TU Delft

Delft University of Technology

Document Version
Final published version

Citation (APA)
Eigbe, E. (2026). *Optimising Discrete Problems: Decision Diagrams and Context-Aware Heuristics*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:c615f31a-f85c-40d2-8ab7-2fa16f33a332

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Optimising Discrete Problems

Decision Diagrams and Context-Aware Heuristics

Eghonghon-Aye Eigbe

Optimising Discrete Problems

Decision Diagrams and Context-Aware Heuristics

Optimising Discrete Problems

Decision Diagrams and Context-Aware Heuristics

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, Prof. dr. ir. H. Bijl,
chair of the Board for Doctorates
to be defended publicly on
Wednesday, 8 April 2026 at 17:30

by

Eghonghon-Aye EIGBE

This dissertation has been approved by the promotors and the copromotor.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Dr. N. Yorke-Smith,	Delft University of Technology, *promotor*
Prof. dr. ir. B. De Schutter,	Delft University of Technology, *promotor*
Dr. M. Nasri,	Eindhoven University of Technology, *copromotor*
Independent members:	
Prof. dr. M. M. de Weerdt,	Delft University of Technology
Prof. dr. ir. R. Sotirov,	Tilburg University
Prof. dr. Q. Cappart,	Université Catholique de Louvain, Belgium
Prof. dr. W. van Hoeve,	Carnegie Mellon University, United States of America
Prof. dr. ir. K. I. Aardal,	Delft University of Technology, reserve member

Nederlandse Organisatie voor Wetenschappelijk Onderzoek

The research presented in this thesis has been partially funded by the NWO project 17931 (SAM-FMS) in the MasCot programme.

Keywords: decision diagrams, scheduling, discrete optimisation, chance constrained optimisation, anytime scheduling, flexible manufacturing systems

Copyright © 2026 by E. Eigbe

ISBN 978-94-93483-99-6

An electronic copy of this dissertation is available at
`https://repository.tudelft.nl/`.

For Matthew Eromosele Eigbe.

Contents

Summary

Optimisation problems are all around us and play a critical role in the outcomes of various sectors of society including scheduling, logistics, network design, and resource allocation. In this thesis, we look at a subset of problems where some or all the choices to be made can only take on values belonging to a discrete set of values – a limitation which increases the difficulty of finding a solution in most cases. We handle this thesis in two parts: first zooming in on a particular class of problems – scheduling – and then on a particular solution method – decision diagrams.

In Part I of this dissertation, we consider the problem of scheduling in manufacturing systems. This popular discrete optimisation problem has been studied extensively; however, advancements in modern manufacturing systems present new challenges and opportunities. A major driver of the changes in manufacturing systems is the integration of the physical processes with computation, networking, and automated control capabilities. Thus, the domain of activities that can directly be decided upon and actuated from software has expanded. We look at one such activity in Chapter 3 namely, sequence-dependent maintenance and propose solution methods that directly integrate maintenance and production planning.

Part II of this thesis looks into decision diagrams as a solution method for discrete optimisation problems. Decision diagrams have existed since the 1950s and were originally introduced as a means to represent boolean functions. Since then they have been used in different fields such as in circuit verification, knowledge representation, and most recently, operations research. While decision diagrams have already been shown to be very promising methods for solving optimisation problems, we push the field forward as follows. In Chapter 5, we propose a decision diagram model for the kind of scheduling problems tackled in Part I of this thesis. We further perform a comparative study of the consequences of heuristic decisions made during the decision diagram compilation process in Chapter 6 and integrate reinforcement learning with decision-diagram-based branch-and-bound in Chapter 7. In Chapter 8 we go further into considering uncertainty in optimisation problems. We focus on the paradigm of finding the best solution while accepting some level of risk, i.e., chance constrained optimisation and present a chance constrained decision diagram formulation. We further provide theoretical guarantees for instances with normally distributed variables.

The contributions of this thesis cover different aspects of solving discrete optimisation problems with a focus on scheduling and logistic applications. The hope is that these advances further the adoption of state of the art research in solving optimisation problems in real-world contexts.

Samenvatting

Optimalisatieproblemen spelen een cruciale rol in verschillende sectoren van de maatschappij zoals roostering/planning, logistiek, netwerkdesign, en de allocatie van resources. In deze thesis kijken wij naar een subset van problemen waarvoor sommige of alle beslissingen die gemaakt moeten worden waarden die behoren tot een discrete set kunnen aannemen - een beperking die het bemoeilijkt om een oplossing te vinden in de meeste gevallen. We organiseren deze thesis in twee delen: eerst zoomen we in op een specifieke groep problemen - roosteringproblemen - en daarna op een specifieke oplosmethode - beslisdiagrammen.

In Deel I van dit proefschrift richten wij ons op het probleem van planning in productiesystemen. Dit populaire discrete optimalisatieprobleem is uitgebreid onderzocht; ontwikkelingen in moderne productiesystemen brengen echter nieuwe uitdagingen en mogelijkheden met zich mee. Een belangrijke drijfveer achter de veranderingen in productiesystemen is de integratie van fysieke processen met rekenkracht, netwerken en geautomatiseerde aansturingsmogelijkheden. Daardoor is het domein van activiteiten dat direct vanuit software kan worden beslist en aangestuurd, uitgebreid. We bestuderen één zo'n activiteit in Hoofdstuk 3, namelijk volgordeafhankelijk onderhoud, en we stellen oplossingsmethoden voor die direct integreren met onderhoud en productieplanning.

Deel II van dit proefschrift kijkt naar beslisdiagrammen als oplossingsmethode voor discrete optimalisatieproblemen. Beslissingsdiagrammen bestaan sinds 1950 en zijn oorspronkelijk geïntroduceerd als middel om booleaanse functies te representeren. Sindsdiens worden ze gebruikt in verschillende velden zoals circuitverificatie, kennisrepresentatie, en meest recent, operations research. Hoewel beslisdiagrammen al veelbelovende methoden zijn gebleken voor het oplossen van optimalisatieproblemen, dragen wij op de volgende manier bij aan de verdere ontwikkeling van het vakgebied. In Hoofdstuk 5 presenteren wij een beslissingsdiagrammodel voor een type roosteringproblemen zoals gepresenteerd in Deel I van het proefschrift. We voeren een vergelijkende studie uit naar de consequenties van heuristische beslissingen die worden gemaakt tijdens het compileren van een beslissingsdiagram in Hoofdstuk 6 en we integreren reinforcement learning met de branch-and-bound methode in Hoofdstuk 7. In Hoofdstuk 8 gaan we dieper in op het meewegen van onzekerheid in optimalisatieproblemen. Wij richten ons op het paradigma van het vinden van de beste oplossing onder aanvaarding van een bepaald risiconiveau, oftewel chance constrained optimalisatie, en presenteren een formulering van beslisdiagrammen met deze kansbeperkingen. Daarnaast geven wij theoretische garanties voor gevallen met normaalverdeelde variabelen.

De bijdragen van dit proefschrift bedekken verschillende aspecten van het oplossen van discrete optimalisatie problemen met een sterke hang naar scheduling en logistieke toepassingen. De hoop is dat deze vooruitgangen de toepassing van state-of-the-art onderzoek bij het oplossen van optimalisatieproblemen in de praktijk verder stimuleren.

1

Introduction

Optimisation problems are ubiquitous in the real word, showing up in multiple applications and systems ranging from route planning in map software available on virtually all smart phones to trajectory planning for space rockets. An optimisation problem typically has a goal or objective value that is to be minimised or maximised and some constraints that place limitations on what choices can be made. A solution to an optimisation problem is thus an assignment of values to all decision variables such that the goal is achieved without violating constraints.

A significant portion of these problems are discrete, i.e., are formulated such that some or all the variables to be decided upon can only take on values belonging to a discrete set of values. This limitation often increases the difficulty of solving the problem (Parker and Rardin 2014).

The general form of a discrete optimisation problem

$$\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f : \mathcal{X} \to \mathbb{R}) \tag{1.1}$$

is such that $\mathcal{X}$ is a finite sequence of variables $\mathcal{X} := x_1, \ldots, x_n$ with domains $D_i \in \mathcal{D}$ associated with each variable where a domain is the set of values a variable is allowed to take, and $\mathcal{C}$ is a finite set of constraints each defined on a subsequence of $\mathcal{X}$. The constraints $\mathcal{C}$ are such that they shrink the domain of the variables to which they apply. The challenge is to find an assignment of values to each variable $x_i \in \mathcal{X}$ such that the assigned value is in the domain D_i, no constraints in $\mathcal{C}$ are violated, and the objective function $f : \mathcal{X} \to \mathbb{R}$ is either maximised or minimised.

1.1. Existing Methods for Solving Discrete Optimisation Problems

Many methods exist to solve discrete optimisation problems. In this section we discuss some of the most prominent ones.

1.1.1. Constraint Programming

Constraint programming (CP) is a solution paradigm for combinatorial problems based on declarative problem description and efficient exclusion of constraint violations from the solution space. CP solvers work by reasoning directly about the constraints (Rossi, Van Beek, and Walsh 2006). The typical flow of a generic CP solver is to begin the search by iterating through the decision variables in some order. In every iteration, a value from its domain is assigned to a decision variable and each assignment triggers *propagation* and *inference* procedures which enforce the consequences of the value assignment. The solution process continues until all variables have a valid assignment. In a constraint satisfaction problem, the problem is solved at this point. In an optimisation problem, the solver continues to find valid assignments until the objective is either minimised or maximised.

1.1.2. Mixed Integer Linear Programming

Linear programs (LPs) are mathematical optimisation methods that optimise a linear function of continuous variables under a set of linear constraints. Mixed-integer

linear programming (MILP) extends LPs such that the set of possible values for some or all variables are limited to integers, essentially; discrete optimisation problems. The standard form of MILPs is:

$$\max \quad \mathbf{c}^T\mathbf{x} + \mathbf{h}^T\mathbf{y} \tag{1.2}$$

$$\text{s.t.} \quad A\mathbf{x} + G\mathbf{y} \leq \mathbf{b} \tag{1.3}$$

$$\mathbf{x} \geq 0 \tag{1.4}$$

$$\mathbf{y} \geq 0 \tag{1.5}$$

$$\mathbf{y} \in \mathbb{Z}, \tag{1.6}$$

where **c** and **h** are vectors of weights, matrices A, G, and vector **b** represent the set of linear constraints, and vectors **x** and **y** hold the decision variables with variables in **y** restricted to be integers. Note that MILP restricts CP as it requires constraints to be expressible as linear equations.

The solution process for MILPs is primarily based on *branch-and-bound*. Branch-and-bound is an algorithmic technique for solving discrete problems employed by a family of algorithms that all share a common core procedure, i.e., enumerating all possible solutions by storing partial solutions in a tree structure and solving these recursively (Land and Doig 2009). The tree structure is constructed by recursively splitting the problem based on the possible values of a decision variables and solving the LP relaxation[1] of the sub-problem in each new node created by a split until a solution that optimally satisfies the integer constraints is reached (Floudas 1995).

1.1.3. Decision Diagrams

Decision diagrams (DDs) are graphical structures capable of compactly representing the solution space of combinatorial optimisation problems. DDs are constructed via recursive dynamic programming formulations and bear very close similarities with Markov Decision Processes (MDP) (Bergman, Cire, Van Hoeve, *et al.* 2016). They are also the first solution method capable of being tuned to produce different quality bounds on the objective function. This tuning is based on the size of the diagram such that the more nodes that are included in a diagram, the tighter the bound is.

Concretely, a DD is a directed acyclic graph $G = (V, E)$ in which every node $v \in V$ represents a state and every edge $e \in E$ is a transition between states based on some variable assignment. In a typical decision diagram, every layer represents a single variable such that all edges leading to the nodes in that layer represent value assignments to the decision variable.

The structure of DDs enables us to recursively divide the problem into smaller sub-problems as every node can be thought of as the root of a sub-graph representing the sub-problem remaining to solve at that node. Consequently, every path in the diagram represents a solution. Given a separable objective function, each edge can also be assigned a weight corresponding to the contribution of that decision to the objective such that the length of a path is then also the objective function value of the solution represented by the path.

[1] LP relaxations of MILPs ignore the integrality constraints and assume all variables are continuous.

1.1.4. Heuristics

Some of the earliest and most popular means of solving optimisation problems are actually heuristics, also known as rules-of-thumb. Heuristics usually consist of a set of rules and do not aim for optimality guarantees but instead try to find good-enough solutions quickly. Such rules are attractive because real-world problems sometimes have a recurring structures that allow us define rules that give good performance in majority of the cases. Heuristics are also attractive as they have the advantage of being easier to implement and explain and do not require dedicated solvers. A downside of heuristic solutions is that they are typically custom made for specific problem classes – though exceptions exist as in the form of meta-heuristics discussed below – and so do not generalise outside of the problems they were designed for.

Meta Heuristics

A meta-heuristic is a problem independent higher level heuristic designed to tune the behaviour of a lower level heuristic.

Some of the most famous examples are *evolutionary algorithms*, which are inspired by the biological process of evolution. They manipulate populations of solutions and iteratively improve them by processes that mimic natural selection and breeding. The particular operations involved in selecting, mutating or combining solutions can be problem dependent but the overall evolutionary framework is not.

In general, meta-heuristics guide us through the solution space towards more promising solutions. In fact, some of them are explicitly search based such as *iterated greedy* (Stützle and Ruiz 2018).

1.1.5. Machine Learning

Machine learning (ML) typically involves giving computers the ability to perform tasks without explicit instructions. Many machine learning methods are dependent on learning from historical data and then generalising to unseen data. Optimisation problems can also be thought of as a task that can be solved without explicit instructions or dedicated algorithms and as many real-life applications solve the same or similar problems over and over, there is often historical data and room to generalise. Thus ML methods have been proposed for solving optimisation problems.

The use of machine learning can be in learning to solve the problem directly as in (Song *et al.* 2022; C. Zhang *et al.* 2020), learning to perform a particular step in the solution process of another algorithm as in (Chalumeau *et al.* 2021), learning appropriate hyper-parameters and settings for an algorithm as in (Reijnen, Y. Zhang, Bukhsh, *et al.* 2022; Reijnen, Y. Zhang, Lau, *et al.* 2022), or using learning methods to make predictions of unknown future events that a solving algorithm then takes as input (Efthymiou and Yorke-Smith 2023).

1.2. The Role of Uncertainty

As described above, solutions to optimisation problems are assignments of values to variables such that certain conditions are met. However, there are many instances

where optimisation involves solving problems without perfect knowledge of what these conditions are or will be. Take for instance, the problem of selecting what inventory to stock with a goal to maximise profit. It could be that prices and demand fluctuate and as such the actual profit of any solution cannot be exactly known at the time of planning. This kind of scenarios are quite common and each of the existing solution methods described have extensions and modifications that handle uncertainty. The solution possibilities depend on the model of uncertainty that we have, i.e. is there a known distribution or are we uncertain about the uncertainty itself?, and the amount of risk, either of constraint violations or optimistic objective function values, we are willing to accept in a solution.

In instances where we are very risk averse such as safety critical operations, the accepted practice is to plan for the worst case, known as robust optimisation. An example is in the field of real-time systems where a schedule must be decided for safety critical tasks. Often, scheduling those tasks is based on the worst case execution time (WCET) – even though the tasks may finish earlier than the WCET – as tasks have strict deadlines and violations can be catastrophic. In other instances, we can plan more stochastically by optimising expected values. We can also rely on predictions of an unknown future and there are multiple lines of work on optimisation algorithms incorporating predictions while limiting the amount of damage or deviation possible should the predictions turn out to be wrong.

1.3. Contents of this Thesis

The methods described above appear in this thesis in different forms. We either extend the methods by establishing new components of the solution process or model problems to fit the methods. Figure 1.1 shows the connection between each content chapter and an existing method.

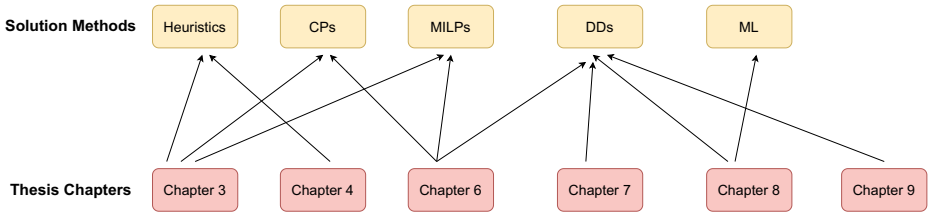

Figure 1.1.: Relationship between thesis chapters and existing methods.

1.3.1. Research Goals

We have two broad research goals corresponding to both parts of the thesis.

How to increase context awareness of heuristic algorithms for scheduling? Scheduling is one of the most common optimisation problems due to the significant economic impact of good scheduling decisions on the outcomes of many industrial

and logistic processes. As such, a host of solution methods exist in the literature for scheduling problems. We find that many real-world systems still use heuristics for scheduling due to their simplicity and explainability. However, in a bid to be simple, heuristic solutions often ignore important parts of the process either by not modelling them at all or not considering them in the decision making process. In Part I of this thesis, we look into increasing the context awareness of heuristic algorithms for scheduling in manufacturing systems without significantly increasing the complexity of the algorithms.

How to leverage decision diagrams to effectively solve real-world problems? In Part II of this thesis, we look into DDs as a solution method, targetting questions on how different means of constructing diagrams affect the solution quality, how to build a generalised DD model for multi-machine scheduling problems, how to integrate learned or heuristic policies with DDs and finally how to handle uncertainty when DDs are used as a solution method.

1.3.2. Contributions

Below, we present an overview of this thesis' main contributions.

