and due dates. We relax the linking of processing time and due dates of each job, and allow for batches to have a un-equal mix of processing times and due dates, defining a new type of schedule (SPT-EDD-dynamic-batch schedule) whose minimum $L_{\max}$ is a lower bound on the optimum schedule for the restricted case. We explain this in more detail below. Later, we propose a dynamic program to find the minimum $L_{\max}$ of this type of schedules, and how to modify it to include information of partially completed batches as we analyse nodes in the tree.

Definition 6.2 *Let α be an SPT ordering of jobs in $\mathcal{J} = \{J_1, \ldots, J_n\}$ (i.e $p_{\alpha(1)} \leq \ldots \leq p_{\alpha(n)}$), and β an EDD ordering (i.e. $d_{\beta(1)} \leq \ldots \leq d_{\beta(n)}$). Let $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$ be a schedule, with batches composed of a collection of processing times and due dates. Let batch $\mathcal{B}_l = \left\{ \begin{matrix} p_{\alpha(j_l)}, & p_{\alpha(j_l+1)}, & \ldots & p_{\alpha(j_{l+1}-1)} \\ d_{\beta(m_l)}, & d_{\beta(m_l+1)}, & \cdots & d_{\beta(m_{l+1}-1)} \end{matrix} \right\}$ where $1 = j_1 < j_2 < \ldots < j_r = n+1$ and, $1 = m_1 < m_2 < \ldots < m_r = n+1$, for $l = 1, \ldots, r$. The processing time of the batch is $p(\mathcal{B}_l) = p_{\alpha(j_{l+1}-1)}$, and the due date of the batch is $d(\mathcal{B}_l) = d_{\beta(m_l)}$. Call this type of schedule an SPT-EDD-dynamic-batch schedule.*

Note that such a schedule is completely specified by the due date and processing times that start each batch. For example, let us think of the instance shown in Table 6.1. A possible sequence α (jobs in SPT) is $J_1, J_3, J_4, J_2, J_5, J_6$

Table 6.1: 6 job example

Job	J_1	J_2	J_3	J_4	J_5	J_6
p_j	2	4	3	3	5	5
d_j	3	6	7	8	11	12

(there are two possible sequences), whereas sequence β (jobs in EDD) is $J_1, J_2, J_3, J_4, J_5, J_6$. Suppose the batch size is $b = 3$, a possible SPT-EDD-dyanmic batch schedule is as follows: $\left\{ \begin{matrix} p_1 \\ d_1, d_2 \end{matrix} \right\}, \left\{ \begin{matrix} p_3, p_4, p_2 \\ d_3 \end{matrix} \right\}, \left\{ \begin{matrix} p_5, p_6 \\ d_4, d_5, d_6 \end{matrix} \right\}$. This

schedule consists of $r = 3$ batches, with $j_1 = 1$, $j_2 = 2$, $j_3 = 5$, and $m_1 = 1$, $m_2 = 3$, $m_3 = 4$. Note that the processing time of the batch $\mathcal{B}$ corresponds to the biggest index on the $p_{\alpha(j)} \in \mathcal{B}$, and the due date is the smallest index $d_{\beta(j)} \in \mathcal{B}$ (in fact, the first due date in the batch). For example, the processing time of batch B_3 is p_6, and its due date d_4. Using equation (6.2) the maximum lateness of this schedule is 3.

Theorem 6.2 *There exists an SPT-EDD-dynamic batch schedule whose maximum lateness is a lower bound on the* $L_{\max}$ *of any schedule constructed from a set of jobs* $J_1, \ldots, J_n$.

Proof: As an overview, the idea behind this proof is 1) to modify the optimal schedule σ obtaining a new schedule σ' where the due dates are in EDD order. We then 2) allow for dynamic batches (where each batch may have an un-equal mix of processing times and due date times) and construct a schedule σ'' where jobs are in EDD order and batches are in SPT order. Finally 3) we construct from σ'' a schedule σ''' that is a SPT-EDD-batch dynamic batch schedule whose $L_{\max}$ provides a lower bound on the optimal schedule.

1) Let σ be a schedule with minimum $L_{\max}$ over $J_1, \ldots, J_n$. Suppose there are two batches in σ, with $\mathcal{B}_H$ scheduled before $\mathcal{B}_K$, such that a job $J_h \in \mathcal{B}_H$ has a later due date than a job $J_k \in \mathcal{B}_K$, that is $d_h > d_k$. Modify the due dates of jobs J_h and J_k such that d_h is now $d'_h = d_k$ and the due date of job J_k is now $d'_k = d_h$. In other words, we interchange the due dates of the jobs. The maximum lateness after modifying these due dates is

$$L_{\max} = \max\{C_k - d'_k, C_h - d'_h, R\}, \text{ where } R = \max_{J_l \in (\mathcal{J} \setminus \{J_h, J_k\})} \{C_l - d_l\}.$$

Since $d_h > d_k$, we have that $C_k - d_k > C_k - d'_k$ so that the lateness of job J_k is now smaller. Recall that $C_k > C_h$ as batch $\mathcal{B}_K$ is scheduled after batch $\mathcal{B}_H$. Thus $C_k - d_k > C_h - d_k = C_h - d'_h$. So that the lateness of job J_h is not bigger than the previous lateness of job J_k. Hence, the maximum lateness of schedule σ is not increased by interchanging these due dates.

We can continue interchanging the due dates of the jobs to construct a new schedule $\sigma' = (\mathcal{B}'_1, \ldots, \mathcal{B}'_r)$ over a new set of jobs $\mathcal{J}' = \{J'_1, \ldots, J'_n\}$ where the processing time of job $J'_j \in B'_l$ is the same as job $J_j \in B_l$ for every $J_j \in B_l$, and $l = 1, \ldots, r$, with the due dates of the jobs in $\mathcal{J}'$ under σ' in earliest due date (EDD) order.

2) Suppose there exist two batches in σ', with $\mathcal{B}'_H$ scheduled before $\mathcal{B}'_K$, such that $p(\mathcal{B}'_H) > p(\mathcal{B}'_K)$. Focus on the processing times and due date times of the jobs in each batch. Batch $\mathcal{B}'_H$ has $|\mathcal{B}'_H|$ due dates and processing times, while batch $\mathcal{B}'_K$ has $|\mathcal{B}'_K|$. Allow batches with different numbers of processing times and due dates. We work now with a different concept of a batch, not as a set of jobs, but rather a collection of due dates and processing times. Accordingly, we adapt the definition of $d(\mathcal{B}'_l)$ to be the minimum of the due dates belonging to batch $\mathcal{B}'_l$ instead of the jobs belonging to the batch, and similarly adapt the definition of $p(\mathcal{B}'_l)$ to be the maximum over the processing times belonging to batch $\mathcal{B}'_l$ $(1 \leq l \leq r)$. No batch is allowed to have more than b due dates or b processing times. We are still able to use equation (6.2) to calculate the maximum lateness, whereas equation (6.1) is no longer useful. Interchange all processing times of batch $\mathcal{B}'_H$ with all processing times of batch $\mathcal{B}'_K$. The other batches remain the same. The new processing time of batch $\mathcal{B}'_H$ is now $p(\mathcal{B}'_K)$, and the completion time of any job in that batch is now smaller as $\sum_{L=1}^{H-1} p(\mathcal{B}'_L) + p(\mathcal{B}'_K) < C(\mathcal{B}'_H)$. Furthermore, if there are batches between $\mathcal{B}'_H$, and $\mathcal{B}'_K$ their completion time is also smaller than before. On the other hand, the completion time of batch $\mathcal{B}'_K$ remains the same, $\sum_{L=1}^{K} p(\mathcal{B}'_L)$. Hence, the completion time of batches after batch $\mathcal{B}'_K$ remain the same. Thus, the maximum lateness is not increased by this interchange in processing times.

We can repeat this same argument, and continue interchanging the processing times of different batches to construct a new schedule $\sigma'' = (\mathcal{B}''_1, \ldots, \mathcal{B}''_r)$, where the batches are in SPT-batch order (i.e. $p(\mathcal{B}''_1) \leq p(\mathcal{B}''_2) \leq \ldots \leq p(\mathcal{B}''_r)$), and the due dates of batch $\mathcal{B}''_l$ are the same as the due dates of batch $\mathcal{B}'_l$ for $1 \leq l \leq r$. Batches do not necessarily have the same number of processing

times and due dates, but $L_{\max}(\sigma'') \leq L_{\max}(\sigma') \leq L_{\max}(\sigma)$.

3) Suppose there are two batches in σ'', with $\mathcal{B}''_H$ scheduled before $\mathcal{B}''_K$, such that there exists a processing time p_h belonging to batch $\mathcal{B}''_H$ that is bigger than a processing time p_k in batch $\mathcal{B}''_K$, that is $p_h > p_k$. Interchange these processing times. The processing time of batch $\mathcal{B}''_H$ is now

$$\max\{p_k, R\}, \text{ where } R = \max_{p_j \in (\mathcal{B}''_H \setminus \{p_k\})} p_j,$$

which is not bigger than before the interchange as $p_k < p_h$. Recall that for σ'', p_h is in batch $\mathcal{B}''_H$ and, as σ'' is an SPT-batch schedule, $p(\mathcal{B}''_H) \leq p(\mathcal{B}''_K)$, so that for σ'' $p_k < p_h \leq p(\mathcal{B}''_K)$. Thus, $p_k \neq p(\mathcal{B}''_K)$, and the processing time of batch $\mathcal{B}''_K$ remains the same after the interchange. Hence, the maximum lateness is not increased by this interchange in processing times.

We can continue interchanging the processing times to construct a new schedule $\sigma''' = (\mathcal{B}'''_1, \ldots, \mathcal{B}'''_r)$, where the processing times are in SPT order and the due dates in EDD order, although batches might contain different numbers of processing times and due dates. As $L_{\max}(\sigma''') \leq L_{\max}(\sigma'') \leq L_{\max}(\sigma)$, the maximum lateness of this schedule is a lower bound on $L_{\max}(\sigma)$. Note that σ''' is an SPT-EDD-dynamic-batch schedule.

□

An SPT-EDD-dynamic-batch schedule with minimum maximum lateness can be calculated by a backward dynamic programming algorithm as explained below.

Using the notation introduced in definition 6.2, α is SPT ordering of jobs in $\mathcal{J} = \{J_1, \ldots, J_n\}$ (i.e $p_{\alpha(1)} \leq \ldots \leq p_{\alpha(n)}$), and β the EDD ordering (i.e. $d_{\beta(1)} \leq \ldots \leq d_{\beta(n)}$). Recall that an SPT-EDD-dynamic-batch schedule $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$, has batches composed of a collection of processing times and due dates, with batch $\mathcal{B}_l = \left\{ \begin{matrix} p_{\alpha(j_l)}, & p_{\alpha(j_l+1)}, & \cdots & p_{\alpha(j_{l+1}-1)} \\ d_{\beta(m_l)}, & d_{\beta(m_l+1)}, & \cdots & d_{\beta(m_{l+1}-1)} \end{matrix} \right\}$ where $1 = j_1 < j_2 < \ldots < j_r = n+1$ and, $1 = m_1 < m_2 < \ldots < m_r = n+1$, for $l = 1, \ldots, r$.

Let $F(j, m)$ be the value of the maximum lateness of an SPT-EDD-dynamic-batch schedule which contains the last $n - j + 1$ processing times

corresponding to jobs $J_{\alpha(j)}, \ldots, J_{\alpha(n)}$, and $n - m + 1$ due date times corresponding to jobs $J_{\beta(m)}, \ldots, J_{\beta(n)}$. If a batch $\mathcal{B}$ which groups processing times $p_{\alpha(j)}, \ldots, p_{\alpha(j'-1)}$ and due dates $d_{\beta(m)}, \ldots, d_{\beta(m'-1)}$ is inserted at the start of a schedule σ containing processing times corresponding to jobs $J_{\alpha(j')}, \ldots, J_{\alpha(n)}$ and due dates to jobs $J_{\beta(m')}, \ldots, J_{\beta(n)}$, then the maximum lateness of jobs in σ is increased by $p_{\alpha(j'-1)}$ (biggest processing time in batch $\mathcal{B}$) while the maximum lateness of batch $\mathcal{B}$ is $p_{\alpha(j'-1)} - d_{\beta(m)}$ (as $d_{\beta(m)}$ is the smallest due date in $\mathcal{B}$).

The initialisation is

$$F(n+1, n+1) = -\infty,$$

and the recursion for $j = n, n-1, \ldots, 1$ and $m = n, n-1, \ldots, 1$ is

$$F(j,m) = \min_{\substack{j<j' \leq \min\{j+b, n+1\} \\ m<m' \leq \min\{m+b, n+1\}}} \left\{ \max\{F(j', m') + p_{\alpha(j'-1)}, p_{\alpha(j'-1)} - d_{\beta(m)}\} \right\}.$$

The minimum value of the maximum lateness is given by $F(1,1)$, and can be computed in $O(n^2 b^2)$ time.

