Job Shop Scheduling with Flexible Maintenance Planning

Ivar Struijker Boudier, B.Sc.(Hons.), M.Res.

Submitted for the degree of Doctor of

Philosophy at Lancaster University.

July 2017

Abstract

This thesis considers the scheduling challenges encountered at a particular facility in the nuclear industry. The scheduling problem is modelled as a variant of the job shop scheduling problem. Important aspects of the considered problem include the scheduling of jobs with both soft and hard due dates, and the integration of maintenance planning with job scheduling. Two variants of the scheduling problem are considered: The first variant makes the classic job shop assumption of infinite queueing capacity at each machine, while such queueing capacity is non-existent in the second variant. Without queueing capacity, the scheduling problem is a variant of the blocking job shop problem. For the non-blocking variant of the problem, it is shown that good solutions can be obtained quickly by hybridising a novel Ant Colony Optimisation method with a novel Branch and Bound method. For the blocking variant of the problem, it is shown that a novel Branch and Bound method can rapidly find optimal solutions. This Branch and Bound method is shown to provide good performance due to, amongst other things, a novel search strategy and a novel branching strategy.

Acknowledgements

I would like to thank the students, staff and management of the STOR-i Centre for Doctoral Training for providing such a stimulating and enjoyable research environment. I would also like to thank my academic supervisors, Michael G. Epitropakis and Kevin Glazebrook, for all their support over the course of my PhD, and for allowing me the freedom to explore the various research directions that interested me. I would also like to thank all those at the National Nuclear Laboratory who spent time with me to help me understand their scheduling challenges, especially my industrial supervisor Paul Jennings. I gratefully acknowledge funding by EPSRC grant EP/H023151/1 and the National Nuclear Laboratory. Finally, I would like to thank my family, and in particular my partner Inés, whose support, understanding, encouragement, patience and love have all made invaluable contributions to the completion of this work.

Declaration

I declare that the work in this thesis has been done by myself and has not been submitted elsewhere for the award of any other degree.

Ivar Struijker Boudier

Contents

List of Abbreviations

ACO Ant Colony Optimisation

BJSSP Blocking Job Shop Scheduling Problem

B&B Branch and Bound

JSP Job Shop Problem

JSSP Job Shop Scheduling Problem

SA Simulated Annealing

TSP Travelling Salesman Problem

List of Symbols

General

J_i	Job i
c_i	Completion time of J_i
d_i	Soft due date of J_i
$\overline{d}_i$	Hard due date of J_i
M_k	Machine k
o_i	The number of operations of J_i
O_{ij}	Operation j of J_i
p_{ij}	Processing time of operation j of J_i
r_i	Release time of J_i
s_i	Starting time of J_i
T_i	Tardiness penalty for J_i

Ant Colony Optimisation

α	Pheromone parameter
β	Heuristic parameter

$\eta_{i,j}$ Heuristic information from operation i to job j

ρ Pheromone evaporation parameter

$\tau_{i,j}$ Pheromone level from operation i to job j

Simulated Annealing

T Temperature

α Cooling parameter

Part I

Introduction

Chapter 1

Problem Introduction

This thesis presents research that addresses the scheduling challenges encountered by the National Nuclear Laboratory (NNL), a company in the nuclear industry. The research in this thesis focuses on one particular NNL facility, where challenging scheduling decisions are made on a daily basis. These scheduling decisions determine where and when jobs are to be carried out. The jobs usually require processing in a prescribed sequence of several different locations at the facility. Each location can only process one job at a time. Thus, it is of interest to schedule work in such a way as to most efficiently use each location. An added complication is that each location must be maintained from time to time. When a location is undergoing maintenance, it cannot process regular jobs. It is therefore also of interest to plan maintenance in such a way as to minimise interference with the processing of jobs. This thesis develops several methods that simultaneously schedule jobs and plan maintenance, with the objectives of completing each

job by its respective due date and performing maintenance within desired time intervals. As there is limited storage available at the facility, this thesis considers both the case where storage is available (Part II) and the case where it is not (Part III).

This chapter continues with a brief introduction to NNL (Section 1.1). This is followed by a more detailed description of their real-world scheduling problem (Section 1.2). A basic model for the scheduling problem is provided in Section 1.3. This chapter then concludes with an overview of the remainder of the thesis (Section 1.4).

1.1 National Nuclear Laboratory

NNL provide technical support to all aspects of the nuclear industry. Their key objective is to help to safeguard and develop nuclear expertise and laboratories across a number of different sites. NNL identify the following three core areas for their business:

- **Waste management and decommissioning:** Development and application of techniques related to decommissioning nuclear facilities.
- **Fuel cycle solutions:** Fuel cycle performance, technology development, safety management and engineering services.
- **Reactor operations support:** This includes post irradiated exam-

ination and the performance of fuel components and graphite.

1.2 A Real-World Problem

The work in this thesis was motivated by the scheduling challenges encountered at a particular NNL facility. This facility provides a range of services. Examples of typical jobs performed at the facility include the examination of reactor fuel and irradiated materials, the processing of radioactive waste, and research into nuclear materials.

1.2.1 Facility Overview

The facility contains 13 shielded cells, known as caves. The caves are designed to protect workers from the radioactive materials they are processing. The workers cannot enter the caves. Any work must be carried out using remotely controlled mechanical arms, known as MSMs (master-slave-manipulators). Each cave has a number of workstations with such mechanical arms. Caves are usually set up to accommodate the processing of one particular type of job. It takes considerable time and effort to modify the setup of a cave to accommodate a different type of job than those it is currently serving. The NNL facility processes jobs for a number of external customers. Some of these customers have long term leases for the exclusive use of a designated cave. All of this means that each job has to be processed in a specific cave and at a specific workstation.

The caves are connected through a single transportation mechanism. The transporter operates within a shielded corridor. Workers do not access this corridor, but instead use remote controlled tools to operate the transporter. Two caves serve as both entry and exit into the system. A typical job will therefore enter the system at one of these two caves. The transporter then moves the job to another cave, where the job is processed. The transporter then moves the job to any other required caves. Eventually, the job leaves the system again through one of the two entry/exit caves. Transporter movements are of relatively short duration, compared to the time jobs spend in the caves. However, the transporter is a very critical part of the system, since jobs cannot be moved around without it.

1.2.2 Facility Maintenance

The jobs processed at the facility rely on a range of specialist equipment, including the MSMs and the transporter. This equipment can break down if it is not properly maintained. Such breakdowns cause delays and must be fixed with *corrective maintenance*. Regular inspection and maintenance of equipment is known as *preventive maintenance*. It keeps the equipment running smoothly, and reduces the probability that a breakdown will occur. NNL use maintenance software to schedule preventive maintenance, which is generally of known duration. The regulations under which the facility operates offer some flexibility with regards to when preventive maintenance can be carried out. Generally, it can be delayed beyond its due date by

16.67% of the maintenance interval, e.g. 61 days for yearly maintenance. This provides a flexible time window during which maintenance can start. If maintenance has not started by the end of this time window, then the equipment must be shut down, due to strict safety regulations in the nuclear industry. When certain equipment is shut down, other parts of the facility may also have to be put out of action, if there is a safety impact. Due to output performance pressures, non-essential maintenance is not always carried out in a timely fashion. This can lead to reduced equipment availability at a later stage, if a breakdown occurs.

When the research in this thesis started, the planning of regular jobs and preventive maintenance at the facility was not properly integrated. Even in the literature, it is not uncommon for researchers to focus either on the scheduling of jobs, or on the planning of maintenance. The integration of the two is much less researched. Therefore we set out to create a scheduling method suitable for the NNL facility, which combines the scheduling of jobs with the scheduling of flexible maintenance activities.

1.2.3 Scheduling Objectives

NNL use both long and short term schedules to plan the work at their facility. The research in this thesis focuses on the 12-week plan. The work at the facility is planned according to the following three important objectives:

1. **Revenue:** Jobs often provide a fixed financial reward upon completion. For some jobs, the full financial reward will only be paid out if a specified deadline is met. If this deadline is missed, the job will attract a reduced financial reward on completion. Some jobs have a second deadline, the exceedance of which may result in a very large financial penalty.

2. **Reputation:** Maintaining a good reputation with customers helps to secure the long term flow of work. A good reputation may be maintained by carrying out a certain amount of work for each customer every year. Therefore it is sometimes important to prioritise work that is not the most financially rewarding.

3. **Reliability:** Reliability refers to the overall availability of resources. Reliability can be maintained by ensuring that maintenance activities start within their flexible time windows.

Once a 12-week plan is in place, it will at some point need to be updated. This can be in response to the availability of new work which needs to be integrated in the schedule. New work often consists of jobs which had been planned, but which had not yet been confirmed. Jobs are sometimes confirmed with only a few weeks to go until the supposed start date. Sometimes there will be some actual new jobs to be included in the schedule. It is possible for new work to arrive at the facility within a few weeks of the initial enquiries from the customer. This means new work may appear into

the middle of the 12-week plan. A schedule update is also required when there is an unexpected breakdown, or when the actual processing times have been markedly different from those anticipated.

Due to the uncertainties outlined above, the 12-week plan is regularly updated. At the time the research presented in this thesis was started, the updating was done manually by the planners. This is a challenging task, as there can be many jobs on the 12-week plan. This thesis develops scheduling tools which the planners can use to explore different scheduling options for the 12-week plan.

1.3 Modelling the Real-World Problem

The real-world scheduling problem described in Section 1.2 must be modelled in a way which captures all its critical aspects. Once the problem is modelled, there will be a foundation on which solution methods can be developed.

The problem at hand shares many similarities with the classic *job shop scheduling problem* (JSSP). In a JSSP, a set of jobs must be scheduled for processing on a set of machines. A job consists of one or more operations. The operations within a job must be processed in a specified order. Each operation must be processed on a specified machine. A basic example with three jobs, each requiring three machines, is illustrated in Figure 1.3.1. In the classic deterministic JSSP, each operation has a fixed and known

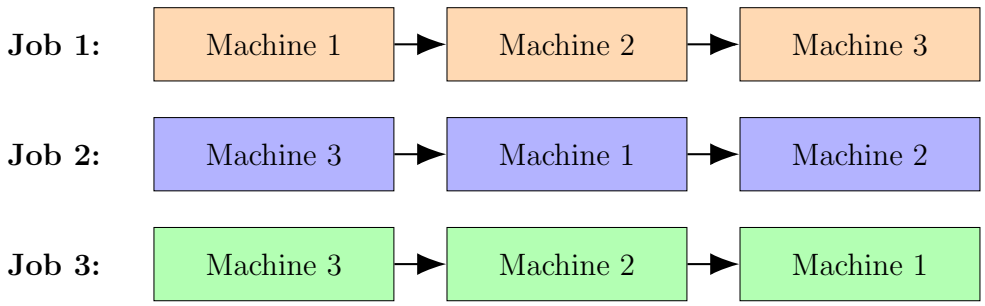

Figure 1.3.1: In the classic job shop scheduling problem, jobs must be processed on specified machines in a specified order.

processing time. Each machine can process at most one job at a time. Once an operation has started, it cannot be interrupted. The JSSP is one of the most notoriously difficult combinatorial optimisation problems (NP-hard, Du and Leung (1990)), even for relatively small instances. JSSPs can be found in the literature with various characteristics and setups that are motivated by specific industrial settings (Pinedo, 2008).

A decision maker has to solve this scheduling problem by finding a sequence of operations on each machine that optimises some performance indicator. Perhaps the most common objective function, at least in the literature, is the latest completion time of all the operations. This is known as the makespan (Artigues and Feillet (2007); Rajendran and Ziegler (2004)). The makespan objective aims to finish the entire set of work as quickly as possible. This is suitable for scenarios in which a single set of jobs has to be scheduled just once, and the overall completion time is all that matters. However, in many applied scenarios this is not the case. Jobs often have

their own individual deadlines. Production processes are often continuously in motion, so that new work has to be integrated into the schedule when the old schedule has not yet completed. One way to achieve this is to create a new schedule, which includes all the new work and any unfinished operations from the old schedule. If such a new schedule is created with the makespan objective, there is nothing to prevent some of the old unprocessed operations to be rescheduled at the end of the new schedule. In the worst case, whenever rescheduling takes place, the same unfinished job is repeatedly put at the end of the new schedule, and forever remains unfinished. There are other objective functions which avoid this scenario, by taking into consideration the deadlines of jobs. Examples of such objective functions include the maximum lateness (Mati et al., 2011; Pinedo, 2008) and the tardiness (Kuhpfahl and Bierwirth, 2016; Singer and Pinedo, 1998; Mattfeld and Bierwirth, 2004; Baker, 1984; Baker and Kanet, 1983; Raman and Brian Talbot, 1993).

An initial model of NNL's scheduling problem can be based on the formulation of the job shop scheduling problem. Each operation requires either a cave or the transporter. Hence, the caves and the transporter represent the machines of the JSSP. It will be assumed that processing times are of known and fixed duration. A precedence constraint is a job which must be completed before another job can start. Each job may have one or more precedence constraints. Each job may have an earliest starting time, also known as a release date. Maintenance activities can be modelled

as single operation jobs.

1.3.1 Novel Model Features

To model the work at the facility of interest, we have enhanced the classic JSSP with a number of novel non-standard features. Jobs at the facility attract a financial reward on timely completion. This reward is reduced when the first due date is missed, and severely reduced when the second due date is also missed. Each maintenance event is modelled as a single-operation job. Maintenance activities have an earliest start time, and a mandatory deadline by which the maintenance must have started. This provides the planners with a flexible time window in which the maintenance can start. To model this behaviour of the jobs and maintenance activities, we introduce both soft and hard due dates as a novel feature in the model. In the literature, hard due dates are sometimes referred to as deadlines. Job due dates refer to the completion time of the job, whilst maintenance due dates refer to the maintenance starting time. We have developed objective functions which return a penalty score based on the number and type (soft or hard, job or maintenance) of due dates which are missed (Equation 3.3.1, Equation 3.3.2, Equation 8.4.1, Equation 8.4.2). Besides striving to meet due dates, our objective functions simultaneously reduce overall tardiness.

The use of both soft and hard due dates does not mean our methods are restricted to only schedule jobs with such due dates. Jobs which have no due dates can still be included, by setting their due dates to infinity. This

is a good option for non-urgent jobs: By setting the due dates to infinity, the jobs will be integrated in the schedule, but only in such a way that they don't cause other jobs to miss their due dates. Jobs with only a soft due date can be scheduled by setting the hard due date to infinity. Jobs with only a hard due date can be included by setting their soft due date equal to their hard due date.

Under the classic JSSP formulation, each machine has a queue with infinite capacity. When one machine is finished with a job, the job can then be put in a queue at the next required machine, even if that machine is currently busy with another job, and even if other jobs are already waiting for that machine. Part II of this thesis considers the case in which such queueing capacity is indeed available. In Part III the scenario in which there are no queues is investigated. In that case, jobs cannot queue at the machine, and will have to wait on their previous machine until the next machine becomes available. This effectively blocks the first machine for use by other jobs. The scenario without queueing capacity is known as a *blocking job shop scheduling problem* (Mascis and Pacciarelli, 2002).

Some jobs at the NNL facility can only start when multiple other jobs come together on completion. This can be thought of as a number of jobs merging into one. For example, consider a job which consists of the processing of some radioactive materials. A second job consists of the preparation of a storage flask for the materials. A third job is to fill the prepared flask with the processed radioactive materials. This implies that the flask filling

can only go ahead once the materials have been processed and the flask has been prepared. The job merging feature is illustrated in Figure 1.3.2. The reverse also happens: There are jobs which, on completion, allow a number of other jobs to start. This can be thought of as a single job splitting into many. In the non-blocking formulation of the problem, this feature can be modelled with regular precedence constraints. However, this is more complicated in the blocking case. The way merging and splitting jobs are modelled under blocking conditions is an additional novel feature (Section 8.7.2).

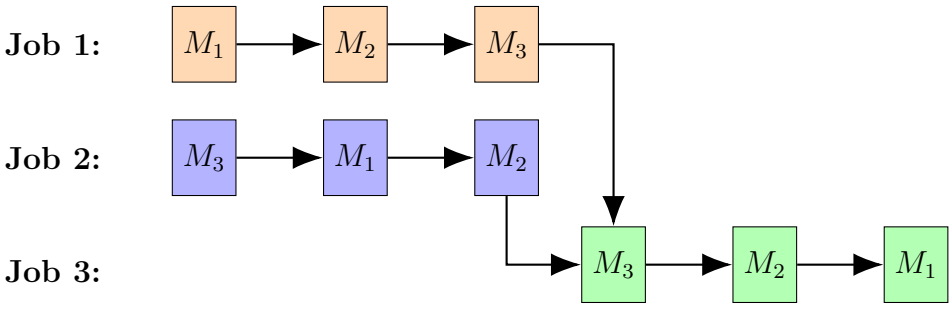

Figure 1.3.2: Job merging: Job 3 can only start once jobs 1 and 2 have been completed. M_k denotes machine k.

1.4 Thesis Structure

An overview of relevant existing scheduling methods is provided in Chapter 2.

Part II of this thesis then develops novel exact, heuristic, and hybrid scheduling methods for the novel job shop scheduling problem with flexi-

ble maintenance, under the classic assumption of infinite queueing capacity (non-blocking). The non-blocking scheduling problem is described and formally defined in Chapter 3. A novel Branch and Bound algorithm for this problem is presented in Chapter 4. This Branch and Bound method employs a novel variable ranking method to speed up the branching process. While this exact method provides optimal solutions, it might require a long computational time to reach optimality. As such, novel heuristic methods have also been developed (Chapter 5). The computational experiments in Chapter 6 show that the Branch and Bound method can be very effective, especially when it is seeded with a good solution from one of the heuristic methods.

Part III of this thesis develops exact and heuristic scheduling methods for the novel blocking job shop scheduling problem with flexible maintenance. The big difference with the problem dealt with in Part II is that there is zero buffer capacity in the system. This means jobs have to remain on their current machine after processing, until such a time that the next required machine is available. While the job waits to be moved, the machine that it is occupying cannot be used by other jobs. This novel blocking job shop scheduling problem with flexible maintenance is described and formally defined in Chapter 8. The same chapter presents a novel adaptation of the alternative graph to model the problem. Under the blocking constraint, it can be difficult to construct feasible schedules. Chapter 8 presents a novel schedule construction method for the blocking job shop,

which is guaranteed to produce feasible schedules. Heuristic solution methods for the blocking problem are presented in Chapter 9. While heuristic methods worked well for the non-blocking version of the problem discussed in Part II, the heuristics presented in Part III generally struggle to produce good solutions. In contrast to this, the Branch and Bound algorithm for the block shop presented in Chapter 10 very quickly produces optimal solutions. The proposed Branch and Bound method employs a novel branching strategy, as well as a novel search strategy. The computational experiments in Chapter 11 show that the Branch and Bound method outperforms the heuristic methods, both in terms of running time and the quality of the final solutions. Final conclusions on the work in this thesis, as well as suggestions for future work, appear in Chapter 13. Details of the problem instances that have been used in this thesis are presented in Appendix A.

Chapter 2

Literature Review

This chapter provides an overview of the job shop scheduling literature and some frequently applied solution methods. Although much research has been done in this area, we are not aware of any existing methods that deal with the job shop scheduling problem with flexible maintenance and double due dates. The methods proposed in this thesis were developed to fill this gap in the literature. The scope of this literature review is in parts somewhat wider than strictly essential, in order to give a sense of the wider scheduling landscape that our own methods reside in. Scheduling problems are discussed in Section 2.1, followed by the integration of job scheduling and machine maintenance in Section 2.2. Various metaheuristic methods are discussed in Section 2.3. An overview of Branch and Bound, an exact solution method, is provided in Section 2.4.

2.1 Scheduling

Scheduling is one of the most important operational features in many manufacturing industries. There exist many types of scheduling problems. The following illustrative example is based on Pinedo (2008). Suppose there is a factory, containing a range of machines, producing a number of different products. Each product has to pass through a number of machines, and some machines will be needed in the manufacturing process of multiple products. It is assumed that each machine can only work on a single job at any one time. The scheduling problem then has a number of constraints to consider, including:

- **Machine availability:** Each machine will have an availability associated with it over the production period. A number of factors could reduce this availability, such as scheduled maintenance, or the time required for setup between the processing of two different products. The setup time may depend on how similar the next product is to the previous one.

- **Machine demand:** To produce one unit of a product, some production time of the associated machines will be required. This production time could be fixed (deterministic), or it may have some variability (stochastic). This means that production is limited by the combination of machine demand and machine availability.

- **Materials:** The raw materials for each product must be available at the correct times.

- **Job processing order:** The order in which each product goes through different machines is often important.

Scheduling problems can have a number of possible objectives. It may be desired to minimise the *makespan*, which is the latest completion time of all operations in the schedule. Some jobs may have due dates, e.g. a time by which it is expected by the customer. In this case the objective is to meet these due dates, or to minimise the expected exceedances of the due dates. Often it is also desirable to minimise costs. These objectives can be considered on their own, or together. Sometimes there are multiple objectives. For example, one might wish to meet the due dates and simultaneously minimise the costs. *Multi-objective scheduling* deals with such situations (Loukil et al. (2005)).

2.1.1 Scheduling Problem Classification

The widely used 3-field problem classification $\alpha|\beta|\gamma$ was introduced by Graham et al. (1979). In this classification, α describes the machine environment, β contains processing characteristics and constraints, and γ holds the objective to be minimised. For a list of a large number of machine environments, constraints, and other scheduling terminology, see Pinedo (2008).

2.1.2 Job Shop Scheduling Problem

The job shop scheduling problem (JSSP) in particular is one of the most popular optimisation problems in both industry and academia. The traditional JSSP schedules n jobs to be processed on m machines. Each job is required to pass through specified machines in a specified order. This order can be different for each job. The problem can be expanded by considering various different complications, based on the characteristics of particular production facilities. For example, the *flexible job shop* is a generalisation of the job shop, in which there may be multiple copies of each machine (Pinedo, 2008). A job requiring a certain machine can then be processed on the first machine of this type to be available. Various different characteristics have been investigated in the literature (Pinedo, 2008), such as due dates (Kuhpfahl, 2016; Mattfeld and Bierwirth, 2004; Kuhpfahl and Bierwirth, 2016; Yang et al., 2012), and flexibility and maintenance (Mokhtari and Dadgar, 2015; Ma et al., 2010; Ruiz et al., 2007), amongst others.

Using the notation introduced in Section 2.1.1, the classic job shop scheduling problem can be expressed as $J \mid\mid C_{\max}$. This problem involves m machines and n jobs, with the objective of minimising the makespan, $C_{\max}$. Additionally, there is no recirculation (i.e. each job requires each machine once at most). This problem is $\mathcal{NP}$-hard (van den Akker et al., 2013), which means it cannot be solved in polynomial time by any known algorithms.

2.1.3 Job Shop Scheduling with Due Dates

Job Shop problems with due date objectives have been widely investigated in the literature (Kuhpfahl, 2016; Mattfeld and Bierwirth, 2004; Kuhpfahl and Bierwirth, 2016; Yang et al., 2012; Raman and Brian Talbot, 1993). A recent overview of local search methods for the job shop scheduling problem with due dates can be found in Kuhpfahl (2016). A survey between the interactions of sequencing priorities and assigning due dates has been performed by Baker (1984), focusing on the average scheduling tardiness. New dispatching rules were introduced in Baker and Kanet (1983) that utilise modified due dates (job's due date or its early finish time). The proposed dispatching rules considered the mean tardiness objective. Experimental results indicate that they were very competitive compared with other prominent dispatching methodologies that optimise the mean tardiness of jobs. Moreover, the effect of various procedures that consider due dates that depend on the expected job processing time, as well as on the level of congestion in the considered shop, has been considered by Eilon and Chowdhury (1976).

2.1.4 The Blocking Job Shop

Under the classic job shop formulation, it is implicitly assumed that each machine has a queue with infinite capacity. If a job requires a machine which is currently in use, the job can be stored in that machine's queue. In

certain systems such storage may not be available. The scenario without queueing capacity is known as a blocking job shop scheduling problem. When the processing of a job has finished on a machine, the job must remain on that machine until the next required machine becomes available. Thus, it blocks the machine that it is currently occupying from use by other jobs. This may lead to what is known as deadlock (Mascis and Pacciarelli, 2002). Deadlock occurs when two or more jobs block each other in a circular way, such that each of these jobs requires a machine currently occupied by one of the other jobs in the deadlock circle. To resolve deadlock, some authors allow for swaps. A swap is when all the jobs in the deadlock circle move simultaneously (Mascis and Pacciarelli, 2002). If swaps are not allowed, deadlock renders a schedule infeasible. Without swaps, it is not always possible to complete a given partial schedule into a full solution. In fact, just determining whether this is possible for a given partial solution is a strongly NP-complete problem (Mascis and Pacciarelli (2002)). The blocking job shop without swaps has received less attention from researchers than the equivalent problem with swaps (Pranzo and Pacciarelli, 2016). Part III of this thesis will consider blocking without swap in the context of production processes, but there are also other situations in which blocking can occur. In railway scheduling, for example, an entire section of track is usually reserved exclusively for a single train, and a train cannot vacate its current section of track until the next section becomes available. See, for example, D'Ariano et al. (2007). Railway scheduling also presents a good

example of the no-swap scenario, since it is physically impossible to swap the positions of two trains, without using a vacant track for one train to pass the other.

The job shop scheduling problem with blocking and no-wait constraints is considered by Mascis and Pacciarelli (2002). They employ the alternative graph (Mascis and Pacciarelli (2000)), an adaptation of the well-known disjunctive graph (Roy and Sussmann (1964)), to model the problem. They consider the case where jobs that block each other may be swapped, and also the case where this is not possible. In the latter case, schedules must be created in such a way that deadlock is avoided. Mascis and Pacciarelli (2002) note that while in the standard (non-blocking) job shop formulation a partial solution can always be extended into a feasible solution, there is no such guarantee for partial solutions of a blocking job shop. There is also no guarantee that, given a feasible solution for the blocking job shop problem, an arc of the alternative graph can be replaced with its alternative to create a new feasible schedule. This implies that local search methods for the blocking job shop incur a greater computational cost than they do in the non-blocking job shop. Mascis and Pacciarelli (2002) present some schedule construction heuristics, with no guarantee of feasibility, as well as a Branch and Bound method. The four heuristics presented by Mascis and Pacciarelli (2002) are designed for minimising the makespan.

The job shop scheduling problem with blocking and no-wait constraints is considered by Meloni et al. (2004). They also employ the alternative

graph to model the problem. A rollout metaheuristic is implemented to obtain feasible solutions by Meloni et al. (2004). The rollout method sequentially fixes all the decision variables in a given problem. At each iteration, the most promising of the remaining decision variables is added to the current partial solution. Meloni et al. (2004) describe three heuristics for selecting the next variable. Each heuristic extends partial solutions by iteratively selecting one of the remaining unselected arcs and adding it to the current partial solution.

More recently, Pranzo and Pacciarelli (2016) propose an Iterated Greedy (IG) algorithm for the job shop scheduling problem with blocking constraints. They consider the problem both with and without swaps allowed. The IG algorithm iteratively destroys part of the current solution, and then repairs it according to a local optimisation procedure.

2.1.5 Other Scheduling Problems

There are many other scheduling problems and solution approaches. Interested readers are referred to the book by Pinedo (2008), which covers a large number of scheduling problems, both for deterministic and stochastic problems.

2.2 Integration of Job and Maintenance Scheduling

The scheduling scenarios and solution methods introduced in previous sections of this chapter often implicitly assume that machines and other equipment are 100% reliable. In reality, this is not the case and equipment failures will occur. The breakdown of equipment at the NNL facility can in some cases lead to severe disruption of the processing of jobs. For instance, problems with the internal transportation mechanism may make it impossible to move jobs in or out of the system for some time. A breakdown of equipment at a single workstation may have a smaller impact on the overall schedule. Regular maintenance can reduce the occurence of such breakdowns. Maintenance planning should therefore be integrated into the job scheduling process. Traditionally, the literature has often dealt with scheduling as a stand-alone problem in which no consideration is given to machine unavailability due to maintenance. Similarly, maintenance planning has often been considered without taking explicitly into account the disruption it causes to the job schedule. The following sections review existing work on the integration of maintenance and scheduling.

2.2.1 Preventive and Corrective Maintenance

The probability of equipment failure can often be reduced by periodically carrying out *preventive maintenance*. This might take place either after a

certain period of time, or after a number of completed jobs. Such maintenance can be planned in advance. Ma et al. (2010) provide an overview of the literature on deterministic scheduling problems with limited machine availability. They consider availability constraints due to planned maintenance, or due to the previous schedule not having finished yet on all machines. It is often assumed that the number of maintenances to be carried out is known in advance. However, there is also the possibility that a machine breaks down, which would require repairs, also known as *corrective maintenance* (Batun and Azizoglu, 2009). It is often reasonable to assume that the probability of a breakdown increases with the time since the last preventive maintenance. It is usually assumed that preventive maintenance returns a system to an 'as new' condition, whereas corrective maintenance only returns the machine to the state it was in just prior to the breakdown. Preventive maintenance helps to reduce the need for corrective maintenance. This then leads to the question of how often preventive maintenance should be carried out. If this is not done often enough, there may be an unacceptably high number of breakdowns. However, if preventive maintenance is carried out too often this will result in considerable machine unavailability.

2.2.2 Single Machine Problems

Single machine scenarios do not capture the complexity of the NNL facility. However, some papers on smaller problems do present interesting ideas

and solutions methods which may be of use in more complex systems. Some papers from the single machine literature are discussed next, before considering more complex scenarios.

Machine Degradation Dependent Maintenance Time

Pan et al. (2010) integrate production scheduling and maintenance planning for a single machine, whilst also considering breakdowns. As machines suffer increasing wear over time they become less reliable. This also means that as time progresses the amount of preventive maintenance required might increase. Therefore the maintenance durations depend on machine degradation. Additionally, it is assumed that some jobs are more important than others. The objective is to minimise the maximum weighted tardiness. The authors explain how to calculate job completion times, taking into consideration the machine failure rate and breakdowns. The expected job completion time for the first job is the sum of the processing time of that job, the expected time spent on preventive maintenance prior to starting this job, and the expected time spent performing corrective maintenance. The expected completion time of the first job is then used to calculate the expected completion time of the second job, and so on. This iterative procedure is rather lengthy, so we refer to the original paper for the full details. Unfortunately, Pan et al. (2010) then jump from their model presentation to their computational results without providing solution methods for the stated problem.

Single Machine with Preventive Maintenance

Batun and Azizoglu (2009) consider a single machine, deterministic scheduling problem with preventive maintenances. They assume known processing times and non-resumable jobs, i.e. any job in progress when the machine is stopped for maintenance will have to be completely restarted once maintenance has been completed. It is also assumed that maintenance activities have fixed starting times and known durations. The objective is to minimise *flow time*, also known as *total completion time*. This is the aggregate of times that jobs spend in the system, which carries an inventory cost. Without maintenance, flow time is minimised on single machines by ordering jobs in increasing order of processing times (Pinedo, 2009). This *shortest job first* principle still holds here for the machine activity periods between maintenances, and so the problem remains to decide which jobs should be carried out during each continuous stretch of machine availability. An outline of how Batun and Azizoglu (2009) proceed is as follows:

1. Use one of two heuristics to construct an initial feasible solution which will yield an upper bound on the optimal flow time. For example, the *improved shortest processing time* heuristic orders the jobs from shortest to longest processing time. This sequence is then divided into batches which fit inside the machine availability windows. For all jobs in all batches, starting from the first batch, the heuristic attempts to exchange each job with longer jobs in later batches in order to reduce

machine idle time. This has complexity $\mathcal{O}(n^2)$.

2. A lower bound on the optimal flow time can be obtained by assuming that jobs are resumable, in which case the shortest job first rule minimises the total flow time. Lower bounds usually result from a relaxation of the problem and help determine the quality of the feasible solutions found (Ben Ali et al., 2011).

3. A Branch and Bound algorithm is used to find the optimal solution. The algorithm builds the feasible sequences by adding one job at a time. It avoids any branches in which batches do not follow the shortest job first rule within activity windows, since such branches cannot produce an optimal solution. Nodes are also *fathomed* (abandoned) when a new batch is started while any remaining jobs would have fitted in the previous batch. Lower bounds for nodes can be calculated using the shortest job first rule. Nodes are fathomed if this lower bound exceeds the best known upper bound.

Batun and Azizoglu (2009) report that, depending on the ratio of the maximum job length and machine operation windows, their algorithm solved to optimality instances with between 35 and 80 jobs. The algorithm tended to perform better as the difference in size of the processing times was reduced. Finally, they note their method could be extended by considering machine breakdowns.

Preventive Maintenance and Breakdowns

Wang (2013) considers a bi-objective single machine environment which requires the integration of job scheduling and preventive maintenance planning. It is assumed that jobs have setup times which depend on the previous job, that machines can suffer a breakdown with increasing probability as time passes, and that machines can be restored to a good-as-new state by performing preventive maintenance. Time to machine failure is modelled by a Weibull distribution. A non-homogeneous Poisson process is used to model the number of failures between each preventive maintenance. The failure rate depends on the time the machine has been in use since the last preventive maintenance. The two objectives are the minimisation of total expected completion times, and minimisation of the expected number of failures. Some compromise will have to be found, since the expected number of failures would be minimised by performing preventive maintenance after each job, but this would cause considerable delays to the production process. Wang (2013) uses genetic algorithms to find good solutions.

2.2.3 Two-machine Flow Shop with Preventive Maintenance

Flow shop problems are concerned with a system in which jobs have to pass through a number of machines in a given order. This sequence of machine visits is identical for all jobs. Allaoui et al. (2008) consider a two-machine

flow shop problem, in which one machine requires preventive maintenance, with the objective of minimising the makespan. This problem is $\mathcal{NP}$-hard. Without maintenance, the two machine flow job has an optimal solution rule derived by Johnson (1954). Let $p_{m,i}$ denote the processing time of job i on machine m. Allaoui et al. (2008) then state aforementioned rule as follows:

> **Johnson's Rule:** "Divide the set of n jobs into two disjoint subsets, S_1 and S_2, where $S_1 = \{J_i : p_{1,i} \leq p_{2,i}\}$ and $S_2 = \{J_i : p_{1,i} > p_{2,i}\}$. Order the jobs in S_1 in the increasing order of $p_{1,i}$ and those jobs in S_2 in the decreasing order of $p_{2,i}$. Sequence S_1 first, followed by S_2".

Allaoui et al. (2008) propose an algorithm that orders the jobs according to Johnson's Rule. It then inserts the maintenance at the end of each job in turn and takes the solution with the shortest makespan. This solution is optimal when all processing times on machine 1 are equal. When the single preventive maintenance activity is carried out on machine 2, it is suggested this is done at time zero to take advantage of the idle time on this machine while the first job is processed by machine 1.

2.2.4 Job Shop Scheduling Integrated with Maintenance Planning

While the papers discussed above present some interesting problems and solution approaches, the complexity of the systems they describe is relatively

trivial compared to the NNL facility under consideration in this thesis. Ben Ali et al. (2011) consider a much more complex problem. They study the simultaneous scheduling of production and maintenance tasks in a job shop. Their multi-objective problem aims to minimise both the makespan and the total maintenance cost. They actually consider two types of preventive maintenance: *Block* maintenance has to be carried out at fixed time intervals of length T, while *Age* maintenance has to be carried out after the machine has been used for a cumulative duration of time T. These Ts can have a different value for both types of maintenance. Their value can also differ between machines. Machines do not necessarily have to be assigned both types of maintenance, but this is possible if so desired. Maintenance does not have to take place exactly every T time units, but can be planned to occur slightly earlier or slightly later. This offers some flexibility which is missing from algorithms in which the maintenance time is fixed. In order to help find solutions where maintenance takes place reasonably close to the required intervals, the maintenance cost function includes a penalty for tasks which are scheduled to occur before or after some tolerance period around time T. Maintenance costs do not incur a penalty if the maintenance is scheduled to start within this tolerance period.

One challenge is to decide the size of the interval T between preventive maintenances. If the interval between successive maintenances is increased, the machine will degrade more over time and become more likely to suffer a failure. Hence, increasing T means that the probability of requiring cor-

rective maintenance is also increased. However, decreasing T means more preventive maintenance periods will be required, which carries an increased machine unavailability cost. Ben Ali et al. (2011) determine T by considering the mean time between failures and multiplying this by a parameter α which balances the preventive and corrective maintenance costs. Another complication may be that different parts of the same machine may require maintenance at different intervals. Using optimal maintenance periods for each individual part would very likely result in unacceptable downtime. One option is to find the optimal maintenance period for each part, plan grouped maintenances with period T, and then service all those parts which are due nearest to the current maintenance slot at the same time. Ben Ali et al. (2011) propose a Genetic Algorithm for solving their multi-objective scheduling problem.

The work by Ben Ali et al. (2011) has some features which deal with NNL's requirements. It integrates job shop scheduling with preventive maintenance planning. The flexibility for maintenance activities is also a nice feature. Unfortunately, this method does not consider jobs with double due dates, and as such is not directly applicable to the NNL facility.

2.2.5 Further Maintenance Literature

A flow shop problem where preventive maintenance is performed before a specified amount of machine usage is considered by Ruiz et al. (2007). They explore several methods, including PACO (Rajendran and Ziegler (2004)),

an Ant Colony Optimisation method for the flow shop. Authors in Li and Pan (2013) propose a hybrid chemical-reaction optimization (HCRO) algorithm for solving the job-shop scheduling problem with fuzzy processing times and flexible maintenance.

2.3 Metaheuristics

The job shop scheduling problem is one of the most notoriously difficult combinatorial optimisation problems (NP-hard, Du and Leung (1990)), even for relatively small instances. As such, metaheuristics are often used to effectively and efficiently solve scheduling problems. Metaheuristics are algorithms which allow the finding of good (near-optimal) solutions in short computational time. A number of metaheuristics which are regularly used to solve scheduling problems are reviewed in the following sections. Those interested in general purpose heuristics for integer programming are referred to Glover and Laguna (1997a) and Glover and Laguna (1997b).

Scheduling problems can come with a variety of objectives. Common objectives include minimisation of the production cost or minimisation of the makespan. The number of potential solutions to a scheduling problem is usually so large that evaluation of every possible schedule is simply not feasible. This means it will not always be possible to solve problems to optimality. Instead, many industrial applications focus on finding good, near-optimal solutions in relatively short time. Many metaheuristics have

been developed to solve such optimisation problems to near-optimality at reduced computational cost. This means there is an optimality versus speed trade-off.

Glover and Kochenberger (2003) define metaheuristics as "solution methods that orchestrate an interaction between local improvement procedures and higher level strategies to create a process capable of escaping from local optima and performing a robust search of a solution space". Although it may feel a little unsatisfactory to only get near-optimal results, this is actually incredibly useful in many real-world applications. A company which needs to schedule its workload might only be willing/able to spend limited processing time on obtaining a near-optimal solution, as opposed to waiting many years to get an optimal result with small additional gain.

2.3.1 No Free Lunch Theorem

The No Free Lunch theorem (Wolpert and Macready, 1997) states that given any pair of two algorithms, both will perform equally well when averaged over all possible problems. This implies that if one algorithm, a_1 say, outperforms another algorithm, a_2 say, on average over a particular set of problems, then a_2 must outperform a_1 over the set of all other problems, on average. As such, there is no single general purpose algorithm available to practitioners that will always give the best results. The best performance can be obtained by tailoring algorithms to the problem of interest. The methods developed in this thesis are therefore also tailored to the problems

under consideration, in order to obtain good results.

2.3.2 Failings of Classical Methods

One of the big challenges of optimisation problems is the possible presence of multiple local minima, as illustrated in Figure 2.3.1. Classical methods can be unsuitable for finding the global minimum in such a scenario. The gradient descent method, for example, starts from some intuitive (or arbitrary) starting point on the objective function. At each iteration, a small step is taken downhill from the current position. If a lower point of the objective function is found, then this will be the starting point for the next iteration. If no nearby improvements exist, then a local minimum has been found (Dréo et al., 2006). The problem with this method is that the local

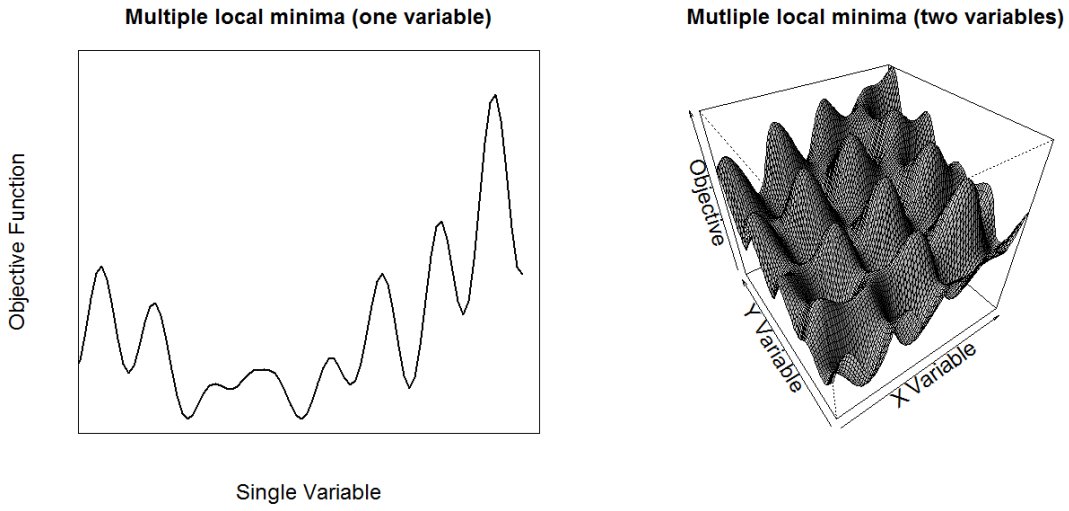

Figure 2.3.1: Multiple local minima. Objective function with a single variable (left) and two variables (right).

minimum found depends on the starting position and that no other local minima are explored. One improvement on this is to explore the solution space from a number of starting positions, but this is computationally expensive and is not guaranteed to lead to an optimal solution, especially when the number of local minima is large (Dréo et al., 2006). Some metaheuristics address this problem by occasionally accepting steps which result in an increase of the value of the objective function.

2.3.3 Simulated Annealing

Optimisation by Simulated Annealing (SA) was first introduced by Kirkpatrick et al. (1983), in the context of efficient electronic circuit board design. Consider a number of computer chips which must be placed on a circuit board, and consider that many of these chips will have to be connected to each other. The problem of interest is then to place the chips in such a way as to minimise the combined length of the links between chips. One approach would be to choose some random starting configuration. At each step, a random pair of chips swaps positions. If this swap results in a reduction of overall chip connection distance, then the swap is accepted and serves as the initial layout for the next iteration. This approach suffers the drawbacks described in Section 2.3.2, in that it will converge on a local minimum which may not be optimal. Simulated Annealing addresses this problem by occasionally accepting moves which increase the value of the objective function.

More generally, Simulated Annealing chooses some random configuration as an initial solution. At each step, some perturbation is made to the existing solution. If this perturbation results in a reduction of the solution score, then it is accepted and replaces the current solution. Moves which increase the value of the objective function are accepted according to a probabilistic rule. The acceptance probability of an uphill move depends on a temperature parameter, T, set by the user. Various temperature reduction rules exist, but a common one is the simple geometric rule $T_{k+1} = \alpha T_k$. For high values of T, the acceptance probability of uphill moves is high and configurations are able to escape from local minima. Periodically, T is decreased. The intensity of this cooling effect depends on the cooling parameter $\alpha \in (0, 1)$. As T decreases, so does the acceptance probability of uphill moves so that, eventually, the system converges to a local minimum. Dréo et al. (2006) note that, "under certain circumstances, Simulated Annealing probably converges towards a global optimum, in a sense that it is made possible to obtain a solution arbitrarily close to this optimum, with a probability arbitrarily close to unity". See, for example, Aarts and van Laarhoven (1985).

Simulated Annealing proceeds as follows (Dréo et al., 2006):

1. Choose an initial configuration.

2. Set an initial temperature, T_0.

3. Modify the configuration. Define the change in objective function as

ΔE.

4. At temperature T, accept the modification with probability

$$\alpha = \begin{cases} 1 & \text{if } \Delta E \leq 0 \\ \exp(-\Delta E/T) & \text{if } \Delta E > 0 \end{cases}$$

5. Continue to step (6) if in thermodynamic equilibrium. Otherwise, return to step (3).

6. Stop if system is frozen. Otherwise, decrease T according to some algorithm and return to step (3).

The thermodynamic equilibrium in step (5) refers to the system being in some sort of balanced state for the given temperature. In practice, this means the temperature is decreased after a predefined number of iterations. The frozen state in step (6) can be said to have been reached once no proposed moves have been accepted for a certain number of iterations, usually spanning a few temperatures.

The efficiency and asymptotic convergence of Simulated Annealing depend on the factors listed below, which must therefore be carefully chosen (Eglese, 1990). The following suggested methods of choosing these parameters have been taken from Dréo et al. (2006), but there are many other ways to set them:

- The **starting temperature**, T_0. This could be chosen as a solution

of

$$\exp\left(-\frac{<\Delta E>}{T_0}\right) = \tau_0,$$

where $<\Delta E>$ is the average of 100 moves from some initial configuration, and τ_0 depends on the quality of this initial configuration.

- The **number of iterations** at each temperature, $N(T)$. The temperature could be changed when $12N$ moves are accepted, or when $100N$ moves have been attempted. Here, N denotes the degrees of freedom of the problem.

- A **temperature adjustment rule**. A simple rule is a geometric one, $T_{k+1} = \alpha T_k$, for some $0 < \alpha < 1$.

- A **stopping rule**. One might choose to stop after three successive temperature stages during which no proposed moves were accepted.

While the rules above may often work well, they are not perfect, and in practice a trial-and-error approach is often used to optimise the parameters. Wright (2010) considered how to automate choosing the Simulated Annealing parameters. They link the initial and final temperatures to the percentage acceptance of worsening solutions (PAWS). Numerical experiments on 100 and 200-city travelling salesperson problems (TSPs) indicated that the best results were obtained when T_0 was associated with a PAWS value in the range $(4, 8)$, for a random starting position. The corresponding value of T_0 could be derived by generating worsening moves from random

starting positions and noting their values of increase in the objective function. The parameter T_0 could then be set in such a way that between 4% and 8% of worsening moves will be accepted initially. Final temperatures T_N were associated with PAWS values in the range (0.02,0.1). Wright (2010) conclude by noting that their experiments were conducted on randomly generated Euclidean TSPs, and that further research is required to investigate whether these conclusions also hold for other problems.

As Simulated Annealing is a very general approach to any combinatorial optimisation problem, it does not necessarily perform as well as some problem-specific heuristics (Eglese, 1990). One disadvantage of Simulated Annealing is that it has no memory and might therefore return to states which it has already explored. Many variants of Simulated Annealing have been developed in order to make it more efficient for particular problems, and it is often desirable to tailor the parameters using knowledge of the problem. We have adapted Simulated Annealing for two versions of the job shop scheduling problem with flexible maintenance, in Section 5.3 and Section 9.4.