Extending list heuristics for scheduling to incorporate maintenance planning (Chapter 3). List schedulers are a class of heuristics for scheduling that work by listing all the operations to be scheduled according to a given *order* and inserting them in a schedule one after the other until all operations are scheduled. An example is the *shortest processing time first* algorithm where operations are ordered according to their processing time. They are usually easy to implement and perform well for many problems (Ruiz 2015). On the other hand, many manufacturing operations also have sequence dependent maintenance triggers as different sequences of operations have different deterioration effects on the machines. To make a list scheduling approach maintenance-aware, we propose to evaluate the effect of any operation placement on maintenance triggering leading to a schedule that already includes the necessary maintenance activities triggered by the chosen operation sequence.

Decision diagram models for multi-machine scheduling problems (Chapter 5). Decision diagrams have proven to be effective tools for solving single resource sequencing problems such as single machine scheduling and the travelling salesman problem (Cire and Van Hoeve 2012). In this thesis, we extend existing models to also cover cases with multiple machines. We focus on the manufacturing use case with multiple machine though the results are generalisable to other multi resource scheduling problems.

Quantifying the effects of decision diagram compilation techniques. (Chapter 6). We study three techniques for DD compilation namely merge heuristics, dominance, and variable ordering. Based on empirical evidence from a set of seven classic optimisation problems, we provide insights on what classes of problems are better

suited to what compilation techniques and how the combination of techniques affect DD based solutions.

Reinforcement learning based restricted decision diagrams (Chapter 7). A key ingredient in DD-based branch-and-bound is the idea of iteratively creating small diagrams from different vertices in the DD to provide solutions and bounds on the objective function with solutions usually created by restricted diagrams. Due to the dependence of both *reinforcement learning* – a learning paradigm for sequential decision making – and DDs on Markovian state transitions, we are able to seamlessly integrate a trained agent to create some of these small diagrams, particularly single-width restricted diagrams.

Bounds for optimisation problems with normally distributed uncertain variables via decision diagrams (Chapter 8). We provide new theoretical results on using decision diagrams to find bounds on optimisation problems in the objective. We show that provided the uncertain variables are normally distributed, chance constrained bounds can be extracted from traditional DDs by swapping out the edge weights with the value of the inverse cumulative distribution function at the required confidence level.

1.3.3. Outline

The rest of the dissertation is structured as follows. In Part I, we deal with scheduling in manufacturing systems beginning with an introduction in Chapter 2 that provides necessary background knowledge. We then follow up with Chapter 3 where we present our contributions and answers to research questions.

Part II begins similarly with an introduction to DDs in Chapter 4. Chapters 5 to 8, discuss specific contributions of this thesis and finally, we end with Chapter 9 where we discuss our main conclusions, recommendations, and opportunities for future work.

I

Context Aware Heuristic Scheduling for Flexible Manufacturing Systems

2

Introduction to Scheduling for Manufacturing Systems

In general, a schedule decides the sequence and timing of a set of tasks or operations to be carried out, typically with some goal in mind to be reached or optimised for. As such, scheduling appears in a wide range of manufacturing and logistical applications. This part of the thesis handles the topic of scheduling in flexible manufacturing systems. In this chapter, we introduce the problem, provide some background information on existing work in this field and introduce the research questions addressed in this part of the thesis.

2.1. Scheduling in Manufacturing

The history of scheduling might as well be as old as the history of human civilisation. It is not far fetched to imagine that the first human settlements, after making tools, took turns to use these tools whether by forcefully acquiring them or collaboratively taking turns. However, the formal history of our current era of scheduling research is thought to go back to World War I with the work of Henry Gantt – author of the famous Gantt chart (Wilson 2003) – and other pioneers such as W.E. Smith (Pinedo 2012). At the turn of the 20th century, as industrialisation came to a head, it became clearer that efficient use of machinery and interconnected systems was a formal discipline and had a significant impact on the economy.

The scheduling problem has only grown more complex since then particularly with the increased integration of computational and physical processes. Such integrated systems are called *cyber-physical systems* and many manufacturing systems today fall under this umbrella due to the complex decisions required to meet on demand production, satisfy real time constraints, and even automatically recover from the influences of uncertainty.

Scheduling problems also fall under the class of discrete optimisation problems. A schedule is basically a sequence of operations. It dictates the order in which operations are to be carried out. Thus, optimising a schedule can be thought of as deciding which operation goes in every position of the sequence. Given that only an operation in the set of operations available to be scheduled can be placed in any position, the domain of every decision variable is discrete[1].

A host of solutions have been proposed to scheduling problems. In this thesis, we group solutions into two (2) broad classes; *heuristic solutions* also known as rules-of-thumb and *exact solutions*, i.e., solutions that are able to provide some guarantee of the optimality of the results they produce. Heuristic solutions make decisions based on pre-defined rules and trade completeness for speed by only considering a (small) portion of the solution space. They however, still provide state-of-the-art performance for many real-world problems.

The simplest scheduling problem in manufacturing is the single machine problem where we are making schedules for a manufacturing system with only one machine. This problem reduces to finding the best permutation of all the operations available given a certain goal – a basic sequencing problem. However, even such a problem is quite complex. First, a single machine manufacturing system often still has pre- and post-processing steps and the steps are often dependent on the state of the machine which is in turn dependent on the last scheduled operation. Such sequence-dependent constraints are one of the most common sources of complexity. Furthermore, the amount of information available to the scheduler can differ greatly depending on the circumstance. In some cases, all the jobs or operations to be scheduled for a time period are known in advance and the scheduler has sufficient time to come up with a plan before production begins – this is an *offline scheduling* scenario. In other cases, jobs are free to arrive on the fly and the scheduler must

[1] Note that we can also formulate the problem such that the only decision is what time each operation should start which gives us variables with continuous domains but the common scenario that operations cannot overlap in time on the same resource still discretises the problem.

make decisions within some limited time, for a limited window of available jobs – this is an *online scheduling* scenario. We have the further complication of uncertainty especially in the time it takes to process an operation. As such, many versions of even the single machine scheduling problem are NP-hard (Lu and Yuan 2007; C. T. Ng, Cheng, and Yuan 2002).

Flexible Manufacturing Systems

Flexible manufacturing systems refer to manufacturing systems that can produce a range of products. The machines in such a system are able to either perform different operations or perform the same operations to different specifications. In advanced cyber-physical systems, changes required for the same machine to perform such flexible operations can often be carried out without any human intervention or manual reconfiguration. Thus, it is also a scheduling decision when switches between product types are made.

A popular example of flexible manufacturing systems is the automotive industry where a single assembly line is able to produce multiple vehicle models (El-Khalil and Darwish 2019). The flexibility is across multiple dimensions ranging from simple switches like producing different coloured vehicles in the finishing stage to more complex operations like substituting entire vehicle components in a way that the system can seamlessly switch modes to process the different components.

2.2. The Production Printing Driver Case

The work in this part of the thesis was carried out in collaboration with Canon Production Printing (CPP) (*Canon Production Printing* 2023). CPP produces a range of printing solution targeted at different use cases. There are printers for individual home and ofice use, wide format printers for advertising purposes and large-scale printers (LSPs) for industrial purposes. Of these, LSPs are the only class of printers that count as a flexible manufacturing system and as such is our driver case for this part of the thesis.

LSPs are cut-sheet printers capable of handling a wide range of sheet types and media at very high throughput in the range of hundreds of pages per minute. In Figure 2.1, we show one of the printers in this product line. Typically, these LSPs consist of three main machines or modules, the paper input module (PIM), the image transfer module (ITM), and the paper finishing module (PFM). In between each module is a transport system for carrying sheets from point to point within the printer. The PIM is responsible for separating and loading sheets onto a track that leads to the ITM. The ITM performs the main function of the printer by releasing ink drops on the sheet that match the requested image. From this point, the sheet is dried and cooled while being transferred to the PFM. Finishing a sheet can involve different steps like stacking, stapling, binding, ete, and a dedicated finisher is ofte attached to the end of the printer to carry out these operations, forming a production line. In this thesis, we assume the production process is complete once the sheet exist the printer and do not consider further processes down the line. This choice is made for multiple reasons. First, the finishers are often from different manufacturers

and the processes are so far a black box. As at the time of completing this work, a generalized model of finishers was still under development; we point the reader to (Farboud 2025) for more understanding of the finisher process. Additionally, we assume a loose coupling to the finisher, i.e., are not connected via an automated transport system. Thus, human intervention is required to transfer printed stacks of paper to the finisher and this transfer process is out of the domain of the scheduler.

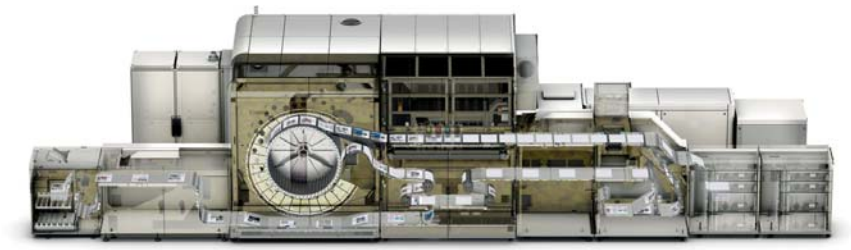

Figure 2.1.: Cross section of the VarioPrint i300 LSP (*Canon Production Printing* 2023)

2.2.1. Re-entrancy

The process of manufacturing a product usually follows some defined flow through the manufacturing system. A flow refers to the order in which different operations must be carried out on the product before it is ready to be delivered. In the example fo the LSP above, the flow is PIM → ITM → PFM.

The flow does not need to be exactly the same for every product, in fact, in flexible manufacturing system it is often the case that different products have different flows. The flow can also be more independent of particular machines, e.g., in plants where machines are multi-purpose or multiple machines are available to perform the same operation. However, there can also be the case where a particular operation has to repeat a step or revisit the same machine in the flow multiple times. Such a requirement is called re-entrancy (Graves *et al.* 1983) and can occur either due to cost concerns – the re-entrant machine is a key machine that is too expensive to simply duplicate and remove the re-entrancy – or quality reasons – some products need to be handled in a delicate state (chemical products for example) and moving the product from one machine to the other would change its state.

Re-entrancy occurs in many production processes, e.g., semi-conductor production, where wafers revisit machines at different stages of the production process and in painting processes where a job may revisit a machine for multiple coats of paint. The

LSP driver case is also a re-entrant problem arising from the duplex printing case, i.e., when we need to print on two sides of a sheet, the same sheet has to be rotated and travel back to the ITM as there is only one such module within the printer due to cost reasons. This is a huge bottleneck in the operation of the LSP and is in fact, the major scheduling challenge.

2.2.2. The Bounded Heuristic Constraint Scheduler (BHCS)

Developed by van Pinxten *et al.* (2017), BHCS is the state of the art heuristic scheduling solution for the LSP. It is a list scheduling approach that builds the sequence of operations to be scheduled and in every iteration, adds one operation to the sequence. The basic skeleton of the algorithm is as shown in Algorithm 1. It works by first identifying feasible sequencing options for the next operation being considered for scheduling in Line 5 and then greedily selecting one of these options in Line 6 before moving on to the next iteration. BHCS improved upon the Heuristic Constraint Scheduler (HCS) introduced in (Waqas *et al.* 2015) by adding more efficient means of generating options and reducing the computational effort for assigning start times.

Additionally, another variant of BHCS known as MD-BHCS exists in the literature where BHCS is combined with a multi-dimensional meta-heuristic to combat the pitfalls of the greedy choices made by BHCS in each iteration. In MD-BHCS, multiple sequencing options are explored simultaneously. In each iteration, instead of picking a single option, a multi-dimensional assessment of the partial solution is made based on different criteria with at most k solutions carried on to the next iteration where k is a tunable hyperparameter.

2.2.3. The Constraint Graph

A schedule can be represented as a sequence of operations. It can also be represented by assigning start times to each operation. The ordering and timing of the operations can either be considered together or one after the other where the order is first decided and the associated timings for the given order are then derived.

To compute such start times, we make use of a constraint graph which is a simple temporal network (Dechter, Meiri, and Pearl 1991) that represents and aids reasoning about temporal constraints on activities.

The constraint graph X is a tuple (N, A) where nodes $n \in N$ represent operations

Algorithm 1 Bounded Heuristic Constraint Scheduler (BHCS)

```
1: function BHCS(scheduling problem f)                ▷ returns schedule Ω
2:     Ω ← <>                                          ▷ empty schedule
3:     Ω' ← ∅                                          ▷ empty set of schedules
4:     for o_c in order do
5:         Ω' ← generateOptions(o_c, f, Ω)
6:         Ω ← selectHighestRanked(Ω')
       return Ω
```

and edges $a \in A$ represent constraints between those operations. The set of nodes N contains two dummy nodes, a source node s and a terminal node z that are connected to all operations with no predecessor and all operations with no successor respectively. Processing and setup times are positive weighted edges, while deadlines are negative weighted edges. A sample constraint graph is shown in Figure 2.2 for a 2-machine problem with 4 operations.

The longest path computation for all nodes in X from the source produces the earliest starting time of all operations est. This is because calculating the longest path from the source to a node tells us the smallest possible time that can elapse before the operation represented by the node is scheduled without violating any constraints.

Representing multi-machine scheduling problems as graphs and extracting the start times of operations by performing longest path computations is a known technique with the *disjunctive graph* (Roy and Sussmann 1964) being one of the most common representations (Błażewicz, Pesch, and Sterna 2000). However, we use the *constraint graph* for the rest of this thesis because it supports more temporal constraints than the disjunctive graph, particularly relative deadlines.

2.3. Contents of this Part of this Thesis

2.3.1. Research Goals

As discussed above, the advent of cyber-physical systems in the manufacturing context give us room to automate more and more decisions during the process. This gives any scheduler a broader view of the system whose operations it is in control of, i.e, it is more context-aware. In this part of the thesis, we look into the general problem of increasing the context-awareness of existing schedulers. This goal leads us to the following research questions:

Research Question 1. *How to design a general method to incorporate maintenance decisions in heuristic scheduling algorithms for our driver case?*

In many cases, the amount of deterioration a machine or system accumulates depends on historical usage. The maintenance required is in turn dependent on the deterioration state of the machine. Thus, there is a link between how the machine is

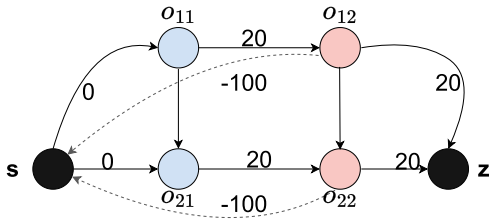

Figure 2.2.: Sample constraint graph for a 2-job 2-machine problem. Solid edges represent processing and setup time constraints while dotted edges represent deadline constraints. Deadlines are negated and their direction reversed. The source and terminals nodes are coloured black.

used and what maintenance is required. In cases where the duration of maintenance actions and average time till a maintenance action is required are on the same scale as duration of operations themselves, integrated production and maintenance scheduling is then necessary. However, many scheduling algorithms already exist with the most popular methods in industrial scheduling still being rule-based or heuristics. It is then useful to find out how to integrate maintenance planning into these algorithms without affecting their performance.

2.3.2. Outline and Contributions

The rest of this part of the thesis contains one main chapter (Chapter 3) where we propose a maintenance insertion algorithm for list-scheduling heuristics; answering the research question posed above.

3

Maintenance-Incorporated Heuristic Scheduling

We posit that anticipatory, integrated scheduling of operational and maintenance tasks leads to superior performance compared to purely 'wait-then-fix' handling of the maintenance tasks. Motivated by an industrial problem with (sequence-dependent) setup times, maximum separation constraints, and a combination of sequence- and time-dependent maintenance tasks, this chapter introduces an integer programming solution, a constraint programming solution and a heuristic solution based on list scheduling. The motivating use case provides a unique combination of concerns that is to the best of our knowledge, not yet studied in the literature. We build on existing work where we can by extending models for sequence-dependent maintenance scheduling to accommodate sequence- and time-dependent maintenance scheduling and also propose other new models. We show the relative performances of our methods through empirical evaluations and also show significant improvements – up to 25% reduction in makespan – when compared to a reactive scheduling approach that does not consider maintenance in its planning. Based on our evaluations on exact methods, constraint programming models scale better than mixed integer programming models for this problem.

Parts of this chapter have been published in IEEE Access Volume 11 (2023) (Eigbe, De Schutter, *et al.* 2023).

3.1. Introduction

We consider a sequence- and time-dependent maintenance scheduling problem. Our problem is motivated by an industrial use case of a large-scale printer (LSP) and is modelled as a flow shop. The operations in this problem have ordering constraints that enforce precedence and also maximum separation constraints that limit the delay between some of the operations. We also face setup time considerations. There are maintenance tasks which depend on the schedule: different sequences of operations have different deterioration effects on the machines. Additionally, the contribution of an operation to the total deterioration effect in a sequence is dependent on the timing of operations. Thus, our maintenance planning problem is both sequence- and time-dependent. The key question is how to handle both operational and maintenance tasks.

This is a challenging question because deteriorated machines produce low-quality jobs; there are thresholds beyond which deterioration of machines must be fixed by carrying out a maintenance activity. The overall objective is to find a feasible schedule that minimises the makespan.