We are now ready to explain how we calculate the lower bound at any node. Recall that each node has an associated partially constructed schedule. The maximum lateness of those batches in the partially constructed schedule is a lower bound on the maximum lateness of any branch emanating from that node. Hence, first we calculate the maximum lateness and makespan of the completed batches forming the partially constructed schedule; let their values be $L'_{\max}$, and $C'_{\max}$, respectively. We then take the jobs of the incomplete batch (there is at least one job when the current batch is not completed), and all remaining unscheduled jobs and find the lower bound on the maximum lateness of these jobs; let its value be F'. If we use the dynamic programming formulation explained above, this will provide a lower bound on the maximum lateness of these jobs if they had started processing at time 0. However, as they start at time $C'_{\max}$, the lateness has to increase by that amount . Thus, the lower bound can then be computed as

$$\max\{L'_{\max}, C'_{\max} + F'\},$$

which again can be computed in $O(n^2b^2)$ time. Note that there might be several jobs in an incomplete batch, and they might change place in the SPT-EDD-dynamic-batch schedule we employ to compute F'. We know that the processing time of the incomplete batch is given by the biggest processing time of the jobs in the b. ch, say p', and its due date is given by the smallest of the due dates, say d'. Thus, to take into consideration that they are already assigned to the incomplete batch we need to change the processing time of the jobs in the incomplete batch to p', and the due dates to d'. We summarise the process as follows.

Lower Bound Procedure

begin

Compute $L'_{\max}$ and $C'_{\max}$

Let b_c be the number of jobs in the current batch B_l.

If $b_c = b$ then compute $L_{\max}$ and $C_{\max}$ of partial schedule.

else compute $L_{\max}$ and $C_{\max}$ of the schedule until batch B_{l-1}.

Set $L'_{\max} = L_{\max}$, and $C'_{\max} = C_{\max}$.

Compute F'.

If $b_c < b$ then

$p' = \max_{J_j \in \mathcal{B}_l} p_j$ and $d' = \min_{J_j \in \mathcal{B}_l} d_j$.

$\mathcal{U}$ is a set comprising all unscheduled jobs, b_c identical jobs with processing time p', and due date d'.

else $\mathcal{U}$ has all unscheduled jobs.

Compute F' over U.

Compute Lower Bound

The lower bound is $\max\{L'_{\max}, C'_{\max} + F'\}$.

end

As an example, consider the instance introduced before in Table 6.1. Let job J_1 be assigned to batch $\mathcal{B}_1$, and jobs J_2 and J_3 assigned to the current batch $\mathcal{B}_2$. Recall the processing times and due dates are as follows.

Job	J_1	J_2	J_3	J_4	J_5	J_6
p_j	2	4	3	3	5	5
d_j	3	6	7	8	11	12

As batch $\mathcal{B}_2$ has less than $b = 3$ jobs, we calculate the makespan and maximum lateness of the partial schedule without the current batch. Thus, computing the makespan and lateness of batch $\mathcal{B}_1$ (containing job J_1), to get $C'_{\max} = 2$, and $L'_{\max} = -1$. The unscheduled jobs are J_4, J_5, and J_6. As there are two job in the incomplete batch $\mathcal{B}_2$, we also take 2 jobs J'_1, J'_2 with processing time $p_{\max} = 4$ and due date $d_{\min} = 6$. Thus, $\mathcal{U}$ is the set of jobs $\{J'_1, J'_2, J_4, J_5, J_6\}$ with the processing times and due dates shown in Table 6.2. After applying the dynamic programming algorithm explained above we

Table 6.2: Set $\mathcal{U}$

Job	J'_1	J'_2	J_4	J_5	J_6
p_j	4	4	3	5	5
d_j	6	6	8	11	12

obtain $F'_1 = -2$, from the schedule $\left\{\begin{smallmatrix}3,4,4\\6,6,8\end{smallmatrix}\right\}, \left\{\begin{smallmatrix}5,\ 5\\11,12\end{smallmatrix}\right\}$. Hence the lower bound is 0. In fact, for this particular instance, the bound is tight as the optimal schedule is $(\{J_1\}, \{J_2, J_3, J_4\}, \{J_5, J_6\})$ with $L_{\max} = 0$.

6.6 Dominance Rules

Recall that we do not need to explore a node if the lower bound is greater or equal to the upper bound. The dominance rules that follow aim at making the pruning process of the tree faster than just by using the simple rule of comparing upper and lower bounds. The basic argument of any dominance rule is that the schedules corresponding to a node (and its branches) are not worth analysing as they are dominated. There are many situations where we are able to check for the dominance of a node. We have applied five main rules, and explain them below.

Dominance Rule A

A situation where we are able to check for dominance, is when a batch is completed. At this stage, we branch form the node representing the last job added to the batch just completed, starting a new batch. Instead of looking at all unscheduled jobs we know that the job with earliest due date of all unscheduled jobs must be in this batch, as explained in Section 6.2. Thus, we need not analyse all the other nodes.

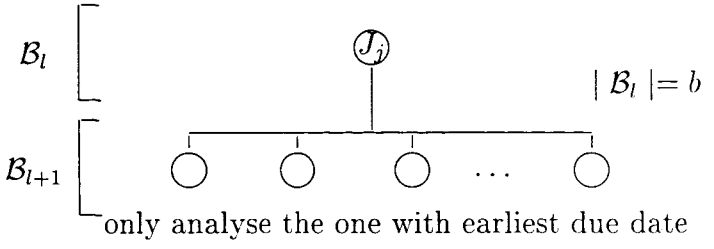

Figure 6.3: Dominance Rule A

Dominance Rule B

Another situation where we can check for dominance is just after we have added a new job to a batch, and the batch is not full. We can check if node ϕ (corresponding to the decision to leave the current batch as it is, adding no more jobs to it) is worth analysing.

Suppose the last job added to the current batch $\mathcal{B}_l$ of the partially constructed schedule σ in our search tree is J_j, and that batch $\mathcal{B}_l$ is not full (i.e. $|B_l| < b$). We can add a job $J_i \in \mathcal{U}$, where $\mathcal{U}$ is the list of unscheduled jobs, to batch $\mathcal{B}_l$ without violating the restriction on the batch size. If there exists a job $J_i \in \mathcal{U}$, such that

$$p_i \leq \max_{J_k \in \mathcal{B}_l} p_k, \text{ and } d_i \geq \min_{d_k \in \mathcal{B}_l} d_k, \tag{6.3}$$

then adding job J_i to batch $\mathcal{B}_l$ does not increase the makespan or maximum lateness of the partial schedule σ, because the conditions in (6.3) are equivalent to $p_i \leq p(\mathcal{B}_l)$, and $d_i \geq d(\mathcal{B}_l)$. Moreover, adding job J_i to batch $\mathcal{B}_l$ will reduce the number of unscheduled jobs remaining to be added to the

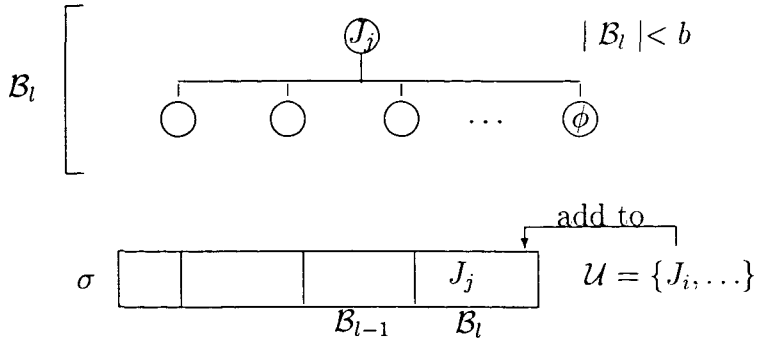

Figure 6.4: Dominance Rule B

partial schedule. Any schedule found by going down the branch indicating that job J_j is the last job to be added to batch B_l (i.e. with the child branch corresponding to node ϕ), is dominated. Hence, the branch ϕ is not worth analysing and should be discarded from the list of nodes to analyse from node J_j.

Dominance Rule C

Now, let us look at the situation where the job that has just been added completes the batch. We analyse whether is worth branching from this node.

Suppose the last job added to the partially constructed schedule σ in the current batch $\mathcal{B}_l$ is J_j, and that this job makes the batch full (i.e. $|B_l| = b$). If there exists a job J_k in batch $\mathcal{B}_l$, and a job J_i in $\mathcal{U}$, where $\mathcal{U}$ is the list of unscheduled jobs, such that $p_k \leq p_i$, $d_k \geq d_i$ (job J_i is more restrictive than job J_k), and

$$p_i \leq \max_{J_h \in (\mathcal{B}_l \setminus J_k)} p_h, \tag{6.4}$$

then the partial schedule is dominated.

Proof: Let σ' be the schedule constructed from σ where job J_k is substituted with J_i in batch $\mathcal{B}_l$. Restriction (6.4) ensures that the processing time of batch $\mathcal{B}_l$ in σ' is the same as in σ. Hence C the completion time of batch $\mathcal{B}_l$ is the same for both (partial) schedules ($C(\mathcal{B}_l(\sigma)) = C(\mathcal{B}_l(\sigma'))$).

Construct a schedule $\sigma^* = \sigma\pi$ that is optimal among schedules that start with σ. The maximum lateness of σ^* is $L_{\max}(\sigma^*) = \max\{L_k^*, L_i^*, L^*\}$,

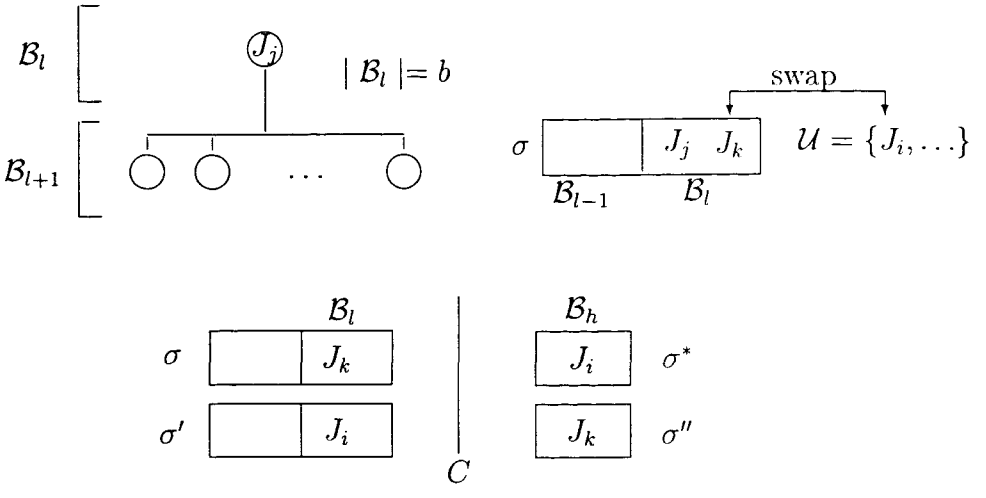

Figure 6.5: Dominance Rule C

where L_k^* and L_i^* is the lateness of job J_k and J_i respectively in σ^*, and L^* is the maximum of the lateness of all other $n-2$ jobs in σ^*. Note that $L_k^* = C(\mathcal{B}_l(\sigma^*)) - d_k = C(\mathcal{B}_l(\sigma)) - d_k = C - d_k$, where C is the makespan of the partial schedule σ. Let $\mathcal{B}_h$ be the batch to which J_i belongs in σ^* (i.e. in π). Thus, $L_i^* = C(B_h(\sigma^*)) - d_i$. Let σ'' be the schedule obtain from exchanging job J_i with J_k in σ^*. We can express the maximum lateness of schedule σ'' as $L_{\max}(\sigma'') = \max\{L_k'', L_i'', L''\}$, the lateness of job k, job i, and the remaining $n-2$ jobs in schedule σ''. Note that, $L_k'' = C(\mathcal{B}_h(\sigma'')) - d_k$, and $L_i'' = C(\mathcal{B}_l(\sigma'')) - d_i = C - d_i$. Hence, $L_i'' < L_i^* \leq L_{\max}(\sigma^*)$. Recall that $p_k \leq p_i$, so $C(\mathcal{B}_h(\sigma'\pi)) \leq C(\mathcal{B}_h(\sigma^*))$. Thus $L'' \leq L^*$, and as $d_k \geq d_i$, we have that $L_k'' \leq L_i^* \leq L_{\max}(\sigma^*)$.

Thus, the partial schedule constructed form this node is dominated.

□

We have to be careful when $p_i = p_k$ and $d_i = d_k$, as not to eliminate equivalent schedules. Hence, in this case we consider the branch dominated if $i > k$.

Dominance Rule D

Again we will look at the situation where a job has just completed the

batch. Only now, instead of looking for dominance by exchanging a job in the partially completed schedule with one that is not, we will look at dominance by exchanging two jobs within the partial schedule.

Suppose the last job added, in the partially constructed schedule σ, to the current batch $\mathcal{B}_l$ in our search tree is J_j (i.e. $|\mathcal{B}_l| = b$). Consider a job J_i in a previous batch, that is, $J_i \in \mathcal{B}_k$ where $k < l$. Let σ' be a schedule obtained by interchanging job J_i and J_j. If the maximum latness of σ' is smaller than the one for σ (i.e. $L_{\max}(\sigma') < L_{\max}(\sigma)$) and $C_{\max}(\sigma') \leq C_{\max}(\sigma)$, or if the makespan of σ' is smaller than the one for σ (i.e. $C_{\max}(\sigma') < C_{\max}(\sigma)$) and $L_{\max}(\sigma') \leq L_{\max}(\sigma)$, then any complete schedule built by continuing the partial schedule σ will be dominated by one bulit by continuing the partial schedule σ'. Thus, the branch corresponding to this last job J_j can be discarded form our search. If both $L_{\max}(\sigma) = L_{\max}(\sigma')$ and $C_{\max}(\sigma) = C_{\max}(\sigma')$, we choose to discard the node if $i > j$.