2.3.4 Tabu Search

Tabu Search was first introduced by Glover (1986), although it did build on existing ideas (Glover (1977);Glover et al. (1985);Glover and McMillan (1986)). It has since been enhanced by others (Dréo et al., 2006). Like Simulated Annealing, Tabu Search explores the solution space by modifying an

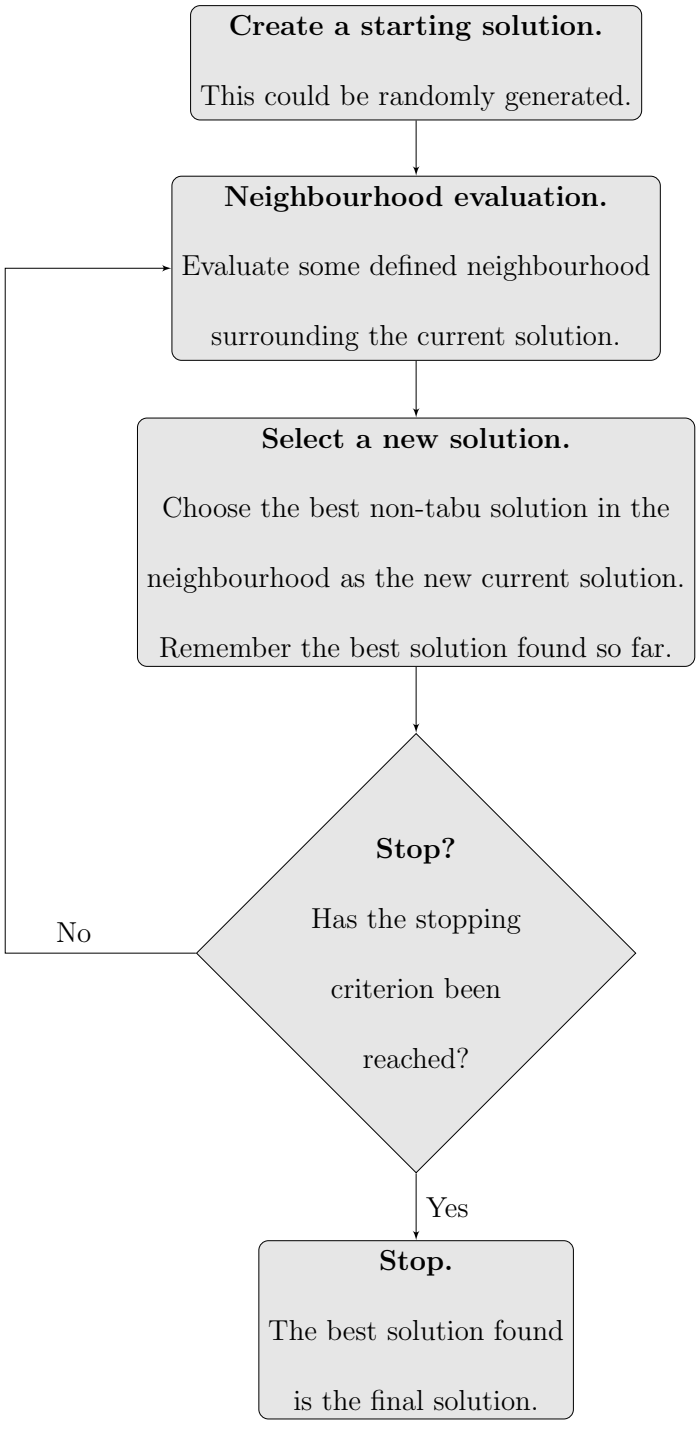

Figure 2.3.2: Flowchart for the basic Tabu Search algorithm (based on Glover (1990)).

existing solution in an attempt to find a better solution. Unlike Simulated Annealing, Tabu Search makes use of a memory function to avoid revisiting solutions multiple times. Using the notation as given in Dréo et al. (2006), let s denote a feasible solution, and let S denote the set of all feasible solutions. Define $N(s) \subset S$ to be the set of neighbouring solutions to s. In the context of the computer chip placement problem (Section 2.3.3), for example, this could be the set of solutions which differs from the current solution by a swap in position of two chips. Tabu Search evaluates all solutions in $N(s)$ and takes the best solution in this neighbourhood to be the new current solution. This is essentially a local search, which would suffer the local minimum convergence issues discussed previously (Section 2.3.2). In order to avoid getting stuck in such local traps, Tabu Search restricts movement in the solution space by preventing a return to recently visited solutions. If $N(s)$ is large, it may be desirable to only consider some of its elements, which could simply be a random subset (Dréo et al., 2006). The flow of the basic Tabu Search is illustrated in Figure 2.3.2.

Tabu Search Memory

Tabu search utilises a short term memory, which seeks to prevent checking the same solution twice within some defined period. Retaining a complete record of all visited solutions can be memory intensive. Additionally, marking them as out-of-bounds for future visits may even prevent the optimal solution being found. This is because the optimal solution could be located

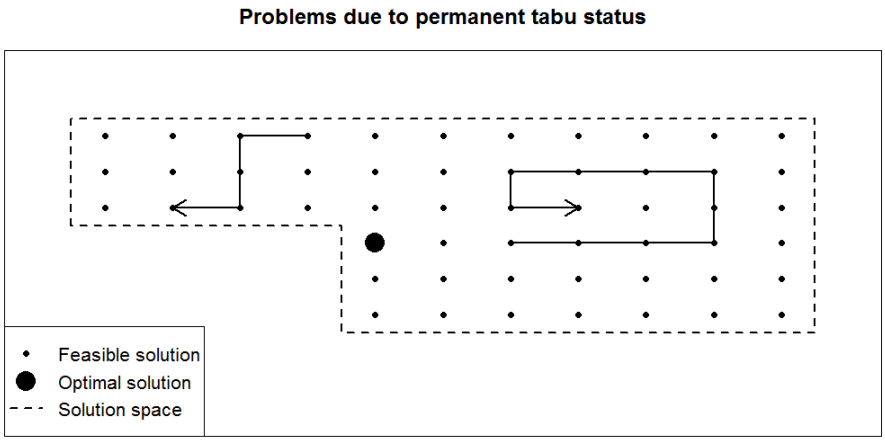

Figure 2.3.3: Permanent tabu status of already visited solutions can prevent the optimum from being reached (illustration based on Dréo et al. (2006)). The iteration path on the left prevents itself from ever reaching the global optimum, while the path on the right does something similar by getting trapped by previously visited solutions.

in an area of the solution space which might become inaccessible for the algorithm once a number of solutions have been marked tabu (Dréo et al., 2006). The algorithm might also lock itself into a local minimum, preventing the optimal solution from being found. These situations are illustrated in Figure 2.3.3.

The issues associated with permanent memory described above can be avoided by using a short term memory instead. For instance, one could restrict the search algorithm to only make moves to solutions which have not been visited during the previous t iterations. Storing complete information

on previously visited solutions can be computationally expensive, so some methods simply store the most recent values of the objective function and prevent moves to solutions with an identical value. There are some variations on this, but this introduces the problem of also preventing moves to unvisited solutions which just happen to have the same value as a recently visited solution (Dréo et al., 2006).

Alternatively, one might consider temporarily banning certain moves, rather than specific solutions. For instance, if (i, j) represents swapping components i and j, then performing this move twice in a row would return the algorithm to the current solution. Therefore, after (i, j) has been performed, this move would be banned for a number of iterations (Dréo et al., 2006). This does not actually prevent recently visited solutions from being visited again, since nothing prevents components i and j returning to their original positions in separate moves. For example, the successive moves $(i, j), (k, p), (i, p), (k, j), (k, i), (j, p)$ bring us back to the solution we started with. In this situation, revisiting solutions can be prevented by using the condition that components i and j cannot both return to the position they occupied before move (i, j) (Dréo et al., 2006).

The size of the tabu memory should be carefully chosen. If the memory is too short, the algorithm will get stuck in a 'valley' of the solution landscape. If the memory is longer, the process will instead be pushed out of this valley and explore other regions of the solution space. If the memory is too long, local minima may not be explored very well (Dréo et al., 2006).

Stopping Criterion

Tabu Search requires a suitable stopping criterion in order to find a good solution within a practical time limit. Gendreau (2003) notes that stopping criteria are often based on one of the following:

- A predetermined length of computing time, or fixed number of iterations;
- A certain number of iterations during which no improvements are made;
- The achievement of a chosen threshold.

The threshold criterion would require a certain knowledge about possible outcomes, since one would like to avoid setting a threshold which could not possibly be reached, or which would be reached too easily.

2.3.5 Evolutionary Algorithms

Evolutionary Algorithms are based on natural evolutionary processes. The basic idea is that we have a population of solutions to an optimisation problem. This population then procreates and gives birth to a second generation of solutions. If two (or more) solutions in the first generation are combined to create a single new solution, then these original solutions are called parents, and the generated solution is their offspring. The offspring generation can then be used as parents for another generation, and so on

(Dréo et al., 2006). In line with evolution, strong solutions should be more likely to reproduce than weak solutions. It may be desirable to allow the strongest existing solutions to not only reproduce, but also survive and be part of the next generation. This is known as *elitism*. Additionally, some offspring may be of poor quality and could be prevented from entering the population (Dréo et al., 2006). It is also possible to allow solutions to reproduce by mutation, which can allow for wider exploration of the solution space than simple reproduction. If stronger solutions are more likely to reproduce than weak solutions, there is a danger of convergence to the strongest solution in the initial population, which is not likely to be the global optimum solution.

If two strong solutions located near two separate local optima produce offspring, there is a danger that this offspring could actually be of poor quality. This is because, in the solution landscape, it can lie somewhere on the mountain separating the two valleys where the good solutions live (Dréo et al., 2006). This situation is illustrated in Figure 2.3.4. This is not to say that parents with large separation should never be matched, as it is always possible that their offspring will inhabit a third, unexplored valley.

There are various ways of creating offspring and mutations. Suppose a solution can be expressed as a binary vector. Mutation can then take place if there is a small chance for each bit to change from 0 to 1, or vice versa. Mating, or *crossover*, is the producing of offspring by parents, by exchanging part of their bit strings. For example, to produce a single child,

the first half of one parent's bit string could be combined with the second half of another parent's bit string. To produce two offspring, a number of elements of the two parent bit strings could be swapped (Dréo et al., 2006). It should be stressed that recombining solutions should be given careful consideration: Most problems will carry a number of constraints, and while the parents may abide by these constraints, their offspring may well violate them. There are different types of Evolutionary Algorithms, including the following (Jansen, 2013):

- **Genetic Algorithms:** First introduced by Holland (1975), genetic algorithms are often concerned with solutions which can be expressed as binary vectors. Crossover is the main method for producing off-

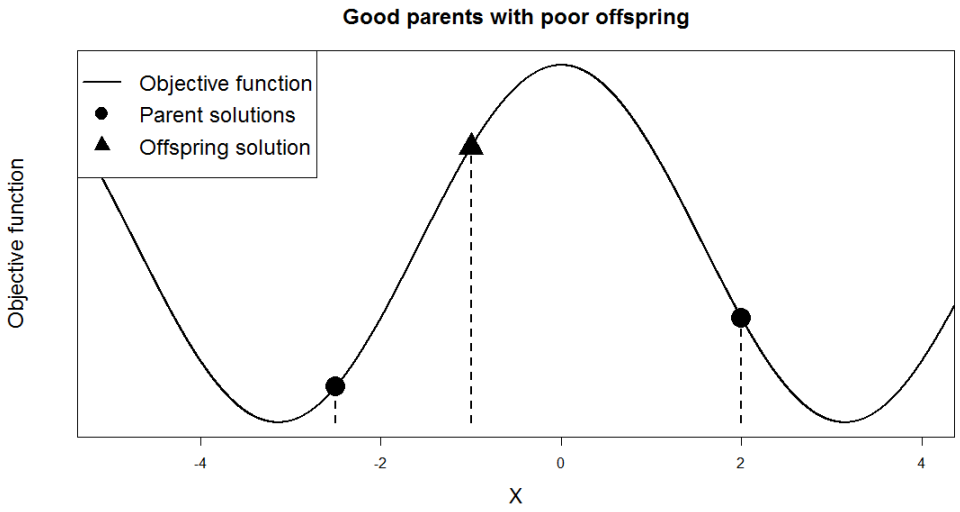

Figure 2.3.4: Parents from separate valleys can produce poorly performing offspring (illustration based on Dréo et al. (2006)).

spring. Mutation is less prevalent but provides some variation in the search. Mutation can be easily implemented by changing the value of random bits (Back, 1996).

- **Evolutionary Strategies:** These often focus on solutions which can be expressed as real-valued vectors. New generations are usually produced by mutation, although recombining existing solutions is also possible. Parent and child are compared in terms of fitness. The stronger of the two survives to be part of the next generation (Boussaïd et al. (2013);Back (1996)).

- **Evolutionary Programming:** This also considers real-valued vectors as solutions and reproduction through mutation. One common way of performing mutations is to add a random Gaussian number to the solution. A tournament is used for selection: Parents and offspring are randomly paired up and the stronger solution 'wins' that round. This is repeated a number of times, and solutions with the largest number of wins form the next generation (Mallipeddi et al. (2010);Back (1996)).

2.3.6 Other Metaheuristics

There are many other metaheuristics available. A survey is provided by Boussaïd et al. (2013). Some other methods include:

- **Swarm Intelligence:** This actually includes a number of meta-

heuristics, such as Ant Colony Optimisation, particle swarm optimisation, and bacterial foraging optimisation. The general idea behind swarm intelligence is that a population of individuals that can locally interact with each other and their environment can work together intelligently. This intelligent cooperation takes place without there being any overall control, but does facilitate the performance of complex tasks. Consider Ant Colony Optimisation as an example. This randomly generates a number of initial solutions. Components of these solutions are marked according to the quality of the solution. Solution components which are marked strongly by multiple individual solutions are then more likely to be included when the next round of solutions is created (Boussaïd et al., 2013). We have adapted Ant Colony Optimisation for both the non-blocking and blocking versions of the job shop scheduling problem with flexible maintenance, in Section 5.2 and Section 9.2, respectively.

- **Greedy Randomized Adaptive Search Procedure (GRASP):** This multi-start method builds an initial solution using a number of components. At each step, a list of good candidate components is generated based on the components already included in the solution. A component is then randomly selected from the best components in this list. Once the initial solution has been constructed, a local search finds the local minimum nearest to this initial solution. This

process is repeated many times. The final solution is then the best one found during all the local searches (Festa and Resende, 2009).

- **Iterated Local Search:** Just as in GRASP, a local search is employed to find the local minimum from some starting solution. However, when a local minimum is found the algorithm does not restart with a new random solution. Instead, the local minimum solution is perturbed with the aim of 'jumping' to another valley in the solution landscape. Local search is then again employed to find the local minimum of that valley, and so on (Hoos and Stützle (2004);Lourenço et al. (2010)).

- **Nested Partitions:** The solution space is divided into regions by fixing some of the decision variables. Sampling is used to determine promising regions, which are then explored further. This can work well when good solutions are clustered together in the solution space (Shi and Ólafsson, 2009).

Sometimes parts of different metaheuristics are combined, possibly with exact methods. Such algorithms are known as hybrid metaheuristics. Blum et al. (2011) provide a survey of the literature on hybrid metaheuristics for combinatorial optimisation. Hybrid methods for multi-objective combinatorial optimisation are discussed by Ehrgott and Gandibleux (2008).

2.3.7 Hyperheuristics

The No Free Lunch theorem (Section 2.3.1) illustrates the advantages that can be gained by tailoring metaheuristics to the problem of interest. However, developing and tailoring metaheuristics is a time consuming process. Practitioners may not have the time, or the experience, to develop brand new metaheuristics for each problem they come across. Hyperheuristics have been developed in an attempt to create general purpose solvers. Hyperheuristics solve the problem of which heuristic to use for a type of problem, rather than solving the problem directly (Burke et al., 2013). The term hyperheuristic was first used by Cowling et al. (2001), and a survey of methods can be found in Burke et al. (2013). They note that there are two classes of hyperheuristic: heuristic selection and heuristic generation. In heuristic selection, the hyperheuristic iteratively selects one heuristic from a fixed set of heuristics that it has been equipped with. The selected heuristic then solves the problem of interest. Finally, the hyperheuristic uses the observed solution quality to learn about the effectiveness of the selected heuristic. Hyperheuristics based on heuristic generation have access to a set of smaller heuristic components. At each iteration, a subset of these heuristic components is combined to generate a new heuristic. The created heuristic then solves the problem of interest. Finally, the hyperheuristic uses the observed solution quality to learn about the effectiveness of the generated heuristic. An example of a hyperheuristic in a scheduling

context can be found in Vázquez-Rodríguez and Petrovic (2010). Their hyperheuristic employs a genetic algorithm to find a good ordering of dispatching rules to be used during schedule creation.

2.3.8 Metaheuristics for the Job Shop

Various heuristic approaches have been successfully applied to address a wide set of different job shop scheduling problem variations, such as Simulated Annealing (Kolonko, 1999; Yamada and Nakano, 1996), Genetic Algorithms (Gonçalves et al., 2005), Evolutionary Algorithm (Tanev et al., 2004), Tabu Search (Nowicki and Smutnicki, 1996; Pezzella and Merelli, 2000), Artificial Neural Networks (Fonseca and Navaresse, 2002), and other approximation approaches (Jansen et al., 2005; Chen and Luh, 2003). For example, the shifting bottleneck procedure was incorporated in the Simulated Annealing algorithm for the job shop scheduling problem in Yamada and Nakano (1996). A Genetic Algorithm solving a job shop with release dates, due dates and tardiness objectives was proposed in Mattfeld and Bierwirth (2004), while the authors in Yang et al. (2012) consider a Genetic Algorithm for a job shop with due dates and deadlines, where due dates are desired completion times, and deadlines are compulsory completion times. Recently, the authors of Kuhpfahl and Bierwirth (2016) look at a job shop where the total weighted tardiness must be minimised. They use the disjunctive graph model to define neighbourhoods in their local search procedure. Moreover, a wide range of solution methods for the classical

job shop problem has been studied. The Shifting Bottleneck procedure by Joseph Adams (1988) is a very popular approach, which is often integrated in other methods. It iteratively solves single-machine problems to optimality, and re-optimises the operations on machines which were scheduled previously. For an overview of other approaches, the reader is referred to Jain and Meeran (1999) and Błażewicz et al. (1996).

The first application of Ant Colony Optimisation (ACO) to the standard job shop problem with makespan objective was introduced in Colorni et al. (1994). A hybridisation of ACO with a Taboo Search algorithm was proposed to address the classical job shop problem in Huang and Liao (2008). The method utilises a novel decomposition method motivated by the shifting bottleneck procedure. The hybridisation achieves competitive performance on a large set of traditional benchmark instances. ACO for group shop scheduling was considered in Blum and Sampels (2004). This is a more general formulation of scheduling problems that includes the job shop and the open shop scheduling problems as special cases. The methodology used incorporates a local search procedure to improve the constructed schedules. A hybridisation with beam search was later proposed by the same author to efficiently address the open shop scheduling problem (Blum, 2005). Alternative solution representations used with ACO were considered in Montgomery et al. (2006) and Montgomery (2007), to tackle real-world job shop scheduling problems. The proposed representations assign alternative dispatching rules for each machine that heuristically

determines the order of operations on the assigned machine. These representations seem to be promising in terms of producing fast near-optimal solutions when optimising processing times and job due dates. In addition, a knowledge-based model was employed in ACO to the flexible job shop scheduling problem (Xing et al., 2010). The introduced model aims to learn potential knowledge acquired from the optimisation procedure of ACO, and subsequently uses it to guide the search towards more promising areas. Experimental results on a large range of problem instances validated its efficiency.

2.4 Branch and Bound

Branch and Bound (B&B) is an exact solution method, first introduced by Land and Doig (1960). A survey of recent advances in B&B is presented in Morrison et al. (2016). Branch and Bound has been applied to solve a wide range of discrete optimisation problems. The advantage of B&B is that it can provide exact solutions to scheduling problems. Its disadvantage is that the required computational time can be much longer than that of metaheuristics. This thesis proposes novel Branch and Bound methods for the non-blocking and blocking variants of the job shop scheduling problem with flexible maintenance, in Chapter 4 and Chapter 10, respectively. A basic overview of the general B&B procedure is presented in Algorithm 1, and described next.

Algorithm 1 Pseudo code of a generic Branch and Bound algorithm.

```
1: Set initial incumbent solution
2: Initialise an empty solution stack
3: Initialise an empty solution and place this on top of the stack
4: while stack not empty do
5:    Take a partial solution from the top of the stack
6:    Select a non-fixed variable in the partial solution
7:    Create one branch for each possible value of the variable
8:    for all newly created branches do
9:       if lower-bound of branch is no better than incumbent solution
         then
10:         Discard branch
11:      else
12:         if branch is complete then
13:            Branch replaces incumbent solution
14:         else
15:            Branch is returned to the top of the stack
16:         end if
17:      end if
18:   end for
19: end while
20: Return the optimal incumbent solution
```

Suppose we wish to solve a discrete optimisation problem, and that we are interested in finding a global minimum. Assume that it is possible to calculate a lower bound for any given partial solution. Also assume we have an incumbent solution, i.e. a best-so-far solution, which provides an upper bound to the optimal solution. If an initial incumbent is not available, the upper bound is set to infinity (Step 1). Starting with an empty solution (Step 3), we select one of the decision variables (Step 6). For each of the discrete values the selected decision variable can take, we create a new partial solution with the variable fixed at that value (Step 7). This step is known as branching, and the partial solutions are referred to as nodes. A lower bound is then calculated for each node. For some problems the lower bound can be improved by solving a linear relaxation of the integer/binary problem with a fast solver. Following the branching operation, it may be that one of the new nodes represents an infeasible solution, or that the lower bound is no better than the best known solution. In either case, it will not be possible to find a new incumbent solution on said branch, so it need not be further investigated. The node can be abandoned (the node is said to be 'pruned', or 'fathomed') (Step 10). If any of the nodes represents a complete solution, it replaces the incumbent solution if it is strictly better (Step 13), or is discarded otherwise. If any open nodes remain, select one to branch on next. Repeat until no open nodes remain (Step 4). At this point, the incumbent solution is proven to be the optimal solution (Step 20).

Branch and Bound methodologies have been successfully applied in the

past to solve versions of the standard job shop scheduling problem to optimality (Brucker et al., 1994b; Artigues and Feillet, 2007; Brucker and Thiele, 1996; Mascis and Pacciarelli, 2002). Fast Branch and Bound algorithms were proposed in Brucker et al. (1994b) and Brucker and Thiele (1996), to solve challenging problem instances of the general job shop scheduling problem. The job shop scheduling problem with sequence-dependent setup times was considered in Artigues and Feillet (2007), where a Branch and Bound algorithm was applied to solve the makespan problem to optimality. Moreover, Singer and Pinedo (1998) present two Branch and Bound procedures for the job shop with the weighted tardiness objective. One branching scheme fixes operations at each branching point. The other branching scheme uses the disjunctive graph model to fix one disjunctive arc at each branching point.

Part II

The Non-Blocking Job Shop with Flexible Maintenance

The chapters in this part of the thesis consider the non-blocking job shop with flexible maintenance. The problem is described and formally defined in Chapter 3. A Branch and Bound algorithm for this problem is presented in Chapter 4. While this exact method provides optimal solutions, it usually has a longer running time than the heuristic methods that are presented in Chapter 5. The computational experiments in Chapter 6 show that the Branch and Bound method can be very effective, especially when it is seeded with a good solution from one of the heuristic methods.

Chapter 3

Introduction to the Non-Blocking Job Shop with Flexible Maintenance

3.1 Problem Overview

The management of the work passing through the NNL facility described in Section 1.2, can be modelled as a job shop scheduling problem (JSSP), with a number of additional non-standard features based on the requirements of the facility.

To briefly recap, the facility comprises shielded workstations, that process sensitive and radioactive materials, handled remotely by expert workers. The facility also contains a single transportation mechanism that connects all workstations, and which transfers the materials across all work-

stations. There are two workstations that both act as entry and exit points for any materials. As such, given a material to be processed, it will have to enter the system at one of these two workstations, and then be moved to the appropriate workstation by the transporter. The materials then may have to undergo further processing at other workstations, and eventually leave the system through one of the two exit points.

The non-standard features faced by schedulers at the facility include: flexible maintenance scheduling; jobs which merge into one, or split into many; jobs with release dates; and jobs with precedence constraints. Existing research might not take into consideration all or most of such non-standard features that are encountered in real-world scenarios. It is more common to consider and study a single complication in isolation.

A crucial aspect of the considered JSSP is the maintenance programme on all machines at the facility. In industrial scenarios, machine maintenance is critical for the good functioning of the facility. However, machine maintenance is often ignored in traditional job shop scheduling formulations. Each workstation at the facility, as well as the transporter, has a preventive maintenance program. These flexible maintenance activities are required to start within a specified time window. The start and end of this time window are the soft and hard due dates, respectively, for the start of the maintenance activity. Due to strict safety regulations in the nuclear industry, a machine must be shut down if its maintenance is not started by the hard due date. In this thesis we integrate the planning of mainte-

nance with the planning of jobs. Each maintenance event is modelled as a single-operation job.

Another feature motivated by the industrial setting is that some jobs can only start when multiple other jobs come together on completion. There are also jobs which, on completion, allow a number of other jobs to start. This merging and splitting behaviour of jobs is modelled with precedence constraints. There is infinite buffer capacity in the system. If a job requires a machine that is currently busy, the job can wait for processing next to its required machine.

Novel features of the problem under consideration include flexible maintenance, soft and hard due dates, and the integrated planning of maintenance alongside the jobs planning procedure.

3.2 Problem Notation

More formally, we wish to schedule a set of $n \geq 1$ jobs $\mathcal{J} = \{J_i\}_{1 \leq i \leq n}$ on a set of $m \geq 1$ machines $\mathcal{M} = \{M_k\}_{1 \leq k \leq m}$. Each job $J_i \in \mathcal{J}$ consists of a number of o_i operations $J_i = \{O_{ij}\}_{1 \leq j \leq o_i}$ to be processed in a given order. Each operation O_{ij} has to be carried out on a specified machine $M_k \in \mathcal{M}$. Each operation has a positive deterministic processing time $p_{ij} \in \mathbb{R}^+$. Each machine M_k can process only one operation at a time. The nature of the operations makes them non-preemptive, i.e. once an operation has started it cannot be interrupted. Each job J_i has a release date r_i. Jobs can have

precedence constraints, which enforce a strict ordering between two jobs. For example, if J_i must be processed before J_j, the precedence constraint is $c_i \leq s_j$, where c_i denotes the completion time of J_i and s_j denotes the starting time of J_j.

The merging of jobs is modelled with the help of precedence constraints. Consider a set of three jobs, $\{J_i, J_j, J_k\}$ say, which merge into a fourth job, J_l. This would be modelled with the constraint $\max\{c_i, c_j, c_k\} \leq s_l$. This type of constraint is known as an in-tree (Cheng and Sin, 1990). Similarly, if job J_i splits into three other jobs, $\{J_j, J_k, J_k\}$ say, this is modelled with the constraint $c_i \leq \min\{s_j, s_k, s_l\}$. This type of constraint is known as an out-tree (Cheng and Sin, 1990).

Let c_i denote the scheduled completion time of job J_i. Each job has a soft due date d_i, and a hard due date $\overline{d}_i$. A tardiness penalty T_i is incurred by J_i if its soft due date is violated, i.e. when $d_i < c_i$. The size of the penalty increases with the delay, and rises rapidly beyond the hard due date. The motivation behind this is that, in reality, the financial reward for a job is reduced if the soft due date is missed, and severely reduced if the hard due date is missed. In addition to this, the linear part of the penalty function that will be introduced in Section 3.3, ensures work is not needlessly delayed when the due dates can be easily met. Completing work sooner helps maintain a good reputation with clients.

Each maintenance activity is modelled as a single-operation job. A flexible maintenance activity J_i receives a tardiness penalty based on its

scheduled starting time, s_i. These flexible maintenance activities are required to start within the time window defined by their soft and hard due dates. More formaly, for maintenance activities there is a hard constraint which requires that $s_i \in [d_i, \overline{d_i}]$. Maintenance can start no earlier than its soft due date d_i. If maintenance has not started by its hard due date $\overline{d}_i$, the corresponding machine must be shut down. Therefore, missed hard due dates for maintenance have to be penalised much more heavily than missed hard due dates for jobs.

As mentioned previously, the objective function considered in this study seeks to minimise the total tardiness penalty T, defined by $T = \sum_{i=1}^{n} T_i$. Depending on the type of job (normal, maintenance), we employ two tardiness penalty functions with different characteristics, as described next in Section 3.3.

3.3 Objective Function

Given that a regular job J_i has soft due date d_i, hard due date $\overline{d}_i$ ($d_i \leq \overline{d}_i$) and, once scheduled, completion time c_i, then job J_i incurs the following tardiness penalty T_i:

$$T_i = \begin{cases} 0 & \text{if } c_i \leq d_i, \\ c_i - d_i & \text{if } d_i < c_i \leq \overline{d}_i, \\ (\overline{d}_i - d_i) + 10^{10} + 100 \times (c_i - \overline{d}_i) & \text{if } \overline{d}_i < c_i. \end{cases} \tag{3.3.1}$$

Intuitively, a job J_i will incur a penalty (tardiness) value only if its completion time c_i exceeds the soft due date threshold d_i. The effect of the penalty in that case has a linear trend proportional to its completion time $(c_i - d_i)$, until the completion time reaches the hard due date $(d_i < c_i \leq \overline{d}_i)$. In the case where the completion time exceeds the hard due date $(\overline{d}_i < c_i)$ a very large fixed penalty is incurred $((\overline{d}_i - d_i) + 10^{10} + \ldots)$. There is a slope to help the optimisation process move towards more desirable solutions $(\ldots + 100 \times (c_i - \overline{d}_i))$.

Given a flexible maintenance activity J_i, a different tardiness penalty is incurred, as follows:

$$T_i = \begin{cases} 0 & \text{if } s_i = d_i, \\ s_i - d_i & \text{if } d_i < s_i \leq \frac{d_i + \overline{d}_i}{2}, \\ \frac{d_i + \overline{d}_i}{2} - d_i + 10 \times \left(s_i - \frac{d_i + \overline{d}_i}{2} \right) & \text{if } \frac{d_i + \overline{d}_i}{2} < s_i \leq \overline{d}_i, \\ \frac{d_i + \overline{d}_i}{2} - d_i + 10 \times \left(\overline{d}_i - \frac{d_i + \overline{d}_i}{2} \right) + & \\ \qquad +10^{15} + 1000 \times \left(s_i - \overline{d}_i \right) & \text{if } \overline{d}_i < s_i. \end{cases} \tag{3.3.2}$$

In the maintenance tardiness penalty function, we add an extra case in the piecewise function that essentially splits the time period between the soft due date d_i and the hard due date $\overline{d}_i$ in half ($d_i < s_i \leq \frac{d_i+\overline{d}_i}{2}$, $\frac{d_i+\overline{d}_i}{2} < s_i \leq \overline{d}_i$). The slope of the penalty function in the second time period is steeper than in the first one, so that larger penalty values apply if the starting time of the maintenance is closer to its hard due date. The aim here is to avoid planning maintenance close to its hard due date when this can be avoided. If maintenance were to be planned close to its hard due date, there is a risk that a small delay would cause the hard due date to be missed, leading to a mandatory shutdown. Similarly, a substantially larger penalty value ($\ldots + 10^{15} + \ldots$) is incurred in the case where the starting time of the maintenance exceeds its hard due date ($\overline{d}_i < s_i$).

Figure 3.3.1 illustrates how the job and maintenance components of the penalty function behave relative to each other. Due to the very large fixed penalties at the hard due date, the Y-axis is not to scale.

These tardiness functions are designed to prioritise the various scheduling objectives. The highest priority is placed on scheduling maintenance tasks to start before their respective hard due dates, to avoid a mandatory shut-down. The second highest priority is to schedule regular jobs to finish before their respective hard due dates, to avoid substantial loss of revenue. While we want to exploit the flexibility of the maintenance window,

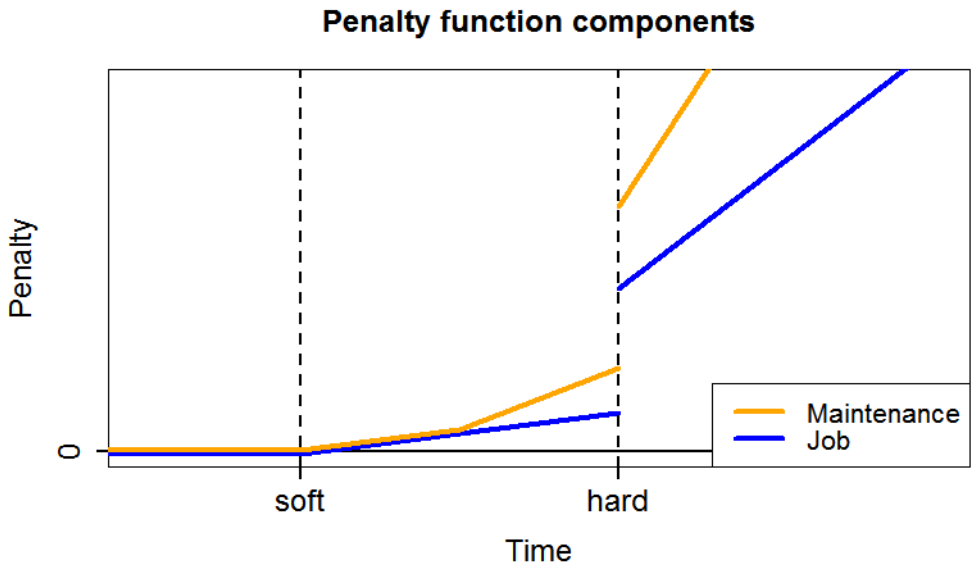

Figure 3.3.1: An illustrative comparison of how the soft and hard due dates influence the job and maintenance components of the penalty function. Y-axis not to scale.

a mandatory shut-down should be avoided at all costs. Therefore, the third priority is to further reduce any remaining delay in maintenance, to below the midway point of its respective maintenance window. The final priority is then to reduce any remaining delays in either jobs or maintenance tasks.

3.4 Problem Classification

The overall objective function (Section 3.3) takes into account job and maintenance tardiness, as well as weighted unit penalties for hard due date violations. Let w_{tj} denote a weight, where t indicates a type of tardiness and where j indicates a job. Then, in terms of the widely used 3-field

problem classification $\alpha|\beta|\gamma$ (Graham et al. (1979); Section 2.1.1), the problem considered here can be represented by $J|prec, r_j|\Sigma_j w_{tj} T_j + \Sigma_j w_{\overline{t}j} \overline{T}_j + \Sigma_j w_{\overline{u}j} \overline{U}_j + \Sigma_j w_{mj} M_j$. The third field contains the objective, and can be broken down as follows:

- Tardiness with respect to the soft due date, T_j, is penalised by $\Sigma_j w_{tj} T_j$.
- Tardiness with respect to the hard due date, $\overline{T}_j$, is penalised by $\Sigma_j w_{\overline{t}j} \overline{T}_j$.
- A weighted unit penalty $\Sigma_j w_{\overline{u}j} \overline{U}_j$ is incurred when hard due dates are missed. Our implementation uses smaller weights for jobs than for maintenance, as maintenance must never be scheduled beyond its hard due date.
- Tardiness M_j with respect to the midpoint between the soft and hard due date is penalised by $\Sigma_j w_{mj} M_j$. This is only applied to maintenance, i.e. the weights are zero for regular jobs. The purpose of this is to discourage maintenance from being scheduled close to its hard due date.

The problem defined above contains as a special case $1|r_j|\Sigma T_j$. This represents a problem with just one machine, in which the jobs have no precedence constraints, the hard due dates are set to infinity, and $w_{tj} = 1$. It has been shown that $1|r_j|\Sigma T_j$ is unary NP-hard (Graham et al. (1979)). Therefore the scheduling problem under consideration in the non-blocking

part of this thesis is also NP-hard in the strong sense.

3.5 Solution Methods

To solve the considered problem, we have developed both exact and heuristic optimisation algorithms that take into account the particular features of the considered problem. Two exact algorithms are presented in Chapter 4: one is a Branch and Bound (B&B) algorithm, the other is a hybridisation of B&B with an Ant Colony Optimisation (ACO) algorithm. In the hybridisation, ACO provides an initial upper bound to the B&B algorithm, to reduce the required execution time. A number of heuristic methods to address the same problem were also developed. These are presented in Chapter 5. They include four ACO algorithms, and a Simulated Annealing (SA) algorithm. The ACO algorithms incorporate different types of heuristic information to efficiently search the optimisation space of the problem. Initial experiments for part of this work were presented in Struijker Boudier et al. (2015).

Thorough experimental comparisons and analysis on 100 constructed problem instances are presented in Chapter 6. It will be shown that three of the ACO algorithms usually find close-to-optimal solutions. Simulated Annealing generally shows a similar performance to this, but occasionally returns markedly worse results. The fourth ACO algorithm generally does not perform well. The pure B&B algorithm also performs worse than the

three best ACO algorithms. It found the optimal solution for 63 out of 100 test problems, and in many instances required many days of running time. The hybrid B&B method, which starts with an initial upper bound obtained by ACO, performed very well. It solved all test instances to optimality, in a relatively short amount of time. This makes it a suitable choice to address challenging real-world problems.

Chapter 4

Non-Blocking Job Shop: Exact Methods

4.1 Introduction

This chapter presents a novel Branch and Bound (B&B) algorithm which solves to optimality the non-blocking job shop scheduling problem with due dates and flexible maintenance activities. This novel problem was defined in Chapter 3. The initial motivation for the development of this B&B algorithm was to obtain optimal solutions for our test problems. Knowing the optimal solutions would make it possible to assess whether or not the heuristic methods under development (see Chapter 5) were showing close-to-optimal performance. However, it eventually became clear that the B&B method is a useful algorithm in practice too. Our B&B algorithm employs a novel variable ranking method to speed up the branching process. The

algorithm also brings together some existing methods, such as Immediate Selection and Strong Branching.

The computational experiments in Chapter 6 indicate that the B&B algorithm can be very efficient, especially when hybridised with one of the heuristic methods proposed in Chapter 5. Branch and Bound methodologies have been successfully applied in the past to solve other versions of the job shop scheduling problem (Brucker et al. (1994b); Artigues and Feillet (2007); Brucker and Thiele (1996); Mascis and Pacciarelli (2002)).

The B&B algorithm makes use of the widely used disjunctive graph representation of the problem. An overview of the disjunctive graph is given in Section 4.2. The complexity of the solution space under the disjunctive graph model is discussed in Section 4.2.1. The main structural components of the B&B algorithm follow in Section 4.3. This chapter then concludes with a look at the complexity of the Branch and Bound algorithm (Section 4.4).

4.2 Disjunctive Graphs

The disjunctive graph formulation of the job shop problem (Roy and Sussmann, 1964) is used to model the problem. It was introduced to address the standard job-shop problem (Roy and Sussmann, 1964) and characterised as an effective representation that has the ability to efficiently limit the solution space of scheduling problems. In this representation, a complete

job shop schedule can be represented by a directed acyclic graph (DAG).

The scheduling decision that has to be performed essentially is to define an ordering over all operations that have to be processed on each machine. In other words, precedence relations have to be fixed between all the operations on each machine. In the disjunctive graph formulation this can be performed by transforming undirected arcs, known as disjunctive arcs, into directed arcs. A set of such "fixed" disjunctions is called a selection. A selection defines a feasible scheduling solution if every disjunctive arc is fixed and the resulting graph is acyclic (Roy and Sussmann, 1964; Brucker et al., 1994b). Such a selection is called a complete selection. For a complete selection (feasible schedule), the completion time of each job can be measured as the length of the longest path from the source node to the final operation of the respective job, plus the processing time of that final operation.

For example, an incomplete schedule appears in Figure 4.2.1, where two dummy nodes represent the source U and the sink V, respectively. Each operation has a single node in the DAG, so that each node requires a specified machine $(A - C)$ and forms part of a job $(1 - 3)$. Directed job arcs (represented by solid lines) determine the order in which the operations of a single job are carried out. The job arcs have length identical to the processing time of their origin node. Since the ordering of operations within a job is fixed, the job arcs are also fixed. An arc of length zero leads from the dummy source node to the first node of each job. The final node of

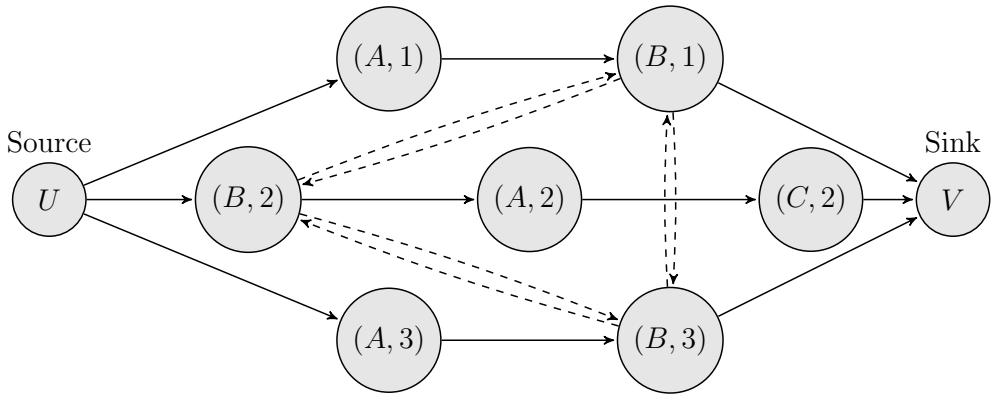

Figure 4.2.1: Directed graph representing a partial schedule for a job shop with 4 machines ($A - C$) and 3 jobs ($1 - 3$). The graph depicts all job arcs (solid lines). The disjunctive arcs for job B are displayed as dashed lines. This figure is a modified version of the presented DAG in Pinedo (2008).

each job has a job arc leading to the sink node. The length of these arcs is determined by the processing time of their origin node. This forms the basis of the disjunctive graph.

To complete the disjunctive graph, machine arcs must be added. If two nodes require the same machine, then one of these nodes must be served before the other. More generally, for all pairs of nodes which require the same machine, a strict ordering must be established. This means a completed graph must include exactly one directed machine arc between every pair of nodes requiring the same machine. Since any two nodes can only be ordered in two ways, there are two mutually exclusive directed machine arcs between the nodes, known as a disjunctive pair of arcs. A valid schedule must include exactly one arc from each disjunctive pair, in such

a way that the resulting graph is acyclic. The length of the machine arcs is also equal to the processing time of their origin node. In the completed DAG, the starting time of any operation can be calculated as the longest path from the starting node to the node corresponding to the operation. Figure 4.2.1 illustrates all the disjunctive arcs for machine B, as dashed lines. In any complete schedule, one arc will have been chosen from each pair. Any selection which introduces a cycle to the graph is invalid.

Since exactly one arc must be chosen from each disjunctive pair, the job shop problem can be formulated as a zero-one integer program. Each pair of disjunctive arcs has an associated zero-one decision variable, which determines which of the two arcs is selected. The proposed Branch and Bound algorithm solves this zero-one integer program. The branching scheme of our Branch and Bound algorithm branches on the disjunctive arcs. One branch will be the partial solution with the first of the two arcs included, while the other branch will have the second arc included instead. This is similar to the arc insertion scheme proposed in Singer and Pinedo (1998).

4.2.1 Disjunctive Graph Complexity

If solutions are represented by binary variables relating to the disjunctive arcs, the solution space generally increases exponentially with the number n of operations to be scheduled. The strongest case for this exponential growth is shown in Table 4.2.1. This table shows the case where there is just one machine, each job consists of a single operation, and there are no

Number of operations	Number of variables	Size of solution space
1	0	2^0
2	1	2^1
3	3	2^3
4	6	2^6
...	...	...
n	$\sum_{0=i}^{n-1} i = \frac{(n-1)n}{2}$	$2^{\frac{(n-1)n}{2}}$

Table 4.2.1: The size of the solution space, in terms of the number of operations n, for a single machine scheduling problem modelled with the disjunctive graph.

precedence constraints. In this case, each additional operation introduces one pair of disjunctive arcs with each of the operations already present. In other words, each k^{th} operation introduces $k - 1$ additional decision variables to the problem. In a more typical problem there will be some precedence constraints. There are usually also some jobs which visit the same machine more than once. Both of these situations reduce the total number of decision variables.

A rather uninformative lower bound for the growth rate of the solution space is zero. This is the case when the order of all jobs and operations is completely defined by precedence constraints.

For a somewhat more realistic example, assume there are no precedence constraints, and that each job visits each machine at most once. Let n_k denote the number of operations to be scheduled on machine M_k, and let m denote the total number of machines. The solution space $\mathbb{S}$ then has size

$$|\mathbb{S}| = \prod_{k=1}^{m} 2^{\frac{(n_k-1)n_k}{2}} = 2^{\sum_{k=1}^{m} \frac{(n_k-1)n_k}{2}} \tag{4.2.1}$$

In a fairly typical problem, at least in the literature, each of the $|\mathcal{J}|$ jobs visits each of the m machines exactly once. There are no precedence constraints. This problem has $n = |\mathcal{J}| \times m$ operations, and each machine must process $\frac{n}{m}$ operations. This typical problem has $\frac{\left(\frac{n}{m}-1\right)n}{2}$ variables, and the solution space has size

$$|\mathbb{S}| = 2^{\sum_{k=1}^{m} \frac{(\frac{n}{m}-1)\frac{n}{m}}{2}} = 2^{\frac{(\frac{n}{m}-1)n}{2}} \tag{4.2.2}$$

The discussion above applies equally to the blocking job shop version of the problem, which will be covered in Part III. In that case, alternative arcs are used rather than disjunctive arcs.

4.3 A Novel Branch and Bound Algorithm

Branch and Bound is introduced in Section 2.4. One of the main factors determining the efficiency of Branch and Bound is the ability to quickly prune branches. We employ several techniques to speed up the pruning process, including a novel ranking method for the decision variables. The current section first describes the main structure of the developed Branch

and Bound algorithm. This is followed by a discussion of the main components of the algorithm.

The main structure of the algorithm is presented in pseudo-code in Algorithm 2. An initial upper bound value has to be provided to the algorithm by the user (Step 1). This can be accomplished by either incorporating a heuristic method, or by generating and evaluating an initial solution by a fast policy rule such as first-come-first-served. If no information is given, the initial upper bound is set to infinity. The initial upper bound acts as an initial best-known solution. The best-known solution is updated any time a better solution is encountered. This allows any branch with a score no better than the best-known solution to be pruned.

The algorithm starts with a single incomplete solution. This is built by only including the job arcs from the disjunctive graph representation, since the job arcs are fixed (Step 2). To complete the solution, exactly one arc must be selected from each pair of disjunctive arcs. For a solution to be feasible, its corresponding graph must be a directed acyclic graph. In a partial schedule, the earliest starting time of each operation can be calculated as the longest path to the associated node. Each time a machine arc is fixed in the partial solution, the longest path to every node either remains the same or is increased. Therefore, the schedule score cannot be decreased by adding arcs, and any partial solution which has a score worse than or equal to the best known solution can be discarded. The branching scheme branches on the disjunctive arcs that have not yet been fixed in the

Algorithm 2 Pseudo code of the developed Branch and Bound algorithm.

```
Input the value of the best-known solution
Create the partial solution with job arcs only
Perform Immediate Selection
Rank and sort the remaining decision variables
while at least one partial solution remains do
    Perform Strong Branching on the single partial solution
    while at least two partial solutions remain do
        Select the most complete partial solution. Ties are broken by lowest score.
        if the current branching level is a multiple of k then
            Perform Immediate Selection
            if solution is now complete or no better than best-so-far then
                Prune branch and restart while loop
            end if
        end if
        Branch on the first non-fixed variable
    end while
end while
Return optimal solution
```

partial schedule.

Having created a partial solution, the algorithm applies the Immediate Selection method (Step 3, described in Section 4.3.1, Carlier and Pinson (1989)) to reduce the number of remaining unfixed disjunctive arcs. The remaining variables are then ranked and sorted by a measure of criticality (Step 4), as described in Section 4.3.3.

The algorithm follows a depth-first Branch and Bound strategy. At each branching point, the lower scoring branch is investigated first, i.e., out of all the partial solutions, the most complete one is branched in the next step. Any ties are broken by the smallest lower bound. Branches are pruned as soon as they are known to be either infeasible or no better than the best-so-far solution. Whenever a single partial solution remains, Strong Branching (Step 6, described in Section 4.3.2, Klabjan et al. (2001)) is applied to select the next variable to branch on. Strong Branching is designed to increase the lower bound of the solution score. In the case where at least two partial solutions remain, the branching variable is the unfixed variable with the lowest rank (Step 15).

Immediate Selection checks are frequently performed to identify fixable variables (Step 10). The motivation behind this is to reduce the remaining solution space. Each time a variable is fixed in the branching process, a binding constraint might be imposed on some other unfixed variables. In other words, some of the remaining variables can now only be fixed in one way. Fixing these variables may then lead to even more variables also being

restricted to particular values.