Integrated production and maintenance planning is a challenge in many industries such as in wind farms (Kang and Guedes Soares 2020; Ren *et al.* 2021), in the capital goods industry (Chansombat, Pongcharoen, and Hicks 2019) and the pulp and paper industry (Avilés *et al.* 2023). In many cases, machine deterioration is dependent on use, i.e., the maintenance required depends on how production operations have been scheduled. Sometimes, this dependence can be ignored and solutions can focus on preventive or policy based maintenance (Gharoun, Hamid, and Torabi 2022; Netto *et al.* 2023; N. Zhang *et al.* 2023). Recent work in this direction has used reinforcement learning to come up with these policies (Valet *et al.* 2022). In other cases, the maintenance and production planning problem can be so integrated that the effect of use patterns on maintenance cannot be ignored. Previous work along these lines has considered different ways in which maintenance planning is integrated with production planning such as time-dependent maintenance (Gawiejnowicz 2020) and position-dependent maintenance (W. Liu, X. Wang, *et al.* 2022; Yang 2010). The literature also considers different models of maintenance activities with one of the most popular models being that maintenance activities affect the processing time of operations (Mor and Mosheiov 2012) . While similar problems have been tackled in the literature, our work deals with a unique combination of maximum separation constraints and a deterioration effect not on the processing times of jobs but on the quality of work produced.

We present three solutions to this problem namely, (i) a mixed integer programming solution, (ii) a constraint programming solution, and (iii) a list-scheduling based heuristic solution that extends the capabilities of existing schedulers to handle the kind of maintenance activities presented in this problem.

Through empirical evaluations, we show that in comparison to the reactive approach of scheduling only production operations and then performing maintenance activities when deterioration thresholds are crossed during a production run, our proactive approach achieves significant improvements in the makespan.

This chapter is organised as follows: Section 3.2 discusses related work, Section 3.3

provides the background and problem definition, Sections 3.4, 3.5 and 3.6 present mixed integer programming, constraint programming and heuristic solutions respectively. We perform empirical evaluations in Section 3.7 and Section 3.8 concludes the chapter.

3.2. Related Work

The literature has investigated the dynamic relationship between machine deterioration and production scheduling from multiple angles ranging from ways to accurately determine the deterioration of a machine (Su *et al.* 2006; Susto *et al.* 2014) to actually generating schedules. We group the research themes in this field based on two categories, namely; (i) the way deterioration is modelled and (ii) the way maintenance activities are modelled.

Based on the *deterioration model*, existing research can be split into three main categories or approaches (Delorme, Iori, and Mendes 2021). The *time-dependent* approach relates deterioration to the time at which a job is scheduled, i.e., scheduling a job later in the schedule incurs some additional deterioration which typically leads to longer processing times compared to scheduling it earlier. Closely related to this is a *position-dependent* approach, where deterioration effect of an operation is dependent on the number of preceding completed operations. Finally, there is the *sequence-dependent* approach in which the deterioration depends on the ordering or sequence of the preceding operations on the machine. As a result of the industrial challenge addressed in this chapter, we focus on the sequence-dependent case with an additional challenge that the deterioration effect of an operation on a machine is not known apriori and is itself time-dependent.

The survey of Gawiejnowicz (2020) into the state of time-dependent scheduling problems has shown that the problem has been studied for single machine, parallel machine and dedicated machine use cases with a wide range of solution methods. However, situations where time-dependence of maintenance activities is coupled with sequence-dependence are unaddressed.

Yang (2011) considers the position-dependent maintenance scheduling problem on a set of parallel machines assuming that machines can only be maintained once within the planning horizon and with a constant maintenance duration. Yang (2010), L. Jin, Yu, and Dong (2018) and W. Liu, X. Wang, *et al.* (2022) all consider position-dependent maintenance on a single machine with varying considerations such as the impact of time-dependent improvements in machine conditions, constraining job processing times to lie within an interval, and a combination of time and position-dependent deterioration respectively. Mor and Mosheiov (2012) and Zhu *et al.* (2016) also consider the position dependent case but both add due-window considerations for just-in-time scheduling considerations.

The *sequence-dependent* approach is a more recent addition to the literature and can be considered as a generalisation of the time- and position-dependent approaches. Notably, Ruiz-Torres, Paletta, and Pérez (2013) and Ruiz-Torres, Paletta, and M'Hallah (2017) study sequence-dependent deterioration on a set of parallel machines without and with maintenance activities respectively. Santos and Arroyo

(2017) consider iterated greedy heuristics for a similar problem and Ding *et al.* (2019) considers the case where the parallel machines are not identical and processing time is based on a combination of deterioration and the speed of the assigned machine. Recently, Delorme, Iori, and Mendes (2021) explored multiple integer programming models for solving the sequence-dependent maintenance problem on parallel machines and provided a heuristic approach for larger instances. The combination of sequence-dependent maintenance with other approaches and its effect in more complex manufacturing systems has not yet been studied.

Based on the *model of maintenance activities*, there are also different approaches in the literature. Some works such as (Ruiz-Torres, Paletta, and Pérez 2013) do not consider the presence of maintenance activities at all and aim to schedule in a way that deterioration is minimized. Other works such as (Delorme, Iori, and Mendes 2021) consider maintenance activities that reset the status of a machine to full health or 0% deterioration while a third category (Mor and Mosheiov 2012) considers rate-modifying maintenance activities that restore machine health by modifying the rate such that machines are able to perform work faster after maintenance. Xanthopoulos *et al.* (2017) and Y. Wang, Elahi, and Xu (2019), classify maintenance activities into those that completely reset the state of the machines and those that restore the machines to some better deterioration state only. Additionally, the maintenance activities can be of fixed duration or can also have varying types based on how deteriorated a machine is.

A core assumption in many scenarios is that deterioration makes machines slower, thus increasing processing times of operations. Our work differs fundamentally in this regard in that using deteriorated machines does not have an effect on processing times, but instead affects the quality of the jobs produced. Our problem defines deterioration thresholds beyond which maintenance activities must be carried out to meet the quality requirements of future jobs. We also consider the case of maintenance activities that reset the state of the machine but also consider that there exist different classes of maintenance activities each with their own deterioration thresholds and incurring different costs.

An additional complication in our problem is the presence of maximum separation constraints, which impose additional feasibility requirements on the problem. Exact solutions are able to easily model these additional requirements but heuristics run the risk of generating infeasible solutions in some cases. We therefore consider it necessary to design a solution for schedules that become infeasible due to the incorporation of maintenance activities. This concept of re-organising or repairing a changed schedule has been studied with various heuristics such as left and right shift (Abumaizar and Svestka 1997; Kutanoglu and Sabuncuoglu 2001). Chan and Wee (2003) combine multiple of these heuristics and a genetic schedule repair algorithm to build a solution that caters to multiple classes of schedule disturbances in a prefabrication plant.

In the context of flow shops, an example of schedule repair algorithms can be found in (Allahverdi 1996) which considers re-scheduling in a two-machine flow shop where schedules are disrupted by machine breakdowns. Additionally, Caricato and Grieco (2008) consider re-scheduling due to inserting new jobs in already planned

schedules and Katragjini, Vallada, and Ruiz (2015) consider re-scheduling due to a wider range of disruptions in flow shop schedules at runtime. These cases all consider unexpected interruptions and do not have the combination of precedence and maximum separation constraints which provide an additional challenge for our problem.

In summary, there is a gap in the literature for sequence-dependent maintenance scheduling where deterioration effects of operations are not known apriori but are themselves time-dependent. The particular industrial challenge we consider has additional requirements of maximum separation that add to the complexity of the problem. Further, the schedule repair that is needed for heuristic schedulers that may produce infeasible schedules when we introduce maintenance activities, also requires new techniques.

3.3. Problem Definition

We consider a *maintenance-aware re-entrant flow shop with setup times and relative due dates* inspired by an industrial use case of a large-scale printer (LSP). The LSP prints different types of duplex sheets that need to be processed twice by the same print head at a speed of 100 or more pages per minute. In this setting, jobs to be scheduled refer to sheets to be printed.

In three-field or Graham notation (Graham *et al.* 1979), the base problem without maintenance is defined as $F|s_i, s_{ij}, limited-wait|C_{\max}$ indicating that it is a flow shop with both sequence-dependent and independent setup times, with maximum

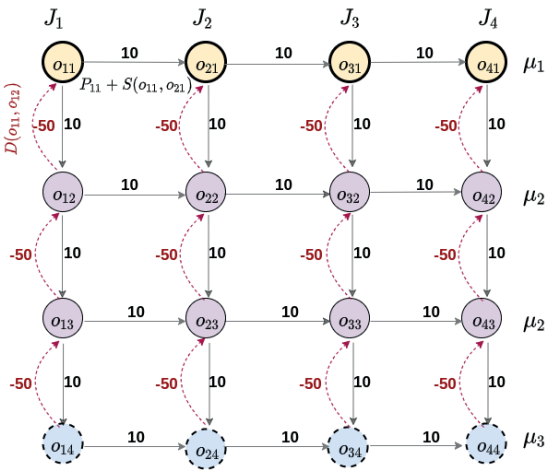

Figure 3.1.: Sample re-entrant flow shop where the operations are represented by circles. Column-wise, we have operations of the same job and row-wise, we have operations on the same machine with one of these being the re-entrant machine that appears on rows 2 and 3. Operations with the same colour or boundary lines are mapped to the same machine. Setup times and maximum separation constraints are shown by solid and dashed edges respectively.

separation constraints between operations of the same job also known as limited-wait constraints, and with an objective to minimise makespan C_{max}. There is no preemption allowed and all jobs are released at time 0.

We represent the n-job m-machine maintenance-aware problem as the tuple $(M, J, O, P, S, D, \delta, X, O^M)$ where $M = \{\mu_1, \ldots, \mu_m\}$ is the set of machines and $J = \langle J_1, \ldots, J_n \rangle$ is the sequence of jobs. The set O represents the set of operations for every job $j_i \in J$ where each operation o_{ij} has a processing time P_{ij}. Each job has the same number of r operations as is in a standard flow shop. Moreover, $S : O \times O \to \mathbb{R}_{\geq 0}$ refers to setup times, which represent the required delay between the completion of an operation and the start of another operation. Setup times can exist between operations of the same job to model travelling time of a job for instance, or between operations on the same machine to model any machine preparation step that is needed between operations. Operations of the same job also have maximum separation constraints between them represented as $D : O \times O \to \mathbb{R}_{>0}$, i.e., the maximum delay between the start times of two consecutive operations of the same job. Such constraints model the fact that operations of a job can often not be delayed indefinitely due to physical constraints in the plant like the buffer size. In a situation where such constraints do not apply, separation constraints can simply be set to infinity and setup times to zero.

The solution to the problem is a schedule Ω, i.e., a sequence of both maintenance activities and production operations where each production operation is assigned a start time such that ω_{ij} represents the start time of operation o_{ij}.

In addition to being a flow shop with the above properties, we have a situation with re-entrancy such that the sequence of machines for each job is $\langle \mu_1, \ldots, \mu_k, \mu_k, \ldots, \mu_m \rangle$, i.e., there is one re-entrant machine that all jobs go through twice. Operations on the re-entrant machines are referred to as *first and second pass operations*. Re-entrancy occurs in many production processes, e.g., semi-conductor production, automated painting, etc. Simple re-entrant setups have been shown to be NP-hard (Emmons, Vairaktarakis, *et al.* 2013; M. Y. Wang, Sethi, and van de Velde 1997). Our motivating use case of production printing has a twice re-entrant setup arising from duplex printing.

We further have the constraints that (i) jobs are not allowed to overtake each other, (ii) the required completion order of jobs is the same as the index of the jobs, and (iii) all setup times and due date constraints are hard constraints that must be obeyed. This situation means that the only scheduling freedom is in the sequence of operations on the re-entrant machine, i.e., first and second passes of the same jobs do not necessarily have to follow each other on this machine. This means we can also think of this as a single machine scheduling problem with precedence and maximum separation constraints.

In the same vein as only needing to schedule the re-entrant machine, we limit our maintenance planning to maintaining the re-entrant machine. While other machines also require maintenance, only the re-entrant machine requires maintenance in the same time scale as the operations being carried out, creating a very tightly coupled problem compared to maintenance of other machines.

Deterioration model. In our motivating industrial problem, there is a deterioration model $\delta : \Omega \to \mathbb{R}_{\geq 0}$, that maps a scheduled sequence of operations on a machine to a deterioration state, i.e., given a sequence of operations on a machine with their corresponding start times, i.e., a schedule, $\delta : \Omega \to \mathbb{R}_{\geq 0}$ informs us of the machine state at the end of the sequence. Here, δ is both sequence- and time-dependent in the sense that deterioration is measured by *idle time* of a machine part, i.e., the longer a machine part has been left idle, the more deteriorated it is. These idle times follow directly not only from the sequence themselves, but also from the assigned start times of operations in these sequences. We do not explicitly model machine parts and instead depend on the fact that different types of jobs use different machine parts and so it can be inferred which machine parts have been idle based on how long it has been since a certain job type has been scheduled. We assume that there is a set of job types $T = \{\tau_1, \ldots, \tau_n\}$ that can be presented to the machine and that there is a lexicographic ordering of job types such that every set of machine parts used by a job type τ_x is contained in the set of machine parts used by a job type $\tau_{y>x}$. It then follows that at the start of an operation of type τ_x, idle time is reset to 0 for all operations of type $\tau_{y\leq x}$. Note that while there could be other kinds of problems where different jobs use completely different machine parts and such a lexicographic ordering of types is not possible, it is still a realistic assumption for many scenarios, e.g., scenarios where jobs come in different sizes and bigger sizes simply use more machine parts for production or scenarios where jobs can be customised with different add-on properties processed by additional machine parts.

Finally, we also take as input a maintenance policy X. In our problem, the policy has a set of maintenance activity classes C. For every class $c \in C$, there is a corresponding maintenance duration P_c similar to processing times of production operations. The maintenance policy further maps intervals of deterioration values $[\theta_c, \Theta_c)$ to classes of maintenance activities such that whenever the deterioration falls in $[\theta_c, \Theta_c)$, a maintenance activity of at least class c is required before further production. Thus, $[\theta_c, \Theta_c)$ defines the interval of deterioration thresholds for a maintenance activity. We assume that these intervals are non-overlapping and that maintenance activities triggered by higher thresholds, i.e., harsher deterioration, are more intense and require longer durations. The deterioration thresholds serve to capture the limits at which the quality of a job would be too low if production carries on without a maintenance activity. In this context, a low-quality job refers to a poor print typically with colours bleeding into each other, blurry prints or unintended lines running across a page.

An example problem is shown in Figure 3.1. The problem is represented as a constraint graph where the due dates and setup times are treated as a system of difference constraints. Operations are represented as circles and each column of operations belong to one job, while each row of operations are mapped to the same machine. Solid arrows represent the minimum separation between operations and are made of the sum of processing and setup times while dashed arrows represent the maximum separation between operations and can be thought of as relative due dates. Minimum separation edges are represented with positive values and maximum separation edges are represented with negative values as they connote *at least* and *at*

most constraints on the difference between the start times of the operations they connect.

3.4. Integer Programming Approach

Mixed Integer Programming (MIP) is one of the most popular exact solving paradigms and has been applied to other maintenance planning problems in the literature (Delorme, Iori, and Mendes 2021; Hnaien *et al.* 2016) with some success. Due to the existence of a wide variety of commercial solvers, mixed integer formulations of a problem are valuable as solutions can be provided by these solvers. Furthermore, MIP models can give additional insight to the structure of a problem. Thus, we also consider such a solution for our problem.

In this section we present an exact MIP model for this problem. The model uses the concept of event based formulation as introduced by Delorme, Iori, and Mendes (2021) and extends this concept to accommodate the kind of maintenance policies in this problem. The key idea here is the notion of *blocks* where a block is defined as a sequence of operations uninterrupted by a maintenance activity, i.e., a block is a sequence of operations separated from other operations in the sequence by at least one maintenance activity. For our model, we extend this idea to also include the effect of job types. A block is then defined as a sequence of operations uninterrupted by either a maintenance activity or an operation with a higher type than any other operation within the block.

We define binary variables to mark whether an operation starts a block or not. These variables are further indexed by job type and maintenance activity class, i.e., an operation can be the start of a block delineated by a class of maintenance activities and/or the start of a block delineated by a job type. Blocks of different types are allowed to overlap with additional constraints added to ensure that maintenance is not triggered more than necessary.

The model retains all variables defined in the problem definition in Section 3.3. Indices of variables corresponding to operations are either of the form x_{ij} when both the job and operation identifier are important or of the form x_a when it is only necessary to differentiate one operation from the other. For ease of modelling, we also define binary variables $\Gamma_{am}, \quad \forall o_a \in O, \mu_m \in M$ to represent machine assignment. Then, Γ_{am} is set to 1 if operation o_a is assigned to machine μ_m. The binary variable Γ_{am} is not a decision variable and is part of the problem description.

Additionally, since we only plan maintenance on the re-entrant machines μ_k, many constraints only apply to operations on this machine and are denoted as R such that $R = \{o_a \in O | \Gamma_{ak} > 0\}$. Furthermore, a dummy operation o_{dummy} of processing time 0 is defined and constrained to be the first operation on each machine. We also extend the use of a job type τ to serve as a function that returns the type of an operation when written as $\tau(o)$.

The following additional variables are developed for the MIP model: ω_{ij} refers to the start time of operation $o_{ij} \in O$, B_{ab} is a binary variable relating to the precedence constraints between operations o_a and o_b. Note that B_{ab} refers only to direct precedence and not the general notion of o_a being scheduled sometime before

o_b. We discretize job types such that τ_a refers to the type of o_a and assume that there is a lexicographic ordering of job types such that processing a job type with a higher value is sufficient to reset the machine for lower job types according to the maintenance policy described in Section 3.3.