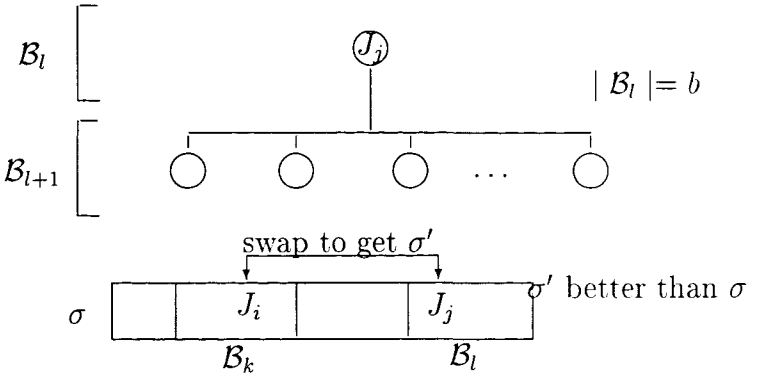

Figure 6.6: Dominance Rule D

Dominance E

In this last rule, we actually store some values as we analyse a branch (recall we are doing depth first). From this stored information, sometimes it is possible to detect that a subsequent branch is dominated.

Suppose that dominance rule D does not eliminate a node, but that we reach a situation where the new schedule σ', found by exchanging job $J_i \in \mathcal{B}_h$

with $J_j \in \mathcal{B}_l$ (with $h < l$ where l is the current batch) in schedule σ, is such that

$$C_{\max}(\sigma') \leq C_{\max}(\sigma), \text{ and } L_{\max}(\sigma) < L_{\max}(\sigma').$$

We store the value of $L_{\max}(\sigma')$ (or the smallest value for any of the previous exchanges of this type). We will continue branching form this node (i.e. building on σ). If at a later stage we have completed the current batch in a partial schedule $\sigma\pi$ and the lower bound of this node is such that $L_{\max}(\sigma') \leq LB(\sigma\pi)$, then the node is dominated, and can be discarded form the search tree.

Proof: Consider a schedule $\sigma\pi\rho$ constructed form $\sigma\pi$ which minimises the maximum lateness. That is, ρ is the optimal continuation of $\sigma\pi$. The maximum lateness of schedule $\sigma\pi\rho$ may be calculated as

$$L_{\max}(\sigma\pi\rho) = \max\{L_{\max}^{(\sigma\pi\rho)}(\sigma), L_{\max}^{(\sigma\pi\rho)}(\pi), L_{\max}^{(\sigma\pi\rho)}(\rho)\}, \tag{6.5}$$

where $L_{\max}^{(\psi)}(\theta)$ is the maximum lateness of the jobs in θ which is a subset of schedule ψ. We have that

$$L_{\max}(\sigma') \leq LB(\sigma\pi) \leq L_{\max}(\sigma\pi\rho).$$

Consider the schedule $\sigma'\pi\rho$, note that $L_{\max}^{(\sigma'\pi\rho)}(\sigma') = L_{\max}(\sigma')$. Recall that $C_{\max}(\sigma') \leq C_{\max}(\sigma)$, hence $L_{\max}^{(\sigma'\pi\rho)}(\pi) \leq L_{\max}^{(\sigma\pi\rho)}(\pi)$, and $L_{\max}^{(\sigma'\pi\rho)}(\rho) \leq L_{\max}^{(\sigma\pi\rho)}(\rho)$. If we express the maximum lateness of schedule $\sigma'\pi\rho$ as the maximum of its three parts, as in equation (6.5), we conclude from the above inequalities that $L_{\max}(\sigma'\pi\rho) \leq L_{\max}(\sigma\pi\rho)$. Thus, schedule $\sigma\pi$ is dominated.

□

We should be careful when $C_{\max}(\sigma') = C_{\max}(\sigma)$. Note that in our branch and bound algorithm we are applying this rule at the same time as checking rule D, which checks for this condition. Hence, we have to use a more restrictive condition for the makespan. We will only store the value of $L_{\max}(\sigma')$ if the inequality $C_{\max}(\sigma') \leq C_{\max}(\sigma)$ is satisfied strictly. That is, if $C_{\max}(\sigma') < C_{\max}(\sigma)$.

6.7 Computational Experience

6.7.1 Experimental Design

We are interested in comparing the behaviour of the branch and bound algorithm over a range of instances. In particular we investigate the number of jobs the algorithm is able to tackle using reasonable computer resources, and we explore the effect of different batch sizes, and due dates.

We have generated sets of instances with $n = 10, 15, 20$, and 25 jobs. We choose different values for b depending on the number of jobs. We also vary the number of instances per set depending on the average time it takes to solve each instance. The following table summarises the different sets of instances we generated. The processing time of the jobs are integers uniformly

Table 6.3: Set of instances for the branch and bound algorithm

n	b	number of instances
10	2, 3, 4, 5, 6, 7, 8	10
15	2, 3, 4, 5, 6, 7, 8, 9, 10	10
20	2, 3, 4	10
25	5, 10, 15	5

distributed from 1 to 100. An estimate on the makespan of the schedule, given the processing times and the batch size b, is $T = \frac{1}{b}\sum_{j=1}^{n} p_j$. An obvious choice for the distribution of the due dates is a uniform distribution from 1 to T. However, to mimic different scenarios for the due dates, we used a parameter λ that takes the following values: $0.5, 1, 1.5$, and integer due dates are distributed uniformly between 1 and λT. A scenario with $\lambda = 0.5$ has restrictive due dates (we expect to have bigger values for the maximum lateness). On the other hand a scenario with $\lambda = 1.5$ has slack due dates (we expect smaller values for the maximum lateness).

The algorithm was encoded in Turbo Pascal and run on a PC with a

Pentium 200MHz processor and 32Mb of ram. For instances with 10 and 15 jobs no time limit was set. However, for 20, and 25, some instances took more than 10 hours to solve, so a limit time for the run was set at 2 hrs and 30 minutes.

6.7.2 Influence of Dominance Rules

There is a substantial reduction on the running time when running the instances with the dominance rules. This was expected as eliminating nodes at an early stage, reduces the search tree considerably. Instances with 10 jobs took several minutes without dominance rules; with the dominance rules the time was reduced to fractions of a second. Rule A and B yield the most reductions in times. Nevertheless, every rule is important. Taking out any rule increases running time of the algorithm, and the number of nodes explored in the search.

In Table 6.4, we present results on how many times the different dominance rules were successfully used for the set of instances with 15 jobs. Each entry is the average over the 10 instances generated per set. We do not show summary results for Rule A, as we consider this rule more as part of the branching scheme, than an elimination criteria. It is always applied, and there is no need to calculate the makespan or lateness of any partial schedule. Its influence in node elimination is vital. We are more interested in comparing the behaviour of the other rules.

As can be seen from Table 6.4, Rule B is the most effective rule (excluding rule A) to eliminate nodes from the tree. As a percentage of the total amount of the nodes analysed it fluctuates form 1% (for the set of instances with $b = 2, \lambda = 1.5$) to 84% (for the set of instances with $b = 7, \lambda = 0.5$). This might be explained from the fact that Rule B is applied when analysing unfull batches (i.e. with $|\mathcal{B}| < b$), whereas Rule C, D, E are applied every time a batch is full (i.e. only when $|\mathcal{B}| = b$). Rule C also seems to play an important part in eliminating nodes, not as much as B, but comparatively more than D or E which are rarely successful. This can also be explained by

Table 6.4: Comparison of different dominance rules for 15-job instances

	b= 2					b=3				
	total	Rule				total	Rule			
λ	nodes	B	C	D	E	nodes	B	C	D	E
1.5	233	3	3	14	0	11849	3061	320	316	42
1	3274	428	76	316	109	27009	7153	521	1097	285
0.5	3085	523	174	380	246	51287	14455	2002	3877	1171

	b=4					b= 5				
	total	Rule				total	Rule			
λ	nodes	B	C	D	E	nodes	B	C	D	E
1.5	42945	15327	267	198	6	85989	44868	408	192	6
1	84969	33924	2888	2590	263	111240	58200	1327	826	40
0.5	143027	63152	5145	7113	609	205459	112793	7366	5768	113

	b=6					b=7				
	total	Rule				total	Rule			
λ	nodes	B	C	D	E	nodes	B	C	D	E
1.5	74150	53407	809	89	0	101372	89239	1422	1	0
1	208305	152197	1606	236	7	128998	106282	2263	2	0
0.5	163878	119532	2302	867	24	149775	126121	2445	7	0

	b= 8					b=9				
	total	Rule				total	Rule			
λ	nodes	B	C	D	E	nodes	B	C	D	E
1.5	36780	29598	1559	0	0	26331	21199	410	0	0
1	52577	43832	2560	0	0	31812	25401	1645	0	0
0.5	138775	113206	2284	0	0	40698	30306	2277	0	0

	b=10				
	total	Rule			
λ	nodes	B	C	D	E
1.5	20426	17397	411	0	0
1	25008	20348	769	0	0
0.5	22510	18167	1247	0	0

Each entry is the average number of nodes over 10 instances

the fact that Rule C is checked before D and E. As a percentage of the total amount of nodes analysed 8% (for $b = 2, \lambda = 0.5$) seems to be the most Rule E comes into use. Rule D and E work better for smaller batch sizes (b form 2 to 6). This might be explained from the fact that this rules come into play only when several batches are present in the partial schedule, and this is less frequent when the batch size is bigger. Rule D however seems to out-perform rule C in eliminating nodes for $b = 3$.

The more restrictive the due dates (λ small), the more nodes that are generated. However this effect seems to diminish for bigger batch sizes (for $b = 9, 10$ the average number of nodes analysed are very similar). Another important factor is the running time, we present a comparative analysis for different sets of instances in the following section.

6.7.3 Analysis of results

In Table 6.5 we present results for the sets with 10 jobs. Each entry shows the average over the 10 instances generated for each set. The time is measured in seconds. The algorithm works very fast, it finds the optimum in less than a second on average. The more restrictive the due dates the more time (and nodes) it takes to solve the problems. Note that as the batch size increases the number of nodes analysed increases at first, reaching a maximum in $b = 5$ (hardest sets of problems), and then decreases. However, as b increases such a smooth (parabolic-like) behaviour is not found, it grows until $b = 5$, decreases slightly, and seems to stabilise for bigger batch sizes.

We show a similar summary of results for the sets of instance with 15 jobs in Table 6.6. We can observe the same smooth behaviour in the average number of nodes analysed as the batch size increases for the 15 job case. However, in this case the hardest problem seems to be for $\lambda = 0.5$ $b = 5$, for $\lambda = 1$ $b = 6$, and for $\lambda = 1.5$ $b = 7$. So that for more restrictive due dates, the hardest problems seem to appear for smaller batch sizes than for less restrictive ones. Our algorithm seem to perform very well for this number of jobs, as on average any instance does not take more than 5 minutes to solve.

Table 6.5: Results for 10-job instances

		10 Jobs					
		$\lambda = 1.5$		$\lambda = 1$		$\lambda = 0.5$	
		Time	Nodes	Time	Nodes	Time	Nodes
b	2	0.02	100	0.04	215	0.06	244
	3	0.13	635	0.26	1326	0.27	1318
	4	0.19	858	0.32	1523	0.37	1797
	5	0.25	1037	0.31	1320	0.46	2211
	6	0.24	710	0.30	1093	0.25	862
	7	0.32	653	0.32	671	0.34	760
	8	0.45	706	0.37	564	0.37	568

Each entry is average over 10 instances

The time measured in seconds

Table 6.6: Results for 15-job instances

		15 Jobs					
		$\lambda = 1.5$		$\lambda = 1$		$\lambda = 0.5$	
		Time	Nodes	Time	Nodes	Time	Nodes
b	2	0.05	233	0.86	3274	2.65	3085
	3	3.28	11849	6.69	27009	39.84	51287
	4	14.10	42945	93.55	84969	112.95	143027
	5	30.47	85989	115.71	111240	200.85	205459
	6	116.76	74150	255.39	208305	233.63	163878
	7	152.73	101372	134.14	128998	191.86	149775
	8	27.84	36780	91.86	52577	103.44	138775
	9	27.92	26331	65.22	31812	41.64	40698
	10	35.03	20426	112.75	25008	45.34	22510

Each entry is average over 10 instances

The time measured in seconds

Table 6.7: Results for 20-job instances

λ	20 Jobs								
	$b = 2$			$b = 3$			$b = 4$		
	Time	Nodes	Sol.	Time	Nodes	Sol.	Time	Nodes	Sol.
1.5	6.66	19644	10	377.54	699170	10	2739.84	2551025	10
1	41.88	40469	10	949.60	1021343	10	13446.12	4885584	6
0.5	50.68	46613	10	915.11	903874	10	8000.82	5678713	6

Sol. = number of optimal solutions found

The number of nodes is the average over 10 instances

The time is measured in seconds, and is the average over 10 instances

In Table 6.7 we show the summary for the set of scenarios with 20 jobs. Where the average for each entry was taken over the 10 instances of each problem. Recall that a time limit was set at 2 hours and 30 minutes for any particular instance. We have made explicit how many instances were solved within that time period, for the sets where not all instances were solved the average was taken over the ones solved. Again as the batch size increases the average time increases. However, the growth in this case is great than in previous sets. For $b = 4$ and $\lambda = 1.5$ the average instance takes 45 minutes to solve, and 4 instances for $\lambda = 1$ and $\lambda = 1.5$ were not solved within the time limit. The number of nodes analysed also increases, considerably more than before. An estimated one million nodes can be analysed in approximately 15 minutes. These instances are sufficiently hard to start using approximation algorithms. In fact, the results obtained form this set of instances are used to evaluate the local search heuristics in Chapter 7.

For the 25-job instance very few optima were reached within the 2 hours and 30 minutes time limit. Three optima were found for $b = 5$, 1 optimal for $b = 10$ and not one for $b = 15$. This results suggest that other techniques should be used if we want to obtain optimal solutions for instances with 25 or more jobs.