Although Immediate Selection can substantially decrease the total number of branching operations, with the possibility of speeding up the algorithm, it carries a considerable computational cost. As such, a balance between execution speed and computational cost has to be considered in this step. After a manual fine-tuning process, we utilise the Immediate Selection check every $k = 15$ branching levels, i.e., whenever there are k fixed variables in a partial solution. A solution is complete when all variables have been fixed on a branch. Complete solutions are compared against the best-so-far solution. The newly completed solution replaces the best-so-far solution if it is better, and is discarded otherwise. Finally, the Branch and Bound algorithm returns the optimal solution found during the search process (Step 18).

Notice that lower bounds of partial solutions could be improved through a linear relaxation of the integer program. However, no linear relaxation is used here, as it was found to be computationally expensive. The main components of the algorithm are described in the following sections.

4.3.1 Variable Reduction: Immediate Selection

The Immediate Selection process is used to reduce the number of remaining unfixed variables. It fixes variables outside of the branching process. The Immediate Selection of disjunctive arcs was introduced by Carlier and Pinson (1989). They identified pairs of nodes which can only be ordered in

one way in the optimal solution. A number of different implementations of Immediate Selection have been proposed in the literature (Brucker et al., 1994a). In our Branch and Bound method, Immediate Selection has been implemented as described below.

Let $\mathcal{I}$ denote the set of disjunctive arcs which have not yet been included in the partial schedule. For each pair of disjunctive arcs $(i,j),(j,i) \in \mathcal{I}$, both orderings of the nodes i and j are investigated. If the ordering (i,j) results in a solution no better than the current upper bound, arc (j,i) is added to the solution, and vice versa. If both arcs cause the partial schedule score to increase to at least the upper bound, then the partial schedule is discarded, as it cannot be the basis of an improved upper bound. A similar procedure was investigated with two variables tested simultaneously. This turned out to be very computationally expensive, and hence it is not used further in this work.

4.3.2 Variable Branching: Strong Branching

The order in which the variables are branched on can have a very large impact on the overall computational cost. Strong Branching aims to quickly increase the score of a partial solution (Klabjan et al. (2001)). It does so by branching on the variable with the largest score on its lowest scoring branch, i.e. the variable with the largest guaranteed increase of the lower bound. While Strong Branching reduces the overall number of branching operations, it is computationally expensive to evaluate the minimum increase in

penalty score for each remaining variable at each branching point. In the proposed algorithm, Strong Branching is performed at the very start of the algorithm. Whenever the first variable has been permanently fixed, because one of its two branches is pruned, the algorithm essentially restarts with one fewer variable. Strong Branching is used whenever such a restart takes place. At any other time, variables are branched in the order determined by their rank, as described in the next section.

4.3.3 Branching Order: Variable Ranking

The Branch and Bound algorithm branches on a decision variable x_{ij} associated with the pair of disjunctive arcs $(i, j), (j, i)$. The branching order of the variables can have a substantial impact on the total running time of the algorithm, because early pruning drastically reduces the remaining unexplored solution space. The likelihood of early pruning can be increased by ordering the decision variables by some measure of criticality (Sadeh and Fox, 1996; Grimes and Hebrard, 2015). When branching is performed on the most critical decision variable, one of the branches can often be pruned quickly. This requires some sensible measure of criticality. To determine the criticality of each variable, we have developed a novel ranking procedure based on two factors: The job urgency, and the busyness of the machines. Our ranking procedure is as follows:

- **Job Ranking:** Jobs are ranked based on slack. Slack is the non-negative difference between the available processing time (the length of time between the job release date and its soft due date) and the required processing time (the sum of the processing times of all operations of the job). Slack is a non-negative quantity that equals zero whenever the required processing time is greater than the available time. More formally, let d_j denote the soft due date of job j, and t_{req} the total processing time required by job j. The slack of job j is then defined as follows:

$$\text{slack}_j = \max\{0, d_j - s_i - t_{\text{req}}\}$$

  The motivation for ranking by slack is that jobs with less slack are at greater risk of being scheduled in such a way as to miss their due dates. Decision variables associated with such jobs should therefore be considered to be more critical. The job with the least slack is given a rank of 1, the job with the next least slack is given a rank of 2, and so on. Jobs with equal slack values share the same rank.

- **Machine Ranking:** Machines are ranked based on the total demand placed on them. The motivation behind this is that decision variables associated with machines with high demand have less flexibility. In the context of a partial schedule, the ordering of operations on a busy machine is more likely to have an effect on job completion times than the ordering of operations on a less busy machine. Demand is

measured as the total processing time required on a machine. A rank of value 1 is given to the machine with the highest demand, a rank of value 2 is given to the machine with the second highest demand, and so on. Machines with equal demand share the same rank.

Each decision variable is associated with at least two jobs, since it relates to machine arcs which lead from one job to another. For each arc, the two job ranks are summed to create a single job rank. This means that if there are R_j unique job ranks $\{1, \ldots, R_j\}$, then there are $2R_j - 1$ possible combined job ranks $\{2, 3, \ldots, 2R_j\}$. Because we will combine job and machine ranks, this naturally presents the question of whether there would be any benefit in scaling the machine ranks, as there are only 10 of these in our test problems. Machine ranks were multiplied by 3, as this was found to work well.

It now remains to combine the selected job and machine ranks into a single rank for each variable. We have tried several ways of combining both ranks, and found the following to work well empirically:

Minimum primary, maximum secondary: Variables are ranked by the minimum of their job and machine ranks. Within these ranks, the variables are then sorted by the maximum of their job and machine ranks. Let r denote the variable rank, let r_m denote the machine rank, R_m the largest of the machine ranks, r_j the job rank, and R_j the largest of the job

ranks. Then

$$r = \{(\min(r_m, r_j) - 1) \times \max(R_m, R_j)\} + \max(r_m, r_j).$$

Once all the variables are ranked, an ascending order defines the criticality of the variables, i.e., the smallest rank is considered the most critical. Any remaining ties are broken based on the duration of the largest processing time associated with the variable.

4.4 Branch and Bound Complexity

In the worst case, a Branch and Bound algorithm will have to explore the entire search tree. The worst-case complexity of Branch and Bound algorithms, for trees with a given depth d and branching factor B, is therefore $\mathcal{O}(B^d)$ (Zhang and Korf, 1992). In practice, however, it is usually not necessary to explore the entire tree. The average-case complexities of a number of Branch and Bound problems are considered by Zhang and Korf (1992). They show that, under certain conditions, depth-first Branch and Bound algorithms can have a linear average-case complexity with respect to the depth d of the branching tree. This is the case when the branching tree is a uniform random tree, and when at each branching point the expected number of child-nodes with score equal to the parent node is greater than 1. However, these conditions do not hold for the branching trees under consideration here. For our scheduling problem it was shown that the depth of the search tree, i.e. the number of variables, increases polynomially with

the number of operations in the scheduling problem (Section 4.2.1). Therefore, even if a Branch and Bound algorithm with linear complexity could be found for this problem, such an algorithm would still have complexity $\mathcal{O}(n^2)$, where n is the number of operations in the problem. The computational experiments on different problem sizes presented in Section 6.6 provide additional insights on how the algorithm performs on both smaller and larger problems.

Chapter 5

Non-Blocking Job Shop: Heuristic Methods

5.1 Introduction

The advantage of exact methods, such as the Branch and Bound (B&B) algorithm presented in Chapter 4, is that they solve combinatorial optimisation problems to optimality. However, the disadvantage of exact methods is that they often require a lot of time to guarantee that an optimal solution has been found. In industrial settings this time is often not available. In practice, it is often more desirable to quickly produce good solutions. This is also the case at the facility introduced in Chapter 1, where production schedules are updated several times per day. The planners at this facility require a scheduling tool which produces a good schedule in a matter of minutes. This need motivated the development of the heuristic methods

presented in this chapter. These heuristics quickly provide near-optimal solutions to the non-blocking job shop scheduling problem with due dates and flexible maintenance activities, as defined in Chapter 3.

The main research outputs of this chapter are the Ant Colony Optimisation (ACO) algorithms presented in Section 5.2. Simulated Annealing (SA) was also implemented for comparison, in Section 5.3. The computational experiments in Chapter 6 show that, on average, ACO obtains better results than SA, although the latter generally requires less running time. Although the heuristic methods cannot prove optimality, some of the ACO variants did manage to find optimal solutions in a short running time. It will also be shown that much can be gained by using the ACO results to obtain an initial upper bound for the B&B algorithm presented in Chapter 4. Such hybridisation drastically reduces the running time of B&B, and facilitates the finding of guaranteed optimal solutions in a short running time.

5.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) (Dorigo and Stützle, 2004) is based on the natural route finding behaviour observed in ants. Ants leave a trail of pheromone as they travel between their nest and a food source, allowing other ants to follow the same path. However, on the initial discovery of the food source, a number of ants may have arrived there by different routes,

some shorter than others. Other ants will follow the various pheromone trails to the food source and deposit additional pheromone. The shorter routes will be travelled more frequently, resulting in greater pheromone deposits compared to the longer routes. This will encourage more ants to follow the shorter route, until eventually most ants will be using the shortest route. ACO algorithms mimic this natural phenomenon in order to find good solutions to difficult problems.

In a scheduling context, an artificial ant can construct a schedule by visiting all the operations to be scheduled exactly once. The order in which the operations are visited then provides a topological ordering of all operations, which defines a complete schedule. Each job has a number of operations, which must be processed in a prescribed order. Some jobs also have precedence constraints. Without additional guidance, ants could easily construct an infeasible route by visiting the operations of a job in the wrong order. To guarantee feasibility of all constructed schedules, ants will not choose to travel to a particular operation, but to a particular job. Having chosen which job to visit next, the actual operation that is added to the schedule is then defined to be the next unscheduled operation of the chosen job. At every step, an ant can travel to every job, including the one it is already at, except completed jobs, or jobs which have not yet had their precedence constraints satisfied. A job is marked as completed as soon as its final operation is added to the schedule. Jobs with precedence constraints become available for scheduling as soon as all operations for all

preceding jobs have been scheduled, but can start no earlier than the latest completion time of all preceding jobs.

The path taken by each ant is governed by probabilistic rules. These rules take into consideration two aspects: Information from previous schedules, and heuristic information. The information from previous schedules accounts for how rewarding it has been in the past to travel from the current operation to each of the available jobs. The heuristic information is often some kind of greedy rule, such as the length of time until each job would be available for further processing. More specifically, the probabilities with which an ant currently at operation i will move to job j depend on the following:

- The **pheromone level** τ_{ij}. Pheromone levels change over time depending on the quality of the routes found by the ants. The pheromone level τ_{ij} will be higher if moving from operation O_i to job J_j has resulted in relatively good schedules previously, and lower if such a move has been associated with less efficient schedules.

- The **heuristic information** η_{ij}. While pheromone levels consider information across previous schedules, the heuristic information accounts for the state of the current partial schedule. It is usually some greedy rule. Four different types of heuristic information have been considered in this thesis. They generally include a myopic measure, "distance", and/or a measure of urgency, "slack" (see Section 5.2.1).

An ant currently at operation O_i will move to job $J_j \in \mathcal{N}_i$ with probability

$$p_{ij} = \frac{{\tau_{ij}}^{\alpha}{\eta_{ij}}^{\beta}}{\sum_{J_j \in \mathcal{N}_i} {\tau_{ij}}^{\alpha}{\eta_{ij}}^{\beta}}, \quad \text{for } J_j \in \mathcal{N}_i, \tag{5.2.1}$$

where $\mathcal{N}_i$ denotes the set of jobs available to the ant when it is at operation O_i, and parameters α and β determine how strongly pheromone levels and heuristic information influence the transition probabilities. Initially all pheromone levels are equal, so that the heuristic information will have the greatest influence on guiding the ants in early iterations. Over time, the pheromone levels start to reflect which moves have previously resulted in good schedules, and the ants will eventually be guided more strongly by the pheromone levels as time progresses.

An initial global pheromone level, τ_0, must be set. For the travelling salesman problem (TSP), a number of suggestions have been made for the value of τ_0 (Dorigo and Stützle, 2004). Most of these suggestions are in part proportional to $\frac{1}{C^{nn}}$, where C^{nn} is the value of the best solution found by the nearest neighbour algorithm. As a rank-based pheromone updating rule is used by the ACO algorithm presented in this chapter, the pheromone level is initialised to closely follow this recommendation according to the following formula:

$$\tau_0 = \frac{0.5 \times w \times (w-1)}{\rho \times C^{nn}},$$

where w is the number of schedules used in the pheromone updating procedure, and ρ is the pheromone evaporation parameter. Attempts were made

to create good schedules with a fast heuristic, e.g. first-come-first-serve, but this often resulted in poor quality solutions. After some initial experimentation, C^{nn} was fixed at the value of 150. This is somewhat larger than the solution values for typical problems at the facility of interest, and therefore gives an approximation to the values which could be produced by a fast heuristic, if one had been available.

At each iteration of the algorithm, n ants visit all operations exactly once, according to the previously described probabilistic rules. We use $n = 100$ ants. This is approximately equal to the total number of operations, as recommended for TSP problems by Dorigo and Stützle (2004). Once the n schedules have been constructed, the pheromone trails are updated. Firstly, pheromone evaporation takes place according to the following formula:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}, \quad \forall\ (i, j), \tag{5.2.2}$$

where ρ is the pheromone evaporation rate. The higher this rate, the quicker the algorithm will converge. Lower values of ρ encourage more exploration of the solution space. Evaporation affects all paths, irrespective of whether ants have recently travelled along them. The result of this is that the paths that are not used much will become less attractive to the ants over time.

After evaporation, rank-based pheromone updating takes place (Bullnheimer et al., 1997). Out of the n schedules, the $w - 1$ best are selected and ranked by due date penalty scores. The best schedule, i.e. that with

the lowest total tardiness penalty (as defined in Section 3.2), is assigned the lowest rank $r = 1$. The schedule with the largest penalty is given the highest rank, $r = w - 1$. Let (i, j) denote the path from operation O_i to job J_j. For each of the $w - 1$ iteration-best solutions, if (i, j) is part of that solution, the corresponding pheromone level τ_{ij} receives a pheromone deposit. The amount of pheromone deposited depends both on the total tardiness penalty of each schedule, $P^{(r)}$, and its rank r. More specifically,

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1}(w - r)\frac{1}{P^{(r)}} \quad \forall\ (i, j) \in \text{ schedule } r. \tag{5.2.3}$$

An additional pheromone deposit is made on the paths of the best-so-far schedule. If this schedule has total tardiness penalty $P^{(bs)}$, the pheromone levels are updated as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + w\frac{1}{P^{(bs)}} \quad \forall\ (i, j) \in \text{ best-so-far schedule.} \tag{5.2.4}$$

The previously mentioned operations are repeated until a specified stopping criterion is reached, e.g. a fixed number of iterations. At that point, the best-so-far schedule is returned as the preferred solution. The structure of the ACO algorithm described in this section is presented in Algorithm 3.

Note that it was also tested whether imposing limits on the pheromone levels was beneficial (Stützle and Hoos, 1997). Imposing limits promotes exploration, with slower convergence as a tradeoff. No evidence was found that this was beneficial for the considered problem.

Algorithm 3 Pseudo code of the developed Ant Colony Optimisation algorithm.

```
Initialise the pheromone matrix
while stopping criterion not met do
    Construct n schedules
    Rank schedules by lowest score
    if schedule with rank 1 is better than best-so-far then
        Schedule with rank 1 replaces the best-so-far schedule
    end if
    Global pheromone evaporation
    Rank-based pheromone deposit on arcs of w-1 best schedules
    Elitist pheromone deposit on arcs of best-so-far schedule
end while
Return the best-so-far solution.
```

5.2.1 Heuristic Information

Four different heuristic information rules have been considered in this study. These are mainly combinations of a myopic measure, "distance", and a measure of urgency, "slack".

The distance from operation O_i to job J_j, dist_{ij}, can be defined as the length of time from the start of operation O_i, until the time at which the next operation of job J_j could be started, given the restrictions imposed by the current partial schedule. If a job's precedence constraints have been fulfilled, an operation can be started when all of the following three conditions have been met: The preceding operation (if any) has been completed, the required machine is available, and the job's release date has been met. Therefore we define dist_{ij} to be the non-negative difference between the latest of these three times, and the starting time of the operation most recently added to the partial schedule. More formally, let s_i denote the starting time of operation O_i, c_j the completion time of the most recently scheduled operation of job J_j (if any), m_j the time at which the next machine required by job J_j finishes its most recently scheduled operation, and r_j the release date of job J_j. The distance from operation O_i to job J_j, dist_{ij}, is then defined as:

$$\text{dist}_{ij} = \max\{0, \max\{c_j, m_j, r_j\} - s_i\}$$

As each job is scored on its completion time, the heuristic information should take into account how likely it is that each job will meet its soft

due date. Jobs with a smaller ratio of available time to required processing time are at greater risk of missing their soft due date. Let slack be defined as the difference between the available remaining time to complete a job by its soft due date, and the remaining processing time required to process all unstarted operations of the job. Slack cannot be negative. More formally, given that d_j denotes the soft due date of job J_j, and t_{req} the total processing time required for all remaining unscheduled operations of job j, the current slack of job J_j can be defined as follows:

$$\text{slack}_j = \max\left\{0, d_j - s_i - t_{\text{req}}\right\}.$$

The following types of heuristic information η_{ij} have been considered:

- **Myopic (ACOm):** The myopic heuristic is based solely on the distance dist_{ij}. It increases as the distance decreases, favouring jobs which are available sooner. Since the distance can equal zero, this is expressed as follows:

$$\eta_{ij} = \exp\left(-\text{dist}_{ij}\right). \tag{5.2.5}$$

- **Due date (ACOd):** The due date heuristic is based on the slack of each job j. It increases as slack decreases, favouring jobs with less slack. Since slack can equal zero, this is calculated as follows:

$$\eta_{ij} = \exp\left(-\text{slack}_j\right). \tag{5.2.6}$$

- **Myopic and due date (ACOmd):** The two previous heuristics are combined here. The contribution of slack is weighted by the

contribution of the distance. If slack is not weighted by the distance, the algorithm has the undesirable property of jumping to jobs which are not available for a long time, if their slack happens to be small. The heuristic information is then as follows:

$$\eta_{ij} = \exp\left(-\text{dist}_{ij}\right)\left\{1 + \exp\left(-\text{slack}_j\right)\right\}. \tag{5.2.7}$$

- **Myopic and due dates rebalanced (ACOmdb):** This is a refinement of the previous heuristic. The additional parameter, $\gamma \in [0, 1]$, adjusts the weight of the contributions of both parts of the heuristic:

$$\eta_{ij} = \exp\left(-\text{dist}_{ij}\right)\left\{(1-\gamma) + \gamma\exp\left(-\text{slack}_j\right)\right\}. \tag{5.2.8}$$

5.2.2 ACO Schedule Construction: An Example

This section contains a visualisation of the schedule construction process of the ACO algorithms. For the purpose of this example, at each step the next job scheduled will be the job which has the highest probability of being scheduled. This probability will be assumed to depend both on distance and slack.

Three unscheduled jobs are displayed in Figure 5.2.1 (top). Each job consists of three operations. All jobs have the same due date at time $t = 18$, illustrated by the dashed line. The earliest availability of each job is indicated by the black triangles. Jobs 1 and 2 (J_1 and J_2) can start at $t = 0$, while J_3 is available from $t = 5$. The slack of each job is the distance from the end of its final operation to the due date (dashed line). Initially,

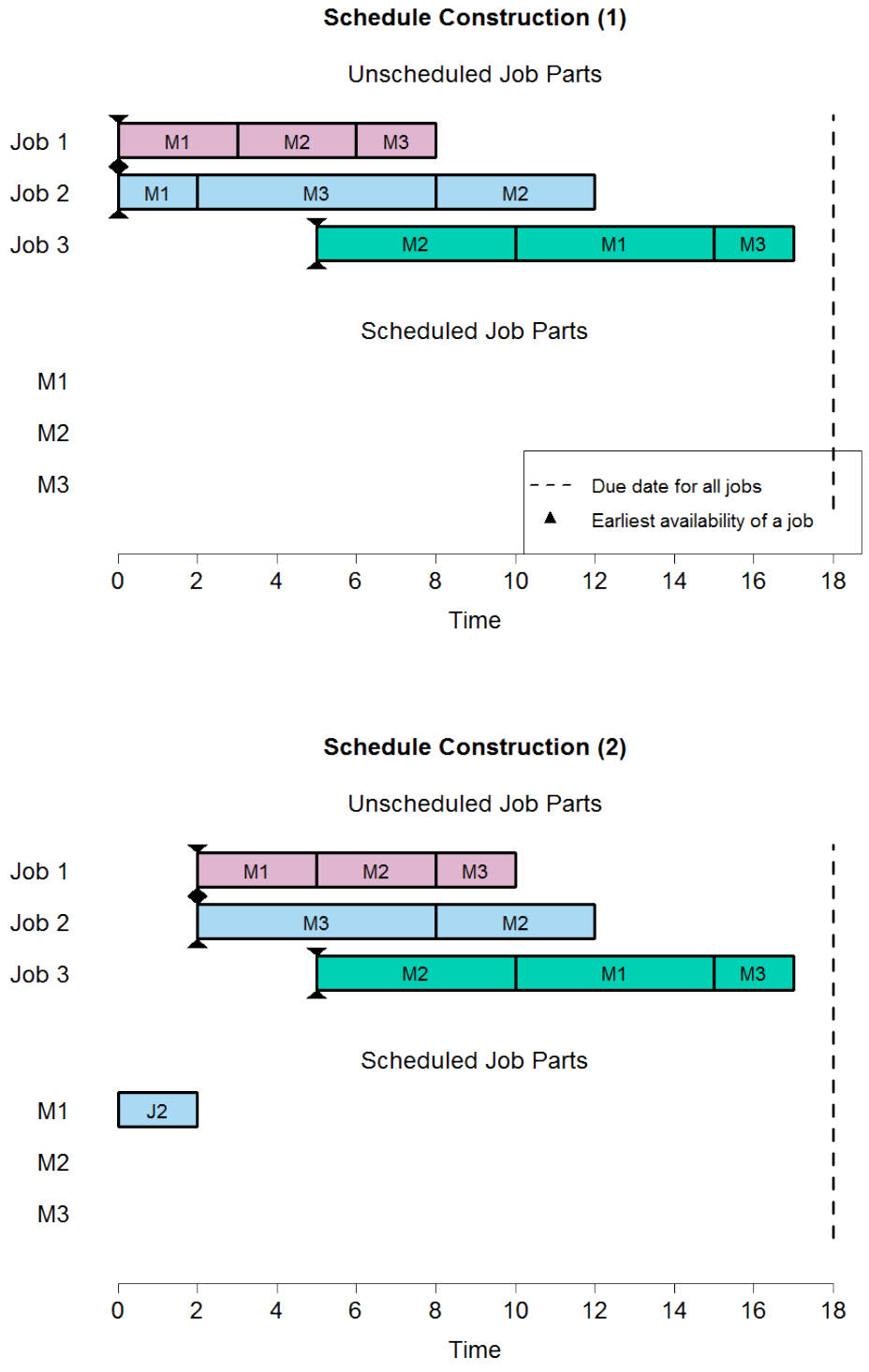

Figure 5.2.1: The unscheduled jobs (top). J_2 has the highest probability of being scheduled (tied for smallest distance wtih J_1, but smaller slack than J_1), and is scheduled first (bottom).

J_1 and J_2 both have the smallest distance. However, J_2 has less slack, and is therefore more likely to be selected first. Thus, the first operation of J_2 is added to the schedule first (Figure 5.2.1, bottom).

The next operation of J_2 can now only start once its first operation has completed. However, since J_1 requires the same machine that J_2 is now being processed on, it will also have to wait at least until the first operation of J_2 has completed. After scheduling this first operation, J_1 and J_2 are again tied for smallest distance, with J_2 still having the least slack. This makes J_2 the most likely job to be scheduled next, as happens in Figure 5.2.2 (top).

This process continues in Figure 5.2.3, Figure 5.2.4 and Figure 5.2.5. At each step, the distance and slack are combined according to the selected heuristic information. At later iterations, pheromone information will also start to play a role. This means that as the algorithm progresses, information from past schedules will become increasingly influential in the schedule construction process.

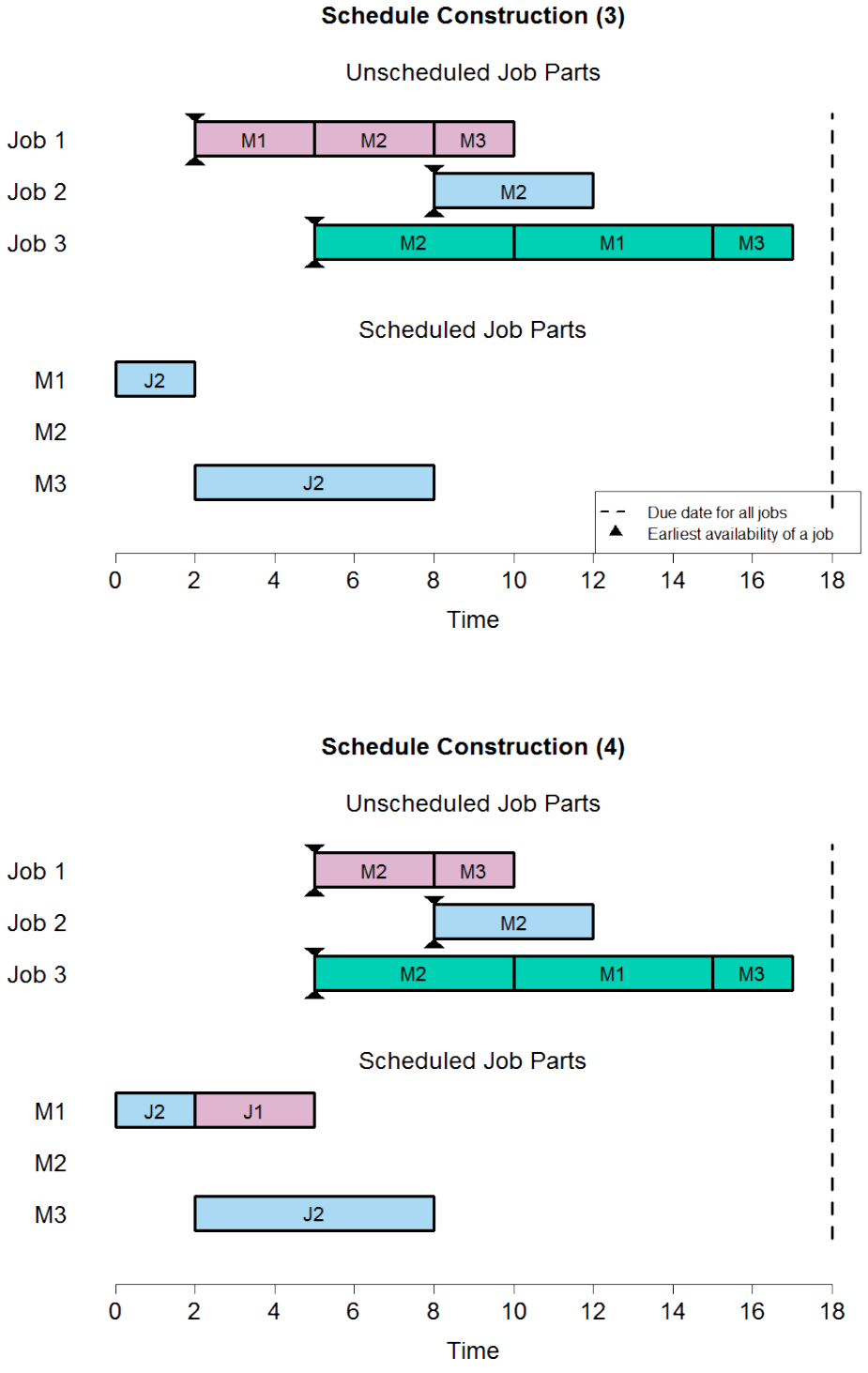

Figure 5.2.2: J_2 has its second operation scheduled (top). J_1 has its first operation scheduled (bottom).

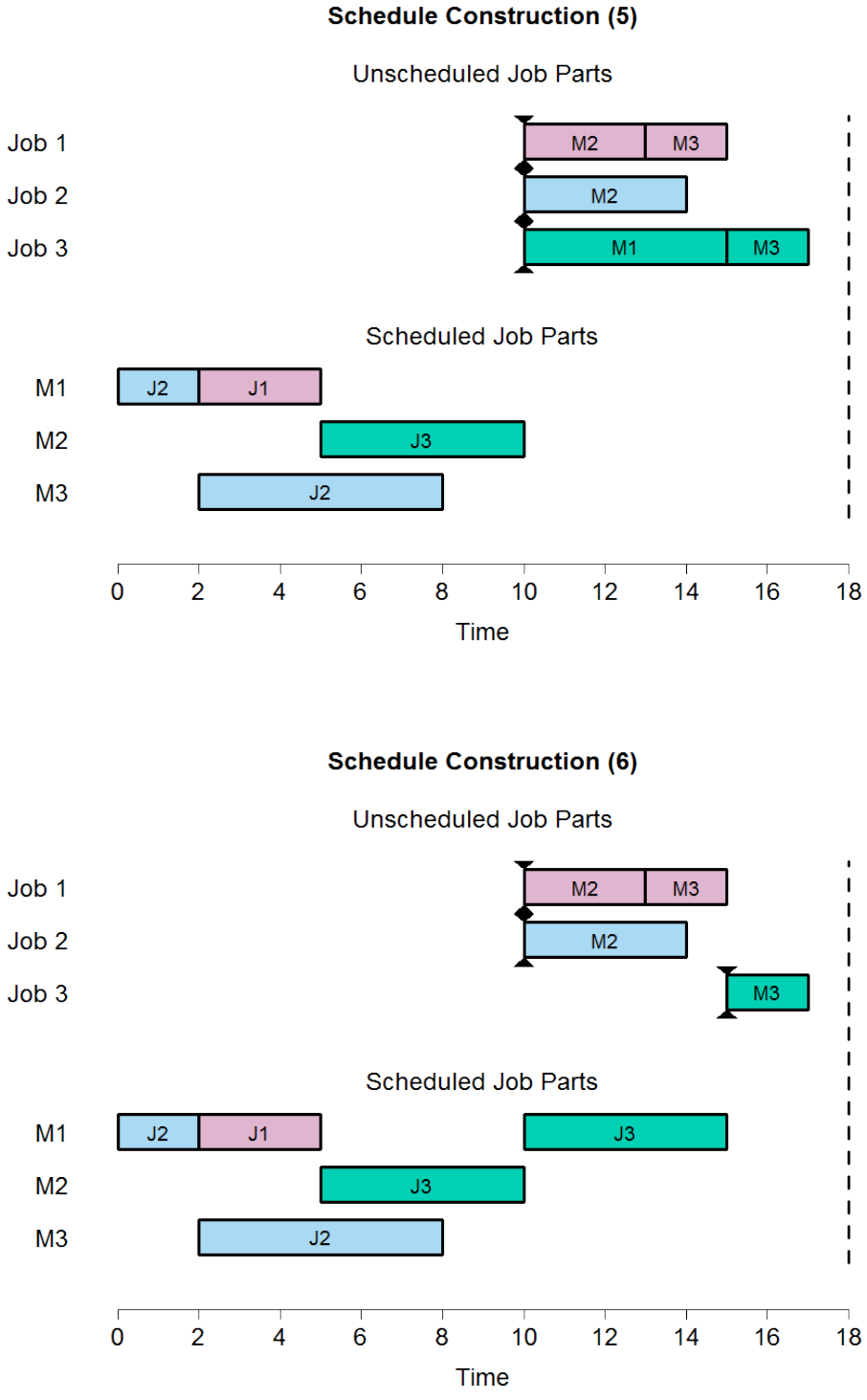

Figure 5.2.3: J_3 has its first (top) and second (bottom) operations scheduled.

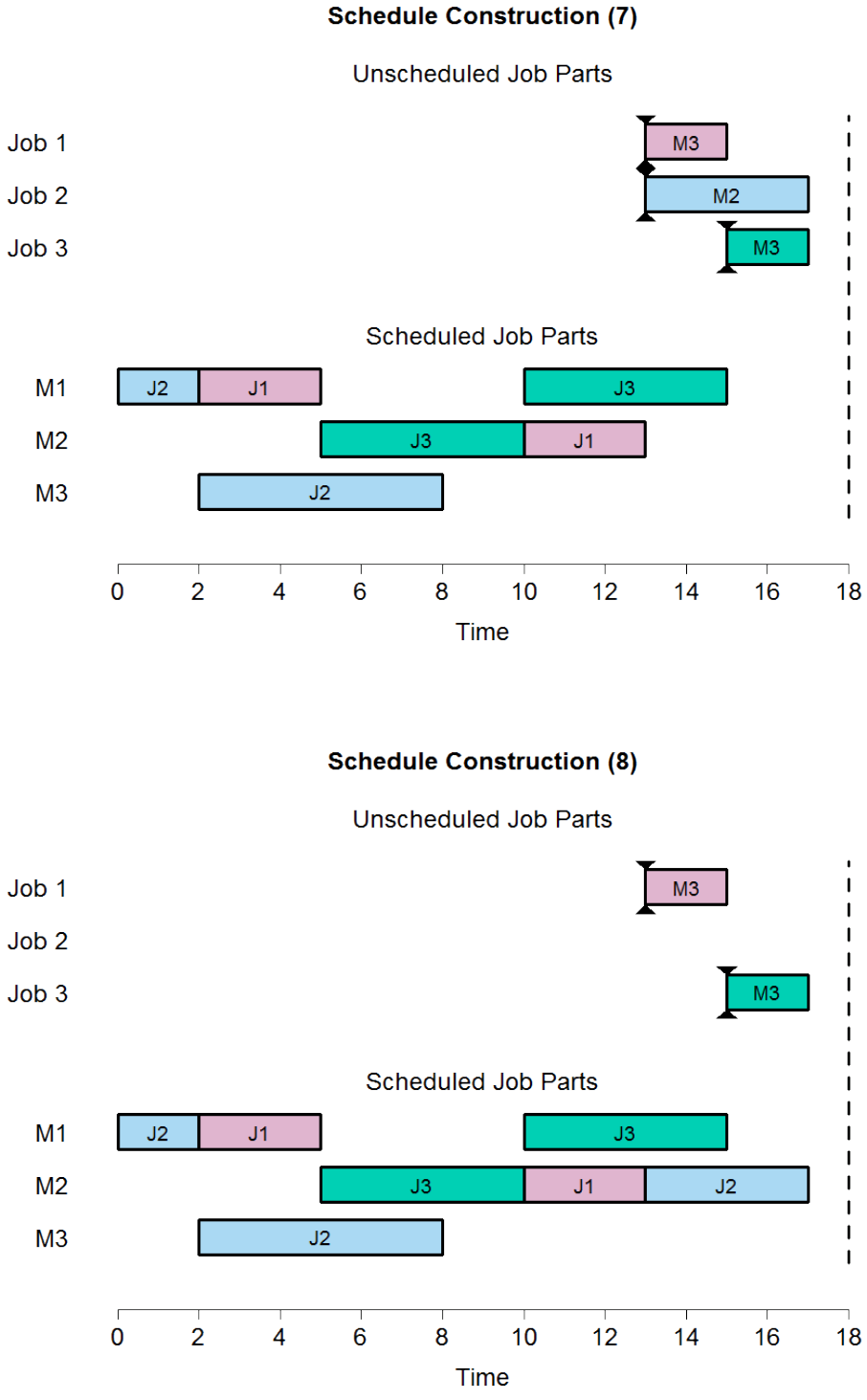

Figure 5.2.4: J_1 has its second (top) and third (bottom) operations scheduled.

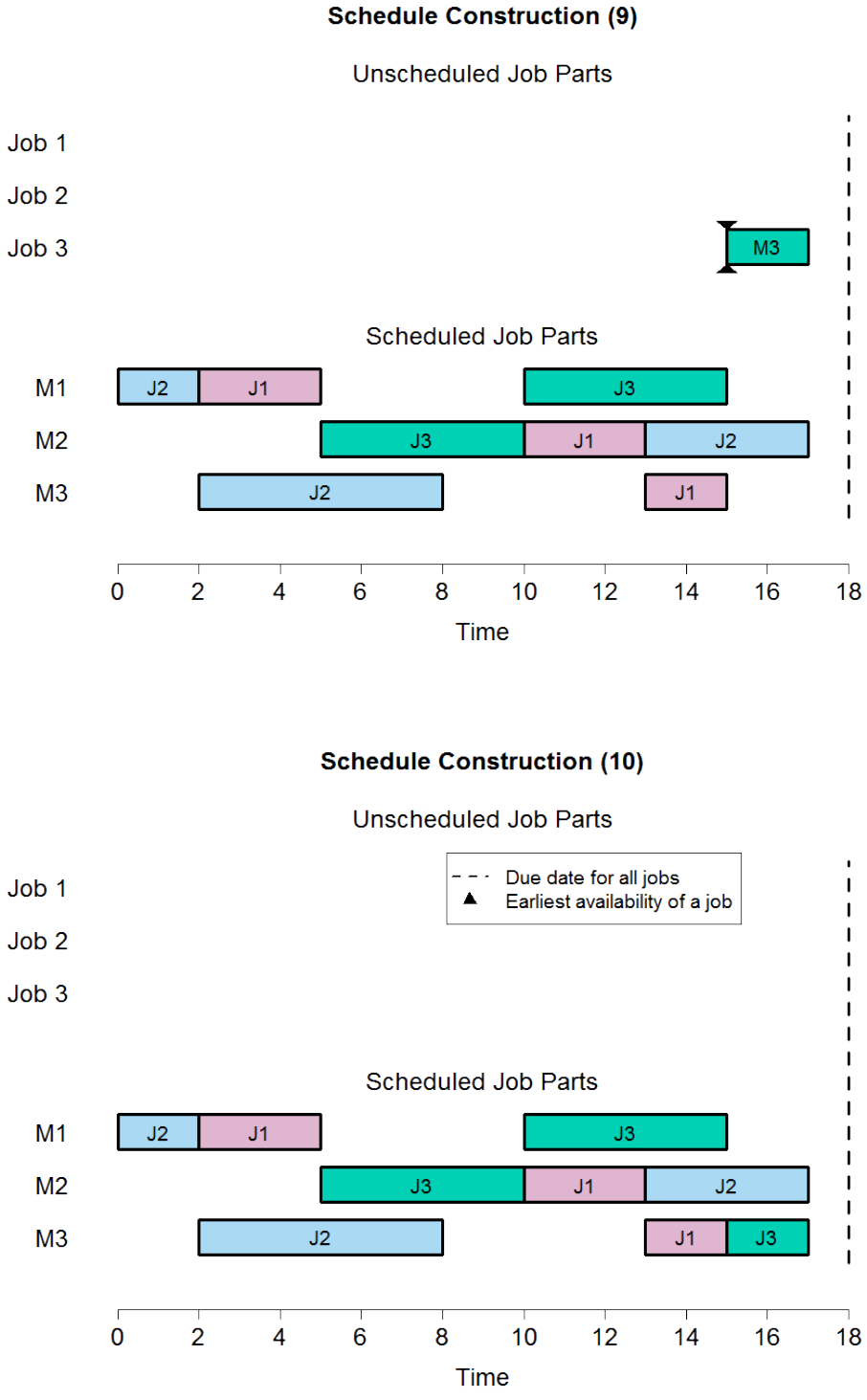

Figure 5.2.5: The schedule is completed by the additon of the third operations of J_1 (top) and J_3 (bottom).

5.3 Simulated Annealing

The final search algorithm that we have adapted for the considered job shop scheduling problem is the well-known and widely used Simulated Annealing algorithm. An overview of Simulated Annealing is provided in Section 2.3.3.

For the purpose of Simulated Annealing, schedules in our job shop problem are represented as a topological ordering of the job operations. In this topological ordering, the operations are only identified by the identity of the associated job. The identity of specific operations can then be deduced from the topological ordering. For example, $\{J_1, J_2, J_1, J_2, \ldots\}$ indicates that the first operation of job J_1 goes before the first operation of J_2, which goes before the second operation of J_1, which goes before the second operation of J_2, etc (Bierwirth, 1995). The problem of interest is then to find a topological ordering which minimises the given objective function.

Two neighbourhoods have been considered to search the space defined by the topological ordering of the nodes, namely 1-swap and 2-swap operators. To update the incumbent solution, both of the search operations are applied alternately. The 1-swap neighbourhood consists of all solutions that can be created by randomly selecting a Job ID from the solution vector, and reinserting it at a random different location in the solution vector. The 2-swap neighbourhood consists of all solutions that can be created by swapping two different and randomly selected Job IDs in the solution vector. One advantage of constructive heuristics, such as ACO, is that

by construction the candidate solutions can be guaranteed to be feasible. However, the modifications carried out by Simulated Annealing provide no such guarantee. Some of the candidate solutions created by 1-swap and 2-swap might be infeasible. Whenever a candidate solution is found to be infeasible, a new candidate is proposed from the same neighbourhood. This is applied repeatedly until a feasible candidate solution is produced.

Given that the number of job operations is denoted by n, and the change in objective function between the current solution and the proposed solution by ΔE, the structure of the developed Simulated Annealing algorithm is represented by Algorithm 4. This closely follows the proposed approach in Dréo et al. (2006).

The temperature is reduced according to a simple geometric rule $T_{k+1} = \alpha T_k$, for some $0 < \alpha < 1$, when one of the cooling criteria is met. Hence the temperature decreases quicker for smaller values of α. We use the cooling criteria proposed in Dréo et al. (2006) to reduce the temperature T. The temperature will be reduced every $12N$ accepted moves or when $100N$ moves have been attempted, where N denotes the total number of operations over all jobs $N = \sum_{i=1}^{n} o_i$. Note that the introduced parameters have a substantial impact on the efficiency and asymptotic convergence of simulated annealing (Eglese, 1990). Thus they have to be carefully chosen.

Algorithm 4 Pseudo code of the developed Simulated Annealing algorithm.

Set the initial temperature, T_0.
Create an initial solution based on first-come-first-serve.
while termination criteria not met **do**
  while cooling criteria not met **do**
    if odd iteration **then**
      Select a candidate solution from 1-swap
    else
      Select a candidate solution from 2-swap
    end if
    At temperature T, accept the candidate solution with probability $\min\left\{1, \exp(-\frac{\Delta E}{T})\right\}$
  end while
  Reduce the temperature: $T_{\text{new}} = \alpha \times T_{\text{old}}$.
end while
Return the best known solution.

Chapter 6

Non-Blocking Job Shop: Hybridisation and Computational Experiments

This chapter presents computational experiments, comparisons and analysis conducted to evaluate the performance of the proposed algorithms for the job shop with flexible maintenance. These algorithms were introduced in Chapter 4 and Chapter 5, and they solve the problem described in Chapter 3. The algorithms have been tested across a wide set of problem instances. These test problems have been constructed in such a way as to capture the main characteristics of the industrial facility described in Chapter 1.

The Branch and Bound algorithm was initially developed to obtain

optimal solutions for our test problems. This would make it possible to assess the performance of the heuristic methods. However, it eventually became clear that it would be beneficial to hybridise the exact and heuristic methods, by seeding the B&B algorithm with upper bounds obtained by one of the heuristic methods. The experimental results show that the ACO-B&B hybrid algorithm finds optimal solutions within a practical timescale.

The problem instances used for the evaluation of the developed algorithms are introduced in Section 6.1. The experimental design of the computational experiments is discussed in Section 6.2. Parameter tuning results appear in Section 6.3. The proposed exact Branch and Bound algorithm, without any initial upper bound information, is analysed in Section 6.4. This is compared against a hybrid approach, in which the proposed B&B algorithm is seeded with an initial upper bound by a heuristic search algorithm. Four variants of the proposed Ant Colony Optimisation algorithm, as well as the adapted Simulated Annealing algorithm, are analysed in Section 6.5. To conclude the experiments, the scalability of the best algorithms across different problem sizes is investigated in Section 6.6.

6.1 Problem Instances

To the best of our knowledge, there are no established test problems for a job shop with flexible maintenance activities and both soft and hard due dates. This section provides an overview of the test problems we have

developed. A fuller description of these appears in Appendix A. The test problems are available at `http://dx.doi.org/10.17635/lancaster/researchdata/160`.

Test problems should capture the typical job structure of jobs at the facility described in Chapter 1. Jobs typically start and end at one of the two combined entry/exit points, and must use the transporter to travel between workstations. Every second operation of a job is therefore usually a transporter operation. In order to assess the performance of the exact and heuristic optimisation algorithms developed in this study, 100 such test problems were randomly constructed, subject to a number of constraints designed to include some typical features of the jobs processed at the facility.

The test instances were designed for a facility with 9 workstations (machine IDs 1-9) and one transporter (machine ID 0). Workstations 1 and 9 both serve as entry and exit points. Each test problem consists of 20 regular production jobs, some with merges or splits. There are also 10 maintenance activities, one for each workstation and the transporter. The total number of operations in each test instance is 102. This is in the order of magnitude of the sets of work being scheduled at the facility. Time is measured in hours, assuming a 40 hour working week. Some jobs are released at time 0, and some jobs are released at time 40.

6.2 Experimental Design

Each heuristic search algorithm was given 30 independent execution runs, on each of the 100 test instances. Each execution run had a pre-defined calculation budget of 150,000 schedule evaluations. Some initial experimentation showed that this budget would be sufficient to obtain good quality solutions within an acceptable running time. Once this budget is spent, the best-so-far solution is returned as the final solution. The objective value of this best solution is used to compare the performance of the algorithms. The B&B algorithms are deterministic, and as such were executed only once on each test problem, with no limit placed on the number of schedule evaluations.

To highlight the general performance of the developed algorithms, the experimental results are aggregated across all problem instances. Statistics on their general performance are presented, in terms of objective values of the best performing solutions, execution time required to reach the reported result, statistics on the tardiness characteristics of the scheduled jobs, and statistical measures that highlight various interesting aspects of the behaviour of the proposed algorithms.

All algorithms considered in this chapter were implemented in C++ and compiled with the GNU g++ compiler under GNU/Linux running on a cluster system equipped with Intel Xeon E5-2650v2 CPU of 2.6 GHz and 4 GB of RAM per processor. All experimental results reported in this study

have been obtained under the same computing conditions.

6.3 Parameter Tuning

To facilitate fair comparisons across all search methodologies, and to avoid manually tuning the hyper-parameters of all algorithms, we employ a well known state-of-the-art off-line automatic hyper-parameter configuration framework, the Sequential Model-based Algorithm Configuration (SMAC, Hutter et al. (2011)). SMAC uses models based on collections of regression trees, known as random forests, to select and evaluate promising parameter configurations. Note that generally, the fine-tuned algorithms exhibit considerable performance gains compared to their original (un-configured) versions. In general, heuristic search algorithms with hyper-parameters often exhibit parameter-dependent performance gains. Inappropriate or thoughtless parameter configurations might lead to substantial deterioration of their performance. The first 10 problem instances were used to fine-tune the parameters for all heuristic search methodologies. As each test instance has a different optimal value, SMAC was used to find parameter values that minimise the average amount by which the optimal score was exceeded.

The five ACO algorithm parameters listed in Table 6.3.1 influence the performance of the algorithm. Hyperparameter optimisation was carried out by SMAC to obtain the recommended values in the table. These are also

Parameter	Recommended value	Range	Parameter description
α	0.8	$\mathbb{R}_{\geq 0}$	Pheromone parameter
β	1.1	$\mathbb{R}_{\geq 0}$	Heuristic parameter
ρ	0.035	$[0, 1]$	Evaporation parameter
w	4	$\mathbb{Z}_{>0}$	Rank parameter
γ	0.87	$[0, 1]$	Heuristic weight

Table 6.3.1: Recommended values for the ACO parameters.