Block starts are marked by binary variables Z_a^c and ζ_a^τ where Z_a^c determines if operation o_a starts a block of operations delineated by a maintenance activity of class c and ζ_a^τ determines if operation o_a starts a block of operations delineated by a job type τ. Idle time values are held by the variables K_a^c and L_a^τ, which correspond to the minimum time elapsed since a maintenance activity of class c preceding o_a and the minimum time elapsed since an operation of type τ preceding o_a respectively. Furthermore, deterioration values at the start of an operation o_a are held by the variable δ_a and are determined by the deterioration values K and L.

Some of the constraints are linearised using big-M variables namely, $\mathcal{M}^\tau$ and $\mathcal{M}^\omega$. We define some bounds for these variables in Section 3.4.1 below.

The integer programming model

In this section, we define the integer programming model made up of an objective function, decision variables and constraints.

Objective

$$\min(C_{max}) \tag{3.1}$$

Decision variables

B_{ab}	Operation o_a directly precedes o_b	$B_{ab} \in \{0,1\}$
ω_a	Start time of operation o_a	$\theta_a \in \mathbb{R}$
L_a^τ	Minimum time elapsed since operation of a type τ preceding o_a	$L_a \in \mathbb{R}$
K_a	Minimum time elapsed since any maintenance activity preceding o_a	$K_a \in \mathbb{R}$
δ_a	Deterioration of machine at start of o_a	$\delta_a \in \mathbb{R}$
Z_a^c	o_a starts a block delineated by a maintenance activity of class c	$Z_a^c \in \{0,1\}$
ζ_a^τ	o_a starts a block delineated by a job of type τ	$Z_a^c \in \{0,1\}$

Constraints

$$\omega_{(i+1)j} \geq \omega_{ij} \quad \forall o_{ij} \in O \tag{3.2a}$$
$$\omega_{i(j+1)} \geq \omega_{ij} + P_{ij} + S(o_{ij}, o_{i(j+1)}) \quad \forall o_{ij} \in O \tag{3.2b}$$
$$\omega_{i(j+1)} \leq \omega_{ij} + D(o_{ij}, o_{i(j+1)}) \quad \forall o_{ij} \in O \tag{3.2c}$$
$$\sum_{o_a \in O} B_{ad} = 0 \tag{3.2d}$$
$$\sum_{o_a \in O \cup \{o_{\text{dummy}}\}} B_{ab} = 1 \quad \forall o_b \in O \tag{3.2e}$$
$$\sum_{o_b \in O} B_{ab} \leq 1 \quad \forall o_a \in O \tag{3.2f}$$
$$B_{ab} \leq \sum_{\mu_m \in \mu} \Gamma_{am}\Gamma_{bm} \quad \forall o_a, o_b \in O \tag{3.2g}$$
$$\omega_b \geq B_{ab}(\omega_a + P_a + S(o_a, o_b)) \quad \forall o_a, o_b \in O \tag{3.2h}$$
$$\omega_b \geq B_{ab}(\omega_a + P_a + Z_b^c(P_c)) \quad \forall o_a, o_b \in O, c \in C \tag{3.2i}$$
$$L_b^\tau \geq B_{ab}(\omega_b - \omega_a - P_a) \quad \forall o_a, o_b \in R, \tau \in T \tag{3.2j}$$
$$L_b^\tau \geq B_{ab}(L_a^\tau + \omega_b - \omega_a) - \mathcal{M}^\omega \zeta_b^\tau \quad \forall o_a, o_b \in R, \tau \in T \tag{3.2k}$$
$$K_a \geq 0 \quad \forall o_a \in R \tag{3.2l}$$
$$K_b \geq B_{ab}(K_a + \omega_b - \omega_a) - \mathcal{M}^\omega \sum_c Z_b^c \quad \forall o_a, o_b \in R \tag{3.2m}$$
$$\delta_a \geq \min(K_a, L_a^{\tau(o_a)}) \quad \forall o_a \in R \tag{3.2n}$$
$$\mathcal{M}^\omega(1 - Z_a^c) + (\delta_a - \theta_c) \geq 0 \quad \forall o_a \in R, c \in C \tag{3.2o}$$
$$\mathcal{M}^\omega Z_a^c - (\delta_a - \theta_c) > 0 \quad \forall o_a \in R, c \in C \tag{3.2p}$$
$$\mathcal{M}^\tau \zeta_b^\tau - B_{ab}(\tau(o_a) - \tau) \geq 0 \quad \forall o_a, o_b \in R, \tau \in T \tag{3.2q}$$
$$\mathcal{M}^\tau(1 - \zeta_b^\tau) + B_{ab}(\tau(o_a) - \tau) \geq 0 \quad \forall o_a, o_b \in R, \tau \in T \tag{3.2r}$$
$$C_{max} = \omega_{|J|r} + P_{|J|r} \tag{3.2s}$$

The objective of the model is to minimise makespan denoted by Equation (3.1). The constraints in Equations (3.2b) to (3.2h) apply to all operations while the constraints in Equations (3.2j) to (3.2r) only apply to operations scheduled on the re-entrant machine. All non-binary variables are constrained to be non-negative, i.e., start times, idle times and deterioration values all have a lower bound of 0.

Equation (3.2a) enforces the fixed-order relationship between operations at the same level of the flow shop. Equations (3.2b) and (3.2c) enforce setup times and maximum separation constraints between operations of the same job respectively, while Equation (3.2d) enforces that the dummy operation has no predecessors. Equation (3.2e) ensures that every operation has exactly one predecessor and Equation (3.2f) enforces that every operation has at most one successor. Equation (3.2g) enforces that operations only follow each other if they are mapped to the same machine and Equation (3.2h) enforces that there is no overlap between operations leaving room for setup times. These make up the constraints that specify the problem without maintenance.

The maintenance constraints follow below. Equation (3.2i) enforces that there is no overlap between operations while leaving enough room for any maintenance activities that may have been triggered. Equations (3.2j) and (3.2k) specify constraints on the minimum time elapsed since an operation of a certain type has come through the machine. Similarly, Equations (3.2l) and (3.2m) specify the minimum time elapsed since a maintenance activity of a certain class has been scheduled. The constraints represented by Equations (3.2j) to (3.2m) are defined in a cumulative way based on predecessor operations. Equations (3.2k) and (3.2m) are activated depending on the presence of a job type or a maintenance activity respectively. This toggle is implemented by big-M values that are activated based on the binary variables Z and ζ.

The actual deterioration value is computed by Equation (3.2n) which is set to the minimum of both K and L. Equation (3.2n) computes deterioration based on the idle time so far and is set to the minimum of K and L so that maintenance is only triggered when necessary. Equations (3.2o) and (3.2p) specify that maintenance activities are triggered whenever deterioration thresholds are crossed thereby starting a new block while Equations (3.2q) and (3.2r) similarly start a new block based on the relationship between types of operations, i..e, a new block is triggered whenever an operations predecessor has a higher type. Note that this model allows multiple maintenance classes to be triggered simultaneously if the threshold violations cross multiple thresholds. However, Equation (3.2h) means that the gap left for maintenance corresponds to the largest processing time of all triggered maintenance activities, thus not paying unnecessary maintenance costs.

Finally, Equation (3.2s) calculates the makespan which in a fixed order problem, is the finishing time of the last operation of the last job.

3.4.1. Bounds for Big-M Values

$\mathcal{M}^{\omega}$

Throughout the model, $\mathcal{M}^{\omega}$ is used as a big-M constraint in two instances. The first is in Equations (3.2k) and (3.2m) to sum up the minimum times since the last maintenance or the last occurrence of a type of job and in Equations (3.2o) and (3.2p) to toggle maintenance if deterioration thresholds are crossed. In each of these cases, the upper bound is the maximum possible deterioration value that can occur. Because our deterioration deals with idle times, we are then looking for a value that is larger than or equal to the maximum time the machine can be left idle.

An idea for this bound is to use an upper bound on the makespan as there always exists a solution with a better makespan than one in which the machine is left idle for the upper bound on the makespan.

This upper bound assumes the worst case which is that every operations incurs the maximum possible setup time and the maintenance activity with the longest duration occurs before every operation. Thus, the bound is

$$\mathcal{M}^{\omega} = \sum_{o_a \in O} \left(P_a + \max_{c \in C}(P_c) + \max_{o_b \in O}(S(o_a, o_b)) \right). \tag{3.3}$$

$\mathcal{M}^\tau$

The tightest bound for $\mathcal{M}^\tau$ is the largest job type available in the problem. This holds because:

- $\mathcal{M}^\tau$ is an upper bound on the types of jobs,
- we assume that job types are all given integer values corresponding to their quality requirements,
- we assume there is a lexicographical ordering of these types that corresponds with the order of the integers representing each job type.

3.4.2. Linearising the Model

Some equations are still quadratic namely Equations (3.2h), (3.2i), (3.2k) and (3.2m). They however all involve the product of a binary variable and a non-negative continuous variable and can also be linearised via the big-M method. The corresponding M is also $\mathcal{M}^\omega$. A detailed explanation of how this linearisation is achieved can be found in (H. P. Williams 2013).

Additionally, Equation (3.2n) requires us to compute the minimum which is also a non-linear equation. We define one more auxilliary binary variable γ_a that is set to 1 if K_a is less than $L_a^{\tau(o_a)}$ and linearise the minimum constraint by replacing it with the following set of equations:

$$\delta_a \leq K_a \quad \forall o_a \in R, \tag{3.4a}$$

$$\delta_a \leq L_a^{\tau(o_a)} \quad \forall o_a \in R, \tag{3.4b}$$

$$\delta_a \geq K_a - \mathcal{M}^\omega(1-\gamma_a) \quad \forall o_a \in R, \tag{3.4c}$$

$$\delta_a \geq L_a^{\tau(o_a)} - \mathcal{M}^\omega(\gamma_a) \quad \forall o_a \in R, \tag{3.4d}$$

$$K_a - L_a^{\tau(o_a)} \leq \mathcal{M}^\omega(1-\gamma_a) \quad \forall o_a \in R, \tag{3.4e}$$

$$L_a^{\tau(o_a)} - K_a \leq \mathcal{M}^\omega(\gamma_a) \quad \forall o_a \in R. \tag{3.4f}$$

Equations (3.4a) and (3.4b) set the deterioration value δ_a to be upper bounded by the minimum of K_a and $L_a^{\tau(o_a)}$. However, this is not enough as δ_a is still free to take any values less than this and can lead to violations of maintenance constraints. We further use Equations (3.4c) and (3.4d) which set δ_a to be lower bounded by the minimum of K_a and $L_a^{\tau(o_a)}$. This lower bound is also achieved via big-M constraints which activate either equation based on γ_a. The combination of the lower and upper bounds ensure that δ_a is exactly set to the minimum of the two values K_a and $L_a^{\tau(o_a)}$. Finally, Equations (3.4e) and (3.4f) set γ_a to 0 if K_a is less than $L_a^{\tau(o_a)}$ and 1 otherwise.

3.5. Constraint Programming Approach

Constraint programming (CP) has recently been shown to perform well for scheduling problems (Laborie *et al.* 2018). This motivates us to also explore a constraint programming solution. In this section, we present a CP model.

Our CP model uses the idea of *interval variables* and *sequence variables*. These are known constraint programming concepts (Laborie *et al.* 2018) with the following definitions. Interval variables refer to operations to be scheduled and are declared with a length equal to the processing time of the operation. The goal of the solver is to assign a start time to each of these variables. An additional characteristic of interval variables are that they have the option to either be compulsory, i.e., they must exist in any schedule produced by the solver, or be optional. Sequence variables on the other hand, represent orderings of interval variables. The solver receives these as a set of interval variables with its goal being to decide on a sequencing of these interval variables.

Apart from constraints and variables, constraint programming also provides some auxiliary functions such as startOf and typeOf which help us access variable properties – in this case, their assigned start times and types respectively.

We define two classes of interval variables, (i) operations which are always present and each retain the representation of o_a and (ii) maintenance activities which are optional and referred to as m_a^c where m_a^c is a maintenance activity of class c that precedes an operation o_a. The variables K, L and R retain their definitions from the MIP model in Section 3.4.

Given that we have $|A|$ maintenance classes and $|R|$ operations in total on the re-entrant machine, we define $|A||R|$ maintenance activities since the worst case is that there is one maintenance activity of a class before every regular operation. We add constraints such that the maintenance activities are included in the sequence only when deterioration thresholds are violated.

Sequence variables are defined per machine and referenced as $Sequence_m$ for corresponding machine μ_m where $Sequence_m$ contains all operations mapped to μ_m including the optional maintenance activities. For the re-entrant machine, we define an additional sequence variable $SequencePlain_m$ as a sequence of only production operations – excluding maintenance activities – and constrain this sequence to follow the same ordering as $Sequence_m$. The purpose of this duplicate sequence is to ensure that sequence-dependent setup times are respected. The details of how we achieve this follow in the constraint definitions below.

Objective

$$\min(C_{max}) \tag{3.5}$$

Decision variables

$Sequence_m$	Sequence of production and maintenance operations on machine μ_m	
$SequencePlain_m$	Sequence of production operations on machine μ_m	
L_a^τ	Minimum time elapsed since operation of a type τ preceding o_a	$L_a \in \mathbb{R}$
K_a	Minimum time elapsed since any maintenance activity preceding o_a	$K_a \in \mathbb{R}$

Constraints

$$C1: \text{before}(Sequence_1, o_{i1}, o_{(i+1)1}) \forall o_{i1} \in O \tag{C1}$$

$$C2: \text{sameSubsequence}(Sequence_1, Sequence_m) \ \forall \mu_m \in \mu \tag{C2}$$

$$C3: \text{sameSubsequence}(Sequence_m, SequencePlain_m) \text{ for re-entrant machine } \mu_k \tag{C3}$$

$$C4: \text{startOf}(o_{i(j+1)}) \geq \text{startOf}(o_{ij}) + P_{ij} + S(o_{ij}, o_{i(j+1)}) \ \forall o_{ij} \in O \tag{C4}$$

$$C5: \text{startOf}(o_{i(j+1)}) \leq \text{startOf}(o_{ij}) + D(o_{ij}, o_{i(j+1)}) \ \forall o_{ij} \in O \tag{C5}$$

$$C6: \text{noOverlapDirect}(Sequence_m, P, S) \ \forall \mu_m \in \mu \tag{C6}$$

$$C7: \text{noOverlapDirect}(SequencePlain_k, P, S) \text{ for re-entrant machine } \mu_k \tag{C7}$$

$$C8: \text{if}(\min(K_a, L_a^{\tau(o_a)}) \geq \theta^c \wedge \min(K_a, L_a^{\tau(o_a)}) < \Theta^c) \Rightarrow \ \text{presenceOf}(m_a^c) = 1 \ \forall o_a \in R, c \in C \tag{C8}$$

$$C9: \text{if}(\text{presenceOf}(m_a^c) = 1) \Rightarrow K_a = 0 \ \forall o_a \in R, c \in C \tag{C9}$$

$$C10: \text{if}(\text{presenceOf}(m_a^c) = 0) \Rightarrow \ K_a = K_{\text{indexOfPrev}(O_a)} + \text{startOf}(O_a) - \text{startOfPrev}(O_a) \quad \forall o_a \in R, c \in C \tag{C10}$$

$$C11: \text{if}(\text{typeOfPrev}(o_a) \geq \text{typeOf}(o_a)) \Rightarrow \ L_a^\tau = \text{startOf}(o_a) - \text{endOfPrev}(O_a) \quad \forall o_a \in R, \tau \in T \tag{C11}$$

$$C12: \text{if}(\text{typeOfPrev}(o_a) < \text{typeOf}(o_a)) \Rightarrow \ L_a^\tau = L^\tau_{\text{indexOfPrev}(o_a)} + \text{startOf}(o_a) - \text{startOfPrev}(o_a) \ \forall o_a \in R, \tau \in T \tag{C12}$$

$$C13: C_{max} = \text{endOf}(o_{|j|r}) \tag{C13}$$

In this model, Constraint C1 enforces an ordering between the first operations of each job. The before constraint enforces precedence relationships between two operations in a sequence. We enforce the order of the first operations of each job which we know will be on the first machine (as we have a flow shop). Constraint C2 builds on C1 to then enforce that this ordering is respected across all other sequences using the sameSubsequence constraint. Note that we only enforce a subsequence because the re-entrant machine has operations on multiple levels of the flow shop. The sameSubsequence is set up such that only operations at the same level are constrained with the fixed ordering, which is in line with the requirements of our problem. Constraint C3 uses the sameSubsequence in a similar way to constrain the duplicate sequences – with and without maintenance activities included – to have the same ordering.

Next, Constraints C4 and C5 enforce the sequence-independent setup times and maximum separation constraints respectively. Both of these apply to operations of the same job as is seen with the index of operations in the constraints.

Sequence-dependent setup time and no overlap constraints are handled by Constraints C6 and C7, which ensure that both the separations required by processing times and sequence-dependent setup times are obeyed. Since maintenance activities are also included in our sequences, we ensure correctness of Constraints C6 by extending the processing and setup times accordingly with setup times set to 0 for operations before or after maintenance activities. The noOverlapDirect constraint works such that the separation denoted by sequence-dependent setup times applies only between direct successors, i.e., say an operation o_a is followed by o_b with a

maintenance activity m_b^c in-between, the setup time between o_a and o_b will not be enforced. Thus, setting the maintenance setup time to 0 can lead to constraint violations as the problem is now under constrained. It is worthy of note that there exists a noOverlapIndirect constraint, which applies sequence dependent setup time constraints to all successors; however, this over-constrains the problem[1]. We use the noOverlapDirect constraint and circumvent under-constraining the problem by using the supporting Constraint C7 on a duplicate sequence without maintenance.