Table 6.8: Results for 25-job instances

λ	25 Jobs								
	$b = 5$			$b = 10$			$b = 15$		
	Time	Nodes	Sol.	Time	Nodes	Sol.	Time	Nodes	Sol.
1.5	5574	2456893	3	7496	5765501	1	9000	865220	0
1	9000	5370042	0	9000	8433876	0	9000	823178	0
0.5	9000	4897654	0	9000	7677432	0	9000	765322	0

Sol. = number of optimal solutions found

The number of nodes is the average over 5 instances

The time is measured in seconds, and is the average over 5 instances

6.8 Concluding Remarks

In this chapter we have presented a branch and bound algorithm to solve the BRMS problem. Our algorithm finds optimal solutions in a reasonable time for instances of up to 20 jobs. We know the problem is NP-hard, but we were hoping to solve bigger instances. Since largest instances occur in practice the need to develop local search heuristics is apparent.

On the other hand, we have shown that we can reduce the search for an optimal solution to EDD-batch schedules. We develop a polynomial time dynamic programming algorithm to calculate a lower bound at each node in the search tree, and describe several dominance rules that are effective in reducing the number of nodes generated. We should point out that this lower bound calculation can also be employed to obtain a lower bound on the maximum lateness of any schedule. In fact, we will use it to provide estimates of the quality of solutions generated by the local search heuristics developed in the next chapter.

We could try improving the branch and bound algorithm by devising a new branching scheme, or developing new lower bounds. Originally we have tried a Lagrangian relaxation approach, but due to the form of the objective function our attempts did not lead to an efficiently solvable relaxed

problem. Even though an optimal schedule for the restricted version of the problem need not be an SPT-batch schedule, we could use the solution for the unrestricted batch size to provide a lower bound on the maximum lateness. It would be interesting to compare this bound with the one we propose. Another avenue for further research is to try and obtain reductions on the time complexity of the lower bound calculation, using efficient data structures like the ones suggested by Wagelmans & Gerodimos (2000).

Chapter 7

Local Search Heuristics

7.1 Introduction

In this chapter we present several local search heuristic for the BMRS problem introduced in Chapter 5 Section 5.5. As explained, the BMRS problem is one of scheduling jobs on a single batching machine with restricted batch size to minimise the maximum lateness. A batching machine is one that can process several jobs at the same time in a batch. In the BMRS problem there is an maximum number of jobs any batch can have. The processing time of a batch is given by the biggest processing time of any job in the batch. We will use the same notation introduced in Chapters 2 and 5. Though, when it is convenient we will talk about job j instead of job J_j, and thus express the sequence of jobs $(J_1, \ldots, J_n)$ by the vector $(1, \ldots, n)$. There are n jobs to schedule $\mathcal{J} = \{J_1, \ldots, J_n\}$, and b denotes the maximum batch size. A schedule σ is determined by a sequence of batches $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$ with no more than b jobs in each batch $\mathcal{B}_l$, $l = 1, \ldots, r$. Applications can be found in the 'burn in' operations in manufacturing of circuit boards, and chemical processes that occur in tanks or kilns.

This BMRS problem is a unary NP-hard problem as shown by Brucker et al. (1998). Not only that, but instances with more than 25 jobs are difficult to solve with the branch and bound algorithm presented in the previous

chapter. All of which justifies the implementation of local search heuristics to find approximate solutions for bigger instances of the problem. The aim of this chapter is to asses the potential of local search heuristics for finding good quality solutions for the BMRS problem. We design a new neighbourhood structure, which is exponentially sized. We refer to it as the split-merge neighbourhood. In particular, we evaluate the use of a swap, an insert, and a split-merge neighbourhood for the problem, in a simple descent heuristic, and a multi-start descent heuristic. We have also developed an iterated descent heuristic in the split-merge neighbourhood which yields the best results.

As explained in Chapter 5, Section 5.4, any local search heuristic has an associated neighbourhood structure. The structure determines which solutions are close to each other (i.e. neighbours), and the search for better solutions will be done over these neighbours. In section 7.2 we present some popular neighbourhoods for sequencing problems, and explain how we can use them to design neighbourhoods for the BMRS problem. In section 7.3 we briefly review recently developed neighbourhoods that are exponentially sized. In section 7.4 we present an exponential neighbourhood (split-merge) for our problem. In section 7.5 we present our computational results, and comparisons for the different local search heuristics developed. Section 7.6 concludes the chapter, and outlines possibilities for further research.

7.2 Classical Neighbourhoods

In Chapter 5, Section 5.4, we introduced the concept of a neighbourhood $\mathcal{N}$, as a function that gives for each solution i (of an instance of a combinatorial optimisation problem) a set $\mathcal{N}(i)$ of solutions that are close to i. However, we have not shown explicitly any particular one for scheduling problems. As explained in Section 5.1 it is common to express an instance of a combinatorial optimization problem by a compact data representation. Such a data representation for some scheduling problems (see Chapter 2) is a sequence of jobs. We now present some classical neighbourhood structures for problems

whose solutions can be expressed as a sequence.

A class of popular neighbourhoods for problems whose solutions can be expressed as a sequence is a k-exchange ($k \geq 2$) neighbourhood. This neighbourhood is composed of all solutions that can be obtained by exchanging k elements of a given sequence. For $k = 2$ the neighbourhood is usually referred to as *swap neighbourhood* (also known as a pair-wise interchange neighbourhood). For example the sequence $(1, 4, 3, 2, 5, 6)$ is a neighbour of sequence $(1, 2, 3, 4, 5, 6)$, obtained by exchanging job 2 with job 4. Verifying local optimality for a k-exchange neighbourhood requires $\Omega(n^k)$ time, where n is the total number of elements. The more exchanges we perform (the bigger the value of k) the more computationaly expensive it becomes to search the neighbourhood. However, when using a simple descent algorithm, the quality of the solution is better. The *size of a neighbourhood* is the number of neighbours for a single solution. For example, the size of the swap neighbourhood ($k = 2$) is $n(n-1)/2$.

Another popular neighbourhood is the *insert neighbourhood.* The insert neighbourhood is composed of all solutions that can be obtained by taking an element and inserting it in another position within the sequence. For example sequence $(1, 5, 2, 3, 4, 6)$ is a neighbour of sequence $(1, 2, 3, 4, 5, 6)$, obtained by taking job 5 and inserting it before job 2. The size of this neighbourhood is $(n-1)^2$. A *block* insertion is when we take a set of more than one element (referred to as a block) and insert it in another position within the sequence. For example sequence $(5, 6, 1, 2, 3, 4)$ is a neighbour of sequence $(1, 2, 3, 4, 5, 6)$, obtained by taking the block with jobs (5, 6) and inserting it before job 1. If we fix the block to be of size m ($m \geq 2$), then the size of the neighbourhood with n jobs is $(n - m + 1)(n - m)$.

A schedule for the BMRS problem can be determined by a job sequence and a partition of the sequence into batches. For example, a particular sequence of 6 jobs might be $\pi = (J_1, J_4, J_3, J_2, J_5, J_6)$, and if $b = 3$ a feasible batching is $\{J_1 J_4\}\{J_3 J_2 J_5\}\{J_6\}$ (the first batch with jobs 1 and 4, the second batch with jobs 2, 3 and 5, finally the third batch with job 6). Recall that

Brucker et al. (1998) gave a dynamic programming algorithm to solve the unrestricted version of the problem (where $b \geq n$), which is explained in more detail in Section 5.5.2. In broad terms a solution was obtain by sorting jobs in SPT (a fixed sequence), and then using the dynamic programming algorithm. We now adapt their algorithm to obtain for any given (fixed) sequence of jobs $\pi = (J_1, \ldots, J_n)$ a partition of the sequence into batches, with no more than b jobs in a batch, that will yield the minimum value for the maximum lateness.

Let F_j be the minimum value of the maximum lateness of a schedule that contains jobs $J_j, \ldots, J_n$ from π, where the processing of the first batch in the schedule starts at time zero . If a batch $\{J_j, \ldots, J_{k-1}\}$, which has processing time $p' = \max_{j \leq l \leq k-1} p_l$, is inserted at the start of a schedule for jobs $J_k, \ldots, J_n$, then the maximum lateness of jobs $J_k, \ldots, J_n$ increases by p', while the maximum latness for jobs $J_j, \ldots, J_{k-1}$ is $\max_{j \leq l \leq k-1} p_l - \min_{j \leq l \leq k-1} d_l$ (the processing time of the batch minus its due date). We can now state the dynamic programming recursion. The initialisation is

$$F_{n+1} = -\infty,$$

and the recursion for $j = n, n-1, \ldots, 1$ is

$$F_j = \min_{j < k \leq \min\{n+1, j+b\}} \{\max\{F_k + \max_{j \leq l \leq k-1} p_l, \max_{j \leq l \leq k-1} p_l - \min_{j \leq l \leq k-1} d_l\}\}. \quad (7.1)$$

The optimal solution value is given by F_1, and the corresponding schedule can be found by backtracking.

Note how similar this formulation is to the one given in Section 5.5.2 (Brucker et al. 1998). The minimum is calculated over a smaller set as we can only fit b jobs into any batch. This is expressed in the range of the first (outer-most) min calculation, instead of the range for k being $j < k \leq n+1$ it is $j < k < \min\{n+1, j+b\}$. Another change is that jobs in sequence π are not necessarily ordered in SPT order. Hence, we need to calculate the processing time of the batch $\{J_j, \ldots, J_{k-1}\}$ inserted at the beginning of the schedule, which is computed as $\max_{j \leq l \leq k-1} p_l$.

We can rewrite the recursion formula (7.1) as

$$F_j = \min_{j<k\leq \min\{n+1,j+b\}} \{ \max_{j\leq l\leq k-1} p_l + \max\{F_k, - \min_{j\leq l\leq k-1} d_l\}\}.$$

Define $G_k = \max_{j\leq l\leq k-1} p_l + \max\{F_k, - \min_{j\leq l\leq k-1} d_l\}$, then

$$F_j = \min_{j<k\leq \min\{n+1,j+b\}} G_k.$$

Note that G_k can be computed for $k = j+1, \ldots, \min\{n+1, j+b\}$, in that order. Hence we can store the previous value of the $\max p_l$ and $\min d_l$ as k progresses on its range, to reduce the number of calculations to constant time for each value of k. Note that for each F_j there are at most b G_k's to compare, and n values of j. Hence, this is an $O(nb)$-time algorithm.

For any feasible schedule σ of the BMRS problem, there is an underlying permutation of jobs π. We say a schedule σ' is a swap (or insert) neighbour of schedule σ if the underlying permutation π of σ is a swap (or insert) neighbour of the underlying permutation π' of σ'. Recall that the swap and insert neighbourhoods are of size $O(n^2)$. Hence, searching the swap or insert neighbourhood of a schedule σ of the BMRS problem will take $O(n^3 b)$ time.

Summarising, in this section we have introduced the swap and insert neighbourhoods. We have also shown that they can be used efficiently (in $O(nb)$ time) via a dynamic programming algorithm to evaluate and obtain solutions to the BMRS problem. This allows us to define both swap an insert neighbourhood structures for the BMRS problem, and tackle them with any of the local search heuristics presented in Chapter 5.

7.3 Exponential Neighbourhoods

In the previous section we defined the size of a neighbourhood as the number of neighbours a single solution has (i.e. $|\mathcal{N}(i)|$). The swap and insert neighbourhoods are polynomial sized neighbourhoods. As mention in chapter 5 it is believed that a bigger neighbourhood often yields better quality solutions. At the same time, the larger the neighbourhood, the longer it

usually takes to search the neighbourhood. A neighbourhood is said to be *exponential* if its sizes grows exponential with respect with the size of the instance. We would think such a neighbourhood to be very time consuming to explore. However, recently some exponential neighbourhoods have been analysed that surprisingly can be be searched in polynomial time.

Congram, Potts & Van de Velde (1998) introduce an exponential neighbourhood (dynasearch swap neighbourhood) for the single machine total weighted tardiness problem, and an iterated local search algorithm which yields the best results found so far in the literature. Dynasearch is equivalent to performing a series of 'independent' swap moves. Moreover, Congram (2000) uses a dynasearch neighbourhood for the linear ordering problem and the traveling salesman problem (TSP), exploring the neighbourhoods in polynomial time, with good results. Hurink (1999) has analysed an exponential neighbourhood for a single batching machine to minimise the total weighted completion time. His model differs form ours in that the processing time of each batch is defined as the sum of the processing time of the jobs that belong to that batch. He uses a multiple transpose neighbourhood with a tabu search heuristic.

Exponential neighbourhoods are also referred to as Very Large-Scale Neighbourhoods (VLSN) by Ahuja, Ergun, Orlin & Punnen (1999). They, distinguish three main methods of exploring VLSN neighbourhoods: (a) partially search using heuristics (variable-depth methods); (b) Network flow based improvement algorithms; (c) polynomial time heuristics for NP-hard problems. Under their classification the neighbourhood used by Hurink (1999) comes under category (b) as he presents the efficient search of the neighbourhood as solving a shortest path problem. Most of the exponential neighbourhoods, however, have been developed for the TSP problem.

In Section 7.4 we present an exponential neighbourhood for the BMRS problem. We call it the split-merge neighbourhood, and we prove that it can be searched in polynomial time.

7.4 Split-Merge Neighbourhood

In this section we present an exponential neighbourhood, which we call split-merge neighbourhood. We calculate its size, and show how it can be explored in polynomial time. We give a dynamic programming algorithm for using the neighbourhood to evaluate and generate solutions for the BMRS problem.