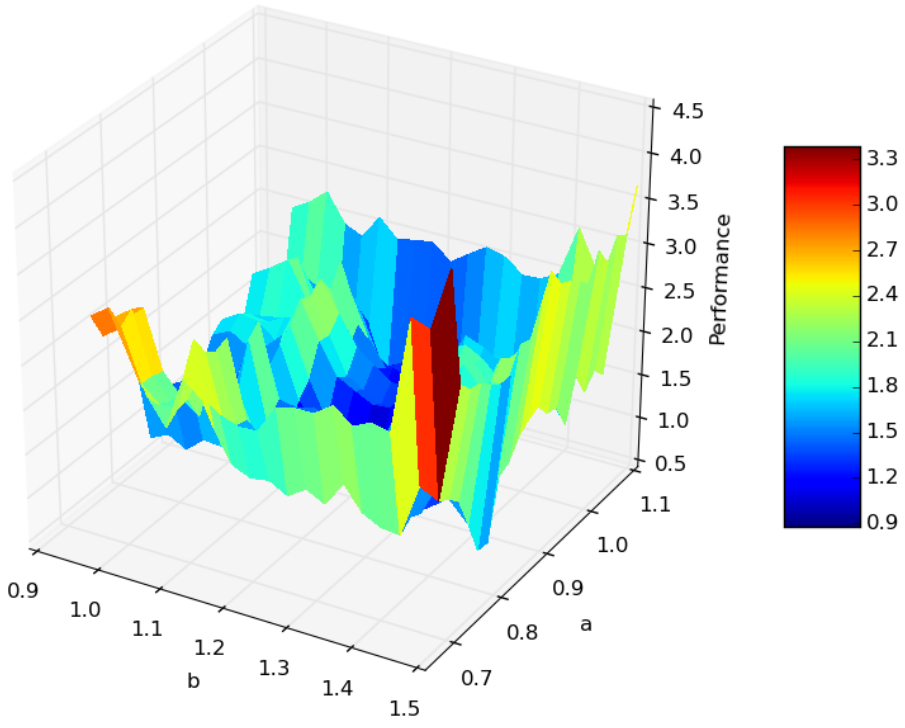

Figure 6.3.1: Estimated performance of the ACO algorithm, in terms of average exceedance of the optimal solution, across values of α (a) and β (b). Plot produced by FANOVA (Hutter et al., 2014).

the values used for our computational experiments. An illustration of the estimated interaction between parameters α and β is shown in Figure 6.3.1.

Regarding Simulated Annealing, SMAC was used to fine-tune the initial temperature, T_0, and the cooling parameter α. Generally good performance was observed for values of the cooling parameter α in the range (0.15, 0.65), and the initial temperature in the range (1, 10). For performance testing on all 100 test instances, the value $\alpha = 0.49$ was used, with an initial temperature of $T_0 = 5$.

6.4 Exact Methodologies: Branch & Bound and Hybrid

Two Branch and Bound algorithms have been developed to optimally solve the considered problem instances. These algorithms, BnB and BnB-UB, are described in Chapter 4. The main difference between them lies in the upper bound information that is provided to the algorithm. The first algorithm, BnB, assumes that no initial upper bound information is available. Therefore, the upper bound is set to infinity. The second approach, BnB-UB, is a hybrid between a heuristic and the BnB algorithm. In this hybrid approach, the considered heuristic is executed initially to acquire a good upper bound estimation. The BnB-UB algorithm then proceeds with the same algorithmic structure as the BnB algorithm. The heuristic method used here to find an initial good upper bound is the Ant Colony Optimisa-

tion algorithm with the balanced myopic and due date heuristic information (ACOmdb, Section 5.2). ACOmdb was selected as it is the best performing of the ACO algorithms (Section 6.5). Any other good heuristic method can be incorporated in such a hybrid method. Notice that this is common practice in order to speed-up and improve the convergence performance of Branch and Bound algorithms (Jourdan et al., 2009). The allowed execution time of both Branch and Bound algorithms has been restricted to a hard time limit of twenty days, due to resource limitations. If an algorithm exceeds this time limit it is stopped and recorded as an unsuccessful execution.

Descriptive statistics on the performance of both Branch and Bound algorithms across all problem instances considered in this work are presented in Table 6.4.1. The mean (μ), median (m), and standard deviation (σ) of the considered performance measures are provided, to capture the main characteristics of the performance value distributions. The best performing cases are highlighted with **boldface** font.

Table 6.4.1 reports for each algorithm the average (μ_s), median (m_s) and standard deviation (σ_s) values for the objective value of the best solution found by the algorithm. BnB-UB shows the best performance, with an average objective value of 32.03, compared to an average of 10^{13} for BnB. The high average score for BnB indicates that, on average, the solutions it finds within the time limit are of poor quality.

The average, median and standard deviation values for the execution

	BnB	BnB-UB
Mean objective value, μ_s	10.00E+13	**32.03**
Median objective value, m_s	36.72	**17.87**
St.dev. objective value, σ_s	30.15E+13	34.51
Mean execution time (seconds), μ_t	262947.322	**3082.692**
Median execution time (seconds), m_t	56167.377	**10.148**
St.dev. execution time (seconds), σ_t	430822.598	10319.805
On time (%), μ_o	82.17	**87.70**
After soft due date (%), μ_s	17.07	**12.30**
After hard due date (%), μ_h	0.77	**0.00**
Average distance, $\mu_{d_{\text{opt}}}$	10.00E+13	**0.00**
Percentage of success, *Success*	63	**100**

Table 6.4.1: Experimental results on the performance of the developed exact methodologies with (BnB-UB) and without (BnB) upper bound information, across all considered problem instances.

time ($\mu_t/m_t/\sigma_t$) required to find that best solution, in terms of CPU Wall-clock time measured in seconds, are also displayed in Table 6.4.1. Note that this is the total time until the best solution was first encountered, and not the total algorithm running time until completion or time-out. BnB-UB also performs best under this measure, as it required an average of 3083 seconds to find the best solution, compared to 262947 seconds for BnB. This shows that without a good initial upper bound, the algorithm running time is much longer.

In addition, to highlight the tardiness of the scheduled jobs, the average percentage of jobs that finish on time (μ_o), finish between soft and hard due dates (μ_s), and after the hard due dates (μ_h), are also shown in Table 6.4.1. BnB-UB again shows the best performance, with 87.7% of jobs starting on time, compared to 82.2% of jobs starting on time for BnB. This shows that BnB-UB is more successful at completing jobs by their soft due dates, compared to BnB.

Finally, Table 6.4.1 reports success percentages of the algorithms across all problem instances (*Success*), and the average distance ($\mu_{d_{\text{opt}}}$), in terms of the difference between the best solution found by the corresponding algorithm and the optimal solution of the problem at hand. BnB-UB finds the optimal solution in all cases, while BnB only achieves this 63% of the time.

It can be observed that the hybrid algorithm, BnB-UB, exhibits superior performance gains against BnB in terms of all performance measures

reported in the tables. It is worth noting that the incorporated heuristic, ACOmdb, provides very good upper bounds for the branch and bound algorithm, since it is able to find the global optimum solution in 87% of the times. In the remaining cases, it is close to the global optimum solution in terms of objective value. The provided upper bound values have average distance from the optimum $\mu_{d_{UB}} = 0.415$, median value $m_{d_{UB}} = 0$, and standard deviation $\sigma_{d_{UB}} = 1.666$, where d_{UB} is defined as the distance between the upper bound (UB) found by the heuristic and the global optimum (f_{opt}), $d_{UB} = UB - f_{opt}$.

BnB-UB exhibits the best average (μ_s) and median (m_s) performance in terms of solution quality across all problem instances, whilst BnB's performance deviates greatly from the global optimum solutions ($\mu_{d_{\text{opt}}}$ =10.00E+13). This high mean value is due to hard due date violations, which are penalised very severely (Section 3.3).

To assess whether quality performance differences between the two algorithms across all problem instances are statistically significant, we employ an exact Wilcoxon-Pratt signed-rank statistical test (Pratt's method was used to handle ties and the zero cases) (Hollander et al., 2013). A non-parametric statistical significance test is employed since the considered samples do not follow a normal distribution, as verified by the Shapiro-Wilk normality test (Royston, 1982). The null hypothesis of the Wilcoxon-Pratt signed rank test states that the compared samples are independent samples from identical distributions with equal median values, against the alterna-

tive hypothesis which states that one of the samples produces either lower, or higher median performance values. Here, we apply a 5% significance level, $\alpha = 0.05$. The statistical test strongly suggests that there are significant differences in performance between the two algorithms ($Z = -6.0318$, p-value =1.455E-11).

Regarding the execution time of the algorithms, BnB-UB is able to locate the global optimum solutions within less than an hour, on average ($\mu_t = 3082.692$ seconds), and in 50% of the time in about 10 seconds or less, as reported in Table 6.4.1. On the other hand, BnB takes on average 3 days to successfully locate the global optimum solution for a given problem instance, which is not adequate for the considered industrial facility.

The optimal solutions do not schedule jobs after the hard due dates, whilst 88% of the jobs are scheduled on time, and about 12% of the jobs are scheduled between the soft and the hard due dates. Notice that the schedules found by BnB for the non-optimal cases contain jobs whose completion time exceeds the hard due date limit.

Overall, the hybrid BnB-UB algorithm successfully solves all problem instances to optimality. The incorporated search methodology, ACOmdb, greatly enhances the proposed Branch and Bound method by providing upper bound values which are either optimal or close-to-optimal. This hybridisation achieves a great speed-up of BnB-UB's execution time. The combination of fast running time and good solution quality make this a viable method for application in demanding industrial settings.

6.5 Search methodologies: ACO & SA

This section analyses the performance of the proposed heuristic search methodologies, in terms of solution quality and speed. The four Ant Colony Optimisation (ACO) variants and the adapted Simulated Annealing (SA) algorithm are compared against each other. These methods are described in Section 5.2 and Section 5.3. The considered ACO variants differ mainly in the type of heuristic information that they employ in their pheromone updating rules. The following four ACO variants are compared:

- **ACOm**: ACO algorithm with myopic heuristic information, as described in Equation 5.2.5.
- **ACOd**: ACO algorithm with due date heuristic information, as described in Equation 5.2.6.
- **ACOmd**: ACO algorithm with a combination of the myopic and due date heuristics, as described in Equation 5.2.7.
- **ACOmdb**: ACO algorithm with a rebalanced combination of the myopic and due date heuristic information, as described in Equation 5.2.8.

The performance in terms of solution quality and its distance from the global optimum solution is investigated first. Note that all problem instances have been solved to optimality by BnB-UB (Section 6.4). Therefore the optimal solutions can be used to measure the quality of the solutions

located by the heuristic algorithms. The performance of the considered search algorithms is summarised by descriptive statistics of the objective values, as well as graphical illustrations of their performance.

Table 6.5.1 exhibits descriptive statistics on the overall performance of the algorithms over all problem instances, in terms of the objective value of the best solution attained by the corresponding algorithm. Specifically, for each algorithm and each problem instance, the objective value of the best solution found by the algorithm is measured. The results are then aggregated over all problem instances. The mean (μ_s), median (m_s), and standard deviation (σ_s) of the final objective values are reported. Two distance measures from the global optimum solution, d_{avg} and d_{best}, are also reported.

Let $f_i(x^\star)$ denote the objective value of the global optimum solution $x^\star$, for the i-th problem instance. Let μ_s^i be the average objective value of the best solutions obtained by an algorithm over its execution runs, for the i-th problem instance. Then, for a given problem instance i out of N, the overall average distance of the objective value from the global optimum solution is given by $d_{\text{avg}}^i = f_i(x^\star) - \mu_s^i$. The quantity d_{avg} then denotes the overall average distance of the objective value from the global optimum solution over all problem instances, so that $d_{\text{avg}} = \frac{1}{N}\sum_{i=1}^{N} d_{\text{avg}}^i$.

Similarly, d_{best} denotes the overall average distance between the best solution found by an algorithm and the objective value of the global optimum solution. Hence, $d_{\text{best}} = \frac{1}{N}\sum_{i=1}^{N} d_{\text{best}}^i = \frac{1}{N}\sum_{i=1}^{N} f_i(x^\star) - b_s^i$, where b_s^i

Algorithm	μ_s	m_s	σ_s	d_{avg}	d_{best}
ACOd	50.46E+8	404.15	90.05E+8	50.46E+8	13.00E+8
ACOm	35.37	23.62	35.78	3.34	0.99
ACOmd	34.33	21.64	35.32	2.30	0.50
ACOmdb	**33.99**	**20.83**	35.69	**1.96**	**0.39**
SA	10.66E+7	24.00	10.27E+8	10.66E+8	1.34

Table 6.5.1: Descriptive statistics on the performance of the developed search algorithms across all considered problem instances, in terms of mean (μ_s), median (m_s) and standard deviation (σ_s) of the objective values. d_{avg} denotes the average difference between the average solution found by the algorithm and the global optimum score. d_{best} denotes the average distance between the best solution found by the algorithm and the global optimum.

denotes the best solution obtained by an algorithm over all its execution runs for the i-th problem instance.

Intuitively, d_{avg} captures the average robustness of the algorithm on producing solutions close to the global optimum, while d_{best} denotes the ability of the algorithm to produce a very good solution. Lower values of d_{avg} and d_{best} indicate better performance. The best result obtained across all algorithms are highlighted in **boldface** font.

Figure 6.5.1 illustrates the distributions of the solution quality found by

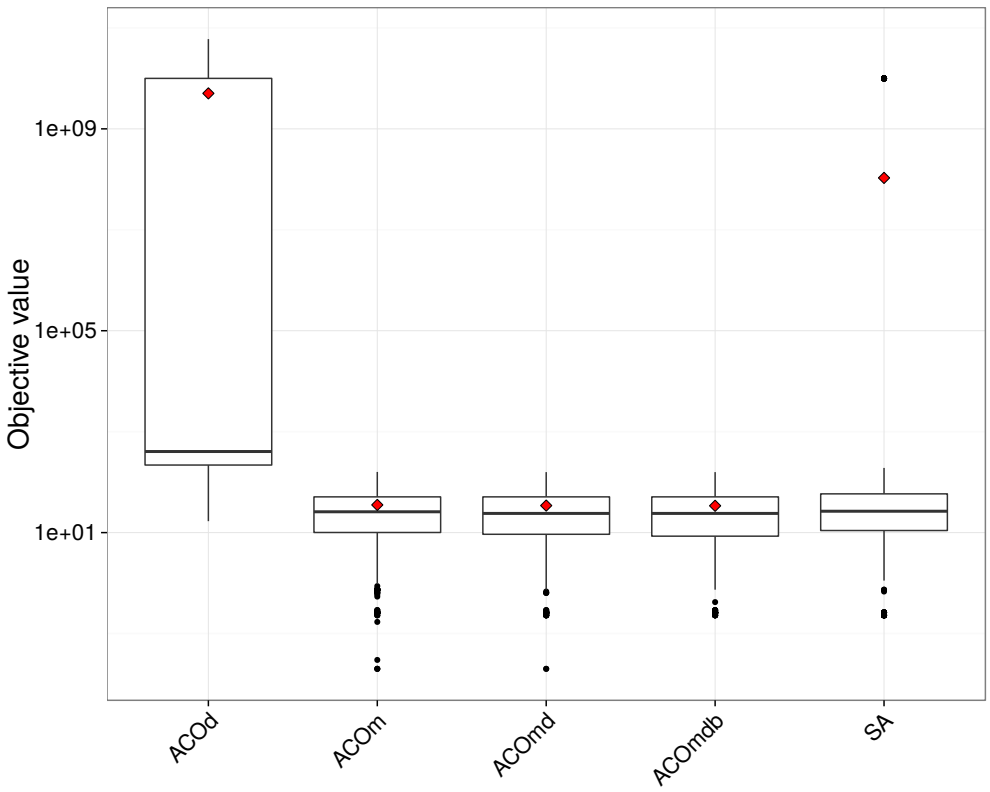

Figure 6.5.1: Box-plots of the objective values for all the heuristic algorithms over all the problem instances and runs (y-axis in $\log_{10}$ scale)

the considered algorithms. Note that in all figures that illustrate boxplots, we highlight the mean value of the observed measure with a red diamond point.

Both Table 6.5.1 and Figure 6.5.1 suggest that three out of four ACO variants (ACOm, ACOmd, and ACOmdb) seem to be the most promising search algorithms. ACOmdb shows the best performance across all considered measures. ACOmdb is able to locate solutions very close to the global optimum. Its average score is only 1.96 points above optimal. ACOm and ACOmd perform similarly and closely follow ACOmdb. However, ACOd

generally does not create good schedules. It is the worst performing algorithm in this chapter. Although SA performs similar to the ACO variants, in some cases it is not able to find a good quality solution. This has a major impact on its overall performance.

The d_{avg} and d_{best} metrics show that ACOmdb is the most promising algorithm. It converges on solutions which are, on average, very close to optimal. ACOm and ACOmd behave similarly. The schedules produced by SA are, on average, far away from the optimal solution. This is mostly due to a few instances in which SA performs very poorly. However, the best solution produced by SA for each problem is, on average, not much worse than the best solutions produced by ACOmdb, ACOmd and ACOm. Finally, ACOd generally creates poor quality solutions. It has the worst performance in this chapter. The distributions of d_{avg} and d_{best}, for each algorithm and across all problem instances, are demonstrated in Figure 6.5.2. The outliers observed in the boxplot for SA explain its poor performance in terms of mean solution score. The distribution shapes of these boxplots further indicate that ACOmdb has the best performance in this chapter.

The tardiness of jobs is considered next, in terms of the average percentages of the jobs scheduled on time (μ_o^t), between the soft and the hard due dates (μ_s^t), and after the hard due dates (μ_h^t). The relevant results are summarised in Table 6.5.2 and Figure 6.5.3. ACOd can be seen to be the worst performing algorithm under this tardiness measure. ACOmdb shows the best performance, closely followed by ACOmd, ACOm, and SA. Note

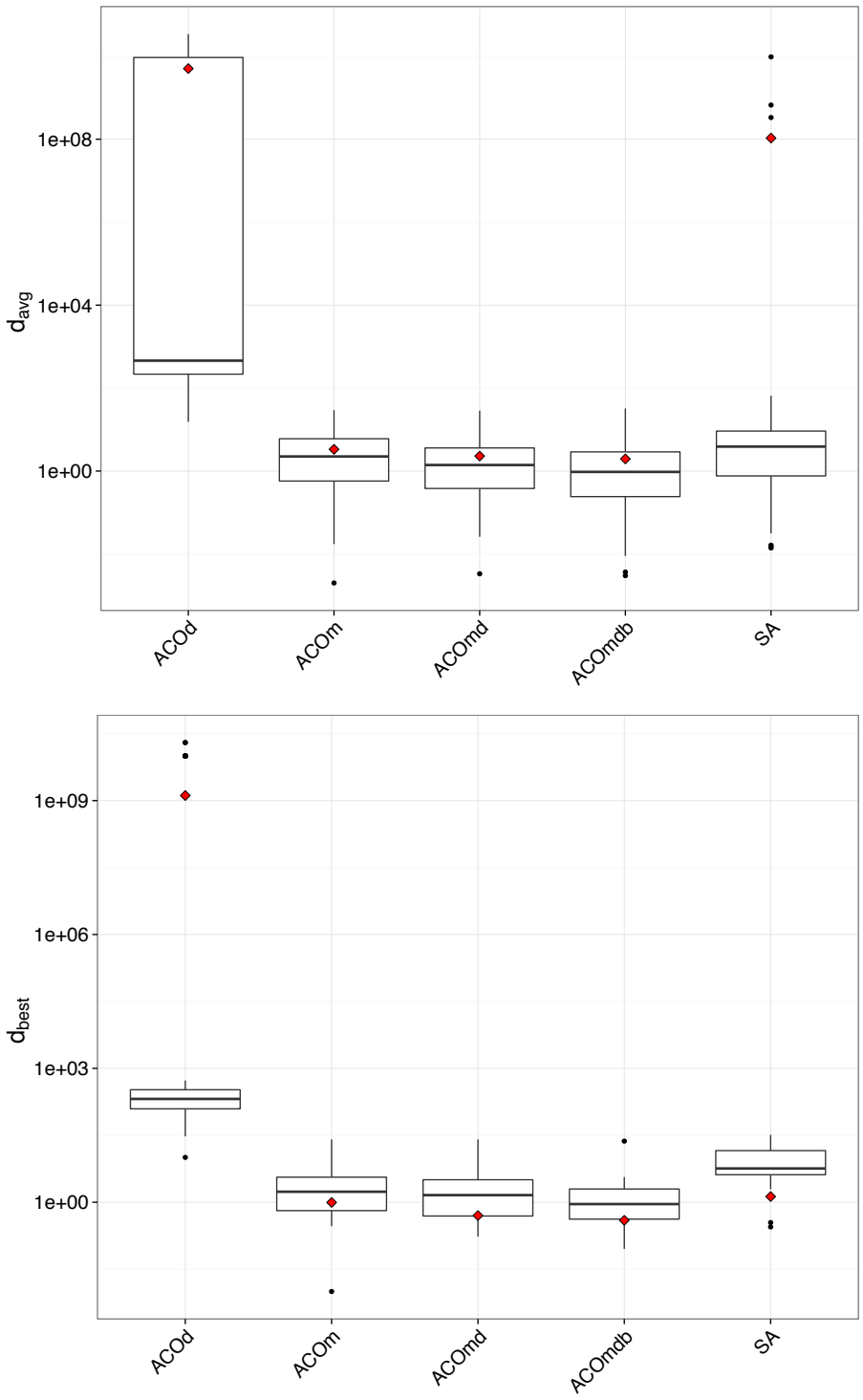

Figure 6.5.2: Box-plots of the d_{avg} and d_{best} measure values for all the heuristic algorithms over all the problem instances and runs (y-axis in $\log_{10}$ scale)

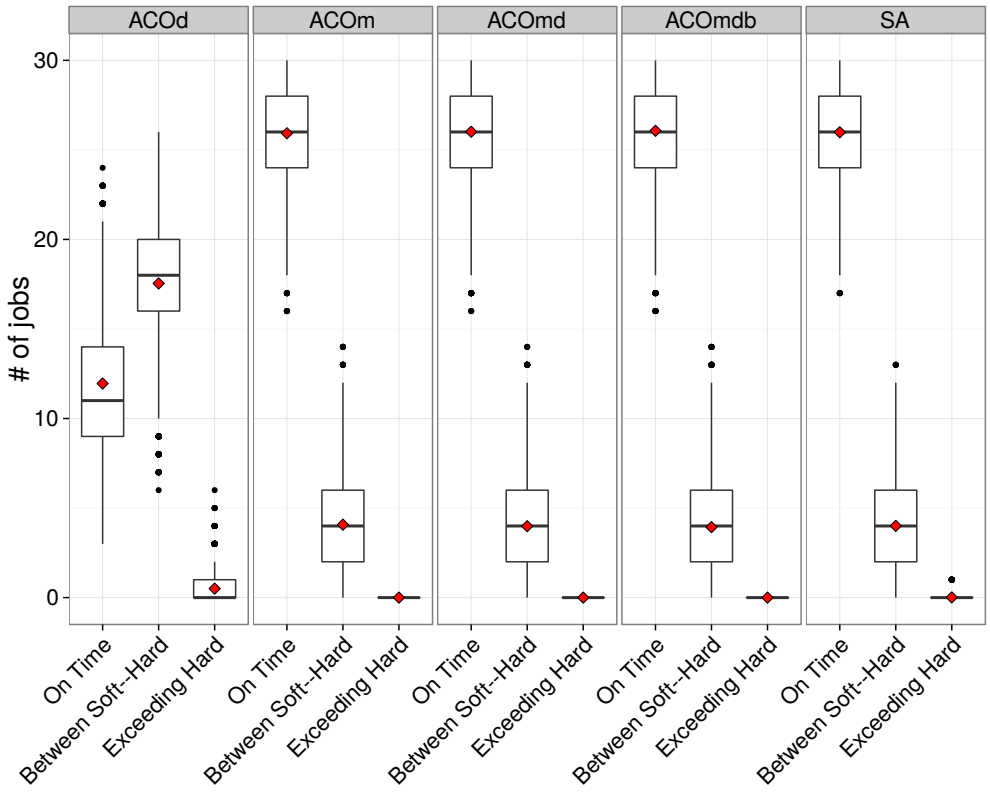

Figure 6.5.3: The distributions of the number of the jobs scheduled on time, between the soft and the hard due dates and after the hard due dates, across all heuristics.

Algorithm	μ_o^t	μ_s^t	μ_h^t
ACOd	39.85	58.47	1.68
ACOm	86.42	13.58	**0.00**
ACOmd	86.71	13.29	**0.00**
ACOmdb	**86.88**	**13.12**	**0.00**
SA	86.61	13.35	0.04

Table 6.5.2: Average percentages of the jobs scheduled on time (μ_o^t), between the soft and the hard due dates (μ_s^t), and after the hard due dates (μ_h^t).

that for SA, a small proportion of jobs ($\mu_h^t = 0.04\%$) exceed their hard due dates. This drastically reduces the performance of the SA algorithm.

Furthermore, to assess whether there exist statistically significant differences in the observed performance between at least two algorithms over all considered problem instances, the Friedman rank sum test (Hollander et al., 2013) was employed on the objective values obtained by the algorithms. The Friedman rank sum test strongly suggests that there exist significant differences in performance between the considered algorithms (p-value ≤ 0.00001, $\chi^2 = 7544.452$). In order to identify these differences, a pairwise post-hoc analysis was performed. For the post-hoc analysis, a pairwise Wilcoxon-signed rank test was employed on the objective value samples. The obtained p-values (p) are reported in Table 6.5.3. In addition,

	ACOd		ACOm		ACOmd		ACOmdb	
	p	p_{bonf}	p	p_{bonf}	p	p_{bonf}	p	p_{bonf}
ACOm	**0.0000**	**0.0000**	–	–	–	–	–	–
ACOmd	**0.0000**	**0.0000**	0.1480	1.0000	–	–	–	–
ACOmdb	**0.0000**	**0.0000**	**0.0149**	0.1495	0.3062	1.0000	–	–
SA	**0.0000**	**0.0000**	0.2239	1.0000	**0.0101**	0.1010	**0.0004**	**0.0041**

Table 6.5.3: Post-hoc statistical analysis for all pairwise combinations of the considered search algorithms across all the problem instances and executions (p: p-value from Wilcoxon-singed rank test, and p_{bonf}: p-value from the Bonferroni correction method)

to alleviate the problem of having Type I errors in multiple comparisons with a higher probability, the Bonferroni correction method was applied. The adjusted p-values, p_{bonf}, are also reported in Table 6.5.3. A 5% significance level ($\alpha = 0.05$) is used for all statistical tests. It can be observed that the performance differences between ACOmdb and the majority of the compared algorithms are significant. All algorithms differ significantly with ACOd. There is no significant evidence that ACOmdb performs differently than ACOmd, or that ACOmd performs differently than ACOm.

The overall success percentage for each heuristic is shown in Table 6.5.4. This is the percentage of times that the optimal solution was found across all test runs for a given problem instance. The mean ($\mu_{\text{Suc.}}$), median ($m_{\text{Suc.}}$), and standard deviation ($\sigma_{\text{Suc.}}$) of these percentages are reported. It can be

Algorithm	$\mu_{\text{Suc.}}$	$m_{\text{Suc.}}$	$\sigma_{\text{Suc.}}$
ACOd	0.00	0.00	0.00
ACOm	36.53	20.00	37.78
ACOmd	45.93	43.33	38.35
ACOmdb	**57.67**	**71.67**	38.83
SA	49.60	48.33	37.34

Table 6.5.4: The mean ($\mu_{\text{Suc.}}$), median ($m_{\text{Suc.}}$), and standard deviation ($\sigma_{\text{Suc.}}$) of the success percentages, for all the heuristic algorithms, over all problem instances.

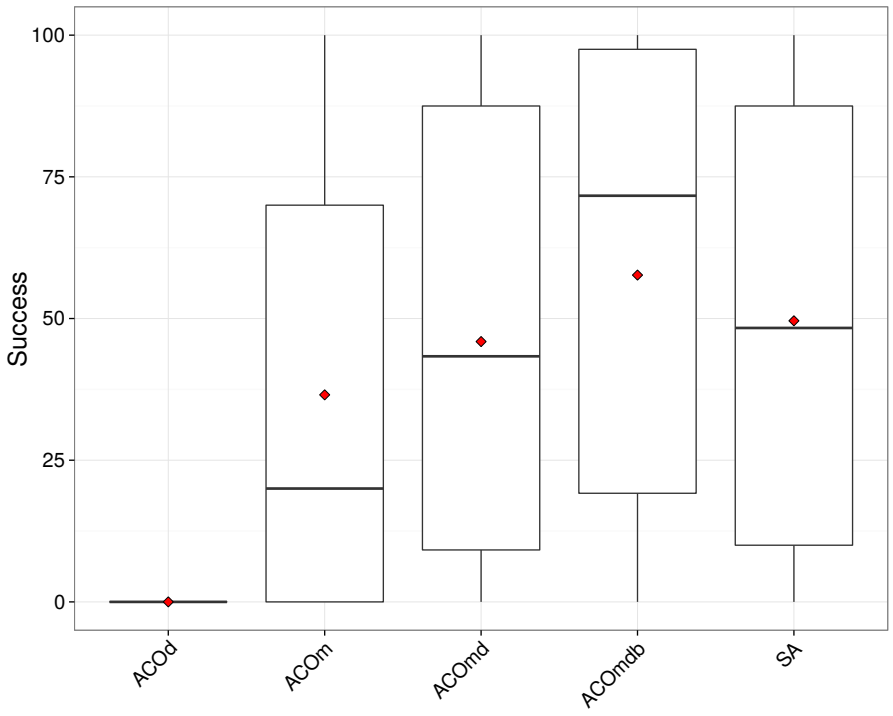

Figure 6.5.4: Box-plots of the distribution of the success percentages for all the heuristic algorithms over all the problem instances.

seen that ACOmdb performs the strongest, with a median success rate of 72%. This table also shows the very poor performance of ACOd, which never found any of the optimal solutions. Boxplots of the success rates appear in Figure 6.5.4. These suggest that SA and ACOmd perform similarly under this measure, with ACOm performing a little worse than both.

Finally, Figure 6.5.5 and Table 6.5.5 show the overall execution times required by the algorithms to reach the reported quality of solutions, across all problem instances. The average (μ_t), median (m_t), and standard deviation (σ_t) of the algorithms' execution times are reported, measured in seconds. Although the ACO variants are fast enough for producing high quality solutions in the industrial environment of interest, the execution time of SA is substantially smaller. This is not unexpected, since SA is a single point search algorithm, while ACO utilises a population of solutions. It is worth mentioning that SA might be a suitable heuristic for quickly seeding the B&B approaches developed in this work.

The experimental results reported in this section show that ACOmdb generally has the best performance of the tested heuristic methods. This is shown by several measures, including the average distance of its solutions from the global optimum, and the percentage of jobs scheduled before their respective soft due dates. Out of the ACO algorithms presented, ACOmdb also required the least time to find its solutions, on average. Simulated Annealing required even less computing time, but did not always perform quite as well in terms of the quality of the solutions found. Based on its

Algorithm	μ_t	m_t	σ_t
ACOd	51.64	46.89	22.37
ACOm	33.89	25.24	27.57
ACOmd	51.94	35.55	50.48
ACOmdb	29.36	19.45	30.53
SA	**2.20**	**1.76**	1.39

Table 6.5.5: Execution times in seconds required by the developed algorithms to reach the reported solution quality across all problem instances.

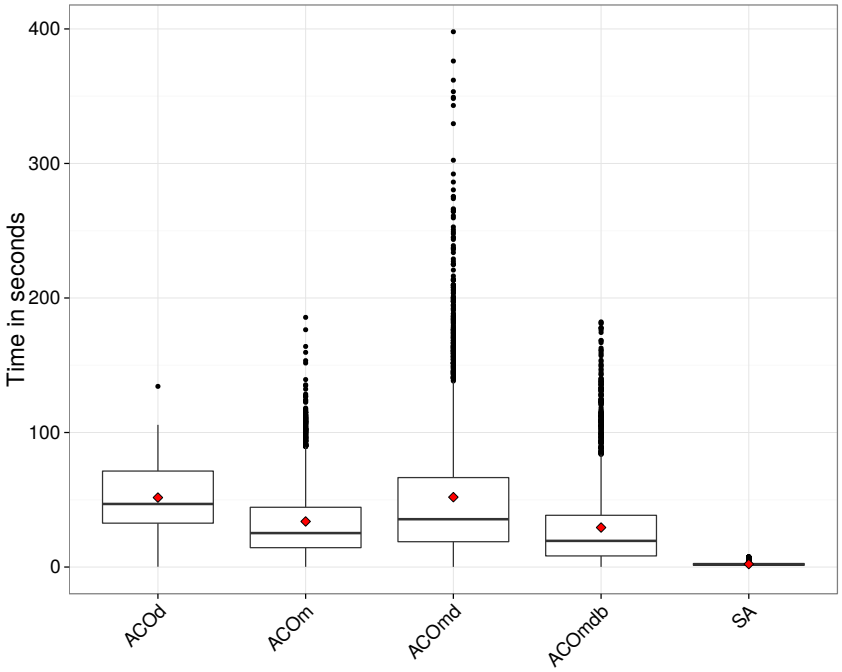

Figure 6.5.5: Overall execution times: box-plots of the execution time (in seconds) required by each algorithm to reach the reported solution quality over all the problem instances

superior solution quality and user friendly running time, ACOmdb is the preferred algorithm in the specific industrial context of this thesis.

6.6 Scalability Analysis

The test problems used for the computational experiments contained 102 operations. This is typical for the workload at the facility for which these algorithms were developed. For applications in other areas the workload and number of machines could be larger or smaller. The performance of the best algorithms was therefore also investigated with larger and smaller test problems. The test instances used in this scaling study contained the following number of operations and machines (machine count in brackets): 64 (6), 82 (8), 102 (10), 122 (12), 140 (14), 160 (16), 178 (18), and 204 (20). While there are countless different options for scaling the number of operations and machines, here it was decided to scale the number of machines roughly proportional to the number of operations. For each problem size, 100 different test instances were created. The other properties of the test instances were the same as before, so that jobs can only enter/exit the system at two locations, and must travel on the transporter between locations. Scaling was investigated for the two best performing algorithms, ACOmdb and B&B-UB.

6.6.1 Scaling: ACO

The best performing ACO algorithm, ACOmdb, solved each of the test problems 30 times. All parameters were fixed at the same values as during the computational experiments: $\alpha = 0.8, \beta = 1.1, \rho = 0.035, w = 4$, and $\gamma = 0.87$. The number of ants was set to 100. The stopping criterion was to terminate after 150,000 schedule evaluations. Figure 6.6.1 shows boxplots of the observed running times, both on the original scale (upper plot) and on a log scale (lower plot). The x-axis is to scale. It can be seen that the running times increase non-linearly with the number of operations.

6.6.2 Scaling: Branch and Bound

The best performing B&B Algorithm, BnB-UB, solved each of the 800 test problems (100 test instances for each of the 8 problem sizes). The initial upper bounds were taken as the best known upper bounds from the ACOmdb scaling experiments. A time limit of one day (86400 seconds) was set, at which point any unfinished runs were terminated. This means some of the observations have censored running times, since all that is known is that the algorithm would take more than 86400 seconds to find the optimal solution. For each of the test problem instances, the number of instances confirmed to have been solved to optimality within the time limit is displayed in Table 6.6.1. It can be seen that as the problem size increases, the success rate decreases.

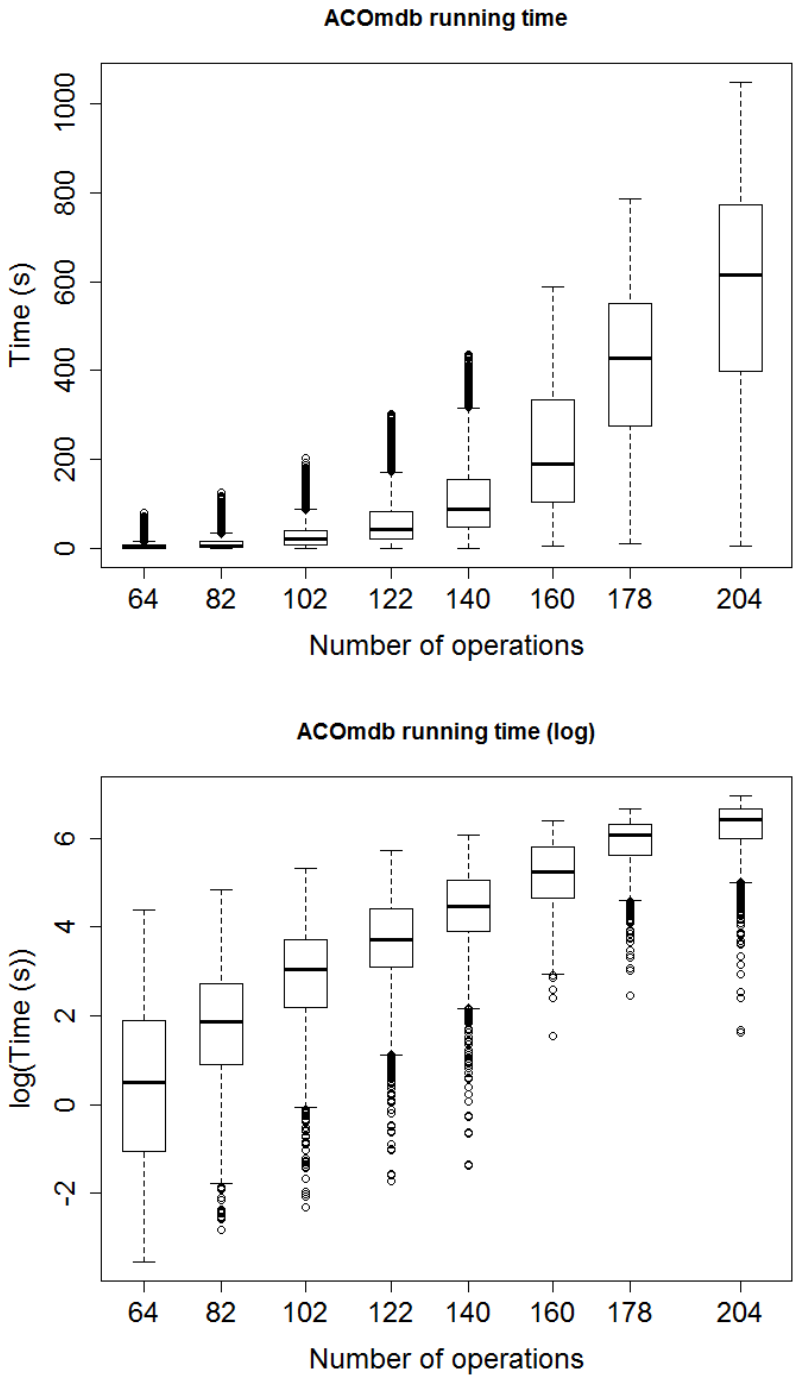

Figure 6.6.1: ACOmdb running times until the final solution was found, across varying problem sizes (x-axis to scale).

Problem Size (operations)	64	82	102	122	140	160	178	204
Solved to optimality (%)	100	99	99	77	67	44	21	2

Table 6.6.1: BnB-UB success rates across problem instance sizes. Success is measured as having found and confirmed an optimal solution within 1 day.

The algorithm running times are displayed in Figure 6.6.2, both in regular values (upper plot) and logged values (lower plot). The log-scale plot suggests that running times increase considerably with problem size. The medians for the three largest test problems were not observed, and as such it is difficult to see exactly how the running times behave beyond the first five problem sizes. A possible confounding factor for larger problem sizes, is that the initial upper bound possibly is not as tight as it is for some of the smaller test problems. This would be a contributing factor to the increase in running times with problem size. However, without knowledge of the optimal solutions, it is not possible to confirm this.

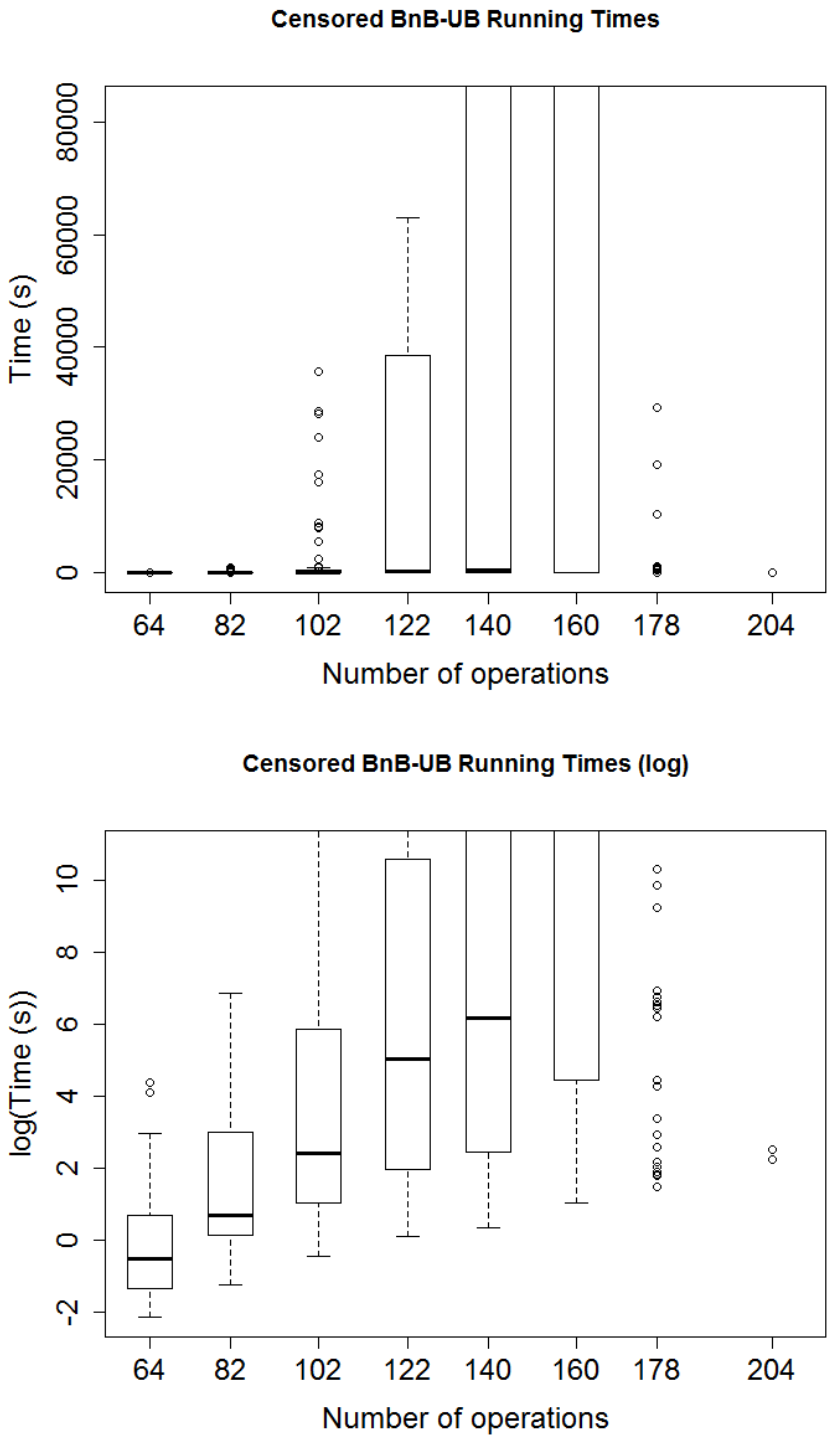

Figure 6.6.2: BnB-UB running times until the optimal solution was confirmed, across varying problem sizes (x-axis to scale). Some data is censored, since runs were terminated after one day (86400 seconds).

Chapter 7

Non-Blocking Job Shop: Conclusions

The work in Part II was motivated by the real life scheduling challenges encountered at a facility in the nuclear power industry, as introduced in Chapter 1. The work at the facility was modelled as a job shop scheduling problem with flexible maintenance, due dates, release dates, and precedence constraints (Chapter 3). Both exact and heuristic solution methods for this problem were developed and presented, including Branch and Bound (Chapter 4), Ant Colony Optimisation (Chapter 5), and Simulated Annealing (Chapter 5). A hybrid method incorporating ACO and B&B was also proposed.

Thorough experimental results, comparisons and analysis on 100 problem instances provide strong evidence that the proposed algorithms produce optimal or near-optimal solutions, within a short running time (Chapter 6).

Specifically, experimental results showed that the ACO algorithm on its own finds good solutions, often optimal, in short computational time. The cooperation between the developed Branch and Bound algorithm and the Ant Colony Optimisation algorithm resulted in a very effective algorithm, which optimally solves the considered problem, within acceptable running time for the industrial problem under consideration. The hybridisation successfully locates optimal solutions on all problem instances. The upper bound information seeded by the ACO algorithm substantially decreases its execution time, making it a valuable and suitable solution to optimally solve challenging real-world scenarios. A scalability analysis showed that running times increase with problem size.

Due to limited storage capacity at the facility of interest, it was desirable that the methods presented in Part II were further developed to be applicable in a blocking job shop. In the blocking job shop, there is no storage capacity at the machines. When an operation is completed by a machine, the materials cannot be moved to the next required machine until the required machine is no longer occupied by any other jobs. Solution methods for the blocking job shop scheduling problem with flexible maintenance and both soft and hard due dates are developed and presented in the next part of this thesis (Part III).

Part III

The Blocking Job Shop with Flexible Maintenance

The chapters in this part of the thesis consider the blocking job shop with flexible maintenance. The big difference with the problem dealt with in Part II is that there is zero buffer capacity in the system. This means jobs have to remain on their current machine after processing, until such a time that the next required machine is available. While the job waits to be moved, the machine that it is occupying cannot be used by other jobs. The problem is described and formally defined in Chapter 8. Heuristic solution methods for this problem are presented in Chapter 9. While heuristic methods worked well for the non-blocking version of the problem discussed in Part II, the heuristics presented in this part generally struggle to produce good solutions. In contrast to this, the Branch and Bound algorithm for the block shop presented in Chapter 10 very quickly produces optimal solutions. The computational experiments in Chapter 11 show that the Branch and Bound method outperforms the heuristic methods, both in terms of running time and the quality of the final solutions.

Chapter 8

Introduction to the Blocking Job Shop with Flexible Maintenance

8.1 Blocking Job Shop

The job shop scheduling algorithms presented in Part II were tested at the NNL facility for which they were developed. Many of their modelling aspects, such as the flexible maintenance windows, and the soft and hard due dates, were found to model the real-world setting very well. However, at this point it also became clear that the algorithms were producing schedules which, if implemented, would lead to storage problems. In some cases this was due to space limitations. In some other cases, the materials belonging to different jobs could not be stored in the same area due to other

considerations, such as differing levels of radioactivity.

Under the classic job shop formulation, it is implicitly assumed that there is ample storage capacity between machines. More specifically, it is assumed that each machine has a queue with infinite capacity. Consider the situation where an operation is finished, and the next machine required by the job is not yet available. Under the assumption of infinite queueing capacity, the materials of the job can immediately be stored in the queue of the next required machine. The machine on which the job was just processed then immediately becomes available again to other jobs. In practice, queueing capacity will always have some limit. However, as long as this limit is unlikely to be reached under normal operating conditions, the assumption of infinite queueing capacity may be quite reasonable for modelling purposes.

After further consultation with the planners at the NNL facility, it was decided that their scheduling problem should be modelled under the assumption that there is no storage capacity inside the facility. This changes the scheduling problem to a variant of the blocking job shop scheduling problem. Machines have no queues within a blocking job shop. To illustrate how this scenario differs from the classic job shop formulation, let us consider an operation which has just finished processing on machine M_1. It is not the final operation of the job, and as such the materials require further processing on another machine, M_2, say. Without queueing capacity at M_2, there are two possible outcomes:

1. If machine M_2 is available, the materials are moved there immediately. The next operation of the job can start, and machine M_1 is now available again to other jobs.

2. If machine M_2 is busy, the materials have nowhere to go and must stay on machine M_1, despite not requiring further processing there. The materials effectively block machine M_1 for use by other jobs, until such a time at which machine M_2 becomes available.

At first glance, it may appear that the blocking effect will simply introduce some delays to the schedules. However, the problem is much more complicated than that. In fact, many schedules produced under the classic job shop assumption of infinite queues will be entirely infeasible in the blocking case, even if the starting times of operations are adjusted to account for delays introduced by machine blocking. The infeasibility encountered is a result of what is known as deadlock (Mascis and Pacciarelli, 2002). Deadlock occurs when two or more jobs block each other in a circular way, such that each of these jobs requires a machine currently occupied by one of the other jobs in the deadlock circle. In the simplest case, such a deadlock circle consists of two jobs blocking each other. Consider, for example, one job on machine M_1, and another job on machine M_2. Each job has finished processing on its current machine. The job on machine M_1 needs to go to machine M_2, and the job on machine M_2 needs to go to machine M_1. In this scenario, neither job can move. More generally,

this situation can occur for any number n of jobs and machines. Deadlock occurs when the job on machine M_1 is waiting for machine M_2, the job on machine M_2 is waiting for machine M_3, ..., the job on machine M_k is waiting for machine M_{k+1}, ..., and finally, the job on machine M_n is waiting for machine M_1. To resolve deadlock, some authors allow for swaps. A swap is when all the jobs in the deadlock circle move simultaneously (Mascis and Pacciarelli, 2002). Due to the single transportation mechanism at the NNL facility, simultaneous movement of jobs is not an option here. Swaps will not be allowed in our problem formulation, and deadlock will have to be avoided at the scheduling stage.