Constraint C8 enforces the presence of a maintenance activity whenever the minimum deterioration is within the limits of threshold violations. We do not explicitly calculate a deterioration variable δ in this model but this is essentially the left hand side of Constraint C8. We depend on the fact that our problem defines non-overlapping maintenance threshold intervals to ensure that at most one maintenance activity is triggered before an operation.

Constraints C9 and C10 deal with the computation of the minimum time elapsed since a maintenance activity has occurred. Since we are guaranteed to trigger at most one maintenance activity per operation, we do not maintain different minimum elapsed times per maintenance class as was done in the MIP model. Similarly, Constraints C11 and C12 compute the minimum time elapsed since a job of a certain type has been through the machine.

Finally, C13 calculates the makespan, which we again know to be the finishing time of the last operation of the last job.

Worthy of note is that Constraints C10 and C12 are cumulative constraints that could be expressed using the cumulFunction constraint, which keeps track of each interval variables contribution to a function (Laborie *et al.* 2018). However, many implementations of this function within available solvers require that the contribution of each interval variable be known apriori whereas, in our case, the contribution of each interval variable is itself based on decision variables (Laborie *et al.* 2018) due to maintenance also being time-dependent[2].

3.6. Heuristic Solution Approach

While exact approaches such as those presented in Sections 3.4 and 3.5 have lots of advantages, they often do not scale well. In this section, we present an alternate heuristic solution approach to handle larger problem instances.

Our heuristic approach is based on extending list schedulers to integrate maintenance activities in the schedule. Heuristic list schedulers have been developed for the industrial problem we consider (van der Tempel *et al.* 2018; van Pinxten *et al.* 2017; Waqas *et al.* 2015) and are also suitable for online scheduling. Thus, we look into extending them to handle integrated production and maintenance scheduling. The typical flow of a list scheduler is to $order$ operations according to some metric and insert them in a schedule one after the other until all operations are scheduled

[1] Given a sequence of operations $o_a \rightarrow o_b \rightarrow o_c$, sequence-dependent setup times will be considered from $o_a \rightarrow o_b$, $o_b \rightarrow o_c$ and $o_a \rightarrow o_c$ whereas the only sequence-dependent setup times that should be considered are from $o_a \rightarrow o_b$ and $o_b \rightarrow o_c$.

[2] Start times of operations are decision variables.

(van der Tempel *et al.* 2018).

3.6.1. Maintenance-aware List Scheduling

To make a list scheduling approach maintenance-aware, we propose to evaluate the effect of any operation placement on maintenance triggering before making a decision. This leads to a schedule with the necessary maintenance activities triggered by the operation sequence already included. This is shown in Algorithm 2. In Line 1, the scheduler takes as input the flow shop to be scheduled, the chosen ordering of the operations $order$, and the ranking of decisions $rank$. Lines 2–6 initialise the variables used in the algorithm, i.e., an empty schedule Ω that is filled with operations by the algorithm, empty sets of schedules Ω' and Ω'' used to keep track of scheduling options, and an operation o_p to track the last operation that was inserted in the schedule. Specifically, o_p is initialised to a dummy operation for the first run where no insertions have occurred yet. In Line 7, the scheduler loops through each operation o_c in the chosen order and Line 8 finds positions to place the operation in the schedule being built with each possible option resulting in a different schedule stored in the set Ω'. For every one of these schedules, we trigger predicted maintenance in Line 10, which updates the schedules with predicted maintenance activities included. We keep track of the last regular operation placed in the schedule o_p to reduce the amount of work it takes to trigger maintenance as the schedule is already evaluated up to that operation o_p. Eventually, we pick the best option in Line 12 where the 'best' is as determined by the supplied ranking $rank$.

The steps shown in Algorithm 2 are generic and can be customised to any list scheduler of choice. However, evaluating maintenance is performed according to the steps described in Algorithm 3. For a given schedule, we first go through the operations in the schedule from the last inserted operation o_p to the current operation being inserted o_c in Line 2. For each operation, we evaluate the deterioration state in

Algorithm 2 Maintenance Aware List Scheduling (MALS)

```
1:  function MALS(flow shop f, operation ordering order, ranking rank)  ▷ returns schedule Ω
2:      Ω ← <>                                                          ▷ empty schedule
3:      Ω' ← ∅                                                          ▷ empty set of schedules
4:      Ω'' ← ∅                                                         ▷ empty set of schedules
5:      o_p ← dummy                                                     ▷ operation initialised to dummy operation
6:      for o_c in order do
7:          Ω' ← generateOptions(o_c, f, Ω)
8:          for ω ∈ Ω' do
9:              ω ← triggerMaintenance(o_c, o_p, f, ω)
10:             Ω'' ← Ω'' ∪ {ω}
11:         Ω ← selectHighestRanked(Ω'', rank)
12:         o_p ← o_c
13:         Ω'' ← ∅
        return Ω
```

Line 3. If a maintenance activity is triggered at any point in the schedule, the action is then inserted and the schedule is re-evaluated in Lines 5–9. We approach this by creating an operation a^c to represent the maintenance activity and adjusting the edges in the graph such that the constraints of the original problem remain intact after the insertion of the new operation. This is illustrated in Figure 3.2 where we show the edges added after inserting a maintenance activity. Since we have hard timing constraints between operations, inserting a maintenance activity can lead to a previously feasible schedule becoming infeasible. In such a case, a schedule repair action is triggered to return the schedule to a feasible state in Line 11. Algorithm 3 assumes that a schedule is always repairable and below in Section 3.6.2, we show what the necessary conditions are for this to be true.

3.6.2. Schedule Repair

Flow shop schedules generally need to obey a certain ordering of operations to be valid. However, re-entrant flow shops with due dates have an additional validity criterion, which is the due date between operations. In a case where operations that are not completely part of the set of input operations – such as maintenance activities – have to be scheduled, due date violations become even more likely. Since these operations are only known when schedules are evaluated, we always have the possibility that a schedule becomes infeasible as a result of these insertions. Furthermore, it is still combinatorial to decide on the repaired version of the schedule that minimizes the makespan after an event that causes infeasibility occurs. We therefore need to develop a schedule repair strategy for this problem.

Our Strategy

Schedule repair entails reorganising a schedule to obtain a state where the schedule is valid again (Vieira, Herrmann, and E. Lin 2003). Since we start from a valid schedule that is rendered infeasible by inserting new operations, the infeasibility is due to a

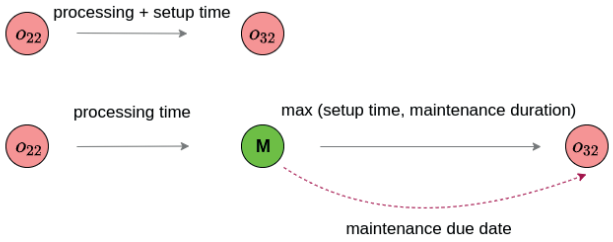

Figure 3.2.: Edge update after inserting a maintenance activity. The original constraints between operations o_{22} and o_{32} (both operations on the second machine) are now between operation o_{22} and the maintenance activity with new edges added to connect the maintenance activity to operation o_{32} This ensures the original constraints of the problem are present and the maintenance activity is scheduled before operation o_{32}.

Algorithm 3 Trigger Maintenance

```
1: function TRIGGERMAINTENANCE(current operation o_c, previous operation o_p, flow
   shop f, schedule Ω)                                  ▷ returns updated schedule Ω
2:     for o_i ∈ ⟨o_p, ..., o_c⟩ do
3:         Δ ← δ(⟨o_1, ..., o_i⟩, μ_k)                  ▷ predict deterioration state
4:         if X(Δ)is defined then                       ▷ deterioration triggers maint.
5:             a^c ← X(Δ)                               ▷ insert maint. activity
6:             Ω ← insertMaintenanceOperation(a^c, Ω)
7:             Ω ← updateStartTimes(f, Ω)
8:             feasible ← checkFeasibility(f, Ω)
9:             if ¬ feasible then
10:                Ω ← repairSchedule(f, Ω)
       return Ω
```

due date violation, i.e., an operation has been delayed too long after its preceding operation. Therefore, the fix is to systematically bring operations closer to their predecessors. However, it is not immediately obvious which operations need to be brought forward and how far this needs to go. As such we define a recursive strategy where we take small steps forward and reevaluate the fix until the schedule is feasible again. Additionally, moving operations around can violate the maintenance policy so after re-organisation, it is necessary to re-evaluate the schedule. This solution falls under the class of proactive-reactive dynamic scheduling (Ouelhadj and Petrovic 2009).

As shown in Algorithm 4, every time we reorganise the operations in the schedule, we first identify three key operations, namely, the *penultimate first pass operation* from the point where the schedule was broken, the *last second pass operation* from the point where the schedule was broken, and finally the *last second pass operation* that has been included in the schedule. This is shown in Lines 4-6 where we identify these key operations and their positions in the schedule. We then move all scheduled second pass operations belonging to jobs ranging from the *last second pass* to the *ultimate first pass* in the schedule – this occurs in the remove and insert calls on Lines 13–17. This way, the schedule has been reorganised such that second pass operations from the point of failure are at least a step closer to their first pass operations. We repeat this process until the schedule becomes feasible[3], moving the point of failure a step backward each iteration – this is as seen on Line 18 where the point of failure is updated ahead of the next iteration. After the schedule is deemed feasible, a last step is taken to trigger maintenance again in Line 20 as re-ordering operations could have invalidated or triggered maintenance activities. This re-ordering works because due dates exist only between consecutive operations of the same job.

Figure 3.3 shows an example of the schedule repair process. In Step 1, the schedule is infeasible after the insertion of a maintenance activity highlighted in green. The ultimate first pass is identified as o_{42}, the penultimate first pass as o_{32} and the last

[3]It is always possible to find a feasible solution as long as the maintenance policy in use is safe. This is as shown in Theorem 1 below.

Algorithm 4 Schedule Repair Strategy

1: **function** REPAIRSCHEDULE(flow shop f, position n, schedule Ω) ▷ **returns** repaired schedule Ω
2: $feasible \leftarrow false$
3: $end \leftarrow false$
4: **while** $\neg$ $feasible$ $\wedge$ $\neg$ end **do**
5: $(fp', o_{fp,k}) \leftarrow penultimateFirstPass(n, \Omega)$
6: $(ffp', o_{ffp,k}) \leftarrow ultimateFirstPass(n, \Omega)$
7: $(sp', o_{sp,k+1}) \leftarrow lastSecondPass(n, \Omega)$
8: **if** $o_{ffp} = o_{1,k}$ **then** ▷ first operation on machine
9: $end \leftarrow true$
10: $i \leftarrow sp' + 1$
11: **while** $i \leq ffp'$ **do**
12: $\Omega \leftarrow removeSecondPassOp(o_{i,k+1}, \Omega)$
13: $\Omega \leftarrow insertSecondPassOp(fp', o_{i,k+1}, \Omega)$
14: $fp' \leftarrow fp' + 1$
15: $i \leftarrow i + 1$
16: $n \leftarrow fp'$
17: $\Omega \leftarrow updateStartTimes(f, \Omega)$
18: $feasible \leftarrow checkFeasibility(f, \Omega)$
19: $\Omega \leftarrow triggerMaintenance(o_{sp}, o_{1,k}, f, \Omega)$
20: **return** Ω

second pass as o_{13}. The operations after the maintenance activity are then brought forward as can be seen in the new placement of o_{23} in Step 2. This continues in Steps 3 and 4 until the schedule is evaluated to be feasible.

It is valuable to point out that the overall algorithm proposed is flexible enough to adopt other repair strategies depending on the use case. An alternate example could be the strategy of reducing the rate of production to prevent or delay maintenance activities. A host of possible rescheduling and repair strategies are surveyed in (Vieira, Herrmann, and E. Lin 2003).

Safe Maintenance Policies

A maintenance policy X maps a deterioration state of the machine to an appropriate maintenance activity. The policy in use determines when and where maintenance activities are necessary. As discussed above, inserting a maintenance activity in a schedule may make the schedule infeasible. We define a *safe* maintenance policy as a policy that ensures that there exists at least one maintenance-aware solution to the flow shop provided there is a feasible schedule for the flow shop alone without considering maintenance activities. Since a schedule becoming infeasible after a maintenance insertion is a result of a violated due date, there should be enough room between consecutive first and second passes of the same job to fit a particular maintenance activity unless the policy is such that the maintenance activity cannot

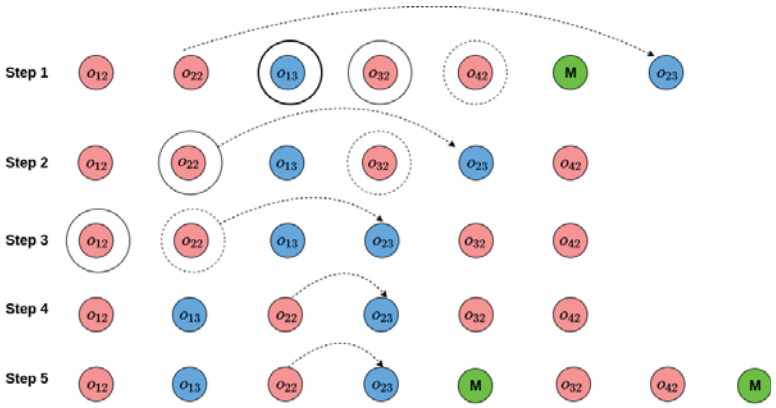

Figure 3.3.: Schedule repair strategy showing progressive steps in the algorithm. In the first step, the schedule is infeasible because of the maintenance activity (highlighted in green). From this point on, the future steps re-organise the schedule until we achieve a feasible schedule in Step 4. In Step 5, a last step is taken to trigger maintenance again as re-ordering operations could have invalidated existing or triggered new maintenance activities. Operations encircled in dotted lines are the ultimate first pass from the point of failure, the ones circled in a thin line are the penultimate first pass, and the ones circled in a thick line are the last higher pass operation.

be triggered between first and second passes of the same job. Concretely, this means that the processing time of any maintenance activity a^c that can be triggered between passes of the same job o_{ik} and $o_{i(k+1)}$ should fit in the available time between them, i.e.,

$$P_c \leq D(o_{ik}, o_{i(k+1)}) - P_{ik} - S(o_{ik}, o_{i(k+1)}) \quad \forall \; o_{ik}, o_{i(k+1)}. \tag{3.7}$$

Theorem 1. *Given an infeasible schedule, the schedule repair strategy defined in Algorithm 4 is always able to return it to a state of feasibility in at most $|J|$ iterations, where $|J|$ is the number of jobs in the schedule, provided that a solution exists for the problem and the maintenance policy in use is safe.*

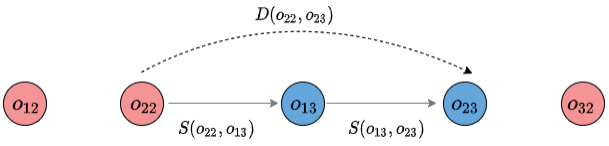

Figure 3.4.: Slack between operations o_{22} and o_{23}

Proof. For an insertion of a maintenance activity a^c between operations o_{ik} and $o_{i(k+1)}$ to become infeasible due to a due date violation, it means that $o_{i(k+1)}$ has been delayed too long, i.e., $\omega_{i(k+1)} - \omega_{ik} > D(o_{ik}, o_{i(k+1)})$. To avert this, the maintenance activity must be able to fit in the slack between both operations. Bearing in mind that

other operations could be placed between o_{ik} and $o_{i(k+1)}$, the slack $\Psi(o_{ik}, o_{i(k+1)})$ left between o_{ik} and $o_{i(k+1)}$ is

$$\begin{aligned}\Psi(o_{ik}, o_{i(k+1)}) = D(o_{ik}, o_{i(k+1)}) - P_{ik} - \\ \max((S(o_{ik}, o_a) + P_a + ... + S(o_a, o_b) + P_b), S(o_{ik}, o_{i(k+1)})),\end{aligned} \tag{3.8}$$

where operations $o_a, ..., o_b$ represent operations possibly placed between o_{ik} and $o_{i(k+1)}$. Figure 3.4 shows an example where the slack between operations o_{22} and o_{23} is

$$\begin{aligned}\Psi(o_{22}, o_{23}) = D(o_{22}, o_{23}) - P_{22} - \\ \max(S(o_{22}, o_{13}) - P_{13} - S(o_{13}, o_{23}), S(o_{22}, o_{23})).\end{aligned} \tag{3.9}$$

The repair algorithm progressively brings operations closer to their direct predecessors by at least one step per iteration. In the last possible iteration of the schedule repair, each operation $o_{i(k+1)}$ follows its direct predecessor o_{ik}. It follows that this occurs in at most $|J|$ iterations of the schedule repair as the re-entrant machine can only have $|J|$ higher pass operations to be re-ordered. At this point, Equation (3.8) becomes

$$\Psi(o_{ik}, o_{i(k+1)}) = D(o_{ik}, o_{i(k+1)}) - P_{ik} - S(o_{ik}, o_{i(k+1)}). \tag{3.10}$$

For this to be infeasible, it means that a^c cannot fit in $\Psi(o_{ik}, o_{i(k+1)})$, i.e., $P_c > D(o_{ik}, o_{i(k+1)}) - P_{ik} - S(o_{ik}, o_{i(k+1)})$, which violates the rules of a safe maintenance policy shown in Equation (3.7). ■

3.7. Experimental Results

This section evaluates the empirical performance of the three solution approaches we propose. We apply the heuristic approach as an add-on to three existing list schedulers in the literature to evaluate the applicability of this approach to list-scheduling. We compare our heuristics against the two exact approaches (integer and constraint programming) to evaluate their accuracy and scalability.