We introduced in section 7.2 the insert neighbourhood in which one item or block is moved. A *multiple insert* move will be one where more than one insert move is allowed to form a neighbour. We expect a multiple insert neighbourhood to be a more powerful neighbourhood than simple insert neighbourhood.

The *split-merge neighbourhood* is a restricted version of a general multiple insert neighbourhood. It is restricted as certain insert moves are not allowed. Let π_0 be an initial sequence. In the split-merge neighbourhood $\mathcal{N}$, certain jobs of the neighbour of sequence π_0 will preserve the order they had in sequence π_0. The idea is to split sequence π_0 into two disjoint subsequences, say π_1 and π_2, and then merge them to obtain a neighbour $\pi \in \mathcal{N}(\pi_0)$ (i.e. $\{J_j|J_j \in \pi_1\} \cap \{J_j|J_j \in \pi_2\} = \phi$, $\{J_j|J_j \in \pi_1\} \cup \{J_j|J_j \in \pi_2\} = \{J_j|J_j \in \pi_0\}$ $= \{J_j|J_j \in \pi\}$, where $\pi \in \mathcal{N}(\pi_0)$ is the neighbour). The merge operation is a multiple insert move, while the splitting operation restricts the feasible moves. Specifically, if job j goes before job i in the same split sequence (either π_1 or π_2), then job j will continue to go before i in the merged sequence π. Also if a job j goes before i in the original solution π_0, and it happens that both belong to the same split sequence (either π_1 or π_2), then they will keep that order (job j sequenced before job i) in the split sequence. This last condition ensures that sequence $\pi_0 \in \mathcal{N}(\pi_0)$ (the initial sequence is a neighbour of itself).

For example, suppose we have $\pi_0 = (1, 2, 3, 4, 5, 6)$, choose $\pi_1 = (2, 4, 5)$, and $\pi_2 = (1, 3, 6)$. Then, a feasible merge is $\pi = (1, 3, 2, 4, 6, 5)$, which is equivalent to performing the following two insert moves in π_0: insert job 3 before job 2, and insert job 5 after job 6. Note how the order of jobs in π, preserves the order of jobs in π_1 and π_2. However, under this choice of

π_1 and π_2, sequence $\pi' = (1, 5, 6, 2, 3, 4)$ is not a neighbour of π_0 as job 4 goes before job 5 in π_1, and job 3 goes before 6 in sequence π_2. On the other hand, if we had chosen $\pi_1 = (1, 2, 3, 4)$ and $\pi_2 = (5, 6)$ this would be a valid merge operation, and $\pi' \in \mathcal{N}(\pi)$. Note that π' could be obtained from inserting block $(5, 6)$ in front of job 2. Hence, different splitting procedures yield different restrictions on the possible neighbours for a specific sequence.

Summarising, the split-merge neighbourhood is a restricted multiple insert neighbourhood, where the restrictions are given by the relative order of jobs in the split sequences, and initial sequence as explained above. The two determining factors for the efficiency of the split-merge neighbourhood are (i) the splitting procedure, and (ii) the merging procedure.

We show in Section 7.4.2 that given the split of the initial sequence a merged sequence and a feasible batching to minimise the maximum lateness of the resulting schedule can be obtained in polynomial time with a dynamic programming algorithm. That is, a collection of restricted multiple insert moves to minimise the maximum lateness (local optiumum) can be obtained in polynomial time in the split-merge neighbourhood.

It remains to see what a good splitting procedure will be. Note that the split sequences can be of un-equal size; if we have a sequence, say π_1, with just one job, then the split-merge neighbourhood is equivalent to a simple insert neighbourhood where we only allow the job in sequence π_1 to be moved. We show in Section 7.4.1 that sequences with (near) equal number of jobs are the best choice in terms of neighbourhood size.

7.4.1 Neighbourhood Size

To calculate the size of the split-merge neighbourhood, consider the two sequences we split, π_1, and π_2. Let n_1 be the number of jobs in sequence π_1, and n_2 in sequence π_2. The job in the i^{th} position in sequence π will be denoted by $\pi(i)$. Once sequences π_1 and π_2 have been specified, any feasible merged sequence π can be represented as a binary string Γ of size n, where

the entry in the i^{th} position, $\Gamma(i)$, takes the following values,

$$\Gamma(i) = \begin{cases} 1 & \text{if } \pi(i) \text{ belongs to sequence } \pi_1, \\ 0 & \text{if } \pi(i) \text{ belongs to sequence } \pi_2. \end{cases}$$

There are $\binom{n}{n_1}$ different Γ binary strings (equivalent to $\binom{n}{n_2}$). Hence, there are $\binom{n}{n_1} = \binom{n}{n_2}$ different sequences π that can be obtained from merging sequence π_1 and π_2. Note that $\binom{n}{n_1} = \frac{n!}{(n-n_1)!\ n_1!} \leq \min\{n^{n_1}, n^{n_2}\}$. Hence, the biggest possible neighbourhood is obtained for $n_1 = \frac{n}{2}$. The neighbourhood size is $\Theta(2^n)$. Neighbourhoods of such a large size can be explored in polynomial time, as explained in section 7.4.2. Furthermore, as larger neighbourhoods tend to yield better quality solutions (better local optima), we are interested in using similarly sized split sequences to have as big a neighbourhood as possible. We have tried different splitting procedures, and compare them in section 7.5.2.

7.4.2 Merging Sequences

We propose the following backward dynamic programing recursion to find a merged sequence, and form batches of jobs with the minimum value of the maximum lateness.

Let $J'_1, \ldots, J'_{n_1}$ be the n_1 jobs in the first sequence, and $J''_1, \ldots, J''_{n_2}$ the n_2 jobs in the second sequence. Let schedule $\sigma(j_1, j_2)$ be a partial schedule with jobs $J'_{j_1}, \ldots J'_{n_1}$ from the first sequence and jobs $J''_{j_2}, \ldots, J''_{n_2}$ from the second sequence, merged in such a way that the maximum lateness is minimized, and the relative order of the jobs in each sequence is preserved as explained before. Let $L(j_1, j_2)$ be the value of this maximum lateness. Then,

$$L(j_1, j_2) = \min_{\substack{j_1 \leq k_1 \leq n_1 + 1 \\ j_2 \leq k_2 \leq n_2 + 1 \\ 1 \leq k_1 - j_1 + k_2 - j_2 \leq b}} \{\max\{L(k_1, k_2) + \max_{J_j \in \mathcal{J}} p_j, \max_{J''_j \in \mathcal{J}} p_j - \min_{J_j \in \mathcal{J}} d_j\},$$

where $\mathcal{J} = \{J'_{j_1}, \ldots, J'_{k_1-1}, J''_{j_2}, \ldots, J''_{k_2-1}\}$ is the batch containing the union of the jobs with indices from $j_1, \ldots, k_1 - 1$ in the first sequence, and indices from $j_2, \ldots, k_2 - 1$ in the second sequence. The maximum lateness of the best

merge is given by $L(1,1)$, and the optimal batching (and merged sequence) can be obtained by backtracking over L.

Note how similar this formulation is to the ones presented before for the problem with restricted batch size. $\mathcal{J}$ is the batch that will be inserted at the beginning of the schedule $\sigma(j_1, j_2)$. The maximum batch size restriction is expressed in the range $1 \leq k_1 - j_1 + k_2 - j_2 \leq b$.

We will evaluate at most b^2 sets $\mathcal{J}$, and there are at most n^2 values of $L(j_1, j_2)$. Hence, this an $O(b^2n^2)$-time algorithm. For a given split of the underlying sequence, exploring the split-merge neighbourhood takes $O(b^2n^2)$ time. We compare this neighbourhood with the swap and insert neighbourhood with different local search heuristics in the next section.

7.5 Computational Experience

7.5.1 Test Problems and Experimental Design

We have used the same set of data generated for instances with 20 jobs, explained in section 6.7, to compare the different local search heuristics we propose. The processing times of the jobs are uniformly distributed from 1 to 100. An estimate on the makespan of the schedule, given the processing times and the batch size b, is $T = \frac{1}{b}\sum_{j=1}^{n} p_j$. An obvious choice for the distribution of the due dates is a uniform distribution from 1 to T. However, to mimic different scenarios for the due dates, we have used a parameter λ that takes the following values: $0.5, 1, 1.5$, and due dates are distributed uniformly between 1 and λT. Hence, $\lambda = 0.5$ is a restrictive due dates scenario and $\lambda = 1.5$ a slack one. Recall that for the 20-job set of instances the maximum batch size is $b = 2, 3$, or 4, and we have 10 instances for each choice of b and λ.

We also generate new sets of data for 50 and 80 jobs to try out the iterated descent algorithm in the split merge neighbourhood we propose in section 7.5.5. The jobs and due dates of this set of instances follow the same distributions explained above. We have choose for $n = 50$ maximum

batch size of $b = 2, 5, 10$, and 25, and for $n = 80$, $b = 5, 10, 15, 20$, and 40. We generated 5 instances for each choice of b, and λ. All our heuristics were encoded in Turbo Pascal and run on a PC with a Pentium 200MHz processor and 32Mb of ram.

7.5.2 Comparing different splitting procedures

In this section we present several splitting procedures, and evaluate their performance using a descent with the split-merge neighbourhood. Recall that the splitting operation is vital in the performance of the split-merge neighbourhood, as it will determine forbidden insert moves. We have tried several procedures to split the sequence, both deterministic and non-deterministic ones. Recall that it is desirable to have similarly sized split sequences. In most of our split procedures we aim for this; however we allow for some to have different sizes, as basis of comparisons.

We propose the following deterministic procedures to split a given sequence π:

S1 *Splitting sequence by half* This is the simplest procedure to split a sequence into two equally sized subsequences. Take jobs $\pi(1), \ldots, \pi(\lfloor \frac{n}{2} \rfloor)$ assign them to the first sequence, and jobs $\pi(\lfloor \frac{n}{2} \rfloor + 1), \ldots, \pi(n)$ to the second sequence.

S2 *Index odd-even splitting* This is another procedure that aims at obtaining equally sized subsequences, assigning jobs one by one alternatively between the two split sequences. Construct one sequence with jobs $\pi(1), \pi(3), \pi(5), \ldots$, (odd index), and another sequence with jobs $\pi(2)$, $\pi(4)$, $\pi(6), \ldots$ (even index).

The non-deterministic procedures to split the sequence we propose are:

S3 *Split balanced parts* This procedure aims at constructing equally sized subsequences. Generate t a random number between 1 and $\frac{n}{2}$. Assign jobs $\pi(1), \ldots, \pi(t)$ to the first sequence, and if $t \neq \lfloor \frac{n}{2} \rfloor$ assign jobs

$\pi(t), \ldots, \pi(\lfloor \frac{n}{2} \rfloor)$ to the second sequence. Assign jobs $\pi(2\lfloor \frac{n}{2} \rfloor - t + 1)$, $\ldots, \pi(n)$ to the second sequence, and if $t \neq \lfloor \frac{n}{2} \rfloor$ assign $\pi(\lfloor \frac{n}{2} \rfloor + 1)$, $\ldots, \pi(2\lfloor \frac{n}{2} \rfloor - t)$ to the first sequence. If $t = \lfloor \frac{n}{2} \rfloor$, then this sequence is equivalent to S1. However, it generally splits sequence π into 4 different parts, and we expected it to behave more like procedure S4 with $\tau = 4$ (see below).

S4 *Split randomly into τ parts* Generate a random number t distributed uniformly between 1 and $2\frac{n}{\tau} - 1$ (i.e $t \sim U[1, 2(\frac{n}{\tau}) - 1]$); $\tau \geq 2$. The expected value of t is n/τ. Our aim is to divide the sequence on average into τ different parts and assign them alternatively between the two subsequences. Let $t_0 = 0$, and $t_1 = \min\{t, n\}$. We assign jobs $\pi(t_0 + 1), \ldots, \pi(t_1)$ to the first sequence. If we have not assigned all jobs of π, generate another random number $t \sim U[1, 2(\frac{n}{\tau}) - 1]$, let $t_2 = \min\{t_1 + t, n\}$, assign jobs $\pi(t_1 + 1), \ldots, \pi(t_2)$ to the second sequence. Continue to generate t and calculating $t_i = \min\{\sum_{k=1}^{i-1} t_k + t, n\}$ to assign jobs $\pi(t_{i-1} + 1), \ldots, \pi(t_i)$ alternatively between the first and second subsequence until the last job $\pi(n)$ of the original sequence has been assigned.

S5 *Split sets of jobs at random* In procedure S4 we divide the sequence into different parts $(\pi(t_{i-1} + 1), \ldots, \pi(t_i))$ and assigned them alternatively; these parts might contain different numbers of jobs. By contrast, in this procedure we want to fix the number of jobs in each part and assign them randomly between the two sequences. The procedure assigns with certain probability ρ $(0 < \rho < 1)$ a set of μ jobs to the first sequence; otherwise it assigns these jobs to the second sequence. Note that $\mu = \lfloor \frac{n}{2} \rfloor$ and $\rho = 0.5$ will yield on average an equivalent split to S1. If $\mu = 1$ and $\rho = 0.5$ we also expect a similar behaviour to S2, though in this case the assignment is random (and we can have a mixture of odd and even indexes on the same split sequence).

The heuristic we used to compare the different splitting procedures is as follows.

Descent Heuristic in split-merge neighbourhood

begin

– Construct an initial sequence π' where jobs are in EDD order.

– Obtain an optimum batching and calculate the maximum lateness $L(\pi')$ for sequence π' using the dynamic programming algorithm explained in Section 7.2, equation (7.1).

repeat

- $\pi = \pi'$, $L(\pi) = L(\pi')$

- Split π into two subsequences with procedure S, one of the splitting procedures mentioned above.