8.2 Problem Overview

The management of the work passing through the NNL facility described in Section 1.2, can be modelled as a blocking job shop scheduling problem (BJSSP), with a number of additional non-standard features based on the requirements of the facility.

To briefly recap, the facility comprises shielded workstations, that process sensitive and radioactive materials, handled remotely by expert workers. The facility also contains a single transportation mechanism that connects all workstations, and which transfers the materials across all workstations. There are two workstations that both act as entry and exit points for any materials. As such, given a material to be processed, it will have to

enter the system at one of these two workstations, and then be moved to the appropriate workstation by the transporter. The materials may then have to undergo further processing at other workstations, and eventually leave the system through one of the two exit points.

The non-standard features faced by schedulers at the facility include: flexible maintenance scheduling; jobs which merge into one, or split into many; jobs with release dates; jobs with precedence constraints; and an absence of storage space within the system. Existing research might not take into consideration all or most of such non-standard features that are encountered in real-world scenarios. It is more common to consider and study a single complication in isolation.

The major difference between the problem presented here and the problem dealt with in Part II, is that there is zero storage capacity in the system. This means jobs have to remain on their current machine after processing, until such a time that the next required machine is available. While the job waits to be moved, the machine occupied by the job cannot be used by other jobs.

A crucial aspect of the considered BJSSP is the maintenance programme on all machines at the facility. In industrial scenarios, machine maintenance is critical for the good functioning of the facility. However, machine maintenance is often ignored in traditional job shop scheduling formulations. Each workstation at the facility, as well as the transporter, has a preventive maintenance program. These flexible maintenance activities are

required to start within a specified time window. The start and end of this time window are the soft and hard due dates, respectively, for the start of the maintenance activity. Due to strict safety regulations in the nuclear industry, a machine must be shut down if its maintenance is not started by the hard due date. In this work we integrate the planning of maintenance with the planning of jobs. Each maintenance event is modelled as a single-operation job.

Another feature motivated by the industrial setting is that some jobs can only start when multiple other jobs come together on completion. There are also jobs which, on completion, allow a number of other jobs to start. This merging and splitting behaviour is modelled with our novel adaptation of the alternative graph, introduced in Section 8.7.2.

Novel features of the problem under consideration include the blocking aspect, flexible maintenance, soft and hard due dates, and the integrated planning of maintenance alongside the jobs planning procedure.

8.3 Problem Notation

More formally, we wish to schedule a set of $n \geq 1$ jobs $\mathcal{J} = \{J_i\}_{1 \leq i \leq n}$ on a set of $m \geq 1$ machines $\mathcal{M} = \{M_k\}_{1 \leq k \leq m}$. Each job $J_i \in \mathcal{J}$ consists of a number of o_i operations $J_i = \{O_{ij}\}_{1 \leq j \leq o_i}$ to be processed in a given order. Each operation O_{ij} has to be carried out on a specified machine $M_k \in \mathcal{M}$. Each operation has a positive deterministic processing time $p_{ij} \in \mathbb{R}^+$. Each

machine M_k can process only one operation at a time. The nature of the operations makes them non-preemptive, i.e. once an operation has started it cannot be interrupted. Each job J_i has a release date r_i. Jobs can have precedence constraints, which enforce a strict ordering between two jobs. For example, if J_i must be processed before J_j, the precedence constraint is $c_i \leq s_j$, where c_i denotes the completion time of J_i and s_j denotes the starting time of J_j.

The merging of jobs is modelled with the help of precedence constraints. Consider a set of three jobs, $\{J_i, J_j, J_k\}$ say, which merge into a fourth job, J_l. This would be modelled with the constraint $\max\{c_i, c_j, c_k\} \leq s_l$. This type of constraint is known as an in-tree (Cheng and Sin, 1990). In the blocking environment, the machine on which the merge takes place must be reserved for said merge as soon as the first of the various materials arrives. This can be modelled with our novel adaptation of the alternative graph, as presented in Section 8.7.2. The splitting of jobs can be modelled with precedence constraints alone. If job J_i splits into three other jobs, $\{J_j, J_k, J_k\}$ say, this is modelled with the constraint $c_i \leq \min\{s_j, s_k, s_l\}$. This type of constraint is known as an out-tree (Cheng and Sin, 1990).

As defined above, let c_i denote the scheduled completion time of job J_i. Each job has a soft due date d_i, and a hard due date $\overline{d}_i$. A tardiness penalty T_i is incurred if the soft due date is violated, i.e. when $d_i < c_i$. The size of the penalty increases with the delay, and rises rapidly beyond the hard due date. The motivation behind this is that, in reality, the financial reward

for a job is reduced if the soft due date is missed, and severely reduced if the hard due date is missed. In cases where all due dates can be met, a small penalty proportional to the completion time of each job ensures work is not needlessly delayed. Completing work sooner helps maintain a good reputation with clients.

Each maintenance activity is modelled as a single-operation job. A flexible maintenance activity J_i receives a tardiness penalty based on its scheduled starting time, s_i. These flexible maintenance activities are required to start within the time window defined by their soft and hard due dates. More formally, for maintenance activities there is a hard constraint which requires that $s_i \in [d_i, \overline{d_i}]$. Maintenance can start no earlier than its soft due date d_i. If maintenance has not started by its hard due date $\overline{d}_i$, the corresponding machine must be shut down. Therefore, a missed hard due date for maintenance has to be penalised much more heavily than a missed hard due date for jobs.

As described previously, the objective function considered in this study seeks to minimise the total tardiness penalty T, defined by $T = \sum_{i=1}^{n} T_i$. Depending on the type of job (normal, maintenance), we employ two tardiness penalty functions with different characteristics, as described next in Section 8.4.

8.4 Objective Function

Given that a regular job J_i has soft due date d_i, hard due date $\overline{d}_i$ ($d_i \leq \overline{d}_i$) and, once scheduled, completion time c_i, then job J_i incurs the following tardiness penalty T_i:

$$T_i = \begin{cases} 0.01 \times c_i & \text{if } c_i \leq d_i, \\ 0.01 \times c_i + 10^3 + c_i - d_i & \text{if } d_i < c_i \leq \overline{d}_i, \\ 0.01 \times c_i + 10^3 + (\overline{d}_i - d_i) + 10^6 + 10 \times (c_i - \overline{d}_i) & \text{if } \overline{d}_i < c_i. \end{cases} \tag{8.4.1}$$

Each job gets a small penalty equal to 0.01 times its finishing time. This ensures that even if a job finishes on time, its completion time is minimised further, as long as this doesn't delay other jobs. The small linear penalty also helps constructive algorithms make better decisions. This is especially true in the early stages of schedule construction, when no due dates are violated, and the partial solution score would otherwise be zero. There is a fixed penalty of 10^3 for any job that exceeds its soft due date d_i. Jobs that exceed their soft due date also incur a linear penalty equal to the delay, $c_i - d_i$, until the completion time reaches the hard due date ($d_i < c_i \leq \overline{d}_i$). In the case where the completion time exceeds the hard due date ($\overline{d}_i < c_i$), a very large fixed penalty is incurred ($\cdots + 10^6 + \ldots$), while there is a slope to help the optimisation process to move towards more desirable solutions ($\ldots + 10 \times (c_i - \overline{d}_i)$).

Maintenance activities don't incur a large fixed penalty when they exceed their soft due date. This allows their flexible maintenance window to be used if needed. Given a flexible maintenance activity J_i, with starting time s_i, the following tardiness penalty is incurred:

$$T_i = \begin{cases} 0.01 \times s_i & \text{if } s_i = d_i, \\ 0.01 \times s_i + s_i - d_i & \text{if } d_i < s_i \leq \overline{d}_i, \\ 0.01 \times s_i + \overline{d}_i - d_i + 10^9 + 100 \times \left(s_i - \overline{d}_i\right) & \text{if } \overline{d}_i < s_i. \end{cases} \tag{8.4.2}$$

In the maintenance tardiness penalty function, a substantially larger fixed penalty value (10^9) is incurred in the case where the starting time of the maintenance exceeds the hard due date ($\overline{d}_i < s_i$). With a gradient of 100, the slope beyond the hard due date is also much steeper for maintenance.

Figure 8.4.1 illustrates how the job and maintenance components of the penalty function behave relative to each other. Due to the very large fixed penalties at the hard due date, the Y-axis is not to scale.

These tardiness functions are designed to prioritise the various scheduling objectives. The highest priority is placed on scheduling maintenance tasks to start before their respective hard due dates, to avoid a mandatory shut-down. The second highest priority is to schedule regular jobs to finish before their respective hard due dates, to avoid substantial loss of

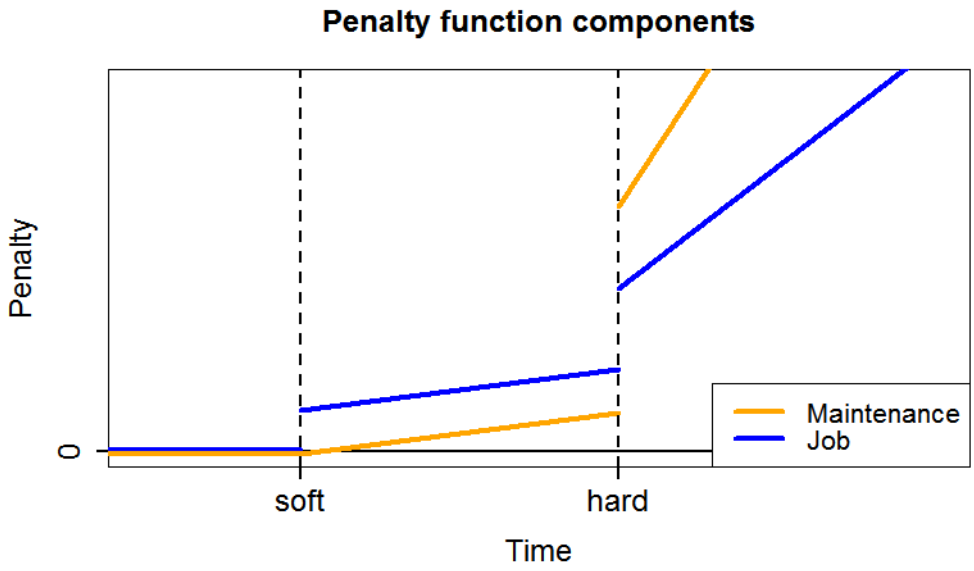

Figure 8.4.1: An illustrative comparison of how the soft and hard due dates influence the job and maintenance components of the penalty function. Y-axis not to scale.

revenue. Next, regular jobs should be completed by their soft due date, if at all possible. The final priority is then to reduce any remaining delays in maintenance tasks. Should it be possible to meet all soft due dates, then the small penalty of 0.01 times the completion time minimises the mean completion time.

8.5 Problem Classification

The overall objective function (Section 8.4) takes into account job and maintenance tardiness, as well as weighted unit penalties for due date violations. Let w_{tj} denote a weight, where t indicates a type of tardiness and

where j indicates a job. Then, in terms of the widely used 3-field problem classification $\alpha|\beta|\gamma$ (Graham et al. (1979); Section 2.1.1), the problem considered here can be represented by $J|prec, r_j, block|\Sigma_j w_{cj}C_j + \Sigma_j w_{tj}T_j + \Sigma_j w_{\overline{t}j}\overline{T}_j + \Sigma_j w_{uj}U_j + \Sigma_j w_{\overline{u}j}\overline{U}_j$. The third field contains the objective, and can be broken down as follows:

- $\Sigma_j w_{cj}C_j$ is the weighted completion time C_j, or starting time in the case of maintenance.
- Tardiness with respect to the soft due date, T_j, is penalised by $\Sigma_j w_{tj}T_j$.
- Tardiness with respect to the hard due date, $\overline{T}_j$, is penalised by $\Sigma_j w_{\overline{t}j}\overline{T}_j$.
- A weighted unit penalty $\Sigma_j w_{uj}U_j$ is incurred when soft due dates are missed. For maintenance, this weight is zero.
- A weighted unit penalty $\Sigma_j w_{\overline{u}j}\overline{U}_j$ is incurred when hard due dates are missed. Our implementation uses different weights for jobs than for maintenance.

8.6 Complexity of the Blocking Job Shop

The problem defined in Section 8.5 contains as a special case $1|r_j|\Sigma C_j$. This is the case when there is just one machine, the jobs have no precedence constraints, due dates are set to infinity, and $w_{cj} = 1$. As there is only a single machine, blocking cannot occur. It has been shown that $1|r_j|\Sigma C_j$ is

unary NP-hard (Graham et al. (1979)). Therefore the scheduling problem considered in this part of the thesis is also NP-hard in the strong sense.

More generally, the standard job shop problem is a special case of the blocking job shop problem. Therefore the blocking job shop is at least as complex as the standard job shop problem. To see this, consider a blocking job shop problem in which each job only has a single operation. Precedence constraints are present between some jobs, so that chains of single operation jobs are formed. Since there are no blocking constraints from one job to the next, one such chain of single operation jobs behaves exactly the same as a sequence of operations in a single job in the standard job shop scheduling problem. Thus, the standard job shop scheduling problem is a special case of the blocking job shop problem, making the latter as least as complex as the former.

8.7 Alternative Graph

To model the blocking job shop problem we use an adaptation of the alternative graph (Mascis and Pacciarelli, 2000), which itself is an adaptation of the well-known disjunctive graph (Section 4.2; Roy and Sussmann (1964)). The alternative graph can be used to model a blocking job shop problem with linear jobs. In this representation, a complete blocking job shop schedule can be represented by a directed acyclic graph (DAG).

8.7.1 Traditional Alternative Graph

The disjunctive graph was introduced in Section 4.2, to model the non-blocking job shop problem. There, the disjunctive arcs defined an ordering over all operations that have to be processed on each machine. In the alternative graph, pairs of alternative arcs are used to define this ordering. Both arcs within a pair belong to the same machine. Each pair of alternative arcs determines the order in which a given machine serves two specified jobs. To model the blocking aspect, alternative arcs originate not directly from the operation they are associated with, but from the next operation in the same job. This ensures that the corresponding machine is not released until the next operation has started, i.e. until the completed operation no longer blocks the machine. Alternative arcs usually have length zero. The exception to this is when they are associated with the final operation of a job. In that case, the alternative arc has its origin at this final operation, and its length is equal to the processing time of that operation. Alternative arcs point directly to the node in the other job that requires the relevant machine. A feasible schedule must contain exactly one arc from each pair, in such a way that no cycles exist in the completed graph. The starting time of each operation is then determined by the longest path from the source to the corresponding node in the graph.

An illustrating example appears in Figure 8.7.1, which shows a single pair of alternative arcs for two operations that both require machine

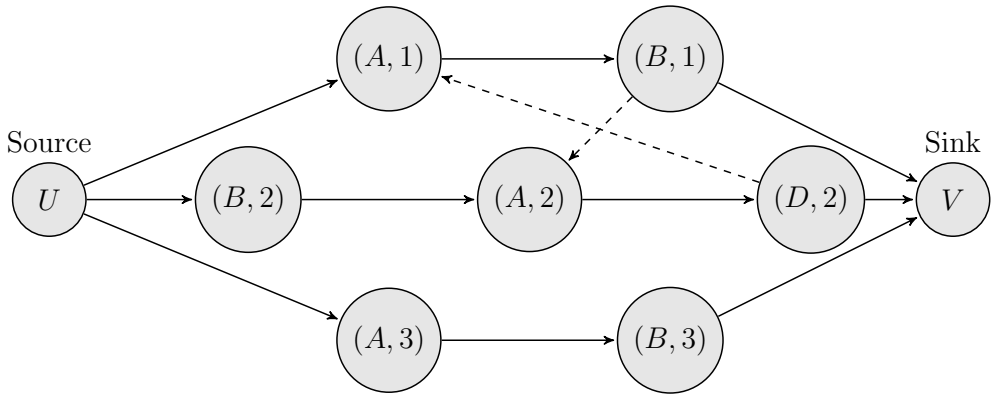

Figure 8.7.1: One pair of alternative arcs (dashed) associated with operations $(A, 1)$ and $(A, 2)$. Figure modified from Pinedo (2008).

A. This particular pair of alternative arcs determines whether machine A serves operation $(A, 1)$ before it serves operation $(A, 2)$, or vice versa. If, for example, operation $(A, 1)$ is served first, the alternative arc with its origin at $(B, 1)$ is selected. This ensures that operation $(A, 2)$ can only commence once both of the following have happened: Job 1 has completed processing on machine A, and job 1 has started processing on machine B.

Our particular problem contains one further complication that is not captured by the alternative graph, namely the merging and splitting of jobs. The following sections introduce our adaptation of the alternative graph, which facilitates these aspects.

8.7.2 Alternative Graph Adaptation: Merging with Blocking

A merge takes place when two or more jobs which previously existed independently, come together and continue as a single job. Consider, for example, some radioactive materials that eventually need to be stored in a flask. The materials will undergo processing on one machine, while the flask is prepared on another machine. Once these separate jobs have been completed, the material is put into the flask at a third location. In the non-blocking case this was simply modelled with precedence constraints, so that the flask filling job could not start until the two pre-merge jobs had been completed. In the blocking case, however, the merging location must be considered reserved, or pre-blocked, as soon as either the material or the flask arrives there. The actual flask filling job will not commence until both have arrived. In addition to this pre-blocking, it must also be ensured that the machine of the first pre-merge job to finish is released for other work as soon as the material is moved to the merging location. We have developed a novel adaptation of the alternative graph to achieve this, as described next.

The following example illustrates the merging of two jobs in our blocking environment. Consider two jobs, job 1 and job 2, which merge into a third job, job 3. The grey nodes in Figure 8.7.2 represent the operations of the three jobs directly involved in this merge. The white nodes represent

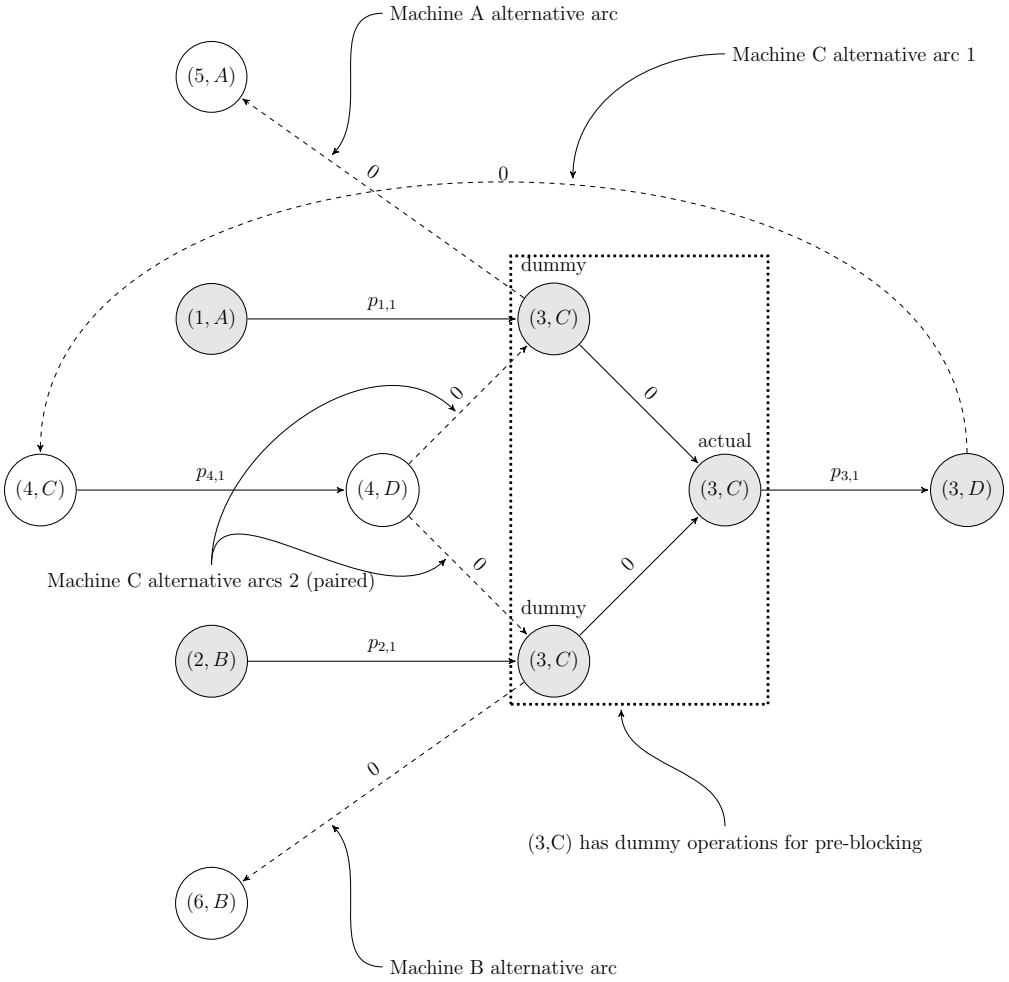

Figure 8.7.2: Operation $(1, A)$ belongs to job 1 and requires machine A. The graph depicts job arcs (solid lines) and alternative arcs (dashed lines). The two dummy operations for $(3, C)$ ensure pre-blocking of machine C takes places as soon as either job 1 or job 2 arrives at machine C. They also allow machines A and B to be released as soon as the materials on them have been moved to machine C.

operations that belong to jobs not involved in the merge. Each node displays a number, indicating the job it belongs to, and a letter corresponding to the machine required by the operation. Fixed job arcs are depicted as solid straight arrows, and the dashed arcs represent alternative arcs. Jobs 1 and 2 merge into job 3. The merge takes place on machine C. Job 4 is another job which requires machine C, so the choice of alternative arcs will determine whether machine C processes job 3 before job 4, or vice versa. More precisely, the choice of alternative arcs for machine C is between the single arc with its origin at $(3, D)$, or the linked pair of arcs originating at $(4, D)$. This pair of linked arcs leads to two dummy nodes with zero processing time. The starting time of each dummy operation represents the moment at which the material of the corresponding preceding machine was moved to machine C. This means that as soon as either job 1 or job 2 completes and sends its materials to machine C, machine C is reserved exclusively for use by operation $(3, C)$. This operation will start processing as soon as the materials from both preceding jobs have arrived. While this example described the merging of two jobs, this method can be used in the same way to model the merging of any number of jobs.

Note that in the case where one job splits into two or more jobs, regular alternative arcs will ensure the original machine remains blocked until the last materials have been moved from there. In this case the machine required for the final operation of the job will have one alternative arc of length zero from the first node of each dependent job.

In the blocking job shop, some consideration must be given as to how to record completion times when jobs merge. When jobs do not merge, the machine of the final operation of a job becomes available again as soon as that final operation is completed. However, when jobs merge, the materials of pre-merge jobs may continue to block the machine of the final pre-merge operation, until they can be moved to the machine on which the jobs merge. Therefore, each of the jobs to be merged will be scored on the starting time of the dummy operation linking it to the dependent job. This is because that is the time at which the machine used by the final pre-merge operation becomes available again to other jobs.

8.7.3 Alternative Graph Adaptation: Arcs

In the traditional alternative graph (Section 8.7.1), alternative arcs are mutually exclusive pairs of machine arcs. In our novel adaptation of the alternative graph, some pairs of alternative arcs now contain two or more arcs on one side of the alternative. More specifically, in our adaptation, the origin, destination, length, and number of alternative arcs are determined as follows:

- All operations O_{ij} of job J_i that are not the final operation of J_i, spawn alternative arcs with origins at the subsequent operation of the same job, $O_{i\{j+1\}}$. These arcs have length zero.

- If job J_i does not split or merge into another job, its final operation

O_{io_i} spawns alternative arcs with origins directly at this final operation O_{io_i}. These arcs have length p_{io_i}, which is the processing time of the final operation O_{io_i}.

- If job J_i splits into a set $\mathcal{J}_{\text{split}}$ of s jobs, so that $\mathcal{J}_{\text{split}} = \{J_j\}_{1 \leq j \leq s}$, then its final operation O_{io_i} spawns sets of alternative arcs from the first operation O_{j1} of each job $J_j \in \mathcal{J}_{\text{split}}$ created by the split. These arcs have length zero. For instance, if the final operation of J_i is performed on machine M_k, then, for each other operation in the schedule requiring machine M_k, a set of alternative arcs points to that operation, one arc from each first operation O_{j1} of each job $J_j \in \mathcal{J}_{\text{split}}$. This ensures that machine M_k is not released until all the jobs created by the split have started processing.

- If job J_i merges into another job, J_m say, a dummy operation $O_{i\{o_i+1\}}$ is appended to its sequence of operations (Figure 8.7.2). The processing time of these dummy nodes equals zero, i.e., $p_{i\{o_i+1\}} = 0$. The resource requirement for dummy node $O_{i\{o_i+1\}}$ is the machine on which the merge will take place, i.e. the machine required by O_{m1}. These dummy nodes do not spawn alternative arcs. Note that in this case, the final physical operation of the job, O_{io_i}, spawns its alternative arcs with origins at the dummy operation $O_{i\{o_i+1\}}$, with length zero.

- Let J_m be a job created by a merge of other jobs. When the destination of an alternative arc would normally have been the first operation

O_{m1} of J_m, a set of alternative arcs is created. This set contains one arc for each dummy operation $O_{i\{o_i+1\}}$ associated with the merge. The arcs point towards the dummy operations preceding O_{m1}, rather than directly to O_{m1} (Figure 8.7.2). The length of these arcs is determined by their origins, as described above.

Some operations to be scheduled on the same machine already have their relative ordering imposed by the job arcs and precedence constraints. Alternative arcs are omitted in those instances.

8.7.4 Alternative Graph Complexity

Under the alternative graph formulation, the blocking job shop scheduling problem can be expressed as a 0-1 integer programme. Each pair of alternative arcs is represented by a binary decision variable. The value of this binary decision variable determines which of the two alternative arcs is included in the solution. The number of variables in this binary variable formulation grows polynomially with the number of operations in the problem. This was shown for the disjunctive graph in Section 4.2.1, and the same reasoning applies here.

8.8 Blocking Job Shop Schedule Construction

These sections introduce a novel schedule construction method, which deals with infeasibility problems associated with the blocking job shop problem. For the non-blocking version of the scheduling problem in Part II, metaheuristics, such as Ant Colony Optimisation (ACO, Section 5.2), were used to quickly obtain near-optimal solutions to the problem. These near-optimal solutions were then used as upper bounds by a Branch and Bound algorithm (Section 4.1), to obtain the optimal solution. This heuristic-exact hybrid approach was found to work well. Therefore it is natural to investigate the performance of similar approaches for the blocking job shop problem. ACO involves the rapid creation of many schedules. As it turns out, the schedule construction method used in the non-blocking case often leads to infeasible schedules when applied to the blocking job shop problem. The problem of infeasibility is discussed next, in Section 8.8.1. This is followed by the presentation of our novel schedule construction procedure, in Section 8.8.2. Our novel schedule construction method guarantees to produce feasible schedules in the blocking job shop.

8.8.1 Schedule Construction: Infeasibility

Constructive algorithms for traditional job shop scheduling problems can be designed in such a way that every constructed schedule is feasible. This

is the case for the schedule construction phase of the ant colony algorithm presented in Section 5.2. It was specifically designed to guarantee feasibility of the produced solutions. This guaranteed feasibility makes it possible to rapidly create large numbers of solutions. Many metaheuristics, including ACO, require a large number of solutions to be created to work well. However, in the case of the blocking job shop, it is not always possible to complete a given partial schedule into a full solution. In fact, just determining whether this is possible for a given partial solution is a strongly NP-complete problem (Mascis and Pacciarelli (2002)).

The open question is then whether it is possible to adapt the ACO method proposed for the blocking case. One approach would be to create schedules as before, and discard any that are found to be infeasible. This procedure could be repeated until the required number of feasible schedules is created. Unless each attempt is successful, this approach will take more time than the non-blocking ACO. However, the non-blocking version of the algorithm had a short running time, so some additional running time might be acceptable.

Some initial experimentation with a blocking version of the ACO algorithm showed that it is actually very challenging to create feasible schedules in a blocking environment. The schedule creation process adds one operation at a time to the back of the existing partial schedule, as in the non-blocking case. In early iterations of the algorithm, the percentage of schedule creation attempts that led to feasible solutions was generally less

than 1%. In later iterations this percentage slowly increased. Overall, this made the algorithm very slow.

One particular challenge for constructive algorithms in a blocking environment is the time wasted between infeasibility being a fact, and its detection. In fact, a partial schedule which only contains two operations may already be impossible to complete, unbeknownst to the algorithm. To see this, consider two jobs, J_1 and J_2, which follow identical paths through the system, but in opposite directions. J_1 requires machines $M_1 \rightarrow M_2 \rightarrow M_3$, and J_2 requires machines $M_3 \rightarrow M_2 \rightarrow M_1$. Suppose now that the first operation of J_1 is added to the partial schedule first, on machine M_1, and that the first operation of J_2 is added second, on machine M_3. It is now inevitable that these two jobs will end up blocking each other in a deadlock situation. Infeasibility is a fact. The difficulty is that, without additional checks, the algorithm will not be aware of this unavoidable deadlock. Blissfully unaware, the schedule construction process continues building on the doomed partial schedule. There is nothing to stop it adding many operations belonging to other jobs to the partial schedule, before the inevitable deadlock is eventually encountered. This wastes valuable computational time.

If the partial schedule is represented by an alternative graph (Section 8.7), feasibility checks can be performed. This can be done by performing repeated Immediate Selection checks (Section 4.3.1), until each variable has been checked once since the last variable was fixed, and testing whether

the alternative graph of the partial solution contains any cycles. Unfortunately, this has a high computational cost, and it gives the algorithm an undesirably long running time. In the absence of feasibility checks, much time is spent attempting to complete partial schedules which are impossible to complete. This also carries a high computational cost. In conclusion, the computational costs are high both when feasibility checks are performed, and when they are not.

8.8.2 Schedule Construction: Guaranteed Feasibility

We have developed a novel schedule construction method that guarantees feasibility of the partial schedule at certain intervals in the construction process. These 'safe points' are partial schedules which are guaranteed to have at least one feasible extension. Whenever infeasibility is encountered beyond one of these safe points, the schedule construction algorithm reverts to the last encountered safe point, and continues from there. The previous section highlighted the high computational costs incurred both when feasibility checks are employed, and when they are not. The method introduced in this section avoids these inefficiencies.

An overview of the schedule construction procedure appears in Algorithm 5. Initially, our schedule construction method creates a topological sorting of all the jobs to be scheduled, subject to their precedence constraints (Step 1). This topological sorting determines the order in which the jobs will be inserted into the partial schedule. A safe point is reached

Algorithm 5 Schedule construction procedure for the blocking job shop. Feasibility of the final schedule is guaranteed.

Create topological sorting $\mathcal{J}^{(\mathrm{s})}$ of all jobs $J_i \in \mathcal{J}$
for all jobs $J_i \in \mathcal{J}^{(\mathrm{s})}$ **do**
  while J_i not completely scheduled **do**
    for all operations $O_{ij} \in J_i$ **do**
      Insert O_{ij} into the partial schedule, either randomly or according to some heuristic rule
      if partial schedule infeasible **then**
        Remove all operations of J_i from the partial schedule
        Break for-loop
      end if
    end for
  end while
end for
Return the feasible solution

whenever the partial schedule contains only complete jobs (Step 2). The first job in the sorting will always be added first. The algorithm proceeds by sequentially scheduling all the operations of the first job (Step 4). This causes no problems, since a single job cannot cause deadlock on its own. The partial schedule now contains all the operations of the first job. Assuming there are no merging or splitting jobs, feasibility is now guaranteed. This is because in the topological sorting, all remaining jobs come after the job that is already scheduled. Therefore, a feasible schedule can always be found by appending the remaining jobs to the partial schedule, in topological order.

Next, the operations of the second job are added to the partial schedule. The operations can be inserted at any point, subject to any precedence constraints. The insertion points are selected by the metaheuristic employing this schedule building method (Step 5). At this stage infeasibility may occur, if the operations of the second job are inserted in such a way that leads to deadlock with the first job (Step 6). If deadlock occurs, the algorithm reverts back to the partial schedule containing only the operations of the first job, and continues from there (Steps 7 and 8). Feasibility of the partial schedule is once again guaranteed once all the operations of the second job have been added to the partial schedule (Step 11).

This process continues until the schedule is complete. Whenever deadlock is encountered during the insertion of the operations of the $(k+1)^{th}$ job, the algorithm reverts to the partial schedule containing the first k jobs.

From that point, a feasible extension is guaranteed to exist.

The topological sorting schedule construction approach described above could fail when some of the jobs to be scheduled contain splits or merges. Consider, for example, the case where two jobs, J_1 and J_2, merge into a third, J_3. The merge takes place on machine M_{merge}. A topological sorting may then be J_1, J_2, J_3. Recall that the final operation of jobs which merge is a dummy operation on M_{merge} (see Section 8.7.2). This means that when J_1 is completely scheduled, the scheduled operations on M_{merge} are as follows:

$$[\ \ldots,\ O_{1,o_1},\ O_{1,o_1+1},\ \ldots] \tag{8.8.1}$$

Since the first operation of J_3, $O_{3,1}$, must immediately follow the dummy operation O_{1,o_1+1}, the partial schedule containing the sequence in Equation 8.8.1 implies the following sequence on M_{merge}:

$$[\ \ldots,\ O_{1,o_1},\ \{\ O_{1,o_1+1},\ O_{3,1}\},\ \ldots] \tag{8.8.2}$$

Thus, J_1 blocks M_{merge} until J_3 starts. Suppose that J_2 contains an operation, $O_{2,j}$ say, which also requires M_{merge}. The feasibility guarantee depends on the possibility of appending $O_{2,j}$ to the end of the current partial schedule. However, the precedence constraints require that $O_{2,j}$ is scheduled before $O_{3,1}$. Thus it would not be possible to append $O_{2,j}$ to the end of the sequence in Equation 8.8.2, and it can no longer be guaranteed that a feasible extension exists. There may even be a fourth job, J_4, with no precedence constraints with respect to the other three jobs. Suppose J_4 also requires M_{merge}. The following would be a valid topological sorting: $[J_1, J_2, J_4, J_3]$.

However, if J_4 is appended to the partial schedule containing only J_1 and J_2, it will not be able to use M_{merge}, since this has already been reserved for J_3. Of course, by chance it may sometimes be possible to insert the relevant operation of J_4 earlier in the schedule. However, the guaranteed feasibility through appending to the back of the schedule no longer exists. To resolve this, the schedule construction algorithm treats all the jobs associated with a merge (or split) as a single job. This means these jobs are put together, uninterrupted, in the topological sorting. There is a safe point before all operations of such a group have been inserted, followed by a safe point after all the operations of the group have been inserted. There are no safe points within such a group, as it was shown that feasibility cannot be guaranteed there. In the preceding example there would be a group, G_1 say, such that $G_1 = [J_1, J_2, J_3]$. A valid topological sorting would then be either $[G_1, J_4]$ or $[J_4, G_1]$.

There is one other case in which the topological sort cannot guarantee feasibility. Consider the set of jobs shown in Figure 8.8.1. J_1 and J_2 initially merge into J_3, which in turn splits into J_4 and J_5. This combination of a merge followed by a split will cause jobs 1-5 to be put into a single group in the topological sorting. There is one other job, J_6, which must be preceded by J_1, and which itself preceeds J_4. However, J_1 and J_4 will be put in the same group, and it will not possible to find a topological sorting when J_6 must simultaneously be scheduled before and after the group. This can be resolved by including J_6 in the group of jobs 1-5.

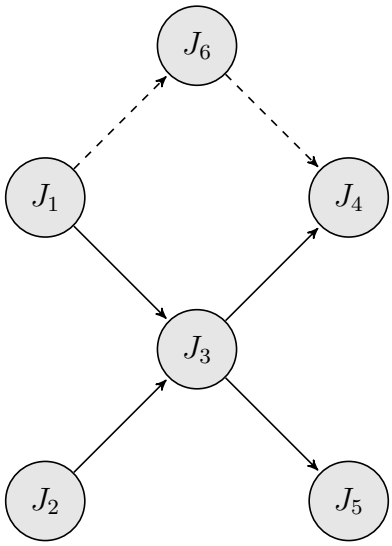

Figure 8.8.1: Jobs 1-5 are bound by merges and splits, and as such will appear as a single group in the topological sorting. Job 6 has precedence constraints with two jobs in this group, which makes it impossible to find a topological sorting, unless J_6 is absorbed in the group.

The guaranteed feasibility schedule construction method presented above, is incorporated in the Ant Colony and Rollout algorithms presented in Chapter 9.

Chapter 9

Blocking Job Shop: Heuristic Methods

9.1 Introduction

This chapter presents a number of heuristic methods that attempt to solve the blocking job shop problem with flexible maintenance, as described in Chapter 8. For the non-blocking variant of this problem, a number of heuristic methods were shown to perform very well (Chapter 5). This makes the adaptation of these algorithms for the blocking problem a logical first step.

A novel Ant Colony Optimisation (ACO) variant for the blocking job shop is presented in Section 9.2. This is followed by a Rollout algorithm in Section 9.3. Finally, a Simulated Annealing (SA) algorithm is presented in Section 9.4. The computational experiments in Chapter 11 show that these

heuristic methods provide better solutions, in terms of the total tardiness penalty T, than those provided by a topological sorting (Section 8.8.2) or a topological sorting followed by a local optimisation routine (Section 9.3.1). SA usually outperforms both ACO and the rollout method, in terms of the overall tardiness penalty T. The trade-off is that SA has a larger median running time (286 seconds) than ACO (206 seconds) and the rollout method (168 seconds). All of the heuristics struggle to find solutions that are close-to-optimal.

9.2 Ant Colony Optimisation for the Blocking Job Shop

Ant Colony Optimisation was shown to be a well performing solution method for the non-blocking variant of our scheduling problem (Section 5.2). It is therefore of interest to investigate whether ACO also performs well in the blocking case.

Ant Colony Optimisation (ACO) (Dorigo and Stützle, 2004) is based on the natural route finding behaviour observed in ants. Ants leave a pheromone trail as they travel between their nest and a food source, allowing other ants to follow the same path. However, on the initial discovery of the food source, a number of ants may have arrived there by different routes, some shorter than others. Other ants will follow the various pheromone trails to the food source and deposit additional pheromone.

The shorter routes will be travelled more frequently, resulting in greater pheromone deposits compared to the longer routes. This will encourage more ants to follow the shorter route, until eventually most ants will be using the shortest route. ACO algorithms mimic this natural phenomenon in order to find good solutions to difficult problems.

9.2.1 Schedule Construction

It was discussed in Section 8.8.1 that schedule construction under the blocking constraint often leads to infeasibility. To reduce the number of times that infeasibility is encountered, the ACO algorithm for the blocking case will use the novel guaranteed feasibility schedule construction method presented in Section 8.8.2.

Initially, the schedule is completely empty. Operations are added to the schedule one at a time. The order in which operations are added is fixed in advance by a topological sorting. A safe point is reached whenever the partial schedule contains only complete jobs. Whenever the addition of an operation results in an infeasible partial schedule, the schedule construction algorithm reverts to the most recent available safe point. For each operation, the scheduling decision to be taken is where to insert the operation in the sequence of operations already scheduled on the required machine. Job precedence constraints are checked to determine the earliest possible insertion point. The actual insertion point of the operation is determined by rolling a weighted die. These weights are based on two aspects: The

pheromone levels, which contain information on how rewarding it has been in the past to insert the node at each available index, and heuristic information based on the quality of the scores of the partial schedules created by inserting the operation at each available index. More specifically, the probability with which operation i will be inserted at index j depends on the following:

- The **pheromone level** τ_{ij}. Pheromone levels change over time depending on the quality of the routes found by the ants. The pheromone level τ_{ij} will be higher if inserting operation i at index j has resulted in relatively good schedules previously, and lower if such a move has been associated with less efficient schedules.

- The **heuristic information** η_{ij}. While pheromone levels consider information across previous schedules, the heuristic information accounts for the state of the current partial schedule. It is usually some greedy rule. Here, the heuristic information is based on the score of the extended partial schedule, the Extended Schedule Score, resulting from insertion of operation i at index j, as follows:

$$\eta_{ij} = \frac{1}{\text{Extended Schedule Score}} \tag{9.2.1}$$

Let $\mathcal{N}_i$ denote the set of available insertion indices. Operation i will then be inserted at index $j \in \mathcal{N}_i$ with probability

$$p_{ij} = \frac{{\tau_{ij}}^{\alpha}{\eta_{ij}}^{\beta}}{\sum\limits_{j \in \mathcal{N}_i} {\tau_{ij}}^{\alpha}{\eta_{ij}}^{\beta}}, \quad \text{for } j \in \mathcal{N}_i \ . \tag{9.2.2}$$

The parameters α and β determine how strongly pheromone levels and heuristic information influence the transition probabilities. Initially all pheromone levels are equal, so that the heuristic information will have the greatest influence on schedule construction in early iterations. Over time, the pheromone levels start to reflect which moves have previously resulted in good schedules, and the schedule construction will eventually be guided by the pheromone levels as time progresses.

In the non-blocking case, we were able to explore the performance of several types of heuristic information, based on slack and distance (Section 5.2.1). Slack and distance were used to select the next operation to be appended to the schedule. However, in the blocking case, the next operation is determined in advance by the topological sorting. Additionally, operations are inserted into the schedule, rather than appended to it. Therefore, slack and distance are not useful heuristic measures in the blocking context. The only usable heuristic information we found was based on the scores of the partial schedules created by inserting the operation at each available index, as specified in Equation 9.2.1. Other alternatives were explored, including heuristics based on the increase in schedule score resulting from insertion at each available point, but these resulted in very poor performance. Such heuristics regularly failed to find any feasible solutions.

9.2.2 Pheromone Updating

ACO algorithms require an initial global pheromone level, τ_0. For the travelling salesman problem, a number of suggestions have been made for the value of τ_0 (Dorigo and Stützle, 2004). Most of these suggestions are in part proportional to $\frac{1}{C^{nn}}$, where C^{nn} is the value of the best solution found by the nearest neighbour algorithm. As a rank-based pheromone updating rule is used by the ACO algorithm presented in this chapter, the pheromone level is initialised to closely follow this recommendation according to the following formula:

$$\tau_0 = \frac{0.5 \times w \times (w-1)}{\rho \times C^{nn}}, \tag{9.2.3}$$

where w is the number of schedules (or the rank) used in the pheromone updating procedure, and ρ is the pheromone evaporation parameter. In the current problem context, poor quality solutions tend to have a few hard due date violations. Recall that each hard due date violation of a job incurs a fixed penalty of 10^6 (Section 8.4). As such, C^{nn} was fixed at the value of 2×10^6, to approximate the order of magnitude for poor solutions.

At each iteration of the algorithm, n ants visit all operations exactly once, according to the previously described probabilistic rules. In the non-blocking case, we used $n = 100$ ants (potential schedules). That is approximately equal to the total number of operations, as recommended for TSP problems by Dorigo and Stützle (2004). However, schedule construction is much slower under the blocking constraint. Therefore it was of interest to

test whether the algorithm can perform well with fewer schedules per iteration, and n was treated as an additional parameter. Once the n schedules have been constructed, the pheromone trails are updated. Firstly, global pheromone evaporation takes place according to the following formula:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}, \quad \forall\ (i,j), \tag{9.2.4}$$

where ρ is the pheromone evaporation rate. The higher this rate, the quicker the algorithm will converge. Lower values of ρ encourage more exploration of the solution space. Evaporation affects every path, irrespective of whether ants have recently travelled along them. The result of this is that the paths that are not used much will become less attractive to the ants over time.

After evaporation, rank-based pheromone updating takes place (Bullnheimer et al., 1997) and, if activated by the user, elitist updating. If elitist updating is active, then out of the n schedules, the $w-1$ best are selected and ranked by schedule score. The best schedule, i.e. that with the lowest total tardiness penalty (as defined in Section 8.4), is assigned the lowest rank $r = 1$. The schedule with the largest penalty is given the highest rank $r = w-1$. Let (i,j) denote a 'path' taken by an ant from operation i to index j. The inclusion of this path (i,j) in a solution specifies that operation i is inserted at index j on its required machine. The quantity τ_{ij} is the corresponding pheromone level. For each of the $w-1$ iteration-best solutions, if (i,j) is part of that solution, it receives a pheromone deposit.

The amount of pheromone deposited depends both on the total tardiness penalty of each schedule, $P^{(r)}$, and its rank r. More specifically, if elitist updating is active, the following deposits are made:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1}(w-r)\frac{1}{P^{(r)}} \quad \forall\ (i,j) \in \text{ schedule } r. \tag{9.2.5}$$

An additional elitist pheromone deposit is then made on the arcs of the best-so-far schedule. If this schedule has total tardiness penalty $P^{(bs)}$, the pheromone levels are updated as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + w\frac{1}{P^{(bs)}} \quad \forall\ (i,j) \in \text{ best-so-far schedule.} \tag{9.2.6}$$

The binary parameter e determines whether elitist updating is to take place ($e = 1$) or not ($e = 0$). If elitist updating is disabled, the additional deposit in Equation 9.2.6 for the best-so-far schedule is not made. In that case, the w iteration-best solutions are used for updating the pheromones. The iteration-best schedule is still assigned rank $r = 1$, but the schedule with the largest penalty now has rank $r = w$. The updating equation then becomes

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w}(w-r)\frac{1}{P^{(r)}} \quad \forall\ (i,j) \in \text{ schedule } r. \tag{9.2.7}$$

The previously mentioned operations are repeated until a specified stopping criterion is reached, e.g. a fixed number of iterations. At that point, the best-so-far schedule is returned as the preferred solution. The structure of the ACO algorithm described in this section is presented in Algorithm 6.

Algorithm 6 Pseudo code of the developed Ant Colony Optimisation algorithm.

Initialise the pheromone matrix (Equation 9.2.3)
while stopping criterion not met **do**
    Construct n schedules (Section 9.2.1)
    Rank schedules by lowest score
    if schedule with rank 1 is better than best-so-far **then**
        Schedule with rank 1 replaces the best-so-far schedule
    end if
    Global pheromone evaporation (Equation 9.2.4)
    Rank-based pheromone deposit on arcs of $w-1$ (w if non-elitist) best schedules (Equations 9.2.5 and 9.2.7)
    if elitist updating active **then**
        Elitist pheromone deposit on arcs of best-so-far schedule (Equation 9.2.6)
    end if
end while
Return the best-so-far solution.

9.2.3 ACO Parameters

All parameters that influence the performance of the ACO algorithm are listed in Table 9.2.1. In order to optimise the performance, the parameters were fine-tuned with SMAC (Section 6.3, Hutter et al. (2011)). The first 10 problem instances (Section 11.1) were used to fine-tune the parameters. The results obtained with SMAC show that the parameter values recommended in Table 9.2.2 perform best. Many configurations within the recommended ranges perform similarly.

The computational experiments in Chapter 11 show that the performance of the ACO method presented above is not very encouraging. Compared to three existing rollout methods (Meloni et al. (2004), Section 9.3), it outperforms just one (SMCP) with respect to solution quality, but not with respect to running time. Our ACO method is outperformed by the remaining two rollout methods (AMCC, SMSP), both in terms of solution quality and running time required.

9.3 A Rollout Algorithm for the Blocking Job Shop

The job shop scheduling problem with blocking and no-wait constraints is considered by Meloni et al. (2004). They employ the alternative graph (Mascis and Pacciarelli (2000); Section 8.7), an adaptation of the well-

Parameter	Type	Range	Parameter description
α	continuous	$\mathbb{R}_{\geq 0}$	Pheromone parameter
β	continuous	$\mathbb{R}_{\geq 0}$	Heuristic parameter
ρ	continuous	$[0, 1]$	Evaporation parameter
w	integer	$\mathbb{Z}_{>0}$	Rank parameter
n	integer	$\mathbb{Z}_{>0}$	Number of ants
e	binary	$\{0, 1\}$	Elitist pheromone updating?