3.7.1. Experimental Setup

All experiments are performed on a 16-core 1.9GHz AMD machine running Ubuntu 20.04 with 32GB RAM. Algorithms are implemented in C++ and the MIP and CP models are solved by CPLEX version 22.1 and CP Optimizer version 22.1, respectively. The exact approaches are all given a 30 minute timeout.

We generate benchmarks according to the types of jobs typically presented in our industrial use case as described in Table 3.1. We generate benchmarks with patterned arrivals of job types such that jobs of a type appear in repeated blocks, e.g., a set of 50 jobs can be made of 20 type 1 jobs followed by 10 type 2 jobs and then 20 type 3 jobs. We randomise the length of the blocks and number of times these blocks repeat to mimic arrival patterns of jobs in practice. We generate 50 instances for each job size in {5,10,50,100,150,200,300,500,1000}.

Type	$P(o_{i1})$	$P(o_{i2})$	$P(o_{i3})$	$P(o_{i4})$	$D(o_{i1}, o_{i2})$	$D(o_{i2}, o_{i3})$	$D(o_{i3}, o_{i4})$
0	0.25	0.30	0.30	0.21	0.85	12.30	1.00
1	0.35	0.42	0.42	0.30	0.95	12.42	1.12
2	0.50	0.59	0.59	0.42	1.10	12.59	1.29
3	0.70	0.84	0.84	0.60	1.30	12.84	1.54
4	0.99	1.19	1.19	0.85	1.59	13.19	1.89

(a) Job processing times and due dates

Machine	Setup Time
μ_1	0.20
μ_2	0.05
μ_3	1.00

(b) Machine setup times

Path	Travelling Time
μ_1 to μ_1	0.60
μ_2 to μ_2	10.00
μ_2 to μ_3	0.70

(c) Job travelling times

Activity Class	Duration	Deterioration States
1	0.5s	10 – 15
2	10s	15 – 30
3	20s	30 – ∞

(d) Maintenance policy

Table 3.1.: Properties of jobs in use case. All timings are in seconds and job travelling times are treated as setup times between operations of the same job.

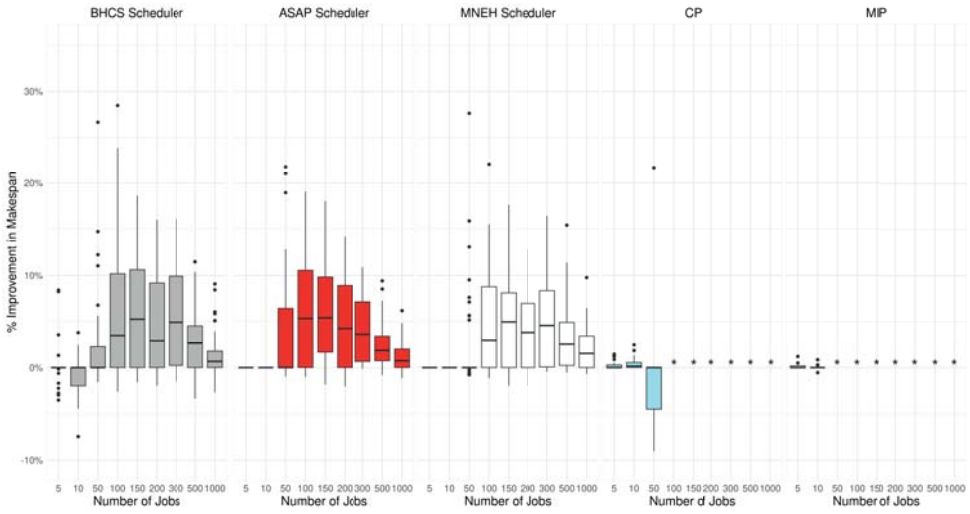

Figure 3.5.: Makespan improvement of maintenance-included versions over base versions

Instances where the solver timed out without providing any solution are marked with *.

Our heuristic approach is implemented as an extension to three schedulers from the literature – Bounded Heuristic Constraint Scheduler (BHCS) (van Pinxten *et al.* 2017), As Soon As Possible (ASAP) Scheduler, and Modified Nawaz-Enscore-Ham (MNEH) Heuristic (Jeong and Kim 2014). BHCS is a list scheduler developed specifically for our use case, while the ASAP scheduler is also a list scheduler that uses the same ordering

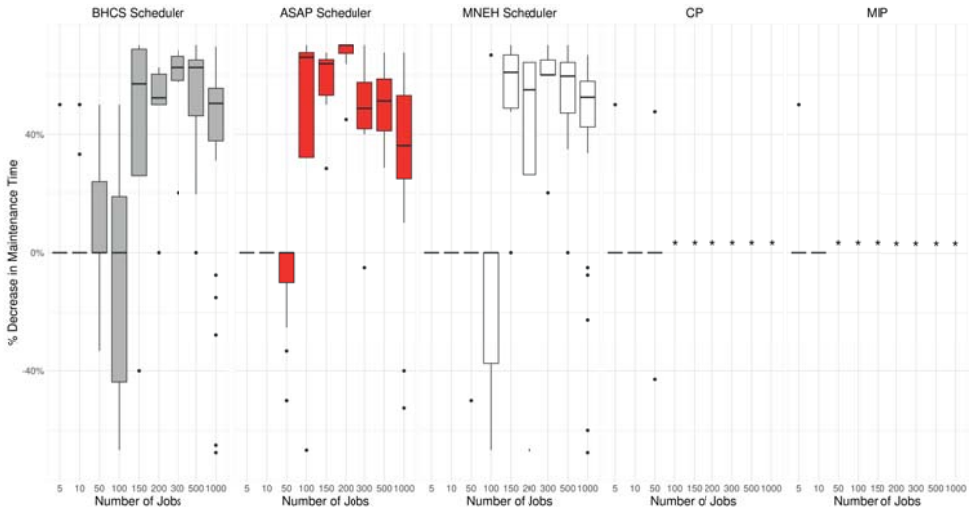

Figure 3.6.: Duration of maintenance activities
Instances where the solver timed out without providing any solution are marked with *.

requirements as BHCS but places operations as soon as possible (ASAP). MNEH is a modification of the popular NEH heuristic (W. Liu, Y. Jin, and Price 2017; Nawaz, Enscore Jr, and Ham 1983) that is suitable for re-entrancy. Maintenance-incorporated versions of these schedulers are referred to as MIBHCS, MIASAP, and MINEH respectively where the MI prefix refers to "maintenance incorporated". In each of these experiments, we tune the heuristic approach to include a maintenance activity if a deterioration threshold is crossed or if 90%[4] of the upper bound of a threshold that affects the quality of an operation further down the line is crossed. Since we insert maintenance between two operations, we always have complete information about the next operation. We can also reliably infer what operations are further down the line for the entire planning window based on which operations have already been scheduled.

In the basic schedulers – BHCS, ASAP, and MNEH – maintenance is reactive and interrupts the schedule during production runs. We simulate the behaviour of reactive maintenance in these schedulers by evaluating the completed schedules they produce for maintenance and compare these with versions of the scheduler that incorporate our proactive maintenance heuristic.

3.7.2. Performance Evaluation

Figure 3.5 compares the makespan of the schedules produced by MIBHCS, MIASAP, and MINEH to the makespan of schedules produced by BHCS, ASAP, and MNEH respectively. We also compare the exact solutions CP and MIP with the best solutions

[4]This value can be tuned. We chose 90% after performing a parameter sweep that showed this value performed best.

provided by MIBHCS, MIASAP, and MINEH. MNEH has the least performance improvement due to it not being a pure list scheduler. With MNEH, only relative positions of operations are decided in each iteration and there is no partial sequence that is guaranteed to remain the same from one iteration to the next; as such the evaluation of the deterioration of a machine loses some meaning from one iteration to the next since sequences change at each iteration. The exact approaches – CP and MIP – should ideally always be better than all of the heuristic approaches but they are sometimes worse because they do not always solve till optimality within the time out.

Figure 3.6 shows the distribution of the time spent on maintenance. We see that with maintenance-included versions, we spend up to 70% less time on maintenance. This is because considering deterioration allows us to perform maintenance before machines deteriorate to a state where we have to pay larger maintenance costs. The difference is also this significant because there is up to one order of magnitude difference between the durations of different maintenance activities for this use case (see Table 3.1d). This difference translates to shorter makespans for the schedulers.

Jobs	Optimality gap (%)		Time to find first solution (s)		% of instances solved[5]	
	CP	MIP	CP	MIP	CP	MIP
5	0.60	0.00	0.82	2.44	100	100
10	1.10	4.60	2.38	419.80	100	26
50	166.01	-	84.95	-	100	0

Table 3.2.: Performance of CP and MIP solutions

In both Figures 3.5 and 3.6, there are instances where the heuristic approach worsens the results particularly for smaller job sets. The instances that are worsened by the heuristic are a result of (i) scenarios where the heuristic maintenance trigger is too conservative and performs maintenance even though the job set could be completed without it, and (ii) scenarios where the list scheduler picks a sequence that triggers shorter maintenance activities.

Neither of the exact solutions are able to scale to provide solutions for larger job sets within the 30 minute time out – this accounts for the missing columns in Figures 3.5 and 3.6. In Table 3.2 we show the performance of the CP and MIP solutions. We see that the CP model is able to solve more instances than the MIP model but for the instances where the MIP model is able to provide solutions, the optimality gap is smaller.

The runtime increases with the number of jobs as expected and Table 3.3 shows the average runtime over the job size of the different schedulers compared in this evaluation. The exact approaches are given a 30 minute timeout and in bold are the solutions with the worst run times for a job size. Instances where no solution was provided by a method before time out are left unfilled and are the worst for that job size. The heuristic solutions are able to provide solutions in runtimes below 350ms for job sizes up to 500. Above that, the runtime grows to 1800ms. The biggest

[5]By solved, we mean that a solution was provided before the timeout, regardless of its quality.

time sink for the heuristic solutions is how often the maintenance evaluation and consequently schedule repair is triggered[6]. This is based on the operation of the base scheduler itself. MNEH evaluates whole sequences while ASAP and BHCS evaluate partial sequences at every decision point thus triggering maintenance evaluations more often, leading to higher runtimes.

Jobs	CP	MIASAP	MIBHCS	MINEH	MIP
5	**505.16**	0.00	0.00	0.00	5.81
10	1083.43	0.01	0.01	0.00	**1522.37**
50	**1611.23**	0.67	0.56	0.07	-
100	-	2.72	1.94	0.32	-
150	-	7.06	4.84	1.12	-
200	-	14.19	8.65	2.23	-
300	-	42.10	23.11	5.94	-
500	-	164.71	84.80	30.19	-
1000	-	1756.40	854.44	303.66	-

Table 3.3.: Average runtime of solution methods (s)

In summary, we find that the heuristic approach is scalable and can produce competitive results compared to exact solvers even for small instance sizes. In general, we also find that apart from improving the actual goal of reduced makespan, integrated production and maintenance planning can also reduce the total time spent on maintenance which can result in reduced costs in some cases.

3.8. Conclusions

Efficient maintenance scheduling is important for sustained productivity of industrial processes. This chapter studied the problem of sequence- and time-dependent maintenance and presented three solution methods namely, mixed integer programming, constraint programming and a heuristic solution. As the problem is motivated by an industrial use case, we have evaluated all the methods on jobs in this case. We show that list scheduling heuristics can be extended to include proactive maintenance with significant performance gains over reactive approaches.

This chapter considers maintenance activities that are on the same time scale as the jobs themselves. An interesting future direction is to include longer-term maintenance planning in the scope and to investigate the combined problem of production and maintenance planning over multiple time scales.

Additionally, we solve the problem from a predictive maintenance perspective, i.e., where maintenance actions are carried out based on the health status of machines. However, this requires knowledge of how machines deteriorate and this information is not always available. Many other works consider a preventive maintenance perspective where the challenge is either scheduling around a set maintenance

[6]Runtimes of ASAP, BHCS, and MNEH are similar.

schedule or determining what the maintenance schedule itself should be. While we know that preventive maintenance runs the risk of either maintaining machine too little or too often compared to the needs-based approach of predictive maintenance, and both preventive and predictive maintenance have been shown to outperform reactive maintenance approaches, it is still interesting to compare both approaches and determine what problem properties make it necessary to use one or the other. This is because even when complete information on the health status of machines is available, the gains made by integrating them in the decision making process may not necessarily be worth the increased runtime.

Acknowledgements

The authors thank Hadi Ara, Joost van Pinxten and Joan Marcè i Igual for their support.

II

Decision Diagrams for Scheduling and Beyond

4

Introduction to Decision Diagrams for Optimisation

Decision diagrams (DDs) are graphical structures capable of compactly representing the solution space of combinatorial optimisation problems. Additionally, they are tunable to represent approximations of the solution space as in restricted *DDs where the solution space is under-approximated and* relaxed *DDs where the solution space is over-approximated. In this chapter, we introduce DD concepts that provide the required foundation for understanding the methods presented in this part the dissertation.*

4.1. Introduction

A typical decision diagram is a directed acyclic graph (DAG) with $G = (V, E)$ with a set of vertices V and edges E. Within the set of vertices are two special vertices; the *root* vertex r and *terminal* vertex t. The *root* vertex is a special source vertex with no incoming edges and the *terminal* vertex is a *dummy* sink vertex with no outgoing edges. All vertices at which there are no more unassigned variables are connected directly to t via a *dummy* edge that only serves to connect a path to the sink. All vertices at the same depth constitute a *layer* in the DD.

These data structures have been used to encode complex functions. Such a function $f : X \to Y$ typically maps a set of variables X to some output Y and the structure is such that each edge represents a value assignment to one of these variables. Consequently, each vertex v in the DD represents a partially evaluated version of f – called a state – where only variables along the path to v have been assigned values. Thus, edges represent transitions made between states. A path in the DD therefore represents a sequence of variable assignments and any $r \to t$ path is a complete evaluation of the encoded function.

Let us consider the boolean function

$$(x_1 \wedge x_2) \vee \neg x_3, \tag{4.1}$$

with three variables x_1, x_2, x_3. Figure 4.1 shows a decision diagram encoding this function.

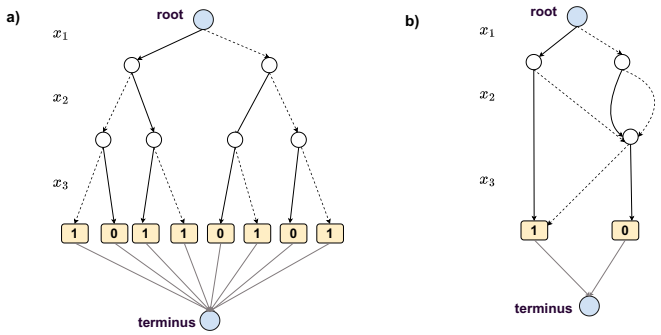

Figure 4.1.: **a)** A binary decision diagram of the ternary boolean function in Equation (4.1). Dotted edges represent a value of 0 and solid lines represent a value of 1. Terminal and root vertices are mapped in blue and function values are within the vertices connected to the terminus.
b) A reduced order binary decision diagram of the same function showing the compacting capabilities of decision diagrams.

4.2. Historical Overview

DDs were first introduced as a means to compactly represent and manipulate Boolean functions as shown in Figure 4.1 (Akers 1978; Lee 1959). They were primarily binary

decision diagrams (BDDs) such that each non-terminal vertex v has at most two outgoing edges with labels 0 and 1 representing possible values for the boolean variable var(x) represented by x. Such diagrams have found uses in circuit verification, knowledge representation, formal methods and many other fields.

Over time, the representative power of DDs continually grew, first with Algebraic Decision Diagrams (ADDs) which maintain the binary branching structure but allow terminal vertices to take any value from a set of constants S and eventually to multi-valued decision diagrams (MDDs) which can represent functions with arbitrary branching possibilities for every vertex.

In the context of optimisation problems, we first see DDs appear in the 1990s (Lai, Pedram, and Vrudhula 1994). They maintain the same DAG structure but in this context are used to compactly represent the solution space of discrete optimisation problems. Recall the general form of a discrete optimisation problem

$$\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f : \mathcal{X} \to \mathbb{R})), \tag{4.2}$$

comprised of a finite sequence of variables $\mathcal{X} := x_1, \ldots, x_n$ with domains $D_i \in \mathcal{D}$ associated with each variable, and a finite set of constraints $\mathcal{C}$ each defined on a subsequence of $\mathcal{X}$ with a goal to find an assignment of values to each variable $x_i \in \mathcal{X}$ such that the assigned value is in the domain D_i, no constraints in $\mathcal{C}$ are violated, and the objective function $f : \mathcal{X} \to \mathbb{R}$ is either maximised or minimised.

In a DD formulation of such an optimisation problem, vertices represent states reached via a (partial) assignment of variables in $\mathcal{X}$ and edges represent transitions between states via feasible value assignments. Additionally, each edge is assigned a cost corresponding to the contribution of the chosen value assignment to the objective function[1]. Thus, the total weight of every $r \to t$ path[2] is the value of the objective derived from the variable assignments along the path and the shortest(longest) path is the optimal solution in a minimisation(maximisation) problem.

DDs for optimisation are often constructed via recursive dynamic programming (DP) formulations (Bellman 1966) and bear very close similarities with Markov Decision Processes (MDP) (Bergman, Cire, Van Hoeve, *et al.* 2016).

In such a dynamic programming formulation, the state space S is partitioned into $n+1$ stages where each stage S_i contains a set of states in the i^{th} stage of the DP model and S_0 contains only the root or initial state r. We also define an infeasible state $\hat{0}$.