- Merge subsequences with the dynamic programming algorithm explained in Section 7.4.2, to obtain a new sequence π', and optimal batching with maximum lateness $L(\pi')$.

until $L(\pi) = L(\pi')$

end

Think of π as the initial solution. The subsequences obtained after splitting sequence π determine feasible merges that can be performed. Applying the algorithm explained in Section 7.4.2 guarantees a resulting sequence π' whose optimal batching has the minimum maximum lateness of all feasible merges. When we use any of the splitting procedures explained previously we guarantee that sequence π is one of the feasible merges. Thus, the maximum lateness of the optimal batching for the new sequence π' is never worst than the maximum lateness of the optimal batching of π. Our aim is to find the best splitting procedure. We consider going from sequence π to π' as a descent move in the split-merge neighbourhood. Initial runs with this descent heuristic for the 20-job set of instances did not give clear evidence on the best splitting procedure. In fact, few (global) optima for the 20-job

BMRS problem were found, and a move (the repeat-until loop) was performed only once or twice. In some sense, what this is telling us is that the split-merge neighbourhood has a flat landscape making it hard to escape the first local optimum found. To avoid this we substituted the stopping criterion $L(\pi) = L(\pi')$ with a criterion to stop after a time limit. We have set this time limit at 0.07 seconds, which is the time it takes to perform on average a first improve descent in the insert neighbourhood (see Section 7.5.3). With this stopping criterion we allow moves where the maximum lateness is not strictly decreasing (i.e. we allow moves to sequences with the same maximum lateness as the previous sequence), and thus making it easier to compare the splitting procedures.

In table 7.1 we show a summary of results comparing the different splitting procedures S1, S2, S3, S4 and S5. For procedure S4 we show results for $\tau = 4, 5, 6$, and 7. For procedure S5 we show results for $\rho = 0.5, 0.7$, and 0.9 and $\mu = 2, 4, 5$. Note that $\rho = 0.7$ is equivalent to $\rho = 0.3$, and more generally S5 with $\rho = k$ is equivalent to $\rho = 1 - k$. Recall we have 10 instances for each pair b and λ. We decided only to show the behaviour over the different due date scenarios (from a slack one, $\lambda = 1.5$, to a restrictive one $\lambda = 0.5$), and summarise the behaviour over different batch sizes (i.e. $b = 2, 3, 4$). Hence, each entry is the average over 30 different 20-job instances. The overall behaviour for each splitting procedure is summarised in the last column. We use the following statistics:

- DV: average percentage deviation from global optima (30 instances),
- NO : number of global optima found (30 instances),
- ODV: average percentage deviation from global optima (90 instances),
- TNO : total number of global optima found (90 instances),

where the percentage of deviation from global optima is calculated as

$$\Big(\frac{L_{\max}(\text{ Local Optimum }) - L_{\max}(\text{ Global Optimum })}{|L_{\max}(\text{ Global Optimum })|}\Big)100,$$

and the local optimum is the one found by the heuristic with the chosen splitting procedure. Recall that the values of the maximum lateness of the

global optima were found for the 20-job instances by the branch and bound algorithm of Chapter 6.

From table 7.1 we see that more global optima are found for less restrictive due dates ($\lambda = 1.5$). These are also the easier problems to solve when running the branch and bound algorithm (Chapter 6). The deterministic procedures (S1, and S2) yield fewer global optima than the non-deterministic procedures. With non-deterministic splitting procedures we are less likely to have the same split sequences from one iteration to the next, thus changing the forbidden insert moves. It seems that the randomness of the non-deterministic procedures allows for diversity in the search. Hence, we prefer non-deterministic splitting procedures.

Splitting procedure S2 performs better than S1. In S1 we keep the first jobs $1, \ldots, \lfloor \frac{n}{2} \rfloor$ together which were consecutive in the initial sequence (the same for the last $n - \lfloor \frac{n}{2} \rfloor$ jobs.) This suggest that leaving a big list of consecutive jobs unchanged from the initial sequence to the split sequences is not desirable. Note that S2 does not leave any consecutive pair of jobs (j and $j + 1$) in the same split sequence. However, this also seems to do poorly. It seems to be desirable to leave at least some consecutive jobs from the initial sequence together in a split sequence. This happens in all other procedures.

For $n = 20$, procedure S5 with $\mu = 4$ will divide sequence π into 5 sets of 4 jobs, so that the split sequences cannot be of the same size. As we have stated before, equally sized split sequences are preferred, this explains why procedure S5 with $\mu = 4$ performs poorly compared with $\mu = 2$ or $\mu = 5$. The best choice of parameters for procedure S5 seem to be $\rho = 0.5$ and $\mu = 2$; confirms that we want some consecutive jobs together and equally sized split subsequences. Procedure S3 appears to perform better than S5, this might be due to the fact that procedure S5 has similarly sized split sequences, but it also allows for consecutive jobs to be separated from each other in a random fashion.

The best results were obtained with splitting procedure S4. This pro-

Table 7.1: Comparing spilt procedures in the split-merge neighbourhood

20-job instances									
Splitting Procedure		Due Dates Scenarios						Overall Totals	
		$\lambda = 1.5$		$\lambda = 1.0$		$\lambda = 0.5$			
		DV	NO	DV	NO	DV	NO	ODV	TNO
S1		87.7	4	26.3	0	11.9	2	41.9	6
S2		43.3	7	17.4	0	12.2	1	24.2	8
S3		32.6	16	7.3	7	4.5	4	14.7	27
S4	$\tau = 4$	23.3	19	6.3	6	3.0	7	10.8	32
	$\tau = 5$	13.6	16	7.5	6	2.9	7	8.0	29
	$\tau = 6$	4.1	23	4.7	9	4.3	5	4.4	37
	$\tau = 7$	6.3	21	5.7	5	3.5	5	5.2	31
S5	$(\rho,\ \mu)$								
	(0.5, 2)	12.1	14	7.4	5	3.1	5	7.5	24
	(0.7, 2)	14.5	12	6.9	3	3.8	6	8.4	21
	(0.9, 2)	18.1	14	10.2	4	5.4	1	11.2	19
	(0.5, 4)	32.9	7	15.3	0	5.6	1	17.9	8
	(0.7, 4)	32.1	4	13.8	0	4.9	2	16.9	6
	(0.9, 4)	67.5	6	11.0	3	6.1	1	28.1	10
	(0.5, 5)	67.8	7	12.2	1	6.5	2	28.8	10
	(0.7, 5)	52.5	8	14.9	1	7.1	1	24.8	10
	(0.9, 5)	45.9	9	14.7	0	4.6	1	21.7	10

DV: average percentage deviation from global optima (30 instances)
NO : number of global optima found (30 instances)
ODV: average percentage deviation from global optima (90 instances)
TNO : total number of global optima found (90 instances)

cedure allows consecutive jobs to be separated and aims on average to have equally sized split sequences. The best choice of parameter for this procedure seems to be $\tau = 6$, as it not only found the greatest number of global optima, but also reduced the average percentage deviation from the global optima.

Hence, form now on in this chapter, when running any heuristics over the split-merge neighbourhood for the 20-job instances we will use the the splitting procedure S4 with $\tau = 6$ with the time limit stopping criterion.

7.5.3 Comparing different neighbourhoods with a simple descent heuristic

In this section we compare the different neighbourhoods presented for the BMRS problem. The swap and an insert neighbourhood for the BMRS problem are defined in Section 7.2. We use a simple descent heuristic as explained in Section 5.4. We can apply either a first improve or a best improve policy for descent. Hence, we can compare four heuristics for the classical neighbourhoods:

IFI: First improve descent in the insert neighbourhood,

IBI: Best improve descent in the insert neighbourhood,

SFI: First improve descent in the swap neighbourhood,

SBI: Best improve descent in the swap neighbourhood,

We use as starting solution a sequence of jobs in EDD order. We compare these heuristics with the descent heuristic in the split-merge neighbourhood as explained in Section 7.5.2, with S4 $\tau = 6$, and time limit stopping criterion of 0.07 seconds. This is the average time it takes to perform a first improve descent in the insert neighbourhood (see table 7.2).

As in the previous section we have used the same set of 20-job instances. We aggregate the behaviour over different batch sizes (i.e. $b = 2, 3, 4$) to show it over the different due date scenarios. Hence, each entry is the average over 30 different 20-job instances. The overall behaviour is summarised in the last row. To compare the performance of the different heuristics we use the

following statistics:

- NO: number of global optima found (30 instances),
- DV: average percentage deviation from global optima (30 instances),
- AT: average time per instance in seconds (30 instances),

The results are shown in table 7.2.

A first improve descent heuristic will stop when the first improving neighbour is found, and a best improve descent heuristic will stop after all neighbours have been analysed returning the best value. Hence, we expect a first improve descent heuristic to take less time than a best improve descent heuristic. This can be seen clearly in table 7.2, where we observe that the best improve descent takes more time. However, the average deviation from global optima is smaller for the best improve policy. This behaviour is more apparent in the insert neighbourhood, where the first improve policy has a 37% deviation whereas the best improve has a 19% deviation. We notice again that it is harder to find global optima for those instances with more restrictive due dates.

The split-merge neighbourhood has the best performance. It finds more global optima, and it had a smaller deviation from the optimum. Furthermore, the deviation from the optimum does not change with how restrictive the due dates are, as it did with the other heuristics. Note that it was less than 5% for any λ. Thus, it seems we have better quality solutions using a descent heuristic in the split-merge neighbourhood whatever the due date scenario. The descent heuristic in the split-merge neighbourhood found 37 out of the 90 global optima. That is only 41% of global optima; hence, it is of interest to apply other local search heuristics. We explore multi-start descent heuristics in the next section.

Table 7.2: Comparing Descent in Swap, Insert, and Split-Merge Neighbourhoods

20-job instances						
	IFI			IBI		
λ	NO	DV	AT	NO	DV	AT
1.5	13	91.13	0.05	14	37.17	0.08
1	4	10.57	0.08	5	12.77	0.12
0.5	2	9.83	0.08	2	8.1	0.12
Totals	19	37.18	0.07	21	19.34	0.11
	SFI			SBI		
λ	NO	DV	AT	NO	DV	AT
1.5	12	51.27	0.03	12	46	0.04
1	5	8.9	0.04	6	8.8	0.06
0.5	4	4.27	0.04	3	3.27	0.07
Totals	21	21.48	0.04	21	19.36	0.06
	S-M					
λ	NO	DV	AT			
1.5	23	4.13	0.07			
1	9	4.77	0.07			
0.5	5	4.33	0.07			
Totals	37	4.41	0.07			

IFI: First improve descent in the insert neighbourhood
IBI: Best improve descent in the insert neighbourhood
SFI: First improve descent in the swap neighbourhood
SBI: Best improve descent in the swap neighbourhood
S-M: Descent in the split-merge neighbourhood
NO: number of global optima found (30 instances)
DV: average percentage deviation from global optima (30 instances)
AT: average time per instance in seconds (30 instances)

7.5.4 Comparing different neighbourhoods with a multi-start descent heuristic

A multi-start descent heuristic is one where a local search heuristic is applied from different starting solutions, as explained in Section 5.4. In this section we compare different neighbourhoods with a multi-start heuristic.

Recall form section 7.5.3 that the best improve descent policy gave better results (both in terms of the number of optima found and deviation from global optima). Hence, we will only use this policy for the insert and swap neighbourhoods. We try the following three multi-start heuristics:

MIBI: multi-start with best improve descent in an insert neighbourhood,

MSBI: multi-start with best improve descent in a swap neighborhood,

MS-M: multi-start descent in the split-merge neighbourhood

We generated 10 random starting solutions (sequences), and run each of the three heuristics (MIBI, MSBI, MS-M) on them. Each random starting solution, yields a local optima, the best of this 10 is then taken as the final solution of the multi-start heuristic. As in the previous section we have used the same set of 20-job instances. We aggregate the behaviour over different batch sizes (i.e. $b = 2, 3, 4$) and show it over the different due date scenarios. Hence, each entry is the average over 30 different 20-job instances. The overall behaviour is summarised in the last row. To compare the performance of the different heuristics we use the following statistics:

- NO: number of global optima found (30 instances),
- AT: average time per instance in seconds (30 instances),
- DV: average percentage deviation from global optima (30 instances),
- NTW: number of times the heuristic found a solution that was no worse than the one found with the other (two) heuristics.

NTW can also be described as the number of times the heuristic finds a 'winning' solution, where by 'winning' solution we mean one that has the smallest value of all the values obtain with all the heuristics for that instance. The smallest value is not necessarily a global optimum. This statistic helps us identify how many times the heuristic arrives at comparatively good solu-

tions. The results are shown in table 7.3.

For multi-start we use randomly constructed sequences instead of the EDD ordered sequence we used before as starting solutions in Section 7.5.3. A good initial solution seems to be of importance for the insert neighbourhood, as observed by comparing with the results given in table 7.2. The overall results for the insert neighbourhood are better with EDD, even though in multi-start we are running the heuristic 10 times (one for each starting solution). In the insert neighbourhood the multi-start heuristic starting with random sequences finds 18 global optima, but using simple descent starting with the EDD sequence finds 21. This is more striking if we notice that the simple descent heuristic only took 0.11 seconds, and the multi-start took 1.6 seconds. Compared with the other two multi-start heuristics it found the winning solution in only 22% of the instances (NTW=20 out of 90 instances).