Table 9.2.1: The block shop ant colony optimisation parameters.

Parameter	Recommended Value	Recommended Range
α	0.823	[0.7, 1.27]
β	0.613	[0.56, 0.85]
ρ	0.334	[0.18, 0.53]
w	3	[2, 4]
n	75	[60, 78]
e	0	0

Table 9.2.2: Recommended values and ranges for the block shop ant colony optimisation parameters.

known disjunctive graph (Roy and Sussmann (1964); Section 4.2), to model the problem. A rollout metaheuristic is implemented to obtain feasible solutions by Meloni et al. (2004). The rollout method sequentially fixes all the decision variables in a given problem. At each iteration, the most promising of the remaining decision variables is added to the current partial solution. Meloni et al. (2004) note that extending a partial solution of a blocking job shop into a complete solution is not always possible. They describe three heuristics for selecting the next variable to be fixed. Each heuristic extends partial solutions by iteratively selecting one of the remaining unselected arcs and adding it to the current partial solution. The three rollout heuristics use different methods to select one of the remaining decision variables, as follows:

1. *Avoid Maximum Current Completion time* (AMCC), identifies the alternative arc that would increase the makespan the most, and then selects its alternative. Any ties are broken by selecting the pair of arcs with the smaller minimum increase.

2. *Select Most Critical Pair* (SMCP), selects the pair of arcs with the largest minimum increase in the makespan. From this pair, the arc with the smaller increase in makespan is selected. Meloni et al. (2004) mention no tie breaking rule. In our implementation, ties are broken by maximum largest increase in penalty score.

3. *Select Max Sum Pair* (SMSP), selects the pair of arcs with the great-

est combined increase in makespan, and out of these two selects the arc with the smallest increase in makespan. Meloni et al. (2004) mention no tie breaking rule. In our implementation, ties are broken by selecting the pair with the smallest minimum increase in penalty score.

We have implemented the three above heuristics for the blocking job shop with due dates and flexible maintenance, in order to compare their performance against our own algorithms. Note that Meloni et al. (2004) were concerned with minimising the makespan, whereas we are concerned with minimising the tardiness penalty. Thus, in order to obtain a description of our implementation, replace "makespan" with "tardiness penalty" in the heuristic descriptions above.

One disadvantage of the rollout methods by Meloni et al. (2004) is that they require a large number of schedule evaluations. At each iteration, both alternatives of each pair of arcs have to be evaluated. Therefore, if a problem consists of n alternative arcs, one of which will be fixed at each iteration, then the total number of schedule evaluations is $2\sum_{i=1}^{n} i = n\,(n+1)$. Our typical test problems have approximately 1050 alternative arcs, requiring in excess of 1.1 million schedule evaluations. In an attempt to reduce the number of schedule evaluations required, we created two rollout algorithms with a predetermined variable selection order. This order is determined by the order of the variables in the problem representation.

Thus, at each iteration, it is only necessary to do two schedule evaluations, one for each of the two alternative arcs. This was tested in two different ways:

1. **Random:** Each variable is fixed in turn, randomly, at 0 or 1. If the first choice is infeasible, the other option is selected instead.

2. **Minimum score:** Each variable, in turn, is evaluated at both values. The selected value is that which results in the partial schedule with the lower score.

Both methods struggled to find feasible solutions. The random method is not deterministic, and was therefore repeated a large number of times. Out of 100,000 schedule creation attempts, only 3 resulted in a feasible solution. This number is far too small to be useful, either on its own or as part of a different metaheuristic.

9.3.1 Rollout with Guaranteed Feasibility

To overcome the infeasibility issues outlined above, we have developed a novel rollout method which incorporates our guaranteed feasibility procedure presented in Section 8.8.2. Our rollout method uses the machine based model of the problem. In this model, the operations on each machine are ordered in the way that they are to be processed. Once each machine has all its operations ordered, this completely defines the schedule.

The rollout procedure is detailed in Algorithm 7. It starts with an empty schedule (Step 2). Operations are added one at a time. The order in which operations are added is fixed in advance by a topological sorting. When this topological sorting is determined, the byproduct is an initial schedule based on this sorting (Step 1). For each operation (Step 4), the scheduling decision to be made is where to insert it in the sequence of operations already scheduled on the required machine. If there is only one feasible insertion point (Step 5), the operation is inserted there and the algorithm moves on to the next operation. When there are multiple available insertion points, a partial solution is created for each feasible insertion point (Step 9). From our guaranteed feasibility schedule creation method, we know that a partial solution can be feasibly completed whenever only complete job groups have been scheduled. The temporary partial solution on each branch is then checked for feasibility (Step 11), by systematically checking whether the remaining nodes in the current job group can be feasibly added to the partial schedule. If this is not possible, there is no feasible extension on this branch and it is discarded. If there is only one remaining branch, this is accepted as the new partial schedule. If multiple branches remain, each branch is randomly completed c times (Step 12). One of the branches is then selected as the new partial schedule (Step 15). This can be done according to one of three policies:

1. Select the branch with the minimum penalty score.

Algorithm 7 Pseudo code of the developed rollout algorithm.

```
Create an initial solution based on topological sorting of the jobs.
Create empty solution
while unscheduled operations remain do
    Select the next operation to be scheduled
    if single feasible insertion point available then
        Insert operation
        Return to Step 3
    else if multiple feasible insertion points available then
        Create a branch for each insertion point
        for each branch do
            Check branch for feasibility
            Complete branch c times
        end for
    end if
    Partial schedule on most promising branch is retained
    if partial schedule is no better than the incumbent then
        Break while-loop
    end if
end while
Return the best-so-far solution.
```

2. Select the branch with the smallest mean penalty score.

3. Select the branch with the smallest mean of the log penalty scores. This option was included because the fixed penalties for due date violations have a very strong influence on the mean penalty score.

Any complete solution encountered during the rollout procedure is compared against the best-so-far solution, and replaces this if it is better. There is also the option to apply a local optimisation routine, based on single-operation swap, to improve any complete solutions. If this local optimisation is activated, each operation, in turn, is removed from the sequence of operations on its respective machine. Every schedule that can be created by reinserting the operation at each possible position on this machine is evaluated. The operation is then returned into the schedule at the place with the minimum schedule score. This continues until no improvement has been found for an entire cycle through all the operations.

9.3.2 Rollout Parameters

The parameters listed in Table 9.3.1 influence the performance of the rollout algorithm. In order to optimise the performance, the parameters were fine-tuned with SMAC (Section 6.3, Hutter et al. (2011)). Each rollout run was terminated after five minutes. This is the same time limit that was given to other methods for parameter optimisation, and therefore provides a fair comparison. The results obtained with SMAC show that the

Parameter	Type	Range	Parameter description
c	integer	$\mathbb{Z}_{>0}$	Number of schedules per branch
l	binary	$\{\text{disabled}, \text{enabled}\}$	Local optimisation
s	categorical	min,mean,meanlog	Branch selection parameter

Table 9.3.1: The blocking job shop rollout parameters.

Parameter	Recommended Value	Recommended Range
c	30	[20,100]
l	Enabled	Enabled
s	minimum	minimum

Table 9.3.2: Recommended values and ranges for the blocking job shop rollout parameters.

parameter values recommended in Table 9.3.2 give the best performance. Performance was clearly better with local optimisation enabled. Selecting the branch with the minimum value gave, on average, a slightly better performance than the other options. With these two parameters fixed, performance of the algorithm does not vary much over the number of times each branch is completed, with values for c over the range $[20, 100]$ giving similar performance.

The computational experiments in Chapter 11 show that the perfor-

mance of the rollout method presented above is not very encouraging. While it outperforms the SMCP method of Meloni et al. (2004), with respect to solution quality, it also requires more running time. Our method is outperformed by AMCC and SMSP, both in terms of solution quality and running time required.

9.4 Simulated Annealing for the Blocking Job Shop

We have developed another heuristic solution method for the blocking job shop problem with flexible maintenance, based on Simulated Annealing (SA). An overview of SA is provided in Section 2.3.3. In this section, the blocking job shop is modelled as a sequence of operations on each machine. The main ingredient of SA is the perturbation mechanism. Here, we employ two types of perturbation, namely the Single Operation Reschedule, and the Job Group Reschedule.

9.4.1 Perturbation: Single Operation Reschedule

The first type of Simulated Annealing perturbation is the Single Operation Reschedule. A single operation is selected, at random and with equal probability. This operation is removed from the sequence of operations on its respective machine. Precedence constraints are used to determine the ear-

liest and latest potential reinsertion points in this sequence. The operation is then randomly reinserted within this range. Due to the blocking constraint, the resulting schedule may be infeasible. In that case, the random reinsert is repeated until a feasible insertion point is found. Reinsertion in the original location is possible, and has the same probability of reinsertion at any other point in the sequence. This provision is needed for those occasions when reinsertion is only feasible at the original position.

While the simplicity of the Single Operation Reschedule is attractive, its drawback is that it is not always able explore the entire solution space. To see this, consider two identical jobs, J_1 and J_2. Both jobs first require machine M_1, and then machine M_2. In a blocking job shop, such identical jobs cannot overtake each other. In any valid solution, J_1 either precedes J_2 on both machines, or follows it on both machines. A single operation swap can only change the order of the jobs on one of the machines. However, this will result in an infeasible schedule, in which one job is overtaken by another. This means it is impossible to reach that part of the solution space in which the order of jobs J_1 and J_2 is swapped.

9.4.2 Perturbation: Job Group Reschedule

The Job Group Reschedule is a perturbation used to ensure that the entire solution space can be explored. In the Job Group Reschedule, one complete job group is selected for rescheduling, with random and equal probability. Recall that a job group contains all jobs connected through merges and

splits (Section 8.8.2). Here, a single job without any merges or splits is also considered to be a job group. All the operations in the selected job group are removed from the schedule. The operations are then randomly reinserted into the sequences on their respective machines. Again, some reinsertions may not be feasible, due to the blocking constraints. Whenever an attempted reinsertion leads to infeasibility, all operations in the selected job group that are already reinserted are once again removed from the schedule, and the reinsertion process is restarted. Once all operations have been successfully returned into the schedule, this is the new proposed schedule. Again, there is no guarantee that the new proposed schedule is not identical to the old one. This is intentional, since there may be occasions where the selected job(s) cannot be returned to any position other than the original one.

9.4.3 Algorithm Structure

Given that the number of job operations is denoted by n, the change in objective function between the current solution and the proposed solution by ΔE, and that g is the interval at which the Job Group Reschedule perturbation is used, the algorithmic structure of the developed Simulated Annealing algorithm follows Algorithm 8. This closely follows the proposed approach in Dréo et al. (2006).

Algorithm 8 Pseudo code of the developed Simulated Annealing algorithm.

Set the initial temperature, T_0
Create an initial solution based on topological sorting of the jobs
while termination criteria not met **do**
  while cooling criteria not met **do**
    if iteration is not a multiple of g **then**
      Select a candidate solution from Single Operation Reschedule
    else
      Select a candidate solution from Job Group Reschedule
    end if
    At temperature T, accept the candidate solution with probability $\min\left\{1, \exp(-\frac{\Delta E}{T})\right\}$
  end while
  Reduce the temperature: $T_{\text{new}} = \alpha \times T_{\text{old}}$
end while
Return the best known solution

9.4.4 Simulated Annealing Parameters

The performance of the SA algorithm depends on a number of parameters, shown in Table 9.4.1. The cooling parameter α lies in the range $(0, 1)$. The initial temperature, $T_0 > 0$, must be set. Let n denote the number of operations to be scheduled. Cooling is triggered when either $k \times n$ moves have been accepted, or $l \times n$ moves have been attempted, whichever happens first (for $k \leq l$). Note that moves only count as accepted if the value of the accepted solution is different from the incumbent solution it replaces. This is because solutions with equal value are always accepted and the algorithm sometimes repeatedly jumps back and forth between solutions of equal quality. Finally, the Job Group Reschedule takes place at every g iterations.

The parameters were optimised with SMAC (Hutter et al. (2011)). Each Simulated Annealing run was terminated after five minutes. This is the same time limit that was given to other methods for parameter optimisation, and therefore provides a fair comparison. The results obtained with SMAC show that the parameter values recommended in Table 9.4.2 give the best performance.

The computational experiments in Chapter 11 show that the performance of the Simulated Annealing method presented above is not very encouraging. While it outperforms the SMCP method of Meloni et al. (2004), as well as our own Rollout and ACO methods, with respect to solu-

Parameter	Type	Range	Parameter description
α	continuous	$(0, 1)$	Cooling rate
T_0	continuous	$\mathbb{R}_{\geq 0}$	Initial temperature
k	integer	$\mathbb{Z}_{>0}$	Acceptance based cooling trigger
l	integer	$\mathbb{Z}_{>0}$	Attempt based cooling trigger
g	integer	$\mathbb{Z}_{>0}$	Group reschedule interval

Table 9.4.1: The block shop Simulated Annealing parameters.

Parameter	Recommended Value	Recommended Range
α	0.88	[0.2,0.89]
T_0	656924	[600000, 850000]
k	107	[80, 125]
l	250	[240, 450]
g	2	[2, 4]

Table 9.4.2: Recommended values and ranges for the block shop Simulated Annealing parameters.

tion quality, it also requires more running time. Our Simulated Annealing method is outperformed by AMCC and SMSP, both in terms of solution quality and running time required.

Chapter 10

Blocking Job Shop: Exact Methods

10.1 Introduction

For the non-blocking variant of the job shop scheduling problem with flexible maintenance, Branch and Bound (B&B) was shown to perform well when hybridised with heuristic methods (Chapter 4). Since the heuristic methods developed for the blocking job shop in Chapter 9 generally showed a disappointing performance, it is of interest to investigate whether exact methods offer a suitable alternative. This chapter presents a novel B&B method which solves to optimality the blocking job shop problem with flexible maintenance. This problem is described in Chapter 8. The proposed B&B method employs a novel branching strategy (Section 10.4.2), as well as a novel search strategy (Section 10.6.5). The computational experiments

in Chapter 11 show that the proposed Branch and Bound method clearly outperforms almost all tested heuristic methods, both in terms of running time and solution quality. Only the heuristics based on Topological Sorting have a shorter running time, but these are not suitable alternatives due to the poor quality solutions they produce.

The Branch and Bound method was first introduced by Land and Doig (1960). A survey of recent advances in B&B is presented in Morrison et al. (2016). Branch and Bound has been applied to solve a wide range of discrete optimisation problems. A basic overview of the general B&B procedure is presented in Section 2.4.

An overview of the proposed Branch and Bound method is given in Section 10.2. The heuristic presented in Section 10.3 provides an initial upper bound. Existing and novel branching strategies are discussed in Section 10.4. This is followed by a discussion of existing and novel search strategies in Section 10.6. The algorithm parameters, and their recommended values, are presented in Section 10.8. The chapter then concludes with a sensitivity analysis of the B&B parameters in Section 10.9.

10.2 Branch and Bound for the Blocking Job Shop with Flexible Maintenance

One of the main factors determining the efficiency of Branch and Bound is the ability to quickly prune branches. We employ several techniques to

speed up the pruning process, including a novel ranking method for the decision variables, as well as a novel node selection procedure. The current section first describes the main structure of the developed Branch and Bound algorithm. This is followed by a discussion of the main components of the algorithm.

The Branch and Bound algorithm presented here uses the novel modification of the alternative graph described in Section 8.7 to model the scheduling problem. Recall that alternative arcs determine the order in which two operations will be processed on the same machine. Each pair of mutually exclusive alternative arcs has a corresponding binary decision variable x_i. If $x_i = 0$, then the first of these two alternative arcs is included in the solution. The other arc is included when $x_i = 1$.

The main structure of the algorithm is shown in Algorithm 9. The best-known solution is updated any time a better solution is encountered, which allows any branch with a score (lower bound) no better than the best-known solution to be pruned. An initial incumbent solution is created through topological sorting followed by local optimisation (Step 1). This solution serves as an initial upper bound. The algorithm then creates a single incomplete solution. This is built by only including the job arcs from the alternative graph representation, since the job arcs are fixed (Step 2). To complete the solution, exactly one arc must be selected from each pair of alternative arcs. For a solution to be feasible, its corresponding graph must be a directed acyclic graph. In a partial schedule, the earliest starting time

Algorithm 9 Pseudo code of the developed Branch and Bound algorithm for the blocking job shop with flexible maintenance.

```
Use a fast heuristic (topological sorting with local optimisation) to obtain the first incumbent solution
Create a partial solution with job arcs only
Perform Immediate Selection (Section 10.5)
Rank and sort the remaining decision variables (Section 10.4.2)
while at least one partial solution remains do
    Reset gradient-based sort origin (Section 10.6.5)
    Perform Strong Branching on the single partial solution (Section 10.4.1)
    Regular Immediate Selection
    Reset the count that triggers solution stack reordering
    while at least two partial solutions remain do
        if stack reorder-threshold reached then
            Reorder the solution stack (Section 10.6.5)
        end if
        Take a partial solution from the top of the stack
        if the current branching level is a multiple of m then
            Perform Immediate Selection
            if solution is now complete or no better than best-so-far then
                Prune branch and restart while loop
            end if
        end if
        Branch on the first non-fixed variable
    end while
end while
Return optimal solution
```

of each operation can be calculated as the longest path to the associated node. Each time a machine arc is fixed in the partial solution, the longest path to every node either remains the same or is increased. Therefore, the schedule score cannot be decreased by adding arcs, and any partial solution which has a score worse than or equal to the best known solution can be discarded. The branching scheme branches on the alternative arcs that have not yet been fixed in the partial schedule. Branches are pruned as soon as they are known to be either infeasible or no better than the best-so-far solution.

The partial solutions, or open branches, are stored in a stack. Each time branching takes place, the two newly created branches are placed on top of the stack. The solution with the lower score is placed on the stack last. Then, the branch on top of the stack is selected next for further branching. Thus, the algorithm partly follows a depth-first Branch and Bound strategy. At certain intervals, the stack is reordered according to one of several search strategies (Section 10.6). This prioritises exploration of promising parts of the solution space which would possibly be left unexplored for much longer by a pure depth-first search procedure.

Having created a partial solution, the algorithm applies the Immediate Selection method (Step 3, described in Section 10.5, Carlier and Pinson (1989)) to reduce the number of remaining unfixed alternative arcs. The remaining variables are then ranked and sorted by a measure of criticality (Step 4), as described in Section 10.4.2.

The solution stack now contains a single partial solution (Step 5). The origin used by the gradient sort search strategy is updated at this point (Step 6, Section 10.6.5). Whenever the stack only contains a single solution, Strong Branching is applied (Step 7, Section 10.4.1, Klabjan et al. (2001)). Any solutions which are not pruned are returned to the top of the solution stack, with the solution with the smaller lower bound being returned last. Strong Branching is always followed by regular Immediate Selection (Step 8, Section 10.5) on all solutions in the stack (at most two). The count which triggers a stack reordering is also reset to 0 when only a single solution is present in the stack (Step 9). Whenever two or more solutions remain in the stack, regular branching takes place, but only after checks are performed to see whether stack reordering or Immediate Selection are due to take place.

Before regular branching takes place, the stack is reordered (Step 12) if during the previous f regular branching operations stack ordering has not taken place, and the incumbent solution has not been replaced. If more than one of the search strategies (Section 10.6.5) are active, the next reordering method is selected according to the round-robin principle.

A partial solution is taken from the top of the stack (Step 14). If this partial solution contains a multiple of m fixed variables, it undergoes Immediate Selection (Step 16). The motivation behind this is to reduce the remaining solution space. Each time a variable is fixed in the branching process, a binding constraint might be imposed on some unfixed variables. In other words, some of the remaining variables can now only be fixed in

one way. Fixing these variables may then lead to even more variables also being restricted to particular values. If the partial solution is now no better than the incumbent solution, it is discarded (Step 18), and the while loop restarts. Otherwise, the partial solution is returned to the top of the stack. If the solution is complete, a comparison with the best-so-far solution is performed. The solution replaces the best-so-far solution if it is better, otherwise it is discarded.

The partial solution at the top of the stack now undergoes regular branching (Step 21), on the first unfixed variable, i.e. the variable with the highest rank. During regular branching, the algorithm follows a quasi-depth-first branching strategy. At each branching point, the lower scoring branch is investigated first. Due to Constant Immediate Selection this is not entirely a depth-first search: Two solutions born from the same node may end up having a different number of additional variables fixed by the Constant Immediate Selection process (Section 10.5.1).

Finally, the Branch and Bound algorithm returns the optimal solution found during the search process.

10.3 Initial Upper Bound

Branch and Bound algorithms perform better with lower upper bounds, since this facilitates quicker pruning. If no initial upper bound is available, it has to be set to infinity. In order to obtain a better initial bound, it may

be possible to create an initial incumbent solution with some fast heuristic. This idea is demonstrated in Maslov et al. (2014), who note that "almost none of the existing exact algorithms apply" this approach of obtaining an initial solution with some heuristic method. We use a specialised heuristic to find an initial upper bound.

The initial incumbent solution is created in two steps. Firstly, a solution is created based on a topological sorting of the jobs. Secondly, a local optimisation routine is applied to improve the quality of this initial incumbent solution.

1. **Topological sorting:** The topological sorting schedule construction method closely follows the guaranteed schedule construction method that was introduced in Section 8.8.2. The main difference here is that jobs are never inserted into the partial schedule, but are always appended to it. Initially, a topological sorting of all the jobs is created, subject to their precedence constraints. Starting with an empty schedule, a full schedule is then created by appending all operations of each job in turn, in topological order. The intuition is that this aims to have at most one job in the system at any given time, so that no deadlock can occur. In reality there probably will be more than one job in the system at times, since a job can start its first operation as soon as its first required machine is vacated for the final time by all preceding jobs. Whenever jobs split or merge there will also be

more than one job in the system. In that case, the topological sorting will be based on groups of jobs, as described in Section 8.8.2.

2. **Local optimisation:** The schedule based on topological sorting of the jobs will most likely be of very poor quality. We apply a local optimisation routine, based on single-operation swap, to improve this initial schedule. Every schedule that can be created by reinserting the operation at a different position on this machine is evaluated. The operation is then returned into the schedule at the place with the minimum schedule score. This continues until no improvement has been found for an entire cycle through all the operations. This local optimum is then passed to the Branch and Bound routine as its first incumbent solution.

10.4 Branching Strategies

The branching strategy determines the branching order of the variables. The order in which the variables are branched on can have a very large impact on the overall computational cost. It is important to use a branching strategy which quickly finds a close-to-optimal solution, to facilitate faster pruning. Various existing branching strategies can be found in Achterberg et al. (2005) and Morrison et al. (2016), some of which are summarised next, in Section 10.4.1. Recently, the development of better branching strategies was identified as an open research direction by Morrison et al. (2016). A

novel branching strategy is presented in Section 10.4.2.

10.4.1 Existing Branching Strategies

The aim of *Strong Branching* is to quickly increase the score of a partial solution (Klabjan et al. (2001)). It does so by branching on the variable with the largest score on its lowest scoring branch, i.e., it selects the variable with the largest guaranteed increase of the lower bound. While Strong Branching reduces the overall number of branching operations required, it is computationally expensive to evaluate the minimum increase in penalty score for each variable at each branching point (Achterberg et al. (2005)). In our algorithm, Strong Branching is performed whenever the solution stack contains just a single partial solution. At any other time, variables are branched on in the order determined by their rank, as described in Section 10.4.2.

For integer optimisation problems in which it is possible to solve the linear relaxation of the problem, a common approach is to select the variable which is most fractional in the relaxed solution. The intuition here is that forcing the most fractional variable to be integer will have the greatest effect on the other remaining variables in the problem. According to results in Achterberg et al. (2005), there is no evidence that this *most infeasible first* rule performs better than selecting variables at random.

Pseudo-cost branching, first introduced by Benichou et al. (1971), learns about the effects of each variable during the branching process. This in-

formation is then used to estimate which variables have the greatest effect on the objective function. One weakness of pseudo-cost branching is that in the early stages there is either no or very little information available for each variable. Hybrid methods between strong branching and pseudo-cost branching try to overcome this by employing strong branching initially, and then pseudo-cost branching once enough information is available. This is the idea behind *reliability branching*, introduced in Achterberg et al. (2005), in which strong branching is applied to each variable initially. Once enough information is available for a variable, pseudo-cost branching is used instead.

More recently, Khalil et al. (2016) present a general purpose method for ranking the variables during the branching process. Their method learns which variables in the current problem are most effective at closing the gap between the current bound and the best known solution. The variable with the current highest rank can then be selected at each branching operation.

10.4.2 Novel Branching Strategy: Variable Ranking

The branching strategies described previously must constantly update the estimated effect of branching on each variable, which carries a computational cost. We have developed a novel branching strategy that uses problem instance specific information to create a predetermined branching order. Since our solution method is tailored to a specific problem, we can gain an advantage by using knowledge about our problem to rank the

variables, in advance of the branching process. We have investigated two variable ranking methods, using both job and machine related information. Initially, for both methods, job and machine ranks are calculated as follows:

- **Job Ranking:** Jobs are ranked based on slack. Slack is the non-negative difference between the available processing time (the length of time between the job release date and its soft due date) and the required processing time (the sum of the processing times of all operations of the job). Slack is a non-negative quantity that equals zero whenever the required processing time is greater than the available time. More formally, let d_j denote the soft due date of job j, and t_{req} the total processing time required by job j. The slack of job j is then defined as follows:

$$\text{slack}_j = \max\{0, d_j - s_i - t_{\text{req}}\}$$

  The motivation for ranking by slack is that jobs with less slack are at greater risk of being scheduled in such a way as to miss their due dates. Decision variables associated with such jobs should therefore be considered to be more critical. The job with the least slack is given a rank of 1, the job with the next least slack is given a rank of 2, and so on. Jobs with equal slack values share the same rank.

- **Machine Ranking:** Machines are ranked based on the total demand placed on them. The motivation behind this is that decision variables associated with machines with high demand have less flexibility. In

the context of a partial schedule, the ordering of operations on a busy machine is more likely to have an effect on job completion times than the ordering of operations on a less busy machine. Demand is measured as the total processing time required on a machine. A rank of value 1 is given to the machine with the highest demand, a rank of value 2 is given to the machine with the second highest demand, and so on. Machines with equal demand share the same rank.

Each decision variable is then assigned a rank, by combining the machine and job ranks associated with it. Let r denote the variable rank, let r_m denote the machine rank, and r_j the job rank. Recall that each decision variable determines which of two alternative machine arcs is included in the solution. A machine arc imposes an ordering on two nodes of different jobs. This means that each variable has two different jobs associated with it, and hence two job ranks, $r_j^{(1)}$ and $r_j^{(2)}$, say. Its combined job rank r_j is then calculated in one of three ways:

$$r_j = \frac{r_j^{(1)} + r_j^{(2)}}{2} \ , \tag{10.4.1}$$

$$r_j = \min\left(r_j^{(1)}, r_j^{(2)}\right) \ , \tag{10.4.2}$$

$$r_j = \max\left(r_j^{(1)}, r_j^{(2)}\right) \ . \tag{10.4.3}$$

The variable's machine rank r_m is simply the rank of the associated machine. Before the job and machine ranks are combined, both are normalised. Let R_j and R_m denote the largest job and machine rank, respectively. The

job ranks are then normalised as

$$r_j \leftarrow 1 - \frac{r_j - 1}{R_j} \ , \tag{10.4.4}$$

and the machine ranks are normalised as

$$r_m \leftarrow 1 - \frac{r_m - 1}{R_m} \ . \tag{10.4.5}$$

All job and machine ranks now lie within the range $(0, 1]$. Note that 0 is excluded from this, to allow scaling, and that 1 is now the highest rank. Let $b \in [0, 1]$ be a balancing parameter. The job and machine ranks are then weighted by the balancing parameter:

$$r_j \leftarrow (1 - b) \times r_j \ , \tag{10.4.6}$$

$$r_m \leftarrow b \times r_m \ . \tag{10.4.7}$$

Finally, each variable has its weighted job and machine ranks transformed into a single variable rank r, according to one of the two following methods:

- **Weighted Sum:** The variable rank r is given by

$$r = r_m + r_j \ . \tag{10.4.8}$$

  The effect of summing the two ranks is illustrated in Figure 10.4.1, for the case where there are five ranks of each type, and $b = 0.5$. The highest rank is in the top-right corner, and the combined rank decreases with each diagonal towards the bottom-left.

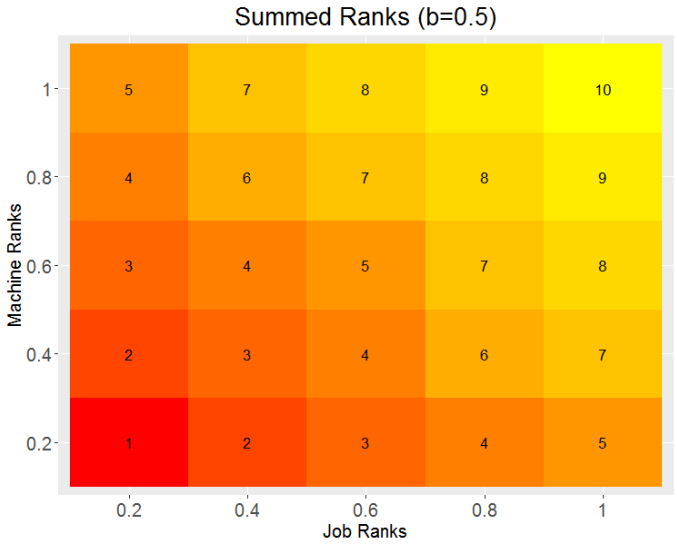

Figure 10.4.1: This heat-map illustrates the effect of combining job and machine ranks through the weighted sum method, for balance parameter $b = 0.5$.

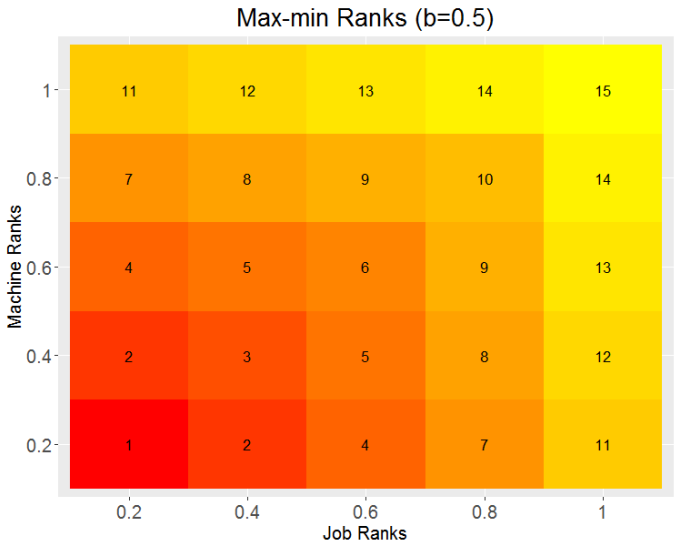

Figure 10.4.2: This heat-map illustrates the effect of combining job and machine ranks through the maximum-primary, minimum-secondary method, for balance parameter $b = 0.5$.

- **Maximum primary, minimum secondary:** Variables are assigned a primary rank, which is the maximum of their weighted job and machine ranks, and a secondary rank, which is the minimum of their weighted job and machine ranks, as follows:

$$r_{\text{primary}} = \max\left(r_m, r_j\right) \ , \tag{10.4.9}$$

$$r_{\text{secondary}} = \min\left(r_m, r_j\right) \ . \tag{10.4.10}$$

  Variables are then ordered by r_{primary}, with ties broken by $r_{\text{secondary}}$. The effect of this approach on the final rank is illustrated in Figure 10.4.2, for the case where there are five ranks of both types, and $b = 0.5$. The highest rank is still in the top-right, but now ranks first decrease along both the horizontal and vertical line, and then according to an inverted L-shape towards the bottom-left.

When $b = 0$, the final rank is only based on job ranks. When $b = 1$ (see Figure 10.4.3), the final ranks are only based on machine ranks. In these two special cases, the weighted-sum and max-min methods will return identical rankings.

10.5 Immediate Selection

Early experimental versions of our Branch and Bound algorithm employed a *depth-first* search strategy. During depth-first search, the algorithm starts with a single partial solution, selects a variable to branch on, and returns

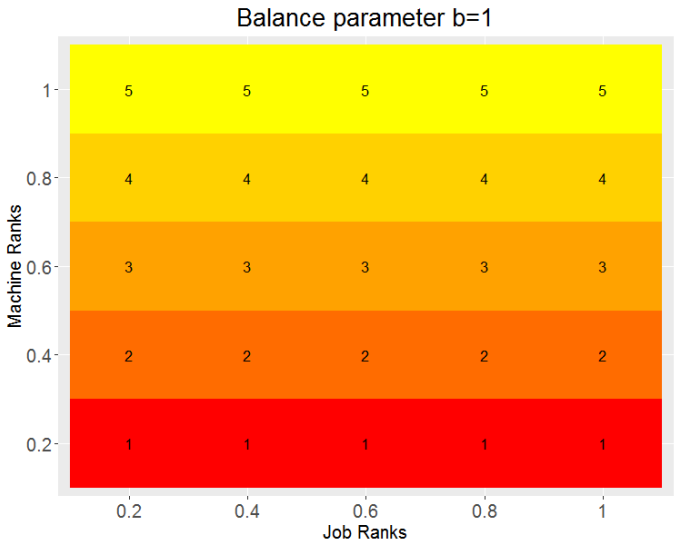

Figure 10.4.3: When $b = 1$, the weighted-sum and max-min ranking methods return identical final rankings.

the new partial solutions to the top of the solution stack. It then takes the next partial solution to branch on from the top of the solution stack, which will be one of the solutions that was just returned to the stack. Infeasible branches are discarded. This implies that, whenever a branch returns no feasible solutions, the algorithm back-tracks up the search tree until a node with open branches is encountered, and continues from there. A drawback of this is that *thrashing* may occur (Kumar (1992)). Thrashing is the repeated encountering of infeasibility due to the same reason. For example, consider that the second and third branching variables, x_2 and x_3 say, have been assigned particular values, which result in another variable further down the tree, x_{103} say, always leading to infeasibility, no matter what value is assigned to x_{103}. Infeasibility will be detected any time x_{103}

is reached, but the algorithm won't be able to associate this with the two branching variables near the top of the tree. The algorithm will simply backtrack, and continue its decent from another branch. If the backtrack did not go high enough, i.e. back to x_3, infeasibility will be detected again and again each time x_{103} is reached. Thrashing can be very computationally expensive. One of the methods we employ to reduce this cost is known as *Immediate Selection*.

The Immediate Selection process is used to reduce the number of remaining unfixed variables. It fixes variables outside of the branching process. The Immediate Selection of disjunctive arcs was introduced by Carlier and Pinson (1989). They identified pairs of nodes which can only be ordered in one way in the optimal solution. A number of different implementations of Immediate Selection have been proposed in the literature (Brucker et al., 1994a). In the context of the blocking job shop, Immediate Selection has been employed by Mascis and Pacciarelli (2002). We employ Immediate Selection in a similar way, as described below.

Let $\mathcal{I}$ denote the set of alternative arcs which have not yet been included in the partial schedule. For each pair of alternative arcs $(i,j),(k,l) \in \mathcal{I}$, each arc is tested in turn. If the inclusion of arc (i,j) results in a solution no better than the current upper bound, arc (k,l) is added to the solution, and vice versa. If both arcs cause the partial schedule score to increase to at least the upper bound, then the partial schedule is discarded, as it cannot be the basis of an improved upper bound.

10.5.1 Constant Immediate Selection

Another way to reduce instances of thrashing and speed up the search process is what we call *Constant Immediate Selection*. Consider the binary variables x and y. When $x = 0$, y can take any value. However, when $x = 1$, the only feasible value for y is 1. Fixing $y = 1$ may in turn impose restrictions on other variables. To take advantage of this, at the start of the algorithm a list is created for each variable x_i, listing all other variables that can be fixed if a particular value is assigned to x_i. To construct the Constant Immediate Selection list, a directed acyclic graph representing a partial solution is created. This graph contains all job arcs, all job precedence constraint arcs, but no machine arcs. Any combination of two machine arcs which would create a circle in this graph is recorded, and their feasible alternatives are added to the Constant Immediate Selection list. This process is quick, as it does not involve any schedule evaluations. Each time a variable is fixed during the Branch and Bound process, any unfixed variables on its Constant Immediate Selection list are also fixed. This is a recursive procedure, so that Constant Immediate Selection is performed for any variables fixed by Constant Immediate Selection. It may be that infeasibility is detected during this process. In that case, the partial solution is discarded.

A similar procedure, referred to as *static implications*, is used by D'Ariano et al. (2007), and later by Pranzo and Pacciarelli (2016).

10.6 Search Strategies

The search strategy of a Branch and Bound algorithm determines the way in which the branching tree is explored. More specifically, it governs the order in which the nodes are selected for branching. A summary of search strategies is provided in Khalil et al. (2016). The following sections describe several search strategies. A novel search strategy is then proposed in Section 10.6.5.

10.6.1 Breadth-first Search

A well known search strategy is breadth-first search, in which all nodes at the current level of the tree are branched on before descending to the next level of the tree. This has a large storage requirement, which makes it unsuitable for many problems. Another drawback is that when upper bounds are based on complete solutions, as is the case for our scheduling problem, this bound cannot be updated until the entire tree has been explored (Morrison et al. (2016)). For these reasons, breadth-first search is not a suitable search strategy for our scheduling problem.

10.6.2 Depth-first Search

During depth-first search, all partial solutions, i.e. the open branches, are stored on a single stack. Initially, the stack contains a single solution with no fixed variables. Whenever branching takes place, a solution is taken

from the top of the stack. This solution is branched, and the resulting partial solutions are returned to the top of the stack. This means the next branching operation takes place on one of the most recently created leaves of the tree, which causes the algorithm to move further down the tree. When a dead-end is reached, due to a solution being complete or infeasible, the algorithm back-tracks up the tree until the first open branch is encountered. This open branch is then explored further, according to the same rules. One advantage of depth-first search is that the number of open branches is relatively small compared to other search strategies, so that it has a relatively small memory requirement. Another advantage is that complete solutions are obtained quicker, with the resulting tighter upper bounds assisting the pruning process.

10.6.3 Best-first Search

The best-first search selects the next leaf to branch on as the leaf representing the most promising partial solution. In the context of our scheduling problem, this would be the problem with the minimum lower bound. The advantage of this strategy is that no time is spent exploring regions which are worse than the optimal solution. The drawback of this strategy is that it leads to many open branches, and therefore has a large memory requirement.

10.6.4 Cyclic Best-first Search

The cyclic best-first search is described in Kao et al. (2008), where it is called distributed best first search. It is a hybrid between depth-first search and best first search. This search strategy cycles through all the levels of the search tree. At each level, the best node is expanded, and its children are stored at the next level. At most one node is expanded at each level during each cycle.

10.6.5 Novel Search Strategy: Stack Reordering

Morrison et al. (2016) identify three potential research directions for Branch and Bound algorithms. The first research direction they identify asks: "How can the search strategy be tailored for different problems?". This section presents a novel search strategy developed for our problem. During initial development of the Branch and Bound algorithm, a depth-first search strategy was used. One restriction of depth-first search is that some promising regions of the solution space may not be explored for some time. It is therefore of interest to consider how to better explore the search space, while retaining the small memory requirements of depth-first search. Some experimentation indicated that it could be beneficial to occasionally reorder the solution stack. This can lead to additional promising regions of the search tree being investigated sooner, whilst maintaining some of the desirable properties of the depth-first approach. The following four

reordering policies were tested:

1. **Completeness:** Most complete solution on top of the stack.

2. **Score:** Smallest score on top of the stack.

3. **Random:** Random ordering of the stack.

4. **Gradient**: Smallest gradient on top of the stack, as calculated next.

In its basic form, we define the gradient to be the total tardiness score divided by the partial solution completeness. The completeness is the number of fixed variables in a partial solution. Therefore,

$$\text{gradient} = \frac{\text{score}}{\text{completeness}} \ . \tag{10.6.1}$$

The gradient is a measure of how much the score has increased, on average, with each variable that has been fixed. The intuition is that this could be somewhat representative of average score increases for the remaining variables, thus making solutions with smaller gradients more attractive. Consider the following two examples as an illustration of how this works:

1. There are two solutions with equal score. The first solution has 20% of its variables fixed, and the other has 80% of its variables fixed. The more complete solution appears more promising, as the less complete solution is likely to have increased in score by the time it reaches 80% completion. Therefore, the more complete solution will be given priority by the gradient sort.

2. There are two solutions of equal completeness, but with different scores. In that case, the solution with the lower score appears more promising and will be given priority by the gradient sort.

The gradient defined in Equation 10.6.1 is based on an implied origin at $[0, 0]$. However, it would be more sensible to use the most recent common predecessor of all remaining partial solutions as the origin. We therefore define the most common predecessor solution as the origin, and calculate the gradient as

$$\text{gradient} = \frac{\text{score} - \{\text{common origin score}\}}{\text{completeness} - \{\text{common origin completeness}\}} \qquad (10.6.2)$$

In practice, this means the origin is updated whenever only a single solution remains in the stack.

Parameter f determines how often stack reordering is triggered, i.e. after f consecutive branching operations during which no new incumbent solution has been found. When two or more reordering methods are active, the round-robin method is used to determine the type of reordering to employ next. Partial solutions are still taken from, and returned to, the stack according to the (quasi) depth-first search strategy. The sensitivity analysis in Section 10.9 shows that, in the context of our test problems, our novel search strategy based on stack reordering outperforms depth-first search.

10.7 Branch and Bound Complexity

It was shown that the depth of the Branch and Bound tree increases polynomially with the number of operations (Section 4.2.1). As such, any Branch and Bound algorithm will at best have polynomial complexity, even if an algorithm could be found which has a linear complexity in relation to tree depth. See Section 4.4 for a discussion on the complexity of the Branch and Bound algorithm for the non-blocking case. The same arguments apply here. The computational experiments on different problem sizes presented in Section 11.4 provide additional insights on how the algorithm performs on both smaller and larger problem instances.

10.8 Parameter Tuning

The Branch and Bound method presented in this chapter has a number of parameters that need to be set before running the algorithm. These values should be carefully chosen, as they have a strong influence on the efficiency of the algorithm. With a good set of parameter values, the algorithm will often only require a few seconds to complete, whereas with the wrong set of parameter values the algorithm might never finish.

The Branch and Bound parameters, including their type and permitted range, are shown in Table 10.8.1. As some of the parameters are continuous, or integers with infinite range, there are infinitely many permitted parameter configurations. This means neither trial-and-error nor complete

Par.	Type	Range	Parameter description
b	continuous	$[0, 1]$	Machine and job ranking balance
c	binary	$\{0, 1\}$	Constant Immediate Selection
d	binary	$\{0, 1\}$	Immediate Selection duration
j	categorical	avg, max, min	Job rank combination type
f	integer	$\mathbb{Z}_{>0}$	Solution stack resorting interval
m	integer	$\mathbb{Z}_{>0}$	Immediate Selection interval
r	categorical	sum, max-first-min-second	Rank combination type

Table 10.8.1: The blocking job shop Branch and Bound parameters.

enumeration of the parameter space are viable parameter tuning options.

In order to optimise the performance of the algorithm, the parameters were fine-tuned with SMAC (Section 6.3, Hutter et al. (2011)). The test instances for the blocking job shop are described in Section 11.1. The first 10 of these test instances were used in the parameter optimisation process. As Branch and Bound is an exact method, the aim of the parameter optimisation was to discover a configuration which minimises the running time of the algorithm. The first part of the algorithm, all the way through to the initial local optimisation procedure, is not affected by the choice of parameters. The algorithm running time was therefore measured from the moment of the first branching operation. Some initial experimentation showed the potential of some configurations to achieve average running times of less than 10 seconds. With this in mind, a hard time limit of 300 seconds was

imposed on individual algorithm runs. This is the same time limit that was given to other methods for parameter optimisation, and therefore provides a fair comparison. If after 300 seconds the algorithm had not terminated, the result was reported as a time-out, and a penalty score of 3000 (i.e. ten times the cut-off limit) was returned.

The results obtained with SMAC show that the parameter values recommended in Table 10.8.2 perform very well. Many similar configurations within the recommended ranges also perform well.

Parameter	Recommended Value	Recommended Range
b	0.99	0.99
c	1	$\{1\}$
d	1	$\{1\}$
j	any	any
f	147	$[100, 500]$
m	6	$\{3, 4, \ldots, 11\}$
r	any	any

Table 10.8.2: Recommended values and ranges for the blocking job shop Branch and Bound parameters.

10.9 Sensitivity Analysis

All Branch and Bound parameters were subjected to sensitivity analysis, in order to better understand their effect on the algorithm running time. The sensitivity analysis investigated one parameter at a time, with the remaining parameters fixed at their recommended values (Table 10.8.2). For each parameter in turn, each of the first 10 test instances were evaluated over a range of valid parameter values. Although the algorithm is deterministic, runs were replicated 30 times at each parameter configuration. This was necessary to account for small natural variations in running times.

In the case of continuous or integer parameters, it was necessary to decide over which range of values, and at what increments, to evaluate the algorithm performance. Knowledge from the parameter tuning process, as well as some manual experimentation, was used to set these ranges. This was done in such a way as to keep the tested ranges reasonably large. Individual algorithm runs had a time limit of 300 seconds. Since the parameters only affect the branching process, the running time was measured from the start of the branching process. This excludes the time taken to load the data, perform various pre-processing operations and find the initial solution based on the topological sorting. These pre-processing operations typically required less than one second of time. Any runs still active after 300 seconds returned their best-so-far solution. The sensitivity analysis was performed on a computer cluster with 2.6GHz Intel Ivy Bridge processors, with 4GB

RAM available for each process.

The sensitivity analysis results are reported in the following sections. At each parameter value, the results have been averaged over all runs for all test instances. The 'optimality zone' in the graphs indicates the largest continuous range of the parameter in which all final solutions were optimal.

10.9.1 Rank Balance Parameter b

Sensitivity analysis was performed for the rank balance parameter b, on the range $[0, 1]$ with increments of size 0.01. The final solutions in the range $[0.36, 1]$ were always optimal (Figure 10.9.1). The best performance with respect to running time was observed in the range $[0.91, 0.99]$ (Figure 10.9.2

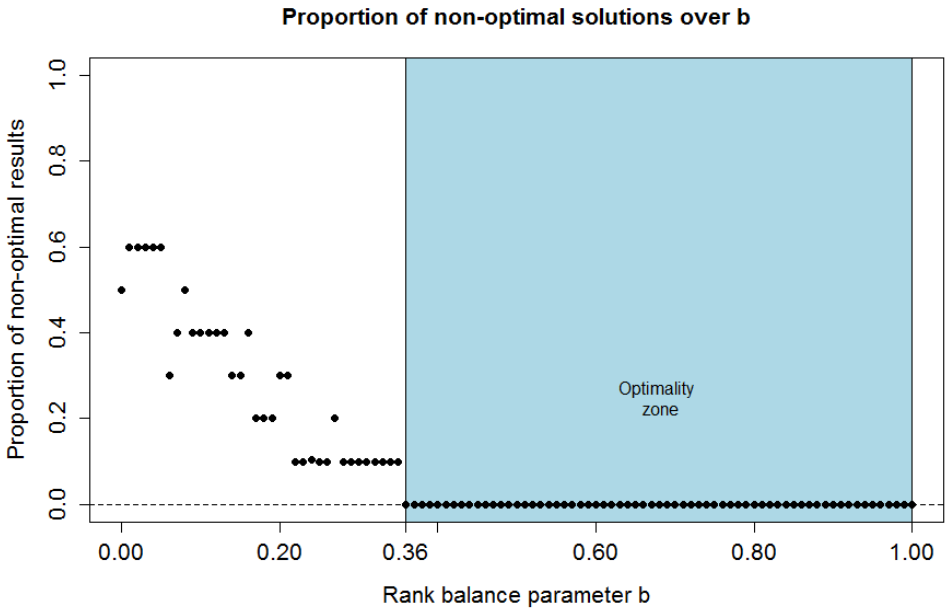

Figure 10.9.1: Proportion of non-optimal results produced by the Branch and Bound algorithm over the range of b.