Typically, each stage in the DP model corresponds to a layer of vertices in the DD[3] and the variables are ordered such that all incoming edges to the i^{th} layer of the DD are via transitions made on feasible value assignments to variable x_i.

We further define the following functions:

- a transition function $\tau : S_i \times D_i \to S_{i+1}$ that maps a state in the i^{th} layer to one in the $i+1^{\text{th}}$ layer given a value assignment in the domain D_i of variable x_i,

[1] This formulation assumes a separable objective function.
[2] Dummy edges that lead to the terminal vertex t are always assigned a weight of 0.
[3] There are exceptions when long arcs are allowed in the DD.

- a cost function $h : S_i \times S_{i+1} \times D_i \to \mathbb{R}$ that provides the contribution of a value assignment to the objective function.

As many combinatorial optimisation problems are NP-hard, DDs are known to have exponentially many vertices, especially when they encode all possible solutions to a problem, i.e., when they are *exact*. Bounded size approximations of DDs have been developed as a means to manage exponential growth. The typical bounding strategy is to limit the width, i.e., the maximum number of vertices in any layer, of the diagram. Specifically, *restricted* DDs encode a subset of possible solutions, while *relaxed* DDs encode a super-set of possible solutions by allowing some infeasible paths to exist in the diagram (Bergman, Cire, Van Hoeve, *et al.* 2016). Additionally, a diagram is *reduced* when it does not contain any isomorphic sub-graphs, i.e., no two vertices encode the same state.

The structure of DDs enables us to recursively divide the problem into smaller sub-problems as every vertex can be thought of as the root of a sub-graph representing the sub-problem remaining to solve at that vertex. Consequently, every path in the diagram represents a solution. Given a separable objective function, each edge can also be assigned a weight corresponding to the contribution of that decision to the objective such that the length of a path is then also the objective function value of the solution represented by the path.

Example 1. Let us consider a KNAPSACK problem instance with capacity 10 and 3 items of weights $6, 4, 2$ and profits $3, 5, 7$ respectively. Here, the challenge is to select a subset of available items to place in a knapsack of given capacity such that the capacity is not exceeded by the total weight of the objects and the profit derived from the objects is maximized. Figures 4.2 and 4.3 show two means of constructing DDs for this problem. Both figures construct a BDD for this problem where the state representation is the tuple $(\mathit{avail}, \mathit{pro})$ such that *avail* is the available capacity and *pro* is the highest profit achievable along any path to that state. Edges in this diagram represent the binary choice of putting an item in the knapsack or not where dotted edges represent the choice of not taking the item. The weight of an edge represents the added profit of the choice made along the edge.

4.3. Compiling a Decision Diagram

The process of constructing a DD given a dynamic programming formulation is called *compilation*. In general, there are two main compilation methods for DDs, namely *top-down compilation* and *incremental refinement* – also known as compilation by separation (Bergman, Cire, Van Hoeve, *et al.* 2016).

Top-down Compilation Top-down compilation works by constructing the diagram layer by layer starting from the root until the terminal vertex where no more variables remain to be assigned values. In every step, all vertices in a layer are expanded by creating child vertices for every feasible value assignment to the variable at that layer. If the diagram is to be restricted, some vertices are *pruned*, and if it is to be relaxed, some vertices are *merged* before continuing to the next layer.

Figure 4.2 shows a top-down compilation process of a relaxed diagram for the running example in Example 1. There are 3 layers of vertices apart from the root and terminal vertices and the relaxation is a result of the merge operation – combining the highlighted vertices – in layer 2. The process of compiling the diagram starts from the the left with the root node having a state (10,0). Recall that the state representation is a tuple $(avail, pro)$ of availability in the knapsack and total profit gained at the state. Thus, at the root, the entire capacity of 10 is available and 0 profit has been made. From the root, we transition by either taking the first item and getting a profit of 3, or not taking the item and getting a profit of 0. These lead to states (4,3) and (10,0) in the second layer of the diagram respectively. The construction then continues this way until all nodes have a path to the terminal node.

Incremental Refinement Incremental refinement, on the other hand, begins with a dummy diagram of limited width (often 1) where all possible variable assignments are available at every vertex. The refinement procedure then progressively expands layers by splitting vertices and filtering out infeasible edges where an edge is considered infeasible if the transition violates constraint(s) of the problem at hand. This compilation method is often used to create relaxed diagrams though exact and restricted diagrams are also possible. Some other variants have been studied in the literature such as A* compilation (Horn, Maschler, *et al.* 2021) and local search refinement (Römer, Cire, and Rousseau 2018), both of which are adaptations of top-down or incremental refinement.

For the same running example in Example 1, Figure 4.3 shows the process of incrementally refining a DD. Here, we begin with a one-width top-down constructed diagram on the left. The progression is again from the root to terminal with the same transition rules as in Figure 4.2. We then begin expanding the diagram. The second layer node is selected to be split and thus the two transitions that led to that node in the previous step are now separated into their own resulting nodes with the consequences of this split cascaded to the next layer of the diagram. The same process is repeated in the following step of the diagram where the third layer node is split. We continue this process until a stopping condition is met, e.g., the desired width per layer.

Some of the earlier uses of DDs for optimisation actually involved aiding the solution process of other solvers. For instance, Andersen *et al.* (2007) used DDs as a domain store in constraint programming and Becker *et al.* (2005) used DDs for *cut generation* in linear programming.

4.4. Decision Diagram Operations

In the process of compiling a decision diagram, some key operations are carried out. We define these below.

Merging. A key step in compiling relaxed DDs is merging, where vertices are combined to limit the size of the diagram. In most relaxation algorithms, the size of the diagram is limited by fixing the width W (number of vertices) of any layer in the

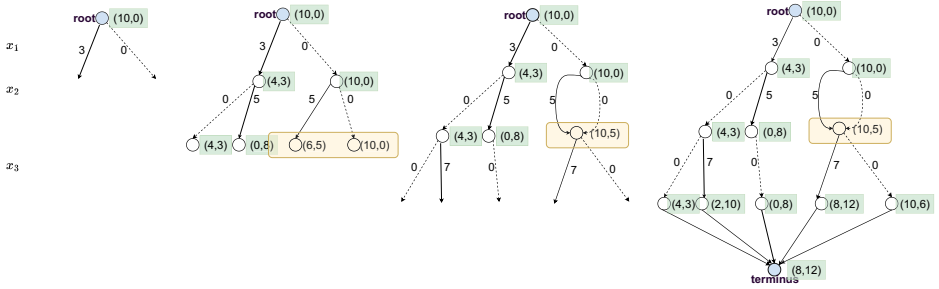

Figure 4.2.: Top-down DD compilation process for the sample problem in Example 1 showing a merging process in yellow highlighted nodes.

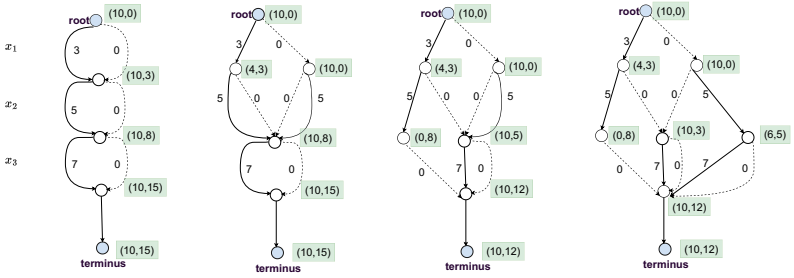

Figure 4.3.: Incremental refinement DD compilation process for the sample problem in Example 1.

diagram. Thus, in layers with vertices that exceed the desired width, some of these vertices are merged such that the target width is achieved. The way this merge is done greatly affects the quality of the bounds produced by the diagram. There are two main questions in any merging process, namely (i) how to select which vertices to merge and (ii) how to combine these vertices.

Splitting. Splitting a vertex can be thought of as the inverse of merging a vertices. One vertex with multiple incoming edges is divided into 2 or more vertices and the incoming edges redistributed among the new vertices. Similar to merging, the two choices to be made are which vertices to split and how to redistribute the edges.

Pruning. Most common in constructing restricted diagrams, vertices can also be pruned where pruning involves complete deletion of a vertex and all its descendants from the DD. This can either be done heuristically – where the resulting diagram is then restricted as there is no guarantee that a pruned vertex may not have led to a better solution, or exactly – where we have such a guarantee. The latter is often ascertained via dominance relations i.e., comparative relations on sub-problems that conclude if a sub-problem can be excluded or pruned from the state space without compromising the objective.

Variable Ordering. In every layer of a typical DD, all edges leading to the vertices in that layer represent value assignments to a particular decision variable. It has been shown that the quality of solutions and bounds produced by a DD is tightly related to the order in which variables are considered (Bergman, Cire, Van Hoeve, *et al.* 2012). However, the problem of finding the best variable ordering is itself NP-hard (Bollig and Wegener 1996), and state-of-the-art heuristics are often problem-specific (Nafar and Römer 2024a).

Algorithm 5 DD-Based Branch-and-Bound (DD-BnB)

```
1:  function DD-BnB(DP formulation of optimisation problem)   ▷ returns optimal solution z*
2:      initialise fringe Q = {r}                             ▷ only root vertex
3:      let z* = −∞
4:      while Q ≠ ∅ do
5:          u → selectVertex(Q)
6:          Q → Q \ {u}
7:          create restricted diagram DD̄ with root u
8:          if DD̄* > z* then
9:              z* → DD̄*
10:         if DD̄ is not exact then
11:             create relaxed diagram DD̲ with root u
12:             if DD̲* > z* then
13:                 let S be an exact cutset of DD̲
14:                 for u' ∈ S do
15:                     add u' to Q
        return z*
```

4.5. Branch and Bound with Decision Diagrams

Bergman, Cire, van Hoeve, *et al.* (2016) put forward the idea of DDs for optimisation in their seminal paper introducing a DD-based branch-and-bound (DD-BnB) algorithm. This algorithm progressively creates small diagrams from different vertices in the DD to provide solutions and bounds on the objective function.

Central to the operation of DD-BnB, is the notion of *exact cutsets*. A cutset $\mathcal{E}$ of a DD is a set of vertices such that any $r \to t$ path in a diagram contains at least one of the vertices in $\mathcal{E}$. We say a cutset is exact when all the vertices in the cutset are exact where a vertex v is *exact* when all $r \to v$ paths lead to the same state and it is *relaxed* when $r \to v$ paths can lead to different states.

The key intuition is then that, keeping an exact cutset is a sufficient representation of the entire diagram as it contains an exhaustive enumeration of all the sub-problems (Bergman, Cire, Van Hoeve, *et al.* 2016). Note that the smallest cutset is the root vertex.

Thus, DD-BnB maintains a queue of open vertices that is always an exact cutset, sometimes also called the fringe, of the DD. Algorithm 5 shows the solution process of DD-BnB assuming a maximisation problem. We begin with a fringe that contains

only the root vertex in line 1. We then continuously pop vertices from the fringe in lines 5 and 6, creating a restricted and relaxed diagram from each selected vertex in lines 7 and 11. Should the restricted diagram result in a better solution, we update our best known solution in line 9. For every relaxed diagram we create with a bound higher than the best known solution, we extract an exact cutset from that vertex and add it to the fringe ahead of the next iteration in lines 13-15. The process is terminated whenever the fringe is empty.

4.6. Contents of this Part of this Thesis

4.6.1. Research Goals

In this part of the thesis we aim to develop a deeper understanding of the consequences of heuristic decisions made during the decision diagram compilation process while expanding the solving capabilities of decision diagrams to handle uncertainty and incorporate learning. These goals result in four (4) research questions.

Research Question 2. *How to formulate a general decision diagram model for optimising manufacturing systems as introduced in Part I of this thesis?*

Many heuristic solutions exist for different multi-machine scheduling problems, as well as MIP and CP models (some of which were presented in Part I for maintenance scheduling) with varying degrees of performance. While Cire and Van Hoeve (2013) present a general DD model for sequencing problems of which the single machine scheduling problem is one, there is no general DD model for the multi-machine case to which many manufacturing systems belong.

Research Question 3. *What is the impact of combining decision diagram compilation techniques on the eventual bounds derived?*

Decision diagrams have shown great promise for solving combinatorial optimisation problems leading to many innovations in diagram compilation from including rough upper bounds, caching, dominance, to even clustering. However, these innovations are often studied in isolation and the combined effects on a decision diagram are not quantified.

Research Question 4. *How to combine learned policies with decision diagrams?*

While traditional optimisation methods have been able to produce impressive solutions to many problems, recent advances in machine learning have propelled researchers to look harder at combining optimisation and machine learning methods. The motivation for this is two-fold. First, as difficult as combinatorial optimisation problems are to solve, many real-life applications solve the same or similar problems over and over. Thus, learning to solve an entire class of problems holds much promise for industry. Secondly, many solution methods still depend on hand-crafted heuristics. Moreover, even within exact solvers, some solution steps are still performed heuristically. This also gives an additional opportunity to learn better heuristics or to automate the design of such heuristics.

Research Question 5. *How can we handle uncertainty in optimisation problems solved by decision diagrams?*

The reality of optimisation problems is one of uncertainty. With almost every problem instance, there is real-world uncertainty that affects problem parameters. Such uncertainty is often ignored or over-estimated to produce robust solutions. MIP solvers have a rich history of means to handle uncertainty ranging from chance constrained formulations to scenario-based models. Many heuristic solutions also consider stochastic parameters. However, DDs do not yet have such capabilities. There are two lines of work that actively consider stochasticity in DDs, (Hooker 2022) which considers DDs with probabilistic transitions and (Perez, Malalel, *et al.* 2023) which designs a DD based propagator for the confidence constraint in constraint programs. Thus, expanding the uncertainty handling capabilities of DDs is of great interest.

4.6.2. Contributions

For research question 2, we propose a model for multi-machine scheduling problems whose objective function is correlated with the makespan. Our results show that solving this problem with decision diagrams is competitive with other state of the art methods like constraint programming and mixed integer programming in terms of both solution quality and runtime.

We answer research question 3 by performing a comparative study on decision diagram compilation techniques for three main tasks: merging vertices, ordering variables and dominance-based node pruning.

To tackle research question 4, we propose an integration of reinforcement learning with DD-based branch and bound such that the restricted DDs are built by learned policies.

Finally, for research question 5, we propose, to the best of our knowledge, the first chance constrained decision diagram formulation. Our solution works for problems with uncertainty in the objective where the objective is a separable monotonically increasing function. We find that there is a significant number of problems with this structure and that we can consider this kind of uncertainty at no extra cost to the decision diagram.

4.6.3. Outline

The rest of this part of the thesis is structured such that each chapter corresponds to exactly one of these research questions. In Chapter 5 we present our DD model for multi-machine scheduling problems followed by the results of our comparative study of DD techniques in Chapter 6. We then present our proposal for RL-assisted branch-and-bound with DDs in Chapter 7 and end this part of the thesis with our theoretical results on DDs for chance constrained problems in Chapter 8.

5

Multi Valued Decision Diagrams for Multi-Machine Scheduling

While decision diagrams have been explored for single and unrelated parallel machine scheduling problems, solutions for multi-machine or shop based manufacturing systems have, to the best of our knowledge, not been explored yet. Such solutions and the required changes to the decision diagram and its rules of construction are still an open question that we address in this chapter.

5.1. Introduction

Manufacturing systems are often modelled as shops. Shop models treat the manufacturing system as a collection of jobs and machines, where jobs are made up of operations and operations are carried out on machines. These models are quite universal and are flexible enough to accommodate many problem-specific constraints and are thus able to cover a significant amount of real-world manufacturing systems (Schmidt 1996). Irrespective of the model, a major determinant of performance is how operations are scheduled. This essential question has received substantial attention over the past few decades (Gupta and Stafford Jr 2006).

Decision diagrams are graphical data structures used in discrete optimisation to represent possible assignments to variables (Bergman, Cire, Van Hoeve, *et al.* 2016) and have shown promise for solving scheduling problems (Cire and Van Hoeve 2013; Matsumoto, Hatano, and Takimoto 2018). Decision diagrams have been explored for scheduling problems from a general framework for sequencing problems (Cire and Van Hoeve 2013) to specific single machine scheduling problems (de Weerdt, Baart, and L. He 2021) and unrelated parallel (van den Bogaerdt and de Weerdt 2018; van den Bogaerdt and de Weerdt 2019) machine scheduling problems.

In this chapter, we introduce a generic decision diagram based scheduling solution for multi-machine scheduling problems. We discuss its operation on a basic flow shop and then a more complex industrial use case with deadline and setup time constraints as introduced in Section 2.2. Motivated by the fact that heuristics still provide state of the art performance for many shop scheduling problems, we also design our approach to accommodate starting out with an initial solution provided by a heuristic and strategically improve it while retaining the properties of exact solvers.

The rest of this chapter is organised as follows: Section 5.2 formally defines the scope of problems considered, Section 5.3 describes our proposed decision diagram representation of the problem, and Sections 5.4 and 5.5 present the compilation of the decision diagram using top down construction and incremental refinement. In Section 5.6 we present dominance rules for this problem. Section 5.7 describes the process of warm starting the decision diagram with an initial solution along with some experimental results in Section 5.8. We conclude the chapter in Section 5.9.

5.2. Problem Scope

We focus on the multi-machine scheduling problem under the objective of makespan minimisation. Let $J = \{j_1, ..., j_n\}$ be the set of jobs to be scheduled on a set of machines $M = \{\mu_1, \ldots, \mu_m\}$. We assume a shop scheduling scenario where there is a set O which represents the set of operations for every job $j_i \in J$ where operation $o_{ij} \in O$ is the jth operation of the ith job. Each operation is assigned to a machine in M. We assume that operations are non-preemptive and every operation o_a has a processing time $p(o_a)$, a release time $r(o_a)$ and is assigned to machine $m(o_a)$.