The multi-start heuristic in the swap neighbourhood, performs better than in the insert neighbourhood. It finds 27 global optima, and found the winning solution in 41% of the instances (NTW=37 out of 90). The starting solution does not seem to play such an important role for multi-start in the swap neigbhourhood, as it finds 6 more global optima than with simple descent and EDD starting sequences (compare with table 7.2). It seems to work well for instances with restrictive due dates ($\lambda = 0.5$). Our intuition suggests that a swap move allows batches to remain intact more often than an insert move, hence keeping more information about the solution. For example if we have the sequence $(1, 2, 3, 4, 5, 6, 7, 8)$, and a maximum batch size of $b = 3$ with the batches $\{1, 2, 3\}\{4, 5\}\{6, 7\}\{8\}$, and we swap 2 with 8, we might keep job 1 and 3 in the same batch and batches $\{4, 5\}\{6, 7\}$ intact; if instead we perform the simple insert move of shifting job 8 to the second position in front of job 2, job 1 cannot be in the same batch with job 3 as $b = 3$, and the batch structure in the middle might have to change.

The best heuristic is multi-start in the split-merge neighbourhood. It finds 77% of the global optima (70 out of 90), and found the winning solutions in 87% of the instances (NTW=79 out of 90). However, it did not find the

Table 7.3: Comparing different multi-start heuristics

20-job instances				
	MIBI			
λ	NO	DV	NTW	AT
1.5	11	34.4	12	1.8
1	6	7.3	7	1.6
0.5	1	4.4	1	1.4
Overall	18	15.3	20	1.6
	MSBI			
λ	NO	DV	NTW	AT
1.5	7	50.17	10	0.8
1	5	6.87	7	0.8
0.5	15	1.03	20	0.7
Overall	27	19.36	37	0.7
	MS-M			
λ	NO	DV	NTW	AT
1.5	26	2	30	0.7
1	22	0.73	24	0.7
0.5	22	0.33	25	0.7
Overall	70	1.02	79	0.7

MIBI: multi-start with best improve descent in an insert neighbourhood,
MSBI: multi-start with best improve descent in a swap neighborhood,
MS-M: multi-start descent in the split-merge neighbourhood
NO: number of global optima found (30 instances),
AT: average time per instance in seconds (30 instances),
DV: average percentage deviation from global optima (30 instances),
NTW: number of times the heuristic found the winning solution.

global optima for 20 instances, it is of interest to see if by keeping some information on previous solutions we can improve our search. We do so by trying iterated local search in the next section.

7.5.5 An iterated descent heuristic with the split-merge neighbourhood

As seen in the previous section the repetitive application of the split-merge heuristic (multi-start) gives the best overall results. Recall that an iterated local search heuristic (Section 5.4.1) also applies several times a local search, but rather than starting from random solutions, it restarts from a modified (or kicked) last local optimum. The idea is to dislodge the search from the current local search to look at other areas of the solution space. It also tries to keep information on previously good solutions by backtracking.

As the split-merge neighbourhood is a restricted version of the multiple insert neighbourhood, we have chosen a kick to be a certain number of random insert moves on the solution. The number of moves is referred to as the *size of the kick*. We want the kick to be sufficiently large to move to a solution that is not too close to the last local optimum, but not too far away that the good characteristics of the previous local optimum are lost, and we effectively have multi-start. As explained before, sometimes instead of using the last optimum to restart the local search, a backtracking procedure is employed, where after several local search runs, a kick is performed on the best overall local optimum, instead of the previous local optimum. The idea of backtracking is to guide the search back to solutions with good values.

There are two basic parameters we have to fix to find a best heuristic of this type: κ which is the number of insert moves (size of the kick), and β which is the number of times a local search is performed before backtracking to the previous best solution.

First to compare iterated with the multi-start, we ran each iterated local search for 0.7 second (the same time it took to analyse the 10 randomly generated instances), with varying values of $\beta = 2, \ldots, 8$ and $\kappa = 2, \ldots 6$.

The best parameter values are $\beta = 5$, $\kappa = 3$ for which the iterated local search found 72 global optima. This does not seem a very big improvement on the 70 global optima that multi-start finds; it is probably partly due to the fact that very few kicks were performed and the backtracking was used very little (once for $\beta = 5, 6, 7, 8$). We then tried a longer run time of 5 seconds, which gives approximately 45 kicks (keeping the stopping criterion of 0.07 seconds for each descent). We tried $\kappa = 3, 4, 5, 6, 7$, and $\beta = 5, 10, 15$. The best parameters are $\beta = 6$ and $\kappa = 10$, where 89 out of the 90 global optima were found. Running the heuristic for 10 seconds per instance yields all the optima. Thus, problems with 20 jobs seem to be easily tackled with this heuristic.

To compare on a consistent basis with respect to computational times we have run the multi-start descent heuristic for the same time (10 seconds) to compare with the iterated descent heuristic in the split merge neighbourhood. The results are shown in Table 7.4. We can see that not all optima were found, so we prefer the iterated heuristic to the multi-start.

Table 7.4: Results for multi-start in split-merge neighbourhood

20-job instances			
λ	NO	DV	AT
1.5	30	0	10
1	28	0.2	10
0.5	27	0.6	10
Overall	85	0.4	10

NO: number of global optima found (30 instances),

AT: average time per instance in seconds (30 instances),

DV: average percentage deviation from global optima (30 instances),

We now discus results for larger instances. Optimal solutions for 50 or 80 jobs are not easily obtainable. To evaluate the quality of the solutions obtained we compute

- The improvement on the starting solution π_{EDD} (sequence in EDD order), with IEED= $\frac{L_{\max}(\pi_{\text{EDD}})-L_{\max}(\pi_{LO})}{L\max(\pi_{\text{EDD}})}$, where π_{LO} is the local optimum obtained from the heuristic.

- The relative deviation from the lower bound of the schedule, as computed in Section 6.5, with DLB= $\frac{L_{\max}(\pi_{\text{LO}})-LB}{LB}$, where π_{LO} is the local optimum obtained from the heuristic and LB is the lower bound on the maximum lateness of any schedule using the dynamic programming formulation explained in Section 6.5.

Note that if DLB =0, then the local optimum is a global optimum. To compare the iterated descent heuristic in the split-merge neighbourhood for the larger instances we also developed an iterated descent heuristic for the swap neighbourhood. Detailed analysis of the solutions for the 20-job instances shows that multi-start in the swap neighbourhood sometimes finds better local optima than in the split-merge neighbourhood. Hence, it is of interest to compare them with an iterated descent heuristic. We ran the iterated heuristic with S4 $\tau = 15$, $\beta = 10$, $\kappa = 10$, for the 50-job instances explained in Section 7.5. Each entry is the average over the 5 instances generated. Each heuristic ran for 10 seconds on each instance. We use the following statistics:

- AIEED: Average improvement on the starting solution (5 instances),
- ADLB: Average deviation from the lower bound (5 instances),
- NO: number of global optima found (5 instances),
- NTW: number of times the winning solution is found in the neighbourhood, either in the split-merge or swap (over the 5 instances).

Our results are summarised in table 7.5

Table 7.5: Results for the 50 job instances

$\lambda = 1.5$	Split-Merge				Swap			
b	AIEDD	ADLB	NO	NTW	AIEDD	ADLB	NO	NTW
2	0	0	5	5	0	0	5	5
5	23	30	0	4	26	38	0	4
10	18	7	1	4	15	20	0	3
25	1	15	0	5	1	24	0	1
$\lambda = 1.0$	Split-Merge				Swap			
b	AIEDD	ADLB	NO	NTW	AIEDD	ADLB	NO	NTW
2	56	147	0	3	40	150	0	3
5	38	133	0	4	31	171	0	1
10	18	103	0	5	10	123	0	1
25	3	39	0	5	3	32	0	4
$\lambda = 0.5$	Split-Merge				Swap			
b	AIEDD	ADLB	NO	NTW	AIEDD	ADLB	NO	NTW
2	26	11	0	4	7	25	0	1
5	30	54	0	5	10	63	0	0
10	18	78	0	2	20	68	0	3
25	2	70	0	5	1	90	0	2
Total	19	57	6	51	13.6	67	5	28

AIEED: Average precentage improvement on the starting solution (5 instances),
ADLB: Average percentage deviation from the lower bound (5 instances),
NO: number of global optima found (5 instances),
NTW: number of times the winning solution is found in using this neighbourhood.

Global optima are found for some instances with slack due dates $\lambda = 1.5$, it seems that in these cases the batch size restriction is less important. In fact for $b = 2$ the EDD sequence also gives the global optimum (AIEDD=ADLB=0 for all 5 instances). It is interesting to note that the deviation from the lower bound was quite big for scenario $\lambda = 1.0$, it might be that the lower bound is not very tight. The best improvements to the starting solutions are found for smaller batch sizes ($b = 2$ and $b = 5$) in the split-merge neighbourhood. For batch size of $b = 25$ the EDD sequence is hardly improved by either heuristic as shown from a small value of AIEDD for all due date scenarios.

However the split-merge neighbourhood seems to perform well compared with the swap neighbourhood in total it finds 85% of the winning solutions (51 out of 60), where as swap finds 46% (28 out of 60). It also improves the starting solution better and has a smaller deviation from the lower bound overall instances. Still it is surprising to find that there were various cases where the winning solution was also found using the swap neighbourhood; in some cases in different instances. It might be interesting to develop a multilevel approach, where both neighbourhood structures (split-merge and swap) were used. As they are now the iterated descent heuristic in the split-merge neighbourhood wins over the iterated descent heuristic over the swap neighbourhood.

For $n = 80$ we ran the iterated descent heuristic in the split-merge neighbourhood with parameters $\tau = 10$, $\kappa = 40$, and $\beta = 15$, time limit per instance of 5 seconds. Again each entry is the average over 5 instances. The results are shown in table 7.6.

The results are similar to the ones found for 50 jobs. The only global optima were found for $\lambda = 1.5$, $b = 5$, where the initial solution (EDD sequence) is also optimal. As can be seen, for maximum batch sizes of $b = 40$ the local optima are not better than the initial solution (EDD sequence). The local optima seem to be further away from the lower bound for the 80-job instances than for the 50-job instances.

Table 7.6: Results for the 80 job instances

Iterated Heuristic in the Split-Merge Neighbourhood									
	$\lambda = 1.5$			$\lambda = 1.0$			$\lambda = 0.5$		
b	AIEDD	ADLB	NO	AIEDD	ADLB	NO	AIEDD	ADLB	NO
5	0.00	0.00	5	0.36	2.54	0	0.24	0.51	0
10	0.16	0.66	0	0.15	2.48	0	0.13	0.91	0
15	0.11	0.66	0	0.10	1.54	0	0.10	1.11	0
20	0.04	0.47	0	0.03	1.21	0	0.02	1.21	0
40	0.00	0.22	0	0.00	0.46	0	0.00	0.72	0

AIEED: Average improvement on the starting solution (5 instances),
ADLB: Average deviation from the lower bound (5 instances),
NO: number of global optima found (5 instances),

7.6 Concluding Remarks

In this chapter we have presented neighbourhood structures for the BMRS problem. In particular the exponentially sized split-merge neighbourhood has proven to be comparatively better for the problem than the insert or swap neighbourhood. However, the swap neighbourhood does seem to obtain better global optima sometimes. It would be desirable to to develop a multilevel approach, where both neighbourhood structures (split-merge and swap) were used, in the hopes of getting better solutions. Another avenue of further research would be the development of other local search heuristic (such as those described in Chapter 5), using the neighbourhood described in this chapter.

For the 20-job instances good quality solutions are found in fractions of a minute, whereas they require hours to solve with the branch and bound algorithm of Chapter 6. The split-merge neighbourhood seems to be good at exploring big areas of the search space, and gives good approximate solutions regardless of the due date scenario, as pointed out in Section 7.5.2.

The efficiency of the neighbourhood search relies on the dynamic programming algorithms to find solutions for the BMRS problem. We could probably

modify the $O(n^3)$ dynamic algorithm in Brucker et. al. (1998) for the total number of late jobs $\sum_{j=1}^{n} U_j$ to tackle the model with restricted maximum batch size which is unary NP-hard, much in the same way we did for the BMRS problem (also unary NP-hard).

Note that heuristics developed for the BMRS problem could also be modified to tackle the two batch machine flow shop problem where the first machine deals with batches of size 1 (classical machine) and the second is a restricted batching machine ($\tilde{F}2\ |b_1 = 1,\ b_2 < n|\ C_{\max}$). Recall the equivalence between $\tilde{1}\ |\ |\ L_{\max}$, and $\tilde{F}2\ |b_1 = 1|\ C_{\max}$ pointed out in Potts & Kovalyov (2000).

Chapter 8

Conclusions and Further Work

This thesis covered two areas of machine scheduling: lot streaming and batching machine scheduling. We used a wide range of techniques throughout this thesis. In the lot streaming chapters we have opted for a network and critical path analysis for the problems. These approaches are useful for finding optimal structures for the models analysed. We believe we have proven that a network representation for lot streaming models is a useful analysis tool. On the other hand, general purpose techniques have also been used, such as branch and bound and dynamic programming, to obtain exact solutions for the batching machine problem we study, and approximation methods such as the local search heuristics we presented in Chapter 7.

We have given a new approach to tackle the $F2|q_j, s_{ij}, t_j|C_{\max}$ model, which gives more insight into why it collapses to a much simpler problem. We think some extension to dominant machines might be possible. We present a form of such analysis for identical jobs in Chapter 4. We also introduce a new model where numbers of sublots are not predetermined. We explain how to add one extra sublot efficiently and presented a heuristic to allocate several sublots. There is plenty more research that can be done on this model. It would be of interest to find lower bounds on the reduction of makespan when adding a given number of sublots. The computational complexity of the model could also be analysed; NP-hardness would justify the development of

local search heuristics for the model.