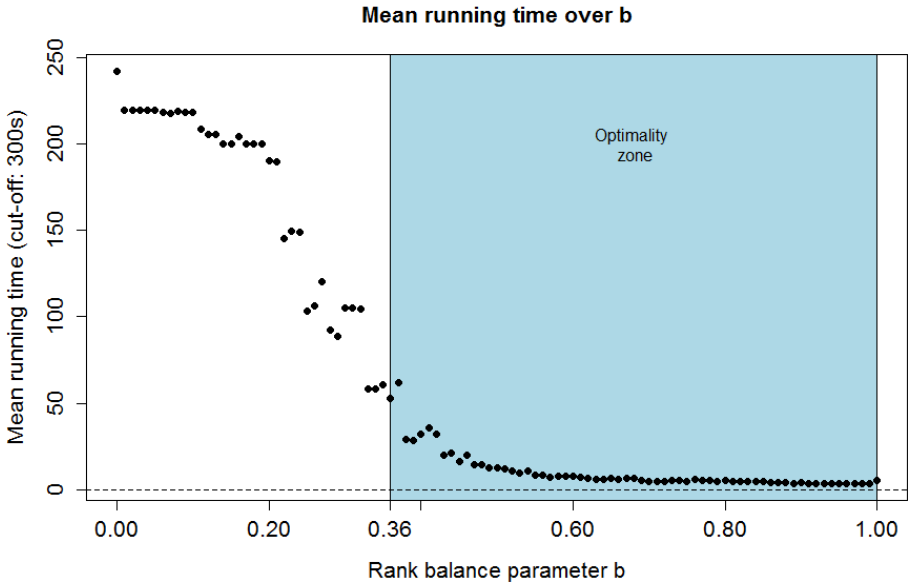

Figure 10.9.2: Mean running time of the Branch and Bound algorithm for the blocking job shop over the range of b.

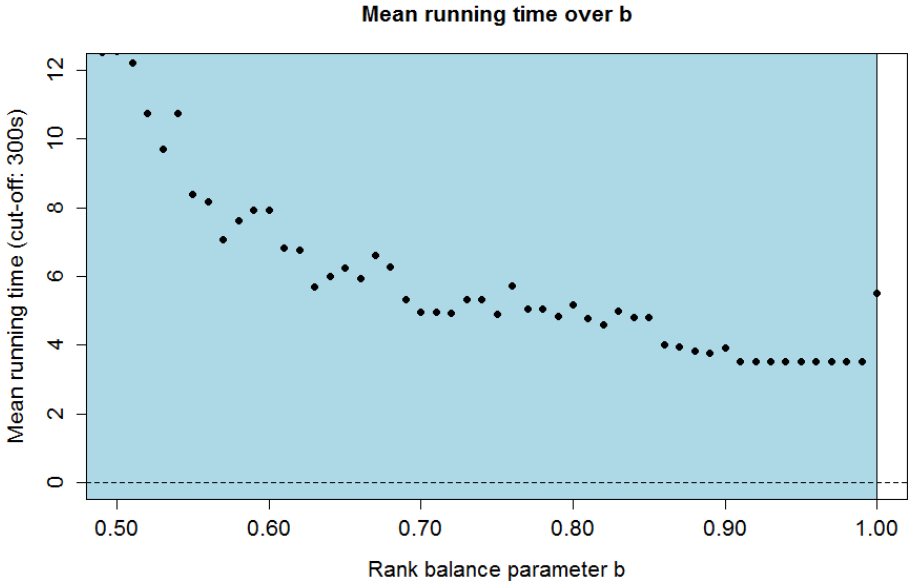

Figure 10.9.3: Mean running time of the Branch and Bound algorithm for the blocking job shop over the upper half of the range of b.

and Figure 10.9.3). Throughout this range, the mean running time was approximately 3.53 seconds. The performance at $b = 1$ is worse, at 5.52 seconds. When $b = 1$, the final rank is completely determined by the machine rank. For values of b just below 1, the balance between job and machine ranks is such that the final rank is based on machine ranks first, with ties broken by job ranks. The range of b in which the final rank is determined by machine ranks first, with ties broken by job ranks, depends on the number of jobs and machines in the problem. For our test problems, this range happens to be $[0.91, 1)$. This is the range in which the best performance was observed. Therefore we recommend that the final rank should be determined by machine rank first, with ties broken by job rank. We have used $b = 0.99$ in our computational experiments.

10.9.2 Immediate Selection Interval Parameter m

Sensitivity analysis was performed for the Immediate Selection interval parameter m, on the range $[1, 100]$ with increments of size 1. The average running time was less than 5 seconds on the range $[3, 11]$ (Figure 10.9.4). The best observed mean running time was 3.53 seconds, at $m = 6$. Therefore we use $m = 6$ in our default configuration. Solutions in the range $[1, 37]$ were always optimal (Figure 10.9.5).

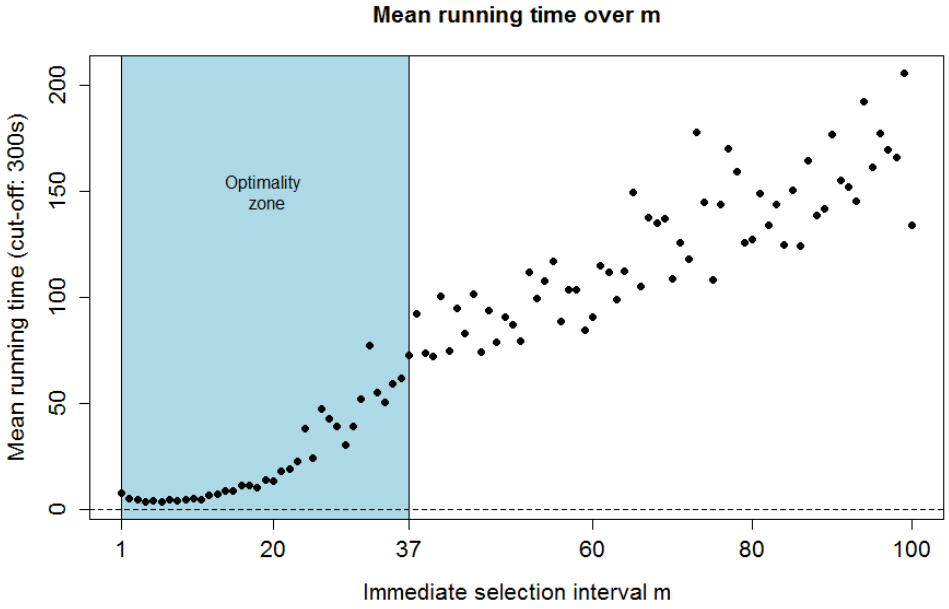

Figure 10.9.4: Mean running time of the Branch and Bound algorithm for the blocking job shop over the range of m.

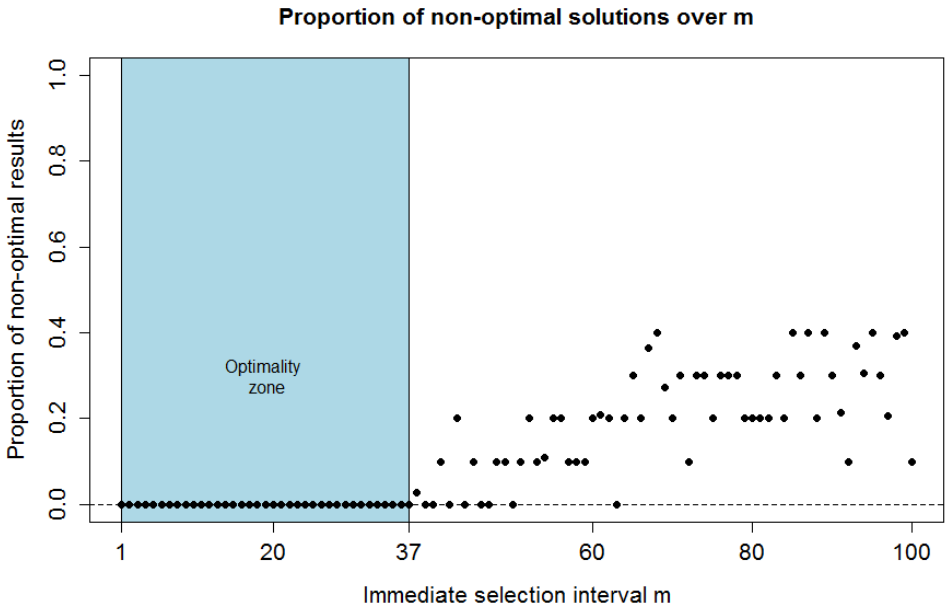

Figure 10.9.5: Proportion of non-optimal results produced by the Branch and Bound algorithm over the range of m.

10.9.3 Stack Reordering Parameter f

Sensitivity analysis was performed for the stack reordering interval parameter f. As the effect of this parameter is best shown when f is plotted on a log scale, the range was set to $[\exp(0), \exp(16)]$, i.e. $[1,\ 8886111]$, with constant increments on the log scale. Good performance with respect to running time (less than 5 seconds on average) was observed in the range $[100,\ 500]$ (Figure 10.9.6). The best observed mean running time was 3.53 seconds, at $f = 147$. Solutions in the range $[20,\ 984609]$ were always optimal (Figure 10.9.7). In fact, almost all solutions for values of f above this range were also optimal. The graph in Figure 10.9.6 flattens out for higher values of f. This plateau represents the region in which each test problem is solved before a single reordering takes place. Reordering was done by score and by gradient, alternately.

For lower values of f, performance worsens due to a number of factors. There is a computational cost associated with reordering the stack. Additionally, between calls to the reordering function, the algorithm performs a depth-first search. This tends to keep the number of solutions in the stack small. If the stack is frequently reordered, the number of solutions in the stack tends to increase, which also affects the computational cost associated with reordering the stack.

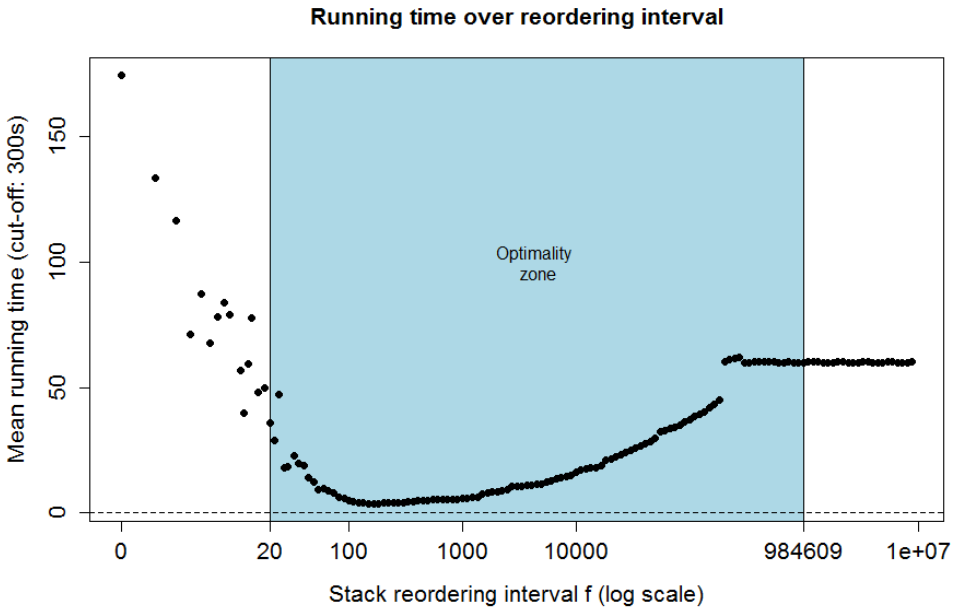

Figure 10.9.6: Mean running time of the Branch and Bound algorithm for the blocking job shop over the range of f.

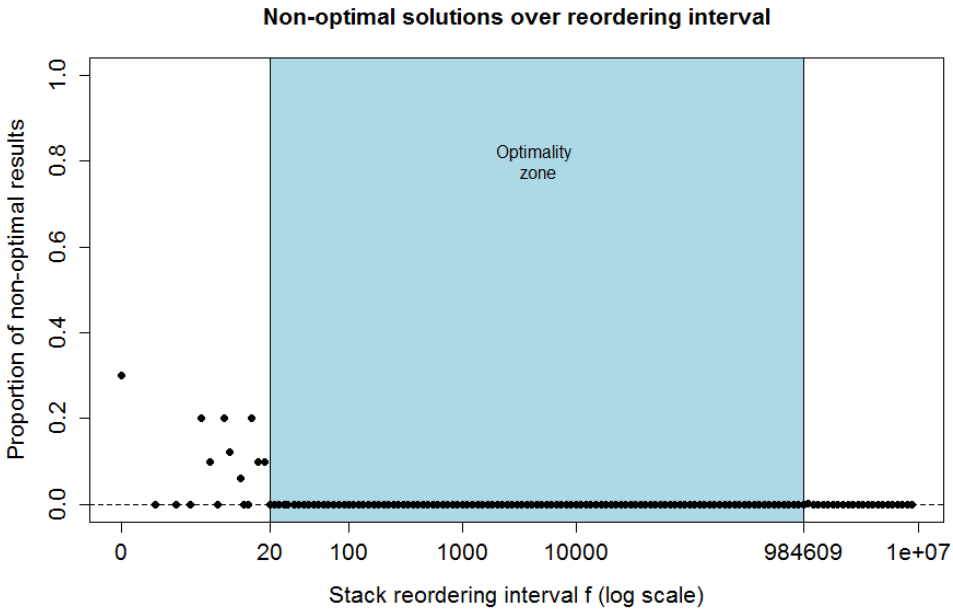

Figure 10.9.7: Proportion of non-optimal results produced by the Branch and Bound algorithm over the range of f.

10.9.4 Constant Immediate Selection Parameter c

Sensitivity analysis was performed for the Constant Immediate Selection parameter c. This binary parameter determines whether Constant Immediate Selection is employed ($c = 1$) or not ($c = 0$). The boxplots in Figure 10.9.8 (left) show some overlap. To assess whether the difference in time performance between the two parameter values across all problem instances is statistically significant, a Wilcoxon rank sum test was employed. A non-parametric test is employed since the results do not follow a normal distribution, as verified by the Shapiro-Wilk normality test. The null hypothesis of the Wilcoxon rank sum test states that the compared samples are independent samples from identical distributions with equal median values, against the alternative hypothesis which states that one of the sam-

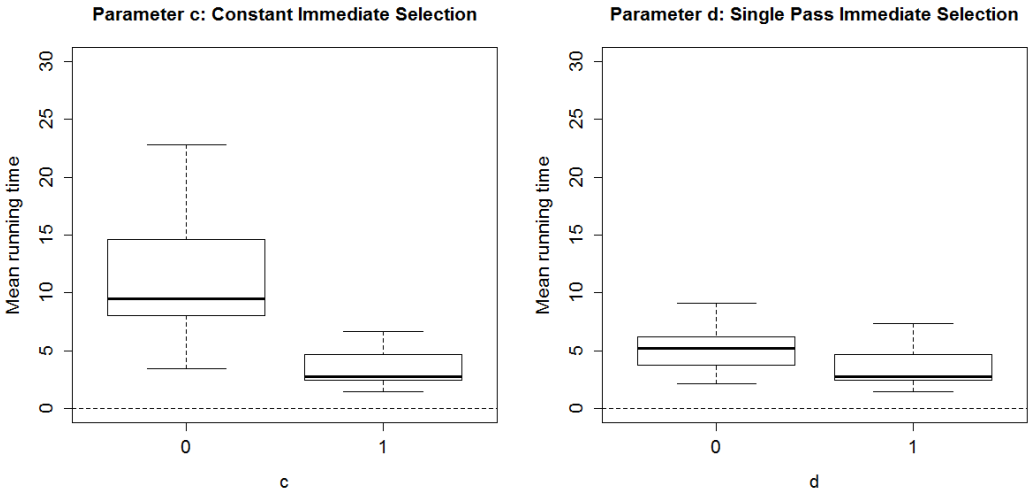

Figure 10.9.8: Spread of the mean B&B running time over binary parameters c (left) and d (right).

ples produces either lower, or higher median performance values. Here, we apply a 5% significance level, $\alpha = 0.05$. The statistical test strongly suggests that there are significant differences in performance between the two parameter values (W = 83730, p-value < 0.0001). With mean running time 3.53 seconds, $c = 1$ generally performs better than $c = 0$. Therefore we use $c = 1$ as our default value. All solutions found with each of the two settings were always optimal.

10.9.5 Immediate Selection Duration Parameter d

Sensitivity analysis was performed for parameter d. This binary parameter determines whether Immediate Selection ends after a single pass over all the variables ($d = 1$), or whether it continues until each variable has been checked once since the last variable was fixed ($d = 0$). The boxplots in Figure 10.9.8 (right) show quite some overlap, with $d = 1$ performing slightly better on average. To assess whether the difference in time performance between the two parameter values across all problem instances is statistically significant, a Wilcoxon rank sum test was employed. A non-parametric test is employed since the results do not follow a normal distribution, as verified by the Shapiro-Wilk normality test. The null hypothesis of the Wilcoxon rank sum test states that the compared samples are independent samples from identical distributions with equal median values, against the alternative hypothesis which states that one of the samples produces either lower, or higher median performance values. Here, we apply a 5% significance

level, $\alpha = 0.05$. The statistical test strongly suggests that there are significant differences in performance between the two parameter values (W = 65051, p-value < 0.0001). Hence, we use $d = 1$ as our default value. All solutions found with each of the two settings were always optimal.

10.9.6 Job Rank Combination Parameter j

Parameter j determines whether each variable has its final job rank created by taking the average of the two job ranks, or by taking the minimum or maximum of the two. Visually, there appears to be a lot of overlap in performance (Figure 10.9.9). To assess whether the difference in time per-

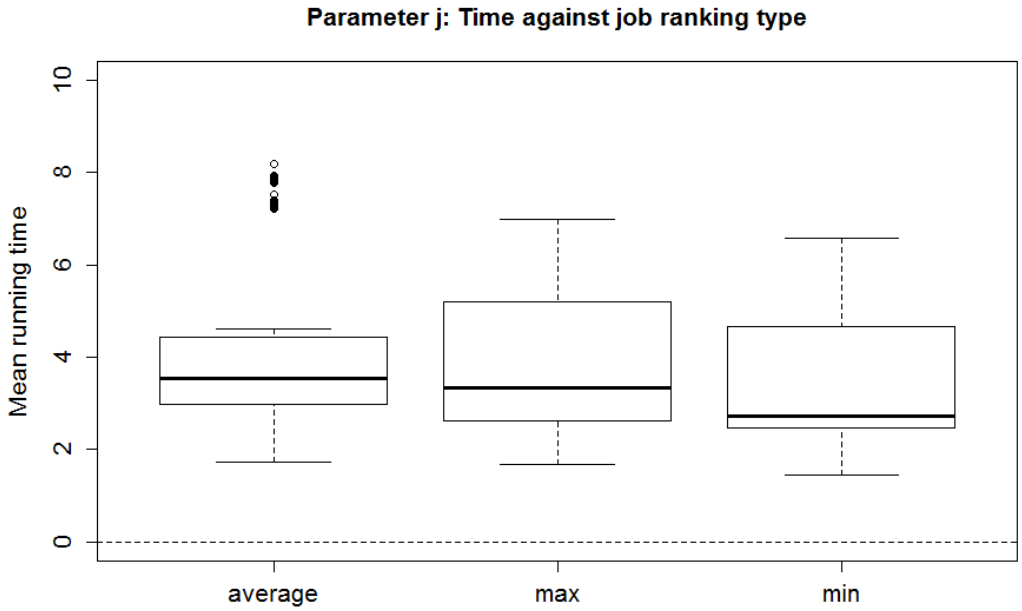

Figure 10.9.9: Spread of the mean running time against job ranking type.

formance between the three parameter values across all problem instances is statistically significant, three Wilcoxon rank sum tests were employed. A Bonferroni correction is applied to the significance level to account for multiple testing. Thus, we apply a 1.666% significance level, $\alpha = 0.0166$. The test results show there is no evidence of a difference in performance between "average" and "max" (p=0.1024). However, "average" performs significantly different than "min" (p<0.0001), and there is also a significant difference in performance between "max" and "min" (p<0.0001). Hence, we use the minimum of the two job ranks, since it has the smallest median.

10.9.7 Rank Combination Parameter r

Parameter r determines whether the final variable rank is determined by taking the sum of the job and machine ranks, or by taking the maximum-primary, minimum-secondary approach. However, for the recommended value for b (close to 1), parameter r has no effect. This is because for such values of b, the largest job rank is smaller than the interval between machine ranks. As a result, the policies selected by r always result in final ranks determined by the machine first, with ties broken by the job rank.

10.9.8 Stack Reordering Policies

The algorithm was tested with all possible permutations of the four stack reordering policies. The results are shown in Figure 10.9.10 (top). The four-

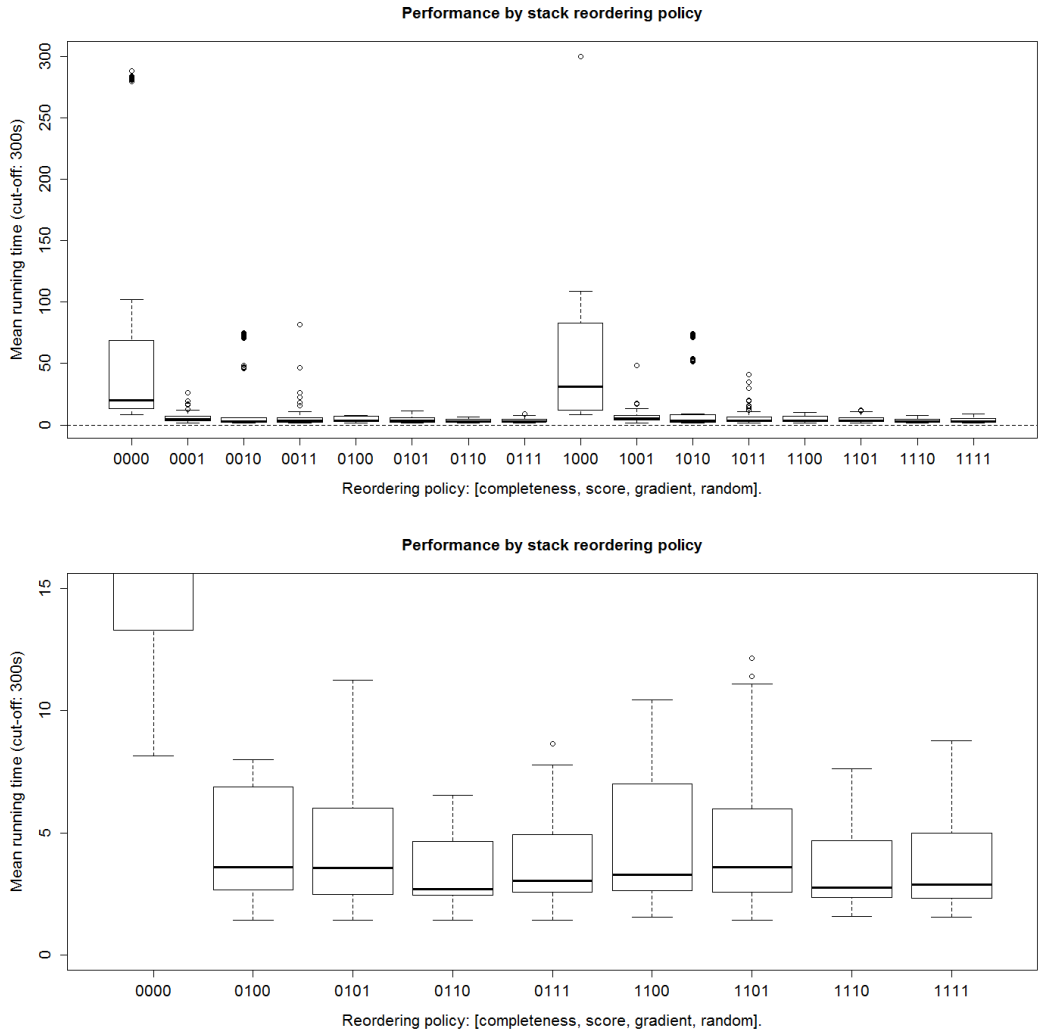

Figure 10.9.10: Top: Mean running time of the Branch and Bound algorithm against all permutations of the four reordering policies: Completeness, score, gradient, and random. The four-digit binary numbers on the x-axis indicate which of the four policies, in the preceding order, are active. **Bottom:** Comparison of all policies with score reordering active, and the policy with no reordering.

digit binary numbers on the x-axis indicate which of the four policies, in the order [completeness, score, gradient, random], are active. If more than one reordering policy is active, the type of reordering is selected in a round-robin fashion. The very first configuration, 0000, represents the case in which no reordering takes place at all. This performs very poorly. Configuration 1000 (completeness only) also performs poorly, in some instances hitting the 300s time limit. More generally, configurations of type *0**, i.e. any in which score reordering is inactive, all have some very visible poorly performing outliers. All configurations of type *1**, i.e. those with score reordering activated, are shown in Figure 10.9.10 (bottom). Policy 0000 (no reordering at all) is also shown again. Note that even the best performance of policy 0000 is worse than the worst performance of three other configurations. This demonstrates the benefit of employing a reordering policy. All policies of type *1** can be seen to perform well. The simplest of these is 0100, in which reordering happens solely by score. The best observed median performance is under policy 0110 (score and gradient).

To assess whether the difference in time performance between stack reordering policies across all problem instances is statistically significant, 120 simultaneous Wilcoxon rank sum tests were employed. A Bonferroni correction is applied to the significance level to account for multiple testing. Thus, we apply a $\frac{5}{120} = 0.0417\%$ significance level. The results are illustrated in Figure 10.9.11. Note that the policies have been ordered by median time performance. At this significance level, there is no clear single

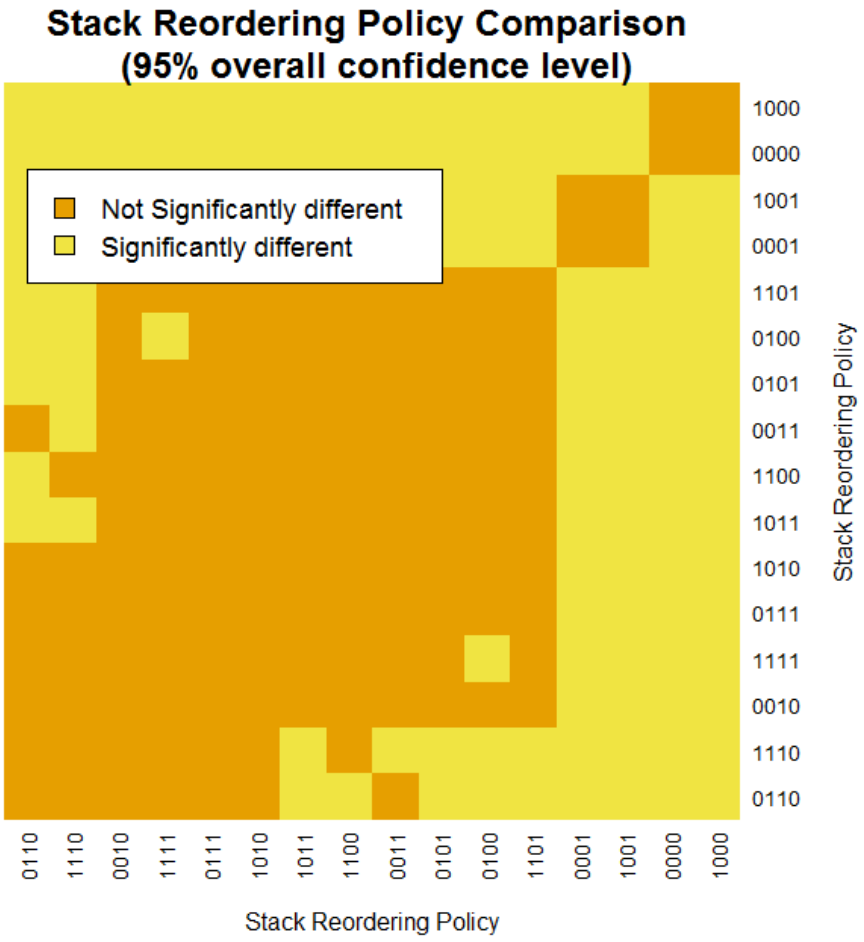

Figure 10.9.11: Results of simultaneous Wilcoxon rank sum tests, with overall significance level of 5%, comparing the running times of all stack reordering policies.

best performing policy. There is some suggestion that the two policies with the best median time performance, 0110 and 1110, outperform the other policies, since each significantly outperforms 9 other policies. Out of these policies, 0110 (score and gradient) is simpler, so we use it as our default policy.

Chapter 11

Blocking Job Shop: Computational Experiments

This chapter presents computational experiments, comparisons and analysis conducted to evaluate the performance of the proposed algorithms for the blocking job shop with flexible maintenance. The problem is described in Chapter 8. The algorithms compared here include the exact Branch and Bound algorithm proposed in Chapter 10, and the various heuristic methods presented in Chapter 9: Ant Colony Optimisation (ACO, Section 9.2), a Rollout method (Section 9.3), the rollout methods proposed by Meloni et al. (2004) (AMCC, SMCP, SMSP), the adapted Simulated Annealing algorithm (SA, Section 9.4), 'instant' solutions based solely on topological sorting (TS, Section 8.8.2), and 'almost instant' solutions based on topological sorting followed by local optimisation (TS+LO, Section 9.3.1). The algorithms have been tested across a wide set of problem instances. These

test problems have been constructed in such a way as to capture the main characteristics of the industrial facility described in Chapter 1.

The computational results show that the exact Branch and Bound method clearly outperforms all tested heuristic methods in terms of the quality of the final solutions found. In terms of running time, only the heuristics based on Topological Sorting are faster than Branch and Bound, however, these are not suitable alternatives due to their poor quality solutions.

The problem instances used for the evaluation of the developed algorithms are introduced in Section 11.1. The experimental design of the computational experiments is discussed in Section 11.2. Computational results for all the methods are presented in Section 11.3. To conclude the experiments, the scalability of the best algorithms across different problem sizes is investigated in Section 11.4.

11.1 Problem Instances

To the best of our knowledge, there are no established test problem instances for a blocking job shop with flexible maintenance activities and both soft and hard due dates. This section provides an overview of the test problems we have constructed. A fuller description of these appears in Appendix A. The test problems are available at `http://dx.doi.org/10.17635/lancaster/researchdata/160`.

Test problems should capture the typical job structure of jobs at the facility described in Chapter 1. Jobs typically start and end at one of the two combined entry/exit points, and must use the transporter to travel between workstations. Every second operation of a job is therefore usually a transporter operation. In order to assess the performance of the developed exact and heuristic optimisation algorithms, 100 such test problems were randomly constructed, subject to a number of constraints designed to include some typical features of the jobs processed at the facility.

The test instances were designed for a facility with 9 workstations (machine IDs 1-9) and one transporter (machine ID 0). Workstations 1 and 9 both serve as entry and exit points. Each test problem consists of 20 regular production jobs, some with merges or splits. There are also 10 maintenance activities, one for each workstation and one for the transporter. The total number of operations in each test instance is 102. This is in the order of magnitude of the sets of work being scheduled at the facility. Time is measured in hours, assuming a 40 hour working week. Some jobs are released at time 0, and some jobs are released at time 40.

11.2 Experimental Design

Due to the stochastic nature of the heuristic algorithms, each of the ACO, Rollout, and SA algorithms were given 30 independent execution runs, on each of the 100 test instances. The B&B, AMCC, SMCP, SMSP, TS and

TS+LO algorithms are deterministic, and hence were only executed once on each test problem. The time limit was set to 300 seconds. If this time limit is reached before the natural conclusion of the algorithm, then the best-so-far solution is returned as the final solution. The total tardiness penalty of this best solution is used to compare the performance of the algorithms.

To highlight the general performance of the developed algorithms, the experimental results are aggregated across all problem instances. Statistics on their general performance are presented, in terms of objective values of the best performing solutions, execution time required to reach the reported result, and statistics on the tardiness characteristics of the scheduled jobs.

All algorithms considered in this study were implemented in C++ and compiled with the GNU g++ compiler under GNU/Linux running on a cluster system equipped with Intel Xeon E5-2650v2 CPU of 2.6 GHz, and 4 GB of RAM per CPU. All experimental results reported here have been obtained under the same computing conditions.

Each algorithm had its parameters set to its recommended values, as presented with the algorithm descriptions in Chapters 9 and 10. Note that AMCC, SMCP, SMSP, TS and TS+LO have no hyperparameters to be tuned.

11.3 Computational Results

Statistics on the performance of the algorithms across all 100 problem instances considered in this work are presented in this section. To enhance the readability of the reported results, the best performing cases are highlighted with **boldface** font.

It should be noted that two of the rollout methods occasionally failed to find a feasible solution. AMCC failed to find a feasible solution for 4 problem instances, while SMSP failed to find a feasible solution for 3 test instances. These seven failed runs provide no usable information in terms of solution score or running time. As such, they are omitted from the analysis of solution scores and running times. The results shown for these two methods should therefore be considered to be optimistic.

Table 11.3.1 reports the average (μ_s), median (m_s) and standard deviation (σ_s) values for the objective value of the best solution found by each algorithm. Recall that the aim is to minimise this penalty score. Branch and Bound clearly outperforms all other methods. It always finds an optimal solution within the time limit, with a median objective value of 5204.27. In terms of the median values, Simulated Annealing, AMCC, and SMSP all roughly rank second. This is followed by ACO, our own Rollout Method, TS+LO, SMCP, and finally TS. Note that the ranking would be different if based on the means. In particular, AMCC and SMSP would be ranked much lower. In terms of mean objective value, both are outper-

Algorithm	μ_s	m_s	σ_s
ACO	4362458.87	4019157.91	2141478.94
B&B	**244971.69**	**5204.27**	474883.45
Rollout	5050913.70	5016386.67	1275299.25
SA	1190007.02	1006727.52	1209283.89
TS	9424264011.02	9014845179.35	637123399.62
TS+LO	23016699.14	13040471.94	100111220.87
AMCC	375746777.70	1007349.00	684697843.14
SMCP	1122572363.08	1002011019.51	1149087612.12
SMSP	320233949.86	1003155.24	715626074.37

Table 11.3.1: The mean (μ_s), median (m_s) and standard deviation values (σ_s) of the objective values.

formed by the simple TS+LO heuristic. The mean values are influenced by the large hard due date penalties, and as such the median provides a more robust measure of performance. The performance of the algorithms is also illustrated in Figure 11.3.1.

Perhaps a more interesting measure than the final score, is the amount by which the final scores exceeds the optimal solution. These values are summarised in Table 11.3.2. This clearly demonstrates the optimality of B&B with respect to schedule quality. These figures are visualised in Figure 11.3.2.

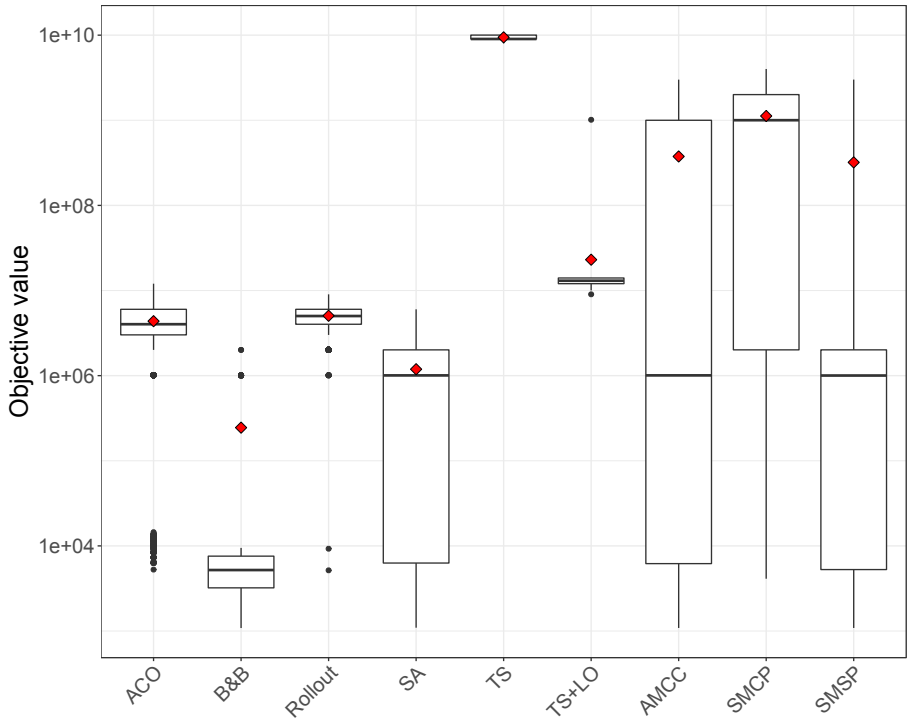

Figure 11.3.1: The objective values for all the algorithms over all the problem instances and runs (y-axis in $\log_{10}$ scale)

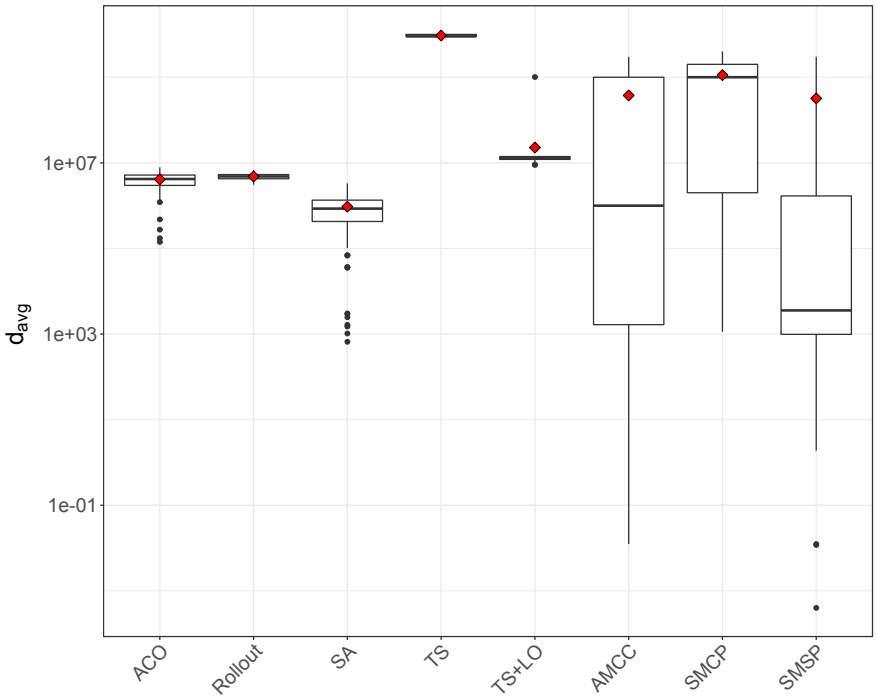

Figure 11.3.2: The difference between the objective values and the optimal solutions, over all the problem instances and runs. Because the y-axis is on a log-scale, B&B is not shown. Note that AMCC and SMSP occasionally located the optimal solution, and the score for this cannot be shown with the log-transform.

Algorithm	μ_d	m_d	σ_d
ACO	4117487.19	4011378.94	2016789.20
B&B	**0.00**	**0.00**	0.00
Rollout	4805942.01	5010151.97	1238444.10
SA	945035.33	1000422.75	1047061.71
TS	9424019039.33	9014840483.68	637137162.31
TS+LO	22771727.45	13031752.37	100035022.05
AMCC	375502191.76	4644.33	684692044.22
SMCP	1122327391.39	1002004622.34	1148980632.23
SMSP	319991852.67	2077.48	715631156.10

Table 11.3.2: Performance in terms of mean (μ_d), median (m_d) and standard deviation values (σ_d) of the distance from the optimal objective values.

Another important measure of performance is the running time required to find the final solution. Each algorithm run was allowed a maximum of 5 minutes running time. The average (μ_t), median (m_t) and standard deviation (σ_t) values for the execution time required to find the best solution, in terms of CPU Wallclock time measured in seconds, are displayed in Table 11.3.3. The running time distributions are also visualised in Figure 11.3.3. B&B only requires a median running time of 5.84 seconds to find the optimal solution. Most other methods are slower, the worst being

Algorithm	μ_t	m_t	σ_t
ACO	199.87	206.02	70.40
B&B	12.50	5.84	25.53
Rollout	159.73	168.41	86.88
SA	274.16	286.42	35.29
TS	**0.00**	**0.00**	0.00
TS+LO	0.11	0.11	0.02
AMCC	19.23	18.76	1.92
SMCP	14.92	14.68	1.64
SMSP	19.09	18.61	1.77

Table 11.3.3: The time (seconds) required to find the best solution, for all the algorithms over all the problem instances and runs.

Simulated Annealing with a median running time of 286 seconds. SMCP has a median running time of 15 seconds, and this increases to 19 seconds for both AMCC and SMSP. This shows that B&B not only performs best in terms of final solution quality, it also requires substantially less running time than most other methods. Only the Topological Sorting methods outperform B&B in terms of time until the best solution was found, requiring only a fraction of a second to run to completion. However, these two methods show appalling performance with regards to solution quality. The 5.8 seconds median running time of B&B makes it a very suitable solution for

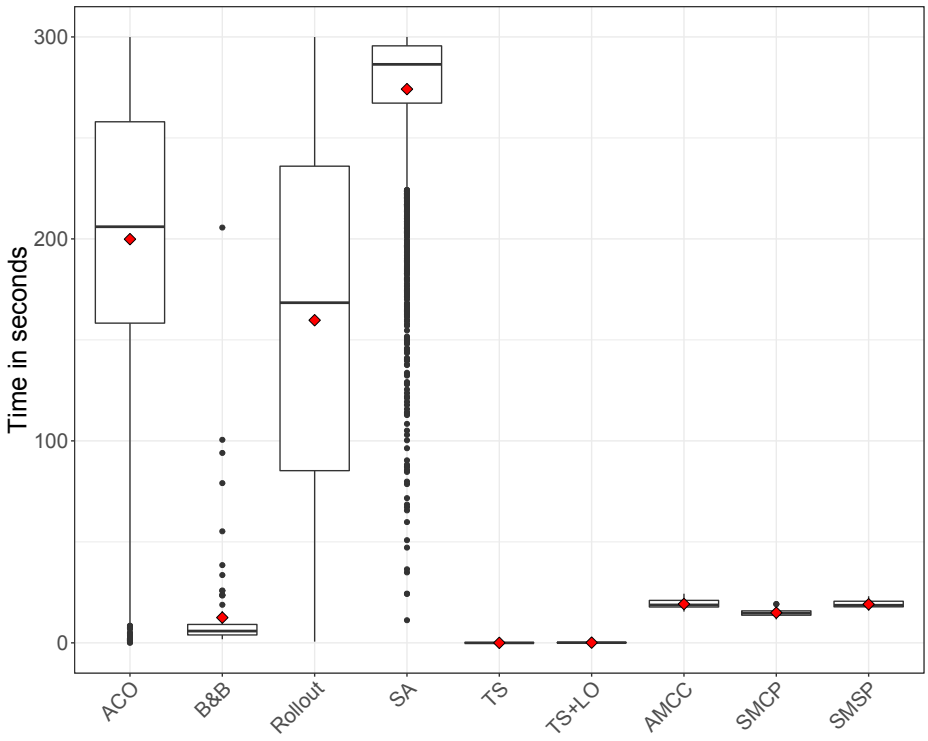

Figure 11.3.3: The time required to find the best solution, over all the problem instances and runs (y-axis in $\log_{10}$ scale).

real-world applications.

The trade-off between running time and solution quality is displayed in Figure 11.3.4. For each algorithm, its respective plotting point represents the median distance from the optimal solution, and the median running time in seconds. It can be seen that B&B dominates all other methods, except TS and TS+LO. However, both TS methods provide very poor quality solutions, so that B&B comes out as the preferred method here.

Recall that the primary objective is to schedule maintenance to start before its hard due date. Almost all algorithms, with the exception of TS, consistently achieve this. Tardiness measures in terms of due date violations will therefore only be presented for regular jobs. To highlight the tardiness of the scheduled jobs, we report the average percentage of jobs that finish on time (μ_o), between the soft and hard due dates (μ_s), and after the hard due dates (μ_h), in Table 11.3.4. The highest percentage of jobs scheduled on time is achieved by B&B, with 76.7%. Since all B&B results were optimal, these are the best percentages that can be achieved with our set of test problems. The lowest percentage of jobs scheduled beyond their hard due dates is also achieved by B&B, at 1.2%. This indicates that in some problem instances, even the optimal solution will contain jobs that exceed their hard due dates. This information can be crucial for decision makers, as early information on potential hard due date violations could be used when accepting new jobs, to protect revenue and reputation. Second place is approximately shared by SA, AMCC, and SMSP, with the

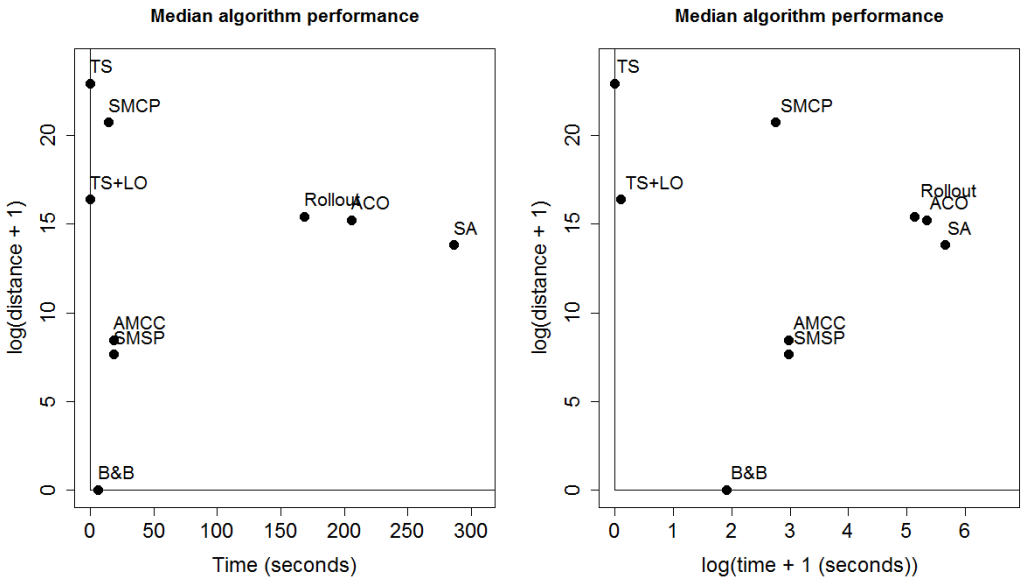

Figure 11.3.4: Median distance from the optimal solution achieved by each algorithm against their respective median running times. Values were increased by 1 before log transforms took place, to deal with log(0) and to map 0 to 0.

Algorithm	μ_o	μ_s	μ_h
ACO	39.34	38.92	21.73
B&B	**76.70**	22.10	**1.20**
Rollout	37.85	36.98	25.17
SA	67.12	26.96	5.91
TS	10.45	20.00	69.55
TS+LO	14.35	20.75	64.90
AMCC	67.86	28.44	3.70
SMCP	57.15	30.05	12.80
SMSP	68.35	28.45	3.20

Table 11.3.4: Average percentages of jobs finishing on time (μ_o), after the soft due date (μ_s), and after the hard due dates (μ_h).

caveat that the latter two occasional fail to find any feasible solution at all. Note that schedules created by TS and TS+LO tend to include a large number of hard due date violations. In fact, the average percentage of jobs scheduled beyond their hard due date is 64.90% for TS+LO, and 69.55% for TS. Detailed information on the number of jobs scheduled on time (or otherwise), across all considered problem instances, is visualised in Figure 11.3.5.

Figure 11.3.6 shows success percentages of the algorithms across all problem instances, with success being defined as whether the optimal solu-

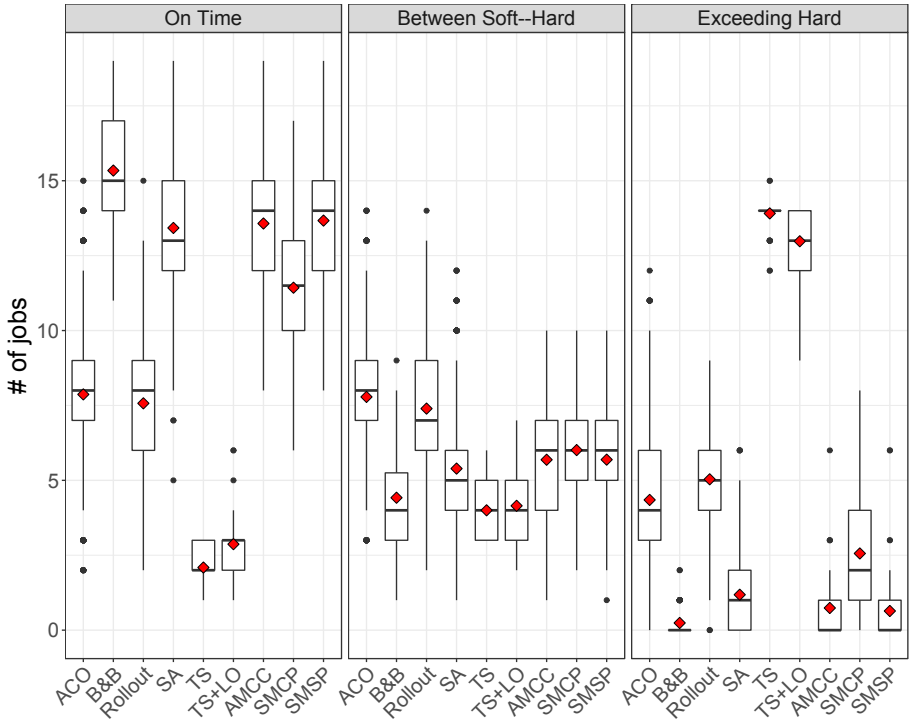

Figure 11.3.5: Distributions of the number of jobs which finish on time, between the soft and hard due dates, and after the hard due dates.

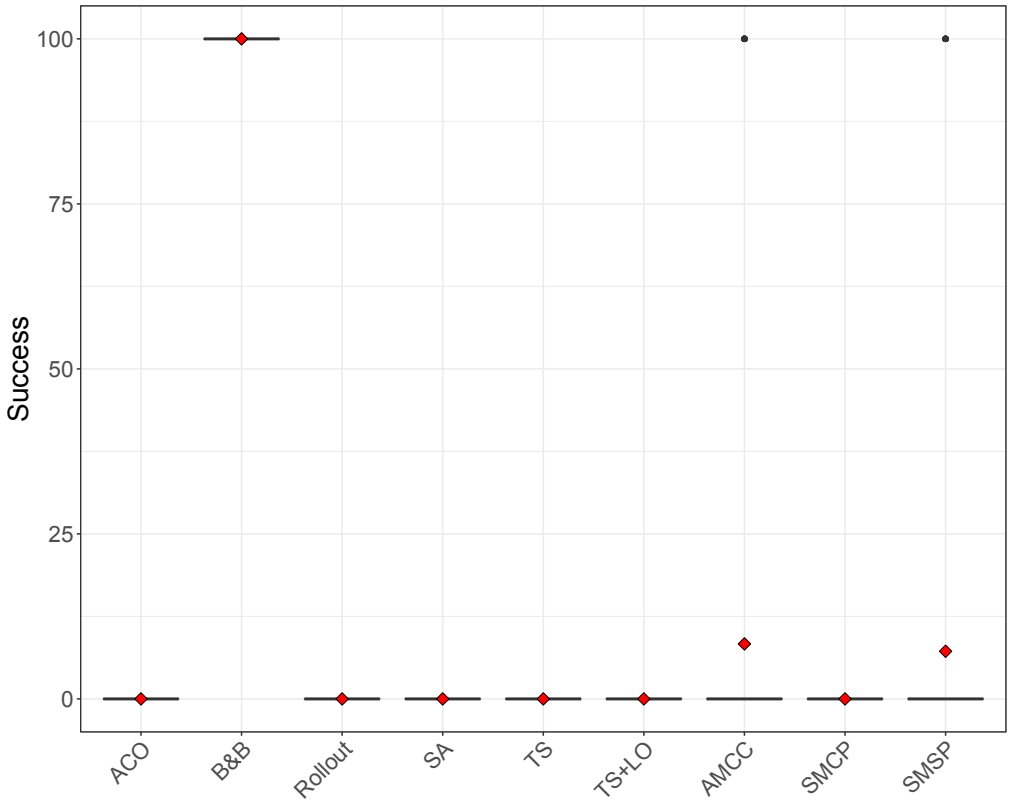

Figure 11.3.6: Box-plots for percentage of times that each algorithm found the optimal solution.

tion was found or not. Note that B&B always found the optimal solution, while most of the other methods never found an optimal solution. Previous results showed that Simulated Annealing occasionally gets close to the optimal solution. AMCC and SMSP have the curious property of being the only other algorithms which occasionally find an optimal solution, whilst also being the only algorithms which occasionally fail to find any feasible solution at all. This again shows the superiority of our novel B&B method over the other algorithms.

To conclude, the B&B algorithm rapidly solves all problem instances

to optimality. B&B outperforms the other algorithms both in terms of solution quality, as well as in terms of running time. The combination of fast running time and good solution quality make this a viable method for application in demanding real-world industrial settings.

11.4 Scalability Analysis

The test problem instances used for the computational experiments contained 102 operations. This is typical for the workload at the facility for which these algorithms were developed. For applications in other areas the workload and number of machines could be larger or smaller. The performance of the best algorithms was therefore also investigated with larger and smaller test problems. The test instances used in this scaling study contain the following number of operations and machines (machine count in brackets): 64 (6), 82 (8), 102 (10), 122 (12), 140 (14), 160 (16), 178 (18), and 204 (20). While there are countless different options for scaling the number of operations and machines, here it was decided to scale the number of machines roughly proportional to the number of operations. For each problem size, 100 different test instances were created. The other properties of the test instances were the same as before, so that jobs can only enter/exit the system at two locations, and must travel on the transporter between locations. Scaling was only investigated for the Branch and Bound, SMSP and AMCC algorithms, since none of the other heuristic

methods showed reasonable performance.

Initial scaling experiments used the B&B hyper-parameter values as reported in Table 10.8.2. However, with these settings it was found that larger problems sometimes ran out of memory. This was due to the solution stack growing very large. The solution stack reordering parameter f was originally optimised on test problems with an average search-tree depth of 1051.6. The largest test problems in this scalability analysis have a search tree with over 4000 levels. If the stack is reordered too frequently, it becomes very difficult for the algorithm to reach the bottom of the tree, and the solution stack grows very large. To address these memory problems, it was therefore decided to scale f relative to the depth d of the search tree. For search trees with an average depth of 1051.6, the optimal value for f was 147. The stack reordering parameter f is therefore scaled with the size of the search tree, as follows:

$$f = \frac{147 \times d}{1051.6} \tag{11.4.1}$$

The two rollout algorithms, SMSP and AMCC, occasionally encounter infeasibility. Both these algorithms are deterministic, so when infeasibility is encountered it can only be recorded that the algorithm failed to find a feasible solution. Table 11.4.1 displays the number of times SMSP and AMCC failed to find a feasible solution, across all problem sizes. There is no obvious relationship between problem size and infeasibility count. As unsuccessful runs provide no information on running time or solution quality,

Problem Size (operations)	64	82	102	122	140	160	178	204
AMCC	3	4	4	1	1	2	5	1
SMSP	2	4	3	1	1	3	2	0

Table 11.4.1: Count of times each algorithm failed to find a feasible solution, out of 100 problem instances for each problem size.

these have had to be omitted from the analysis. Hence, any results relating to AMCC and SMSP should be considered to be somewhat optimistic.

A time limit of one day (86400 seconds) was set for all runs in this scalability analysis. Any runs that reached this time limit were terminated. This means some of the observations have censored running times, since all that is known is that the algorithm would take more than 86400 seconds to find the optimal solution. For B&B, the number of instances confirmed to have been solved to optimality within the time limit, for each problem size, is displayed in Table 11.4.2. It can be seen that as the problem size

Problem Size (operations)	64	82	102	122	140	160	178	204
Solved to optimality (%)	100	100	100	95	89	55	20	0

Table 11.4.2: B&B success rates across problem instance sizes. Success is measured as having found and confirmed an optimal solution within 1 day. 4GB of RAM was available to each run.

increases, the success rate decreases. Since the optimal solutions for many of the larger test problems are not known, it is not known what percentage of SMSP and AMCC runs returned an optimal solution.

The algorithm running times are displayed in Figure 11.4.1, both in regular values (upper plot) and log-transformed values (lower plot). It should be noted that some of the B&B test runs for larger test problems terminated before the time limit, without having confirmed the optimal solution. This is due to the solution stack growing very large and eventually exceeding the available memory (4GB RAM). The total number (and percentage) of early terminations due to memory shortage is displayed in Table 11.4.3. This memory shortage would not be an issue for a pure depth-first search algorithm, where the size of the solution stack is limited by the depth d of the search tree. Here, however, the solution stack is able to grow large due to the stack reordering procedure. The benefit of this reordering procedure was demonstrated in the sensitivity analysis for B&B (Section 10.9.3): It offers a speed increase for the problem sizes that the algorithm was developed for. However, to avoid memory issues with larger test problems,

Problem Size (operations)	64	82	102	122	140	160	178	204
Memory shortage (%)	0	0	0	0	1	0	11	73

Table 11.4.3: B&B test runs which terminated early due to memory shortage.

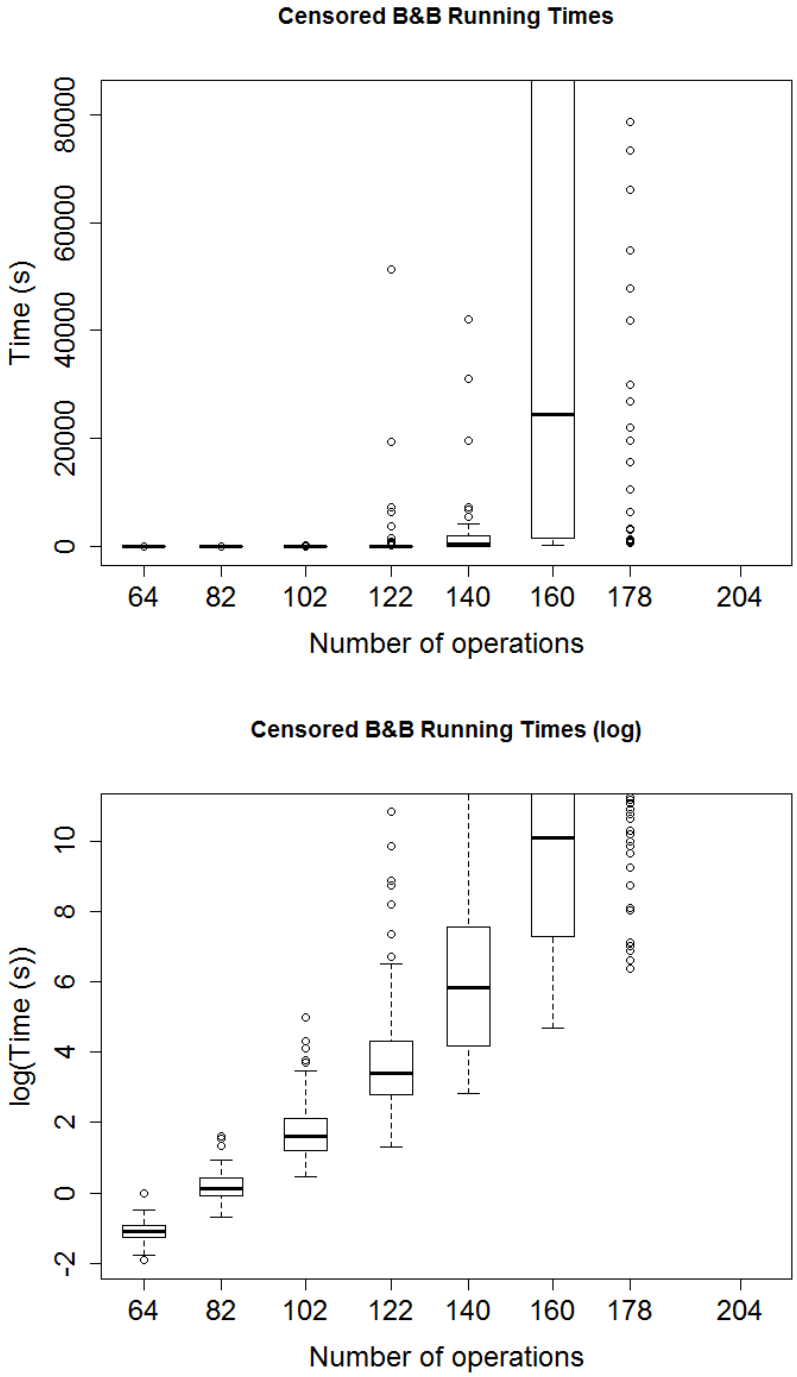

Figure 11.4.1: B&B running times until the optimal solution was confirmed, across varying problem sizes (x-axis to scale). Some data is censored, since runs were terminated after one day (86400 seconds).

it may be better not to use reordering, or to ensure there is a sufficient amount of memory available.

Due to the memory issues encountered by B&B for some of the larger instances, the scaling experiment was repeated with 8GB RAM available to each run. None of these runs terminated due to a shortage in memory. The corresponding success rates are displayed in Table 11.4.4. It can be seen that these are almost identical to the success rates with 4GB RAM, except for one additional successful run for problem size 178. It would appear that runs which previously ran out of memory, still required a lot of additional running time due to the large number of open branches.

Since B&B was unable to confirm all optimal solutions for larger problem sizes, it is not possible to assess how close to optimal the solutions obtained by AMCC and SMSP are. Instead, the median observed penalty scores for the three algorithms across all problem sizes are displayed in Table 11.4.5. Recall that this excludes the cases in which AMCC and SMSP were unable to find a feasible solution, as reported in Table 11.4.1. It can

Problem Size (operations)	64	82	102	122	140	160	178	204
Solved to optimality (%)	100	100	100	95	89	55	21	0

Table 11.4.4: B&B success rates across problem instance sizes. Success is measured as having found and confirmed an optimal solution within 1 day. 8GB of RAM was available to each run.

Problem Size	B&B	AMCC	SMSP
64	**3278**	5176	5171
82	**3216**	6150	6200
102	**5204**	1007504	1004688
122	**5184**	9403	10328
140	**6299**	1010350	1508925
160	**7367**	2011212	3012441
178	**12107**	1000014363	1000013932
204	**1012109**	2000513749	1007021738

Table 11.4.5: Median penalty scores across all problem sizes for Branch and Bound, AMCC, and SMSP.

be seen that, on average, B&B finds the best solutions across all problem sizes.

The running times for AMCC are displayed in Figure 11.4.2, both in regular values (upper plot) and logged values (lower plot). A similar plot for SMSP appears in Figure 11.4.3. Both algorithms show very similar time requirements. For larger problem sizes, AMCC and SMSP require less total running time than B&B, but the latter produces better quality solutions.

It should be noted, though, that B&B usually produces a number of ever-improving solutions, before it eventually locates an optimal solution. Therefore, a better comparison between B&B and the rollout methods can be made by giving B&B the exact same running time as required by AMCC and SMSP. This allows the comparison of solutions which were produced with equal running time. Table 11.4.6 and Table 11.4.7 show such a comparisons. For each problem instance, it compares the quality of the rollout solutions against the B&B incumbent solution at the time that the rollout method terminated. The first column shows the percentage of instances for which B&B had actually already obtained a better solution than that produced by the rollout method. For AMCC this ranges between 71% and 95% across different problem sizes. The results for SMSP are similar, with B&B outperforming SMSP between 71% and 96% of the time. Each table has a second column, containing the median difference in objective score between B&B and AMCC/SMSP, at the time of completion of the respective rollout method. The difference is calculated as B&B minus AMCC/SMSP. It

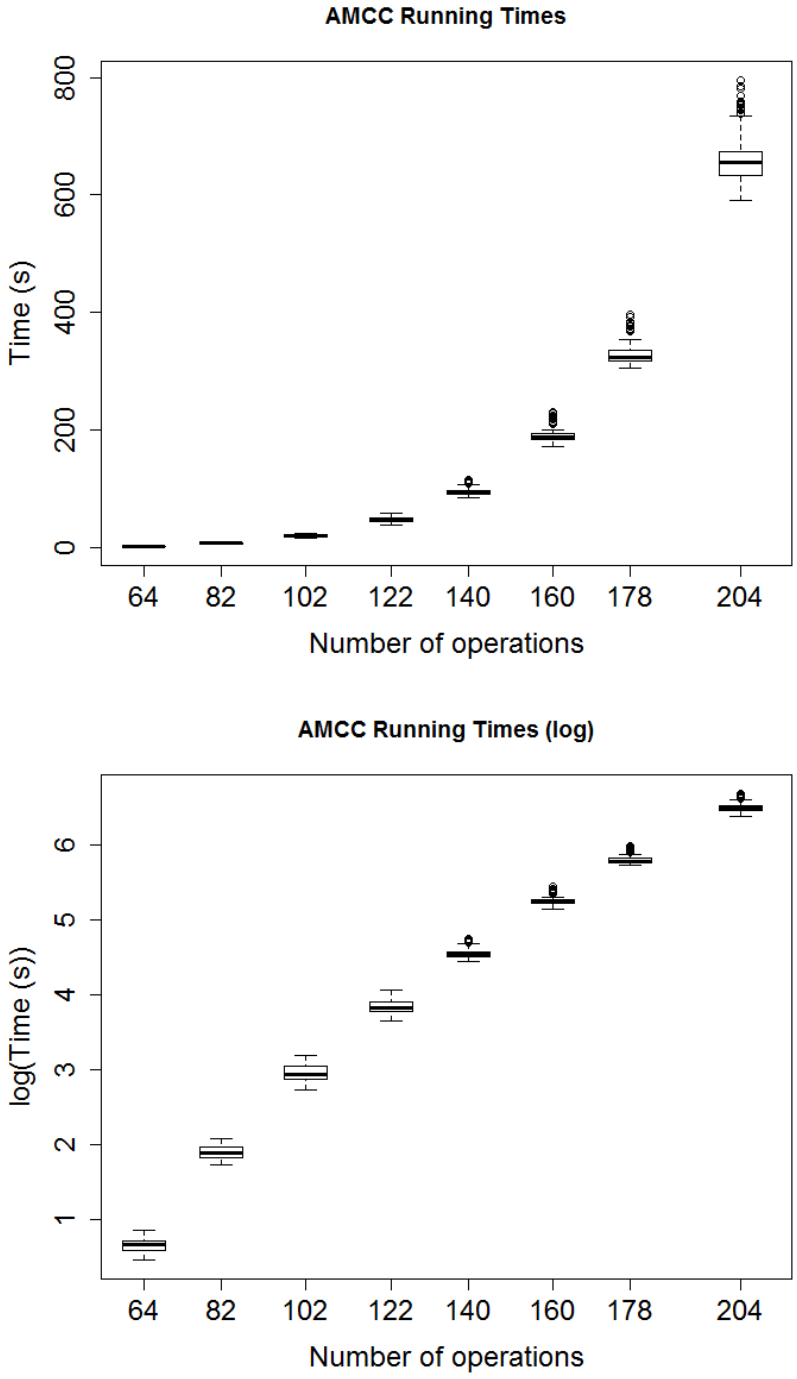

Figure 11.4.2: AMCC running times until the solution was found, across varying problem sizes (x-axis to scale).

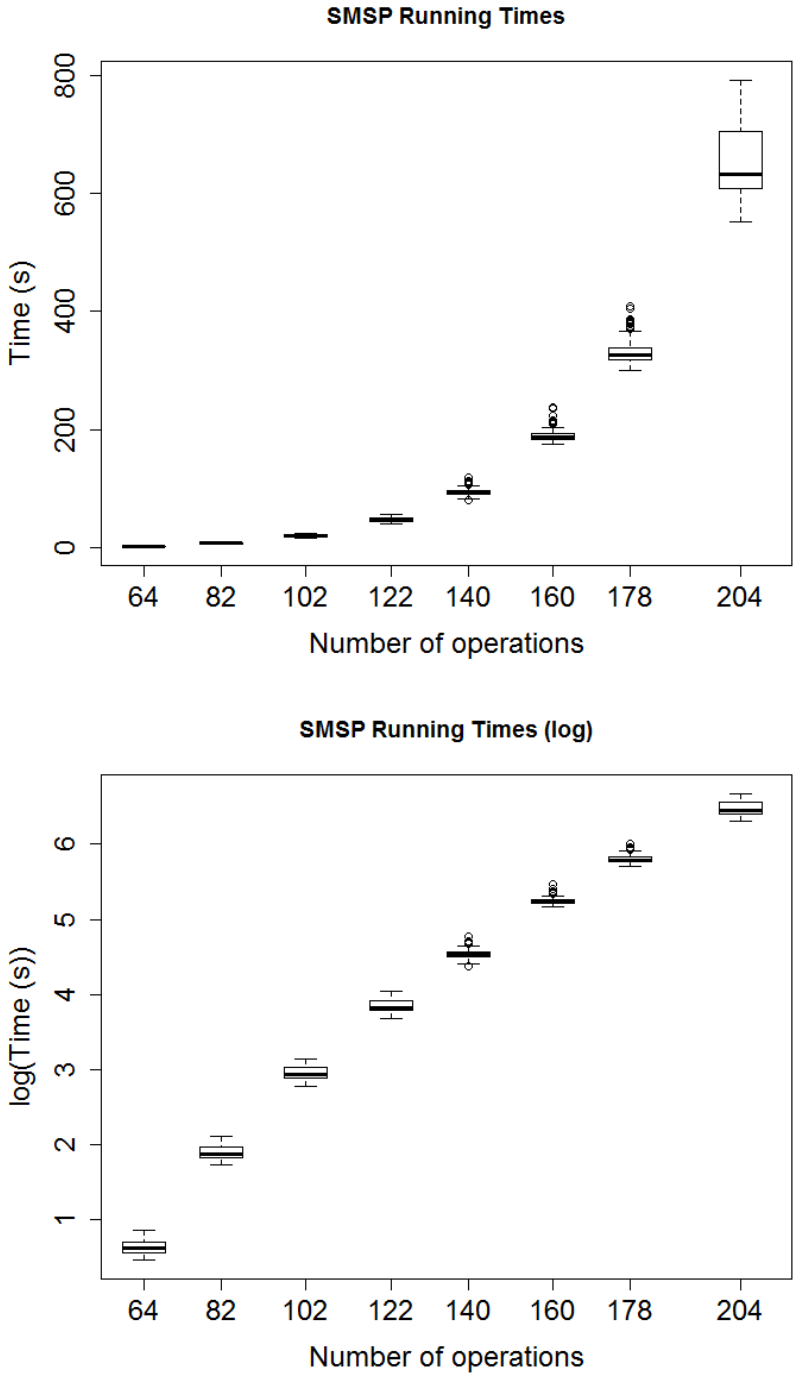

Figure 11.4.3: SMSP running times until the solution was found, across varying problem sizes (x-axis to scale).

Problem Size	B&B Better (%)	Median difference (B&B-AMCC)
64	71	-993
82	75	-1022
102	92	-503031
122	95	-4037
140	88	-1001985
160	86	-1003090
178	80	-998995848
204	85	-1990983639

Table 11.4.6: The percentage of runs in which B&B had obtained a better result by the time AMCC finished, and the median difference in total tardiness score between the two methods (B&B - AMCC).

can be seen that, without exception, these differences are negative. Mean and standard deviation values of the differences cannot be provided, since some of the AMCC/SMSP runs had an infinite penalty score, as they did not produce a feasible solution. Even if B&B could not locate the optimal solutions, the solutions it found are substantially better than the ones generated by the rollout heuristics. Thus, in an industrial context where the time budget is similar, B&B is the most appropriate choice to be made. To conclude, B&B can be seen to outperform the rollout methods, on average, when scaled across all problem sizes, when the same amount of running

Problem Size	B&B Better (%)	Median difference (B&B-SMSP)
64	71	-529
82	77	-1041
102	92	-2954
122	96	-2077
140	91	-1002341
160	87	-2001557
178	79	-998992788
204	84	-1004001403

Table 11.4.7: The percentage of runs in which B&B had obtained a better result by the time SMSP finished, and the median difference in total tardiness score between the two methods (B&B - SMSP).

time is available to each method.

Chapter 12

Blocking Job Shop: Conclusions

The work in Part III was motivated by the real life scheduling challenges encountered at a facility in the nuclear power industry, as introduced in Chapter 1. The work at the facility was modelled as a blocking job shop scheduling problem with flexible maintenance, due dates, release dates, and precedence constraints (Chapter 8). Heuristic solution methods for this problem were presented in Chapter 9, including methods based on Ant Colony Optimisation, Simulated Annealing, rollout procedures, and topological sorting. An exact Branch and Bound method was also developed (Chapter 10). It includes both a novel branching strategy based on variable ranking, and a novel search strategy based on periodic solution stack reordering.

Thorough experimental results, comparisons and analysis on 100 prob-

lem instances show that the proposed Branch and Bound algorithm produces optimal solutions, within a short running time (Chapter 11). The computational results show that the exact Branch and Bound method clearly outperforms all tested heuristic methods in terms of the quality of the final solutions found. In terms of running time, only the heuristics based on Topological Sorting are faster than Branch and Bound, however, these are not suitable alternatives due to their poor quality solutions. Most of the heuristic methods generally produce low quality solutions. The heuristics also tend to require much more computational time, compared to the proposed Branch and Bound method. A scalability analysis showed that Branch and Bound performs best, both in terms of running time and solution quality, across a range of problem sizes. This makes it a very effective algorithm, which optimally solves the considered problem, within acceptable running time for the industrial problem under consideration.

The Branch and Bound method not only performs well, but it also captures the main characteristics of the facility for which it was developed. At the time of writing, the algorithm is being trialled by the schedulers at this facility, as they found that the algorithm models their requirements very well.

Part IV

Conclusion

Chapter 13

Conclusions and Future Work

The computational results for the non-blocking job shop scheduling problem with flexible maintenance (Part II) show that the hybridisation of heuristic and exact methods can be used to obtain optimal solutions, in a short running time. When a hybridisation is not used, the exact Branch and Bound method requires excessive running time, while the heuristic methods rapidly find optimal or near-optimal solutions. In contrast to this, the opposite holds true in the blocking variant of the problem (Part III). There, the heuristics generally struggle to produce good solutions. In fact, just producing feasible solutions of any quality was found to be problematic for the heuristic methods. Our novel guaranteed feasibility schedule construction method now ensures the heuristic solutions are at least usable, but the heuristic algorithms still struggle to produce good quality solutions. In contrast to this, the Branch and Bound algorithm for the block shop presented in Chapter 10 very quickly produces optimal solutions.

13.1 Future Work

Following the work presented in this thesis, there are a number of open research directions. We would like to further enhance the methods presented here to also handle the rescheduling of existing schedules. Rescheduling may be required when new jobs have to be integrated in a schedule which has already been partially processed, or in response to unforeseen circumstances, such as machine breakdowns. The developed Branch and Bound algorithm can be used for total rescheduling in response to such events. However, it would be of interest to develop robust rescheduling methods, which aim to minimise disruption to the existing schedule. Such a robust rescheduling method would have to balance a trade-off between minimising the total tardiness score, and minimising 'change' between the old and new schedule, for some measure of change. There is also the potential to do future work on the inclusion of other resource requirements. Currently, the only resource taken into account is the location where a job is to be processed. However, there are certain jobs which can only be performed by appropriately trained workers, and jobs which require specialist equipment. Thus, it would be of interest to build on the work presented in this thesis by developing a scheduling method that accounts for multiple resource requirements.

Appendix A

Test Problems

This appendix describes the properties of the 100 test problems created for the job shop scheduling problem with flexible maintenance and due dates. This problem is defined in Chapter 3. The same test instances are also suitable for the blocking version of the same problem, as defined in Chapter 8. The test problems were randomly generated, but under some constraints so as to include some typical features of the facility that the scheduling algorithms were developed for. This facility is described in Chapter 1. The test problems are available at `http://dx.doi.org/10.17635/lancaster/researchdata/160`.

Recall that the facility comprises of shielded workstations. It also contains a single transportation mechanism that connects all workstations and transfers materials across all workstations. There are two workstations that both act as entry and exit points for any materials, which is a physical constraint for the facility. As such, given a material to be processed, it will

have to enter the system from one of these two workstations, and be moved to the appropriate workstation by the transporter. The materials may then have to undergo further processing at other workstations, and eventually leave the system through one of the two exit points.

Jobs typically start and end at one of the two combined entry/exit points, and must use the transporter to travel between workstations. Every second operation of a job is therefore usually a transporter operation. The test instances were designed for a facility with 9 workstations (machine IDs 1-9) and one transporter (machine ID 0). Workstations 1 and 9 both serve as entry and exit points. Each test problem consists of 20 regular production jobs, some with merges or splits. There are also 10 maintenance activities, one for each workstation and the transporter. The total number of operations in each test instance is 102. Time is measured in hours, assuming a 40 hour working week. Some jobs are released at time 0, and some jobs are released at time 40.

A.1 Job Properties

Each job has the following properties:

- Jobs consist of a sequence of operations, where each operation requires a single specified machine (0-9) and has a specified deterministic processing time.

- Machines 1 and 9 provide access to the system, so that jobs must

start and finish at one of these (except jobs which merge or split).

- Machine 0 is the transporter between machines 1-9, which means every second operation of a job must be on machine 0.
- A machine (2-8) was randomly assigned to each operation (with equal probabilities), except for entry/exit and transport operations. Machines were assigned in such a way that transportation always takes place between two different machines, i.e. avoiding the situation where a job is collected from and then delivered to the same machine.
- Some jobs are released at time 0 (the start of week one), and some at time 40 (the start of week two).
- Processing times are randomly generated from the following uniform distributions:
  - Machines 1 and 9 (entry/exit): Unif(2,6)
  - Machine 0 (transport): Unif(0.5,1.5)
  - Machines 2-8: Unif(8,32)
- Each job has a soft due date, and some jobs also have a hard due date.
- Due dates are randomly generated, taking into account the total processing time required for the job. This is done in such a way that

each job in isolation could always be completed before its soft due date, as follows:

  - Soft due date = Release date + (Total processing time) × RandomUnif(1.75,2.25)
  - Hard due date = Release date + (Total processing time) × RandomUnif(2.75,3.25)

- Some pairs of jobs merge into one job, and some jobs split into two jobs.

A.2 Maintenance Activity Properties

Maintenance activities are modelled as jobs with a single operation. During maintenance, the machine is unavailable to other jobs. All the maintenance activities in the test problems have a duration of 8 hours, and a flexible starting window of duration 26.67. This is based on current practice for periodic maintenance at the facility of interest.

A.3 Test Problem Properties

Each test problem has the following properties:

- Total number of operations: 102
- The following jobs are included in each problem:

- One job with five operations, release date 0, and both soft and hard due dates.
- One job with five operations, release date 0, and a soft due date.
- One job with five operations, release date 40, and both soft and hard due dates.
- One job with five operations, release date 40, and a soft due date.
- One job with seven operations, release date 0, and both soft and hard due dates.
- One job with seven operations, release date 0, and a soft due date.
- One job with seven operations, release date 40, and both soft and hard due dates.
- One job with seven operations, release date 40, and a soft due date.
- Two jobs (release dates 0 and 40) which each split into two jobs (giving a total of six jobs). There are three operations before the split (entry → transport → processing) and four operations on each fork of each split (transport → processing → transport → exit). All six jobs have soft and hard due dates calculated as described above, except that the processing time required for jobs after the split is the total processing time required by both

the job before and the job after the split.

  - Two pairs of jobs (release dates 0 and 40) that merge into a third job (giving a total of six jobs). There are four operations in each job before the merge (entry → transport → processing → transport) and three operations in the jobs after the merge (processing → transport → exit). All six jobs have soft and hard due dates calculated as described above, except that the processing time required for jobs after the merge is the total processing time required by both the job before and the job after the split.

- There are 10 maintenance activities, one for each machine. The first maintenance activity has release date 0. A single maintenance activity is released every 16 hours (two days). The ordering in which the machines receive maintenance is random, with equal probabilities.

A.4 File Format

Each of the 100 test instances is stored in a separate .csv file. A typical job entry has the following format:

Job ID,1,,,,,,,,,,,,,,,,,,

Type (job/maintenance),job,,,,,,,,,,,,,,,,,,

Resource,9,0,3,0,9,,,,,,,,,,,,,,

Processing time,5.36,1.41,19.41,1.04,4.84,,,,,,,,,,,,,,

Due date (soft and hard),61.3,102.83,,,,,,,,,,,,,,,,,,

Release date,0,,,,,,,,,,,,,,,,,,

Preceding Jobs,,,,,,,,,,,,,,,,,,,,

Precedence Type,,,,,,,,,,,,,,,,,,,,

Each entry can be described as follows:

- **Job ID:** A unique job identification number.
- **Type (job/maintenance):** Identifies task as a regular job ("job") or maintenance ("maintenance").
- **Resource:** The resource (machine ID) required by each of the operations of this job. The ordering of the operations within the job is defined by the ordering of the data.
- **Processing time:** The processing time required for each operation.
- **Due date (soft and hard):** The soft and hard due dates. Some jobs have their hard due dates set as "Inf", this denotes infinity and indicates there is no hard due date for the job.
- **Release date:** The release date (time) of the job.

- **Preceding Jobs:** The precedent constraints, if any. These are the job IDs of jobs which must be completed before this job can start.

- **Precedence Type:** Indicates whether the precedence constraint is part of a job split ("split") or job merge ("merge").

Maintenance entries have the same format as job entries, but are marked as maintenance on the "Type (job/maintenance)" line. A typical maintenance entry has the following format:

Job ID,40,,,,,,,,,,,,,,,,,,

Type (job/maintenance),maintenance,,,,,,,,,,,,,,,,,,

Resource,0,,,,,,,,,,,,,,,,,,

Processing time,8,,,,,,,,,,,,,,,,,,

Due date (soft and hard),144,170.67,,,,,,,,,,,,,,,,,

Release date,144,,,,,,,,,,,,,,,,,,

Preceding Jobs,,,,,,,,,,,,,,,,,,,

Precedence Type,,,,,,,,,,,,,,,,,,,

Please note that the job IDs range from 1 to 40, but not every number in that range is used. These should not be changed, as the precedence constraints depend on them.

Bibliography

E. H. L. Aarts and P. J. M. van Laarhoven. Statistical Cooling: A General Approach to Combinatorial Optimization Problems. *Philips Journal of Research*, 40(4):193–226, 1985.

T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42 – 54, 2005.

H. Allaoui, S. Lamouri, A. Artiba, and E. Aghezzaf. Simultaneously scheduling n jobs and the preventive maintenance on the two-machine flow shop to minimize the makespan. *International Journal of Production Economics*, 112(1):161–167, 2008.

C. Artigues and D. Feillet. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals of Operations Research*, 159(1): 135–159, December 2007.

T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

K. R. Baker. Sequencing Rules and Due-Date Assignments in a Job Shop. *Management Science*, 30(9):1093–1104, 1984.

K. R. Baker and J. J. Kanet. Job shop scheduling with modified due dates. *Journal of Operations Management*, 4(1):11–22, November 1983.

S. Batun and M. Azizoglu. Single machine scheduling with preventive maintenances. *International Journal of Production Research*, 47(7):1753–1771, 2009.

M. Ben Ali, M. Sassi, M. Gossa, and Y. Harrath. Simultaneous scheduling of production and maintenance tasks in the job shop. *International Journal of Production Research*, 49(13):3891–3918, 2011.

M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

C. Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *Operations-Research-Spektrum*, 17(2-3):87–92, 1995.

J. Błażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1 – 33, 1996.

C. Blum. Beam-ACO-hybridizing ant colony optimization with beam search: an application to open shop scheduling. *Computers & Operations Research*, 32 (6):1565–1591, June 2005.

C. Blum and M. Sampels. An ant colony optimization algorithm for shop scheduling problems. *Journal of Mathematical Modelling and Algorithms*, 3(3):285–308, 2004.

C. Blum, J. Puchinger, G. R. Raidl, and A. Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135 – 4151, 2011.

I. Boussaïd, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237(0):82 – 117, 2013.

P. Brucker and O. Thiele. A branch & bound method for the general-shop problem with sequence dependent setup-times. *Operations-Research-Spektrum*, 18 (3):145–161, September 1996.

P. Brucker, B. Jurisch, and A. Krämer. The job-shop problem and immediate selection. *Annals of Operations Research*, 50(1):73–114, 1994a.

P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(13):107–127, March 1994b.

B. Bullnheimer, R. F. Hartl, and C. Strauß. A new rank based version of the ant system - a computational study. *Central European Journal for Operations Research and Economics*, 7:25–38, 1997.

E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, Dec 2013.

J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

H. Chen and P. B. Luh. An alternative framework to Lagrangian relaxation approach for job shop scheduling. *European Journal of Operational Research*, 149(3):499–512, September 2003.

T. Cheng and C. Sin. A state-of-the-art review of parallel-machine scheduling research. *European Journal of Operational Research*, 47(3):271 – 292, 1990.

A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian. Ant system for job-shop scheduling. *Belgian Journal of Operations Research, Statistics and Computer Science*, 34(1):39–53, 1994.

P. I. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *Selected Papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, PATAT '00, pages 176–190, London, UK, UK, 2001. Springer-Verlag.

A. D'Ariano, D. Pacciarelli, and M. Pranzo. A branch and bound algorithm for scheduling trains in a railway network. *European Journal of Operational Research*, 183(2):643 – 657, 2007.

M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, 2004.

J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Metaheuristics for Hard Optimization*. Springer, Berlin, 2006.

J. Du and J. Y. Leung. Minimizing total tardiness on one machine is np-hard. *Math. Oper. Res.*, 15(3):483–495, July 1990.

R. Eglese. Simulated annealing: A tool for operational research. *European Journal of Operational Research*, 46(3):271 – 281, 1990.

M. Ehrgott and X. Gandibleux. Hybrid metaheuristics for multi-objective combinatorial optimization. In C. Blum, M. Aguilera, A. Roli, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 114 of *Studies in Computational Intelligence*, pages 221–259. Springer Berlin Heidelberg, 2008.

S. Eilon and I. G. Chowdhury. Due dates in job shop scheduling. *International Journal of Production Research*, 14(2):223–237, March 1976.

P. Festa and M. G. C. Resende. An annotated bibliography of GRASP - Part I: Algorithms. *International Transactions in Operational Research*, 16(1):1–24, 2009.

D. J. Fonseca and D. Navaresse. Artificial neural networks for job shop simulation. *Advanced Engineering Informatics*, 16(4):241–246, October 2002.

M. Gendreau. An introduction to tabu search. In *Handbook of Metaheuristics*, pages 37–54. Springer US, Boston, MA, 2003.

F. Glover, C. McMillan, and B. Novick. Interactive decision software and computer graphics for architectural and space planning. *Annals of Operations Research*, 5(3):557–573, 1985.

F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533 – 549, 1986.

F. Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74 – 94, 1990.

F. Glover and M. Laguna. General purpose heuristics for integer programming—part i. *Journal of Heuristics*, 2(4):343–358, 1997a.

F. Glover and M. Laguna. General purpose heuristics for integer programming—part ii. *Journal of Heuristics*, 3(2):161–179, 1997b.

F. Glover and C. McMillan. The general employee scheduling problem. an integration of ms and ai. *Computers & Operations Research*, 13(5):563 – 573, 1986.

F. W. Glover and G. A. Kochenberger, editors. *Handbook of Metaheuristics*, volume 114 of *International Series in Operations Research & Management Science*. Springer, 1 edition, 2003.

J. F. Gonçalves, J. J. de Magalhães Mendes, and M. G. C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*, 167(1):77–95, November 2005.

R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In E. J. P.L. Hammer and B. Korte, editors, *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

D. Grimes and E. Hebrard. Solving variants of the job shop scheduling problem through conflict-directed search. *INFORMS Journal on Computing*, 27(2): 268–284, 2015.

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

M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric Statistical Methods*. Wiley, 3 edition edition, 2013.

H. H. Hoos and T. Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

K. Huang and C. Liao. Ant colony optimization combined with taboo search for the job shop scheduling problem. *Computers & Operations Research*, 35(4): 1030 – 1046, 2008.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION-5*, LNCS, pages 507–523, 2011.

F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML 2014)*, page 754762, June 2014.

A. S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present and future. *European journal of operational research*, 113(2):390–434, 1999.

K. Jansen, M. Mastrolilli, and R. Solis-Oba. Approximation schemes for job shop scheduling problems with controllable processing times. *European Journal of Operational Research*, 167(2):297–319, December 2005.

T. Jansen. *Analyzing Evolutionary Algorithms: The Computer Science Perspective*. Springer, 2013.

S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

D. Z. Joseph Adams, Egon Balas. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

L. Jourdan, M. Basseur, and E.-G. Talbi. Hybridizing exact methods and metaheuristics: A taxonomy. *European Journal of Operational Research*, 199(3): 620 – 629, 2009.

G. K. Kao, E. C. Sewell, and S. H. Jacobson. A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem. *Journal of Scheduling*, 12(2): 163–175, 2008.

E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

D. Klabjan, E. Johnson, G. Nemhauser, E. Gelman, and S. Ramaswamy. Solving large airline crew scheduling problems: Random pairing generation and strong branching. *Computational Optimization and Applications*, 20(1):73–91, 2001.

M. Kolonko. Some new results on simulated annealing applied to the job shop scheduling problem. *European Journal of Operational Research*, 113(1):123–136, February 1999.

J. Kuhpfahl and C. Bierwirth. A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective. *Computers & Operations Research*, 66:44 – 57, 2016.

J. Kuhpfahl. *Job Shop Scheduling with Consideration of Due Dates*. Springer Fachmedien Wiesbaden, Wiesbaden, 2016.

V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

J. Li and Q. Pan. Chemical-reaction optimization for solving fuzzy job-shop scheduling problem with flexible maintenance activities. *International Journal of Production Economics*, 145(1):4 – 17, 2013.

T. Loukil, J. Teghem, and D. Tuyttens. Solving multi-objective production scheduling problems using metaheuristics. *European Journal of Operational Research*, 161(1):42 – 61, 2005.

H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*, pages 363–397. Springer US, Boston, MA, 2010.

Y. Ma, C. Chu, and C. Zuo. A survey of scheduling with deterministic machine availability constraints. *Computers & Industrial Engineering*, 58(2):199 – 211, 2010.

R. Mallipeddi, S. Mallipeddi, and P. Suganthan. Ensemble strategies with adaptive evolutionary programming. *Information Sciences*, 180(9):1571 – 1581, 2010.

A. Mascis and D. Pacciarelli. Machine scheduling via alternative graphs. Technical Report DIA 46-2000, Dipartimento di Informatica e Automazione, Università Roma, 2000.

A. Mascis and D. Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3):498–517, 2002.

E. Maslov, M. Batsyn, and P. M. Pardalos. Speeding up branch and bound algorithms for solving the maximum clique problem. *Journal of Global Optimization*, 59(1):1–21, 2014.

Y. Mati, S. Dauzère-Pérès, and C. Lahlou. A general approach for optimizing regular criteria in the job-shop scheduling problem. *European Journal of Operational Research*, 212(1):33 – 42, 2011.

D. C. Mattfeld and C. Bierwirth. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *European Journal of Operational Research*, 155(3):616 – 630, 2004.

C. Meloni, D. Pacciarelli, and M. Pranzo. A rollout metaheuristic for job shop scheduling problems. *Annals of Operations Research*, 131(1-4):215–235, 2004.

H. Mokhtari and M. Dadgar. Scheduling optimization of a stochastic flexible job-shop system with time-varying machine failure rate. *Computers & Operations Research*, 61:31–45, September 2015.

J. Montgomery. Alternative Solution Representations for the Job Shop Scheduling Problem in Ant Colony Optimisation. In M. Randall, H. A. Abbass, and J. Wiles, editors, *Progress in Artificial Life*, number 4828 in Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, December 2007.

J. Montgomery, C. Fayad, and S. Petrovic. Solution Representation for Job Shop Scheduling Problems in Ant Colony Optimisation. In M. Dorigo, L. M. Gambardella, M. Birattari, A. Martinoli, R. Poli, and T. Stützle, editors, *Ant Colony Optimization and Swarm Intelligence*, number 4150 in Lecture Notes in Computer Science, pages 484–491. Springer Berlin Heidelberg, September 2006.

D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79 – 102, 2016.

E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

E. Pan, W. Liao, and L. Xi. Single-machine-based production scheduling model integrated preventive maintenance planning. *The International Journal of Advanced Manufacturing Technology*, 50(1-4):365–375, 2010.

F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120(2):297–310, January 2000.

M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

M. L. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2nd edition, 2009.

M. Pranzo and D. Pacciarelli. An iterated greedy metaheuristic for the blocking job shop scheduling problem. *Journal of Heuristics*, 22(4):587–611, 2016.

C. Rajendran and H. Ziegler. Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research*, 155(2):426 – 438, 2004.

N. Raman and F. Brian Talbot. The job shop tardiness problem: A decomposition approach. *European Journal of Operational Research*, 69(2):187–199, September 1993.

B. Roy and B. Sussmann. Les problemes d'ordonnancement avec contraintes disjonctives. *Note D.S.*, 9, 1964.

J. P. Royston. An extension of Shapiro and Wilk's W test for normality to large samples. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):115–124, 1982.

R. Ruiz, J. C. García-Díaz, and C. Maroto. Considering scheduling and preventive maintenance in the flowshop sequencing problem. *Computers & Operations Research*, 34(11):3314 – 3330, 2007.

N. Sadeh and M. S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1 – 41, 1996.

L. Shi and S. Ólafsson. *Nested Partitions Method, Theory and Applications*. Springer, 2009.

M. Singer and M. Pinedo. A computational study of branch and bound techniques for minimizing the total weighted tardiness in job shops. *IIE Transactions*, 30(2):109–118, 1998.

I. Struijker Boudier, K. Glazebrook, M. Wright, and P. Jennings. Ant colony optimisation for a job shop with flexible maintenance. In *Proceedings of the* 7^{th} *Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2015)*, pages 614–616, 2015.

T. Stützle and H. Hoos. Max-min ant system and local search for the traveling salesman problem. In *IEEE International Conference On Evolutionary Computation (ICEC'97)*, pages 309–314. IEEE Press, 1997.

I. T. Tanev, T. Uozumi, and Y. Morotome. Hybrid evolutionary algorithm-based real-world flexible job shop scheduling problem: application service provider approach. *Applied Soft Computing*, 5(1):87–100, December 2004.

M. van den Akker, K. van Blokland, and H. Hoogeveen. Finding robust solutions for the stochastic job shop scheduling problem by including simulation in local search. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 402–413. Springer Berlin Heidelberg, 2013.

J. A. Vázquez-Rodríguez and S. Petrovic. A new dispatching rule based genetic algorithm for the multi-objective job shop problem. *Journal of Heuristics*, 16 (6):771–793, Dec 2010.

S. Wang. Bi-objective optimisation for integrated scheduling of single machine with setup times and preventive maintenance planning. *International Journal of Production Research*, 51(12):3719–3733, 2013.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

M. Wright. Automating parameter choice for simulated annealing. Management science working paper series, The Department of Management Science, Lancaster University, 2010.

L. Xing, Y. Chen, P. Wang, Q. Zhao, and J. Xiong. A Knowledge-Based Ant Colony Optimization for Flexible Job Shop Scheduling Problems. *Applied Soft Computing*, 10(3):888–896, June 2010.

T. Yamada and R. Nakano. Job-shop scheduling by simulated annealing combined with deterministic local search. In I. H. Osman and J. P. Kelly, editors, *Meta-Heuristics*, pages 237–248. Springer US, 1996.

H. Yang, Q. Sun, C. Saygin, and S. Sun. Job shop scheduling based on earliness and tardiness penalties with due dates and deadlines: an enhanced genetic algorithm. *The International Journal of Advanced Manufacturing Technology*, 61(5-8):657–666, 2012.

W. Zhang and R. E. Korf. An average-case analysis of branch-and-bound with applications: Summary of results. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 545–550. AAAI Press, 1992.