We also developed solutions for the problem of scheduling jobs on a batching machine with restricted batch size to minimise the maximum lateness. We relied on efficient dynamic programming algorithms to find lower bounds on the maximum lateness of a schedule in the model, and to search efficiently an exponential neighbourhood. The results obtained for the split-merge neighbourhood are very encouraging. As pointed out previously, we could develop the neighbourhood further, by designing new local search heuristics, or trying a multi-level approach with the swap neighbourhood.

It would be of interest as well to try and develop new exponential neighbourhoods for batching problems that have polynomial time dynamic programming solutions when the batch size is not restricted. In particular we could tackle the single batching machine model with restricted maximum batch size and total number of late jobs $\sum_{j=1}^{n} U_j$ objective function.

Finally, both lot streaming and batching machine scheduling are extensions of classical machine scheduling models. We hope that this work has contributed to the development of further knowledge in this area. It has certainly given us many possibilities for further research.

Bibliography

Aarts, E. H. L., Korst, J. H. M. & Van Laarhoven, P. J. M. (1997), *Simulated Annealing* Local Search in Combinatorial Optimization, John Wiley & Sons Ltd., chapetr 4, pp 91-120.

Ahuja, R. K., Ergun, O., Orlin, J. B. & Punnen, A. P. (1999), 'A survey of very large-scale neighbourhood search techniques', MIT, MA 02139, USA To appear in *Discrete Applied Mathematics* .

Albers, S. & Brucker, P. (1993), 'The complexity of one-machine batching problems', *Discrete Applied Mathematics* **47**, 87-107.

Baker, K. (1974), *Introduction to Sequencing and Scheduling*, Wiley, New York.

Baker, K. R. (1988), Lot streaming to reduce cycle time in a flow shop. *Working Paper No. 203* Amos Tuck School of Business Administration, Dartmouth College, Hannover, USA.

Baker, K. R. (1995), 'Lot streaming in two-machine flow shop with setup times', *Annals of Operations Research* **57**, 1–11.

Baker, K. R. & Pyke, D. F. (1990), 'Solution procedures for the lot streaming problem', *Decision Science* **21** (3), 457-492.

Bogaschewsky, R. W., Buscher, U. D., & Lindner, G. (2001), 'Optimizing multi-stage production with constant lot size and varying number of unequal sized batches', *Omega* **29**, 183–191.

Brucker, P., Gladky, A., Hoogeveen, H., Kovalyov, M. Y., Potts, C. N., Tautenhahn, T. & Van de Velde, S. L. (1998), 'Scheduling a batching machine', *Journal of Scheduling* **1**, 31–54.

Carlier, J. (1982), 'The one-machine sequencing problem', *European Journal of Operational Reserach* **11**, 42–47.

Cetinkaya, F. C. (1994), 'Lot streaming in a two-stage flow shop with set-up, processing, and removal times separated', *Journal of the Operational Research Society* **45**(12), 1445–1455.

Chen, J. & Steiner, G. (1996), 'Lot streaming with detached setups in three-machine flow shops', *European Journal of Operational Research* **96**(3), 591–611.

Chen, J. & Steiner, G. (1997), 'Approximation methods for discrete lot streaming in flow shops', *Operations Research Letters* **21**, 139–145.

Chen, J. & Steiner, G. (1998), 'Lot streaming with attached setups in three-machine flow shops', *IIE Transactions* **30**, 1075–1084.

Chen, J. & Steiner, G. (1999), 'Discrete lot streaming in two-machine flow shops', *INFOR* **37**(2), 160–173.

Congram, R. K. (2000), Polynomial Searchable Exponential Neighbourhoods for Sequencing Problems in Combinatorial Optimisation, PhD thesis, University of Southampton.

Congram, R. K., Potts, C. N. & Van de Velde, S. L. (1998), An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem, Preprint OR 95, University of Southampton.

Conway, R. W., Maxwell, W. L. & Miller, L. W. (1967), *Theory of Scheduling*, Addison-Weseley Publishing Company.

Cook, S. (1971), 'The complexity of theorem-proving procedures', *Proceedings of the Third Annual ACM Symposium on the Theory of Computing* pp. 151–158.

Cook, W. J., Cunningham, W. H., Pulleyblank, W. R. & Schrijver, A. (1998), *Combinatorial Optimization*, John Wiley & Sons, USA.

Cormen, T. H., Leiserson, C. E. & Rivest, R. L. (1993), *Introduction to Algorithms*, MIT Press, Massachusetts, USA.

Dantzig, G. B. (1949), 'Programming of interdependent activities, II, mahtematical model', *Econometrics* **17**(3 & 4), 200–211.

Dauzère-Pérès, S. & Lasserre, J.-B. (1997), 'Lot streaming in job-shop scheduling', *Operations Research* **45**(4), 584–595.

Dreyfus, S. E. & Law, A. M. (1977), *The Art and Theory of Dynamic Programming*, Mathematics in Science and Engineering Vol. 130, Academic Press.

Dupont, L. & Dhaenens-Flipo, C. (2002), 'Minimizing the makespan on a batch machine with non-identical job sizes: an exact procedure', *Computers and Operations Research* **29**, 807-819.

Fisher, M. L. (1981), 'The lagrangian relaxation method for solving integer programming problems', *Management Science* **27**, 1–18.

Fisher, M. L. (1985), 'An application oriented guide to lagrangian relaxation', *Interfaces* **27**, 10–21.

French, S. (1982), *Sequencing and Scheduling, an Introduction to the Mathematics of the Job-Shop*, John Wiley & Sons.

Garey, M. R. & Johnson, D. S. (1978), 'Strong NP-completeness results: Motivation, examples, and implications', *J. Assoc. Comput. Mach.* **25**, 499–508.

Garey, M. R. & Johnson, D. S. (1979), *Computers and Intractability a Guide to the Theroy of NP-Completeness*, W.H. Freeman and Company, USA.

Glass, C. A., Gupta, J. N. D. & Potts, C. N. (1994), 'Lot streaming in three-stage production processes', *European Journal of Operational Research* **75**, 378–394.

Glass, C. A. & Potts, C. N. (1998), 'Structural properties of lot streaming in a flow shop', *Mathematics of Operations Research* **23**(3), 624–639.

Glover, F. (1986), 'Future paths for integer programming and links to artificial intelligence.', *Computers & Operations Research* **13**, 533–549.

Goyal, S. K. (1976), 'Note on "manufacturing cycle time determination for a multi-stage economic production quantity model"', *Managment Science* **23**(2), 332–333. the rejoinder, 334-338.

Graham, R. L., Lawler, E. L., Lenstra, J. K. & Rinnoy Kan, A. H. G. (1979), 'Optimization and approximation in deterministic machine scheduling: A survery', *Annals of Discrete Mathematics* **5**, 287–326.

Grötschel, M., Lovasz, L. & Schrijver, A. (1988), *Geometric Algorithms and Combinatorial Optimisation*, Springer, Berlin.

Heizer, J. & Render, B. (1996), *Production and Operations Management: Startegic and Tactical Decisions*, Prentice Hall, New Jersey.

Held, M. & Karp, R. M. (1970), 'The traveling salesman problem and minimum spanning trees', *Operations Research* **18**, 1138–1162.

Held, M. & Karp, R. M. (1971), 'The travelling salesman problem and minimum spanning trees: Part II', *Mathematical Programming* **1**, 6–25.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, University of Michingan Press, Ann Arbor, MI.

Hopfield, J. J. & Tank, D. W. (1985), 'Neural computation of decisions in optimization problems', *Biological Cybernetics* **52**, 141–152.

Hurink, J. L. (1999), 'An exponential neighborhood for a one-machine batching problem', *OR-Spektrum* **21**, 461–476.

Jackson, J. (1955), Scheduling a production line to minimize maximum tardiness. Research Report 43, Magnagement Science Research Project, University of California, Los Angeles.

Johnson, D., Papadimitriou, L. A. & Yannakakis, M. (1988), 'How easy is local search', *Journal of Computer and System Sciences* **37**, 79–100.

Johnson, S. (1954), 'Optimal two- and three-stage production schedules with setup times included', *Naval Research Logistics Quarterly* **1**, 61–67.

Kalir, A. A. & Sarin, S. C. (2000), 'Evaluation of the potential benefits of lot streaming in flow-shop systems.', *Int. J. Production Economics* **66**, 131–142.

Kalir, A. A. & Sarin, S. C. (2001), 'A near-optimal heuristic for the sequencing problem in multiple-batch flow-shops with small equal sublots.', *Omega* **29**, 577–584.

Karmarkar, N. (1984), 'A new polynomial time algorithm for linear programming', *Combinatorica* **4**, 373–395.

Karp, R. M. (1972), *Reducibility Among Combinatorial Problems*, Plenur Press, New York. R.E. Miller and J.W. Thatcher (eds.) 85-103.

Khachian, L. G. (1979), 'A polynomial algorithm for linear programming', *Doklady Akad. Nauk USSR* **244**(5), 1093–96. Translated in *Soviet Math. Doklady*, 20, 191-194.

Khachiyan, L. G. (1997), 'A polynomial algorithm for linear programming', *Soviet Mathematics Doklady* **20**, 191–194.

Kirkpatrick, S., Jr., C. G. & Vecchi, M. P. (1983), 'Optimization by simulated annealing', *Science* **220**, 671–680.

Kumar, S., Bagchi, T. P. & Sriskandarajah, C. (2000), 'Lot streaming and scheduling heuristic for m-machine no-wait flowshops', *Computers & Industrial Engineering* **38**, 149–172.

Lageweg, B., Lawler, E. L., Lenstra, J. K. & Rinnooy Kan, A. H. G. (1978), 'Computer aided complexity classification of deterministic scheduling problems', *Mathematisch Centrum* . unpulished manustript, Amsterdam.

Land, A. H. & Doig, A. G. (1960), 'An automatic method for solving discrete programming problems', *Econometrica* **28**, 497–520.

Lee, C.-Y., Uzsoy, R. & Martin-Vega, L. A. (1992), 'Efficient algorithms for scheduling semiconductor burn-in operations', *Operations Research* **40**(4), 764–775.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. & Teller, E. (1953), 'Equation of state calculations by fast computing machines', *Journal of Chemical Physics* **21**, 1087–1092.

Mühlenbein, H. (1997), *Genetic Algorithms*, John Wiley & Sons Ltd., chapter 6, pp. 137–171.

Mühlenbein, H., Gorges-Schleuter, M. & Krämer, O. (1988), 'Evolution algorithms in combinatorial optimization', *Parallel Computing* **7**, 65–85.

Papadimitriou, C. H. (1994), *Computational Complexity*. Addison-Wesley.

Papadimitriou, C. H. & Steiglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, New York.

Pinedo, M. (1995), *Scheduling Theory, Algorithms and Systems*, Prentice-Hall.

Pinedo, M. & Chao, X. (1999), *Operations Scheduling with Application in Manufacturing and Services*, Irwin/McGraw-Hill, USA.

Potts, C. N. & Baker, K. R. (1989), 'Flow shop scheduling with lot streaming', *Operations Research Letters* **8**, 297–303.

Potts, C. N. & Kovalyov, M. Y. (2000), 'Scheduling with batching: A review', *European Journal of Operational Research* **120**, 228–249.

Potts, C. N. & Van Wassenhove, L. N. (1992), 'Integrating scheduling with batching and lot-sizing: A review of algorithms and complexity', *Journal of Operational Research Society* **43**(5), 395–406.

Ramasesh, R. V., Fu, H., Fong, D. K. H., & Hayya, J. (2000), 'Lot streaming in multistage production systems', *International Journal of Production Economics* **66**, 199–211.

Schrijver, A. (1986), *Theory of Linear and Integer Programing*, Wiley, Chichester.

Sen, A., Topaloglu, E., & Benli, O. S. (1998), 'Optimal streaming of a single job in a two-stage flow shop', *European Journal of Operational Research* **110**, 42–62.

Smith, W. E. (1956), 'Various optimizers for single stage production', *Naval Research Logistics Quarterly* **3**, 59–66.

Sung, C. S., Choung, Y. I., Hong, J. M., & Kim, Y. H.(2002), 'Minimizing makespan on a single burn-in oven with job families and dynamic job arrivals', *Computers and Operations Research* **29**, 995-1007.

Szendorvits, A. Z. (1975), 'Manufacturing cycle time determination for a multi-stage economic production quantitiy model', *Managment Science* **22**(3), 298–308.

Trietsch, D. & Baker, K. R. (1993), 'Basic techniques for lot streaming', *Operations Research* **41**(6), 1065–1076.

Van de Velde, S. L. (1991), Machine Scheduling and Lagrangian Relaxation, PhD thesis, Eindhoven Techinal University.

Černý, V. (1985), 'Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm', *Journal of Optimization Theory and Applications* **45**, 41–51.

Vickson, R. G. (1995), 'Optimal lot streaming for multiple products in a two-machine flow shop', *European Journal of Operational Reserach* **85**, 556–575.

Vickson, R. G. & Alfredsson, B. (1992), 'Two- and three-machine flow shop scheduling problems with equal sized transfer batches', *International Journal of Production Research* **30**(7), 1551–1574.

Wagelmans, A. & Gerodimos, A. (2000), 'Improved dynamic programs for some batching problems involving the maximum latneness criterion', *Operations Research Letters* **27**, 109–118.

Webster, S. & Baker, K. R. (1995), 'Scheduling groups of jobs on a single machine', *Operations Research* **43**(4), 692–703.

Yannakakis, M. (1997), *Computational Complexity*, Local Search in Combinatorial Optimization, John Wiley & Sons Ltd, chapter 2, pp. 19–55. Aarts, E.H.L., and Lenstra, J.K. (Eds.).