The entire problem is also subject to a set of constraints $\mathcal{C}$ where every $c \in \mathcal{C}$ enforces a relationship between a set of operations such that any solution that violates a constraint c is infeasible.

A solution to a multi-machine scheduling problem is a sequence of operations Ω.

We can safely extract start times from the sequence by assigning them based on an as-soon-as-possible (ASAP) strategy, i.e., assigning the minimum starting times of operations that still satisfy the constraints (See Lemma 1).

Lemma 1. *Given a sequence of operations, assigning ASAP start times produces the optimal makespan for that sequence.*

Proof. We prove it by contradiction. Assume, towards a contradiction, that there is a schedule Ω such that the ASAP start times are not optimal for Ω. If that is the case, it means that there is at least one operation o_1 for which the constraints between o_1 and any other operation $o_2 \in O$ are such that the minimum relative start time is smaller than that assigned by the ASAP strategy. However, because (i) the ASAP start times are based on the constraints between operations as they appear in Ω and (ii) the ASAP schedule provides the minimum starting times of operations that still satisfy the constraints, it is not possible that the minimum relative start times are smaller than those assigned by the ASAP strategy. We have reached a contradiction. ■

Note that while we assume a shop scheduling model in this chapter, the ideas are transferable to any disjunctive multi resource scheduling problem. For instance, non-preemptively scheduling tasks on multi processors.

5.3. Decision Diagram Representation

A decision diagram $G = (V, E)$ is made up of nodes V and edges E. In general, nodes $v \in V$ and edges $e \in E$ can hold state information with a key difference being that edges also represent transitions between states. We generally transit between states by scheduling operations; thus edges hold information of which operation(s) fueled the transition they represent.

A decision diagram contains two special states, namely the root state r and the terminal state t. At the root state r, no decisions have been made, and at the terminal state t, all decisions have been made, i.e., all operations have been scheduled. Thus, every path from $r \rightarrow t$ represents a schedule and its timing can be assigned according to Lemma 1.

Example 2. As a running example, we take a set of 4 operations $\{o_{11}, o_{12}, o_{21}, o_{22}\}$ with processing times $\{1,2,3,5\}$ on 2 machines $\{\mu_1, \mu_2\}$ such that $o_{11} \prec o_{12}$, $o_{21} \prec o_{22}$, o_{11} and o_{21} are assigned to μ_1 and o_{12} and o_{22} are assigned to μ_2.

Definition 1 (decision variable)**.** *We define a decision variable x_i as a discrete variable that takes values from a finite domain D_i.*

5.3.1. Operation Transitions

In this section, we describe transitions in the decision diagram. There is the question of how to represent transitions in a multi-machine scenario as different jobs could be dispatched to different machines simultaneously. In our representation, we dispatch only one job at a time and term this a flat-arc representation.

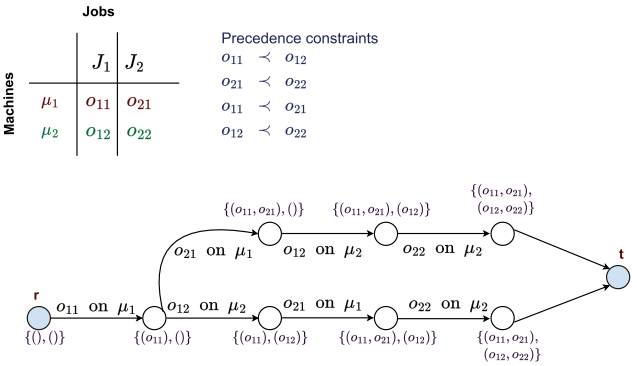

Figure 5.1.: Sample decision diagram showing all allowed paths for the sample job set in Example 2

Flat Arc Representations

Flat arc representations are a natural extension of the single machine sequencing formulation where one operation is scheduled at a time along each edge irrespective of machine assignments. In this representation every layer l_i represents a decision variable x_i namely the operation scheduled in the ith position of the schedule. At every node v in layer l_i, the domain of D_i – which we also refer to as $\text{dom}(v)$ for simplicity – is the set of operations whose constraints have not been violated on all paths leading to v. This representation flattens the multi-machine dynamic and treats all operations to be scheduled as a single set such that there are $m \times n$ positions in the schedule. Similarly, there are $m \times n$ decision variables.

In order to execute the schedule produced by a path in such a decision diagram, we still have to unravel the single sequence to execute on different machines. Each operation is scheduled on the machine it is assigned to and the order of operations on a given machine μ_m is the non-consecutive sub-sequence of all operations assigned to μ_m. Considering that we compute start times of any sequence using an ASAP strategy and machine assignments are known, this still produces a correct schedule for any path in a flat arc decision diagram. In the case where machine assignments are unknown and there are multiple *identical* machines that can carry out the same operation, it is also sufficient to unravel the schedule by placing operations on the machine with earliest completion time at every step (van den Bogaerdt and de Weerdt 2018).

The flat arc representation creates a complete solution space, i.e., all possible orderings are contained in the resulting DD. This is because we effectively consider all permutations of the flattened sequence of operations. Thus, when paths are unravelled to make a solution and we extract non-consecutive sub-sequences per machine, every operation is considered in all possible positions in the sub-sequence of its machine. In addition to completeness, this representation also makes a lot of the developed rules for single resource sequencing using DDs directly applicable to the multi-machine case.

Merged Arc Representations

We can alternatively think about the DD as every layer representing a position across all machines such that there are n positions in the schedule, e.g., layer 1 represents the set of operations in the first position on all machines. In order to transition from one layer to the next, an edge represents an element in the Cartesian product of the sets of operations and machines. Thus, there are $|n|^{|m|}$ branching possibilities for every node while there are only $|n|$ branching possibilities for the flat arc representation. Due to this large width of the merged representation, we use the flat arc representation for the rest of this chapter as it is more computationally manageable.

5.3.2. State Information

The exact information held in states is up for design and different approaches have been taken for instance van den Bogaerdt and de Weerdt (2019) hold most of the state information in the nodes while Cire and Van Hoeve (2013) have most of the information in the edges. In this chapter, we keep state information mostly in the edges because when nodes have multiple edges leading to them – as is the case in most relaxed decision diagrams –, having state information represented by nodes approximates the state more than if state information was held in edges. Thus, we define the following representations for nodes and edges.

Edges

Edges retain information on both the operations scheduled along them and the possible state of the partial solution to which they belong. Concretely, we associate an edge $e = (u, v)$ – where u and v refer to source and destination nodes – with the tuple $(val_e, w_e, est_e, lst_e, a_e)$ where:

- val_e is the operation scheduled along e,
- w_e is the weight of the edge e computed as its contribution to the objective,
- est_e and lst_e refer to timing vectors with elements associated with each operation in O. We refer to start times of a particular operation as $est_e^{o_a}$ and $lst_e^{o_a}$ for earliest and latest start times, respectively. For every operation o_a, $est_e^{o_a}$ and $lst_e^{o_a}$ correspond to the earliest and latest times o_a can start and still be part of a feasible schedule in the state of edge e,
- a_e is the earliest availability of all machines – $a_e^{\mu_k}$ is the earliest availability of machine μ_k.

All timing values in the tuple $(val_e, w_e, est_e, lst_e, a_e)$ represent what happens after the operation(s) in val_e have been scheduled. We also define the functions $source(e)$, $destination(e)$, $layer(e)$ which return the source node, destination node and depth or layer of the source node of e respectively.

Nodes

Nodes mainly keep track of what edges have led to or away from them. We retain the definitions in (Cire and Van Hoeve 2013) and associate the tuple $(All_v^{\downarrow}, Some_v^{\downarrow})$ with each node v such that $All_v^{\downarrow}$ is the set of edge labels that appear in *all* paths from the root r to v and $Some_v^{\downarrow}$ is the set of edge labels that appear in *some* path from the root r to v. We define the functions $layer(v)$, $out(v)$ and $in(v)$ which respectively return (i) the layer to which node v belongs, (ii) the set of outgoing edge labels from v, and (iii) the set of incoming edge labels to v.

In general, a node v is *exact* when all paths that lead to v results in the same state. For single resource sequencing problems, an exact node is one where $All_v^{\downarrow} = Some_v^{\downarrow}$, i.e., every path leading to the node has sequenced the exact same set of operations (Rudich, Cappart, and Rousseau 2023). For the multi-machine problem, we enforce a stricter condition that every exact node has a unique path leading to it. This is due to the flat-arc representation and how it affects sequence-dependent behaviour on the same machine. In this representation, operations that follow each other on a machine are not necessarily adjacent, thus even though two paths have scheduled the same operations, they may not necessarily exhibit the same sequence-dependent behaviour and as such, the state computation of a node with two or more paths leading to it is still inexact.

As discussed in Section 4.3, there are two main ways in the literature to compile decision diagrams, namely top-down compilation and incremental refinement. We formulate the construction such that both compilation methods result in the same diagram when fully expanded. We provide details of each compilation method in Sections 5.4 and 5.5 below.

5.4. Top Down Compilation

In this paradigm, the decision diagram is constructed from root to terminal, most often layer by layer although recent work has explored compiling in an A* fashion (Horn, Maschler, *et al.* 2021).

We begin construction of the decision diagram from the root node r; r has no in-degree edges as no sequencing decision is made for any machine. The state values of the root node are:

$$All_r^{\downarrow} = \emptyset \tag{5.1}$$

$$Some_r^{\downarrow} = \emptyset \tag{5.2}$$

For any edge transition leading from node u to v by scheduling an operation o_e along edge e, we update all node state values same as Cire and Van Hoeve (2013) such that:

$$All_v^{\downarrow} = \bigcap_{k=(u,v)\in in(v)} (All_u^{\downarrow} \cup \{val_k\}) \tag{5.3}$$

$$Some_v^{\downarrow} = \bigcup_{k=(u,v)\in in(v)} (Some_u^{\downarrow} \cup \{val_k\}). \tag{5.4}$$

We update edge value of e such that the est_e and lst_e vectors are updated by evaluating the constraint graph at the new state – details of this evaluation follow shortly in Section 5.4.1 – and:

$$val_e = o_e, \tag{5.5}$$

$$w_e = \min\{\max\{a_e\} - \max\{a_{e'}\} : e' \in in(u)\}, \tag{5.6}$$

$$a_e^{m(o_e)} = est_e^{o_e} + p(o_e). \tag{5.7}$$

We continue to expand and transition edges until all paths lead to the terminal state – any state where all operations in the system have been scheduled is connected to the terminal node via an edge with weight 0 – and the diagram is complete. In every transition, we prune those that violate any problem constraints – ordering constraints are checked explicitly and timing constraints are violated whenever $lst_e^o < est_e^o$ for any operation. We maintain the partial makespan at each node and assign edge weights as the difference in partial makespan from the source and destination nodes of an edge. The problem is *infeasible* whenever there is no path that leads from the root to the terminal node. The set of root-to-terminal paths represents the set of feasible solutions and the shortest root-terminal path is *optimal*, i.e., the best possible solution.

In Figure 5.2, we show a visual of this construction process for the problem in Example 2 showing the growth of the $All^{\downarrow}$ and $Some^{\downarrow}$ sets as we transition from state to state. In Step I of Figure 5.2, we begin compiling the diagram from the root note. As no operations have been scheduled yet, both the $All^{\downarrow}$ and $Some^{\downarrow}$ sets are empty. In Step II, we see the case where two transitions lead to the same node, thus the $All^{\downarrow}$ set remains the same – as we cannot guarantee any of the newly added transitions exist on all paths – while the set $Some^{\downarrow}$ is updated to include the operations along both transitions.

5.4.1. Calculating Start Times

An edge e has two explicit timing vectors namely, est and lst. We have the choice to make est and lst only represent start times of the operations scheduled along that edge but for more involved analysis such as dominance comparisons discussed in Section 5.6, it is necessary to hold values for all the operations in the problem. To compute such values, we make use of a constraint graph which is a simple temporal network (Dechter, Meiri, and Pearl 1991) that represents and aids reasoning about temporal constraints on activities.

Constraint Graph for Start Time Evaluation

As introduced in Section 2.2.3, the constraint graph X is a tuple (N, A) where nodes $n \in N$ represent operations – including two dummy nodes, a source node s and a terminal node z that are connected to all the first and all the last operations of the jobs respectively – and edges $a \in A$ represent constraints between those operations. A sample constraint graph is shown in Figure 5.3 for a 2-machine problem with 4 operations.

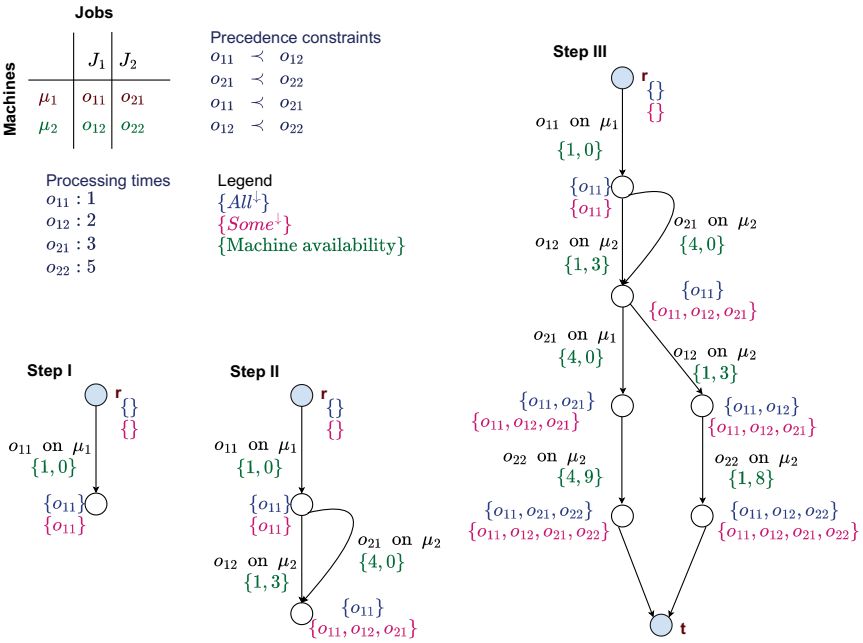

Figure 5.2.: Sample decision diagram showing the top-down construction process for the sample job set in Example 2

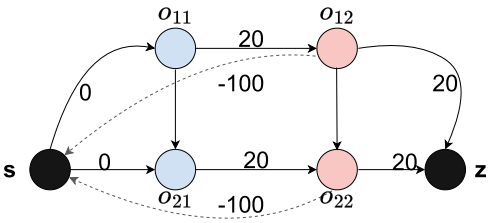

Figure 5.3.: Sample constraint graph for a 2-job 2-machine problem. Solid edges represent processing and setup time constraints while dotted edges represent deadline constraints. Deadlines are negated and their direction reversed.

The longest path computation for all nodes in X from the source node produces the earliest starting time of all operations est. Similarly, the latest starting time lst can be computed this way but with a negated relaxation condition in the longest path algorithm.

In the case where operations have non-zero release times, the longest path to an operation is always the maximum of the release time and the longest path found by the algorithm.

At the root of the decision diagram, only constraints that exist regardless of scheduling decisions are included in X. As we evolve the shop from state to state by making scheduling decisions, the constraint graph is updated per state by adding edges corresponding to the operation ordering in a state as constraints. The edges corresponding to a scheduling decision represent any mandated temporal delays between those operations. Earliest and latest start times can be updated for every state based on computations on a constraint graph updated with the scheduling decisions made at that state.

Updating Scheduling Decisions

A scheduling decision to place an operation o_a after an operation o_b results in an edge that has a weight equal to the sum of the processing time of o_a and any setup times that may exist between o_a and o_b. This edge update accounts for sequence-dependent constraints that only become active when certain scheduling decisions are made and thus, ensures that the start times derived from the constraint graph are indeed feasible.

Note that although longest path computations are known NP-hard problems, we can update start times in polynomial time because (i) whenever positive weighted cycles are found in this graph, we know such a graph represents an infeasible schedule and (ii) positive weighted cycles can be spotted in at most $O(|N||A|)$ using the Bellman-Ford-Moore longest path finding algorithm.

5.4.2. Merge Operators

In this section, we discuss a merge operator we have designed to achieve a relaxation for this problem and discuss its validity according to established conditions for valid merge operators in (Hooker 2017).

In order to create relaxed diagrams, we place a limit on the maximum number of nodes in a layer. To do this during top down construction, merging is typically applied where merging combines multiple nodes into one. However, since we place state information in the edges, we do not have to merge by explicitly recomputing state properties as in past work (Hooker 2017; van den Bogaerdt and de Weerdt 2018; van den Bogaerdt and de Weerdt 2019) but we can simply combine incoming edges to meet at a single node, making the set of outgoing edges a union of all outgoing edges and updating the node properties according to Equations (5.3) and (5.4) concludes the merge.

Concretely, to merge 2 nodes v and v' into a node v'', we create v'' as follows:

$$in(v'') = in(v) \cup in(v') \tag{5.8}$$

$$out(v'') = out(v) \cup out(v') \tag{5.9}$$

$$All^{\downarrow}_{v''} = All^{\downarrow}_{v} \cap All^{\downarrow}_{v'} \tag{5.10}$$

$$Some^{\downarrow}_{v''} = Some^{\downarrow}_{v} \cup Some^{\downarrow}_{v'}. \tag{5.11}$$

We can then update all the outgoing edges of v'' according to Equations (5.3), (5.4), (5.6) and (5.7), compute the est and lst vectors as detailed below, and delete v and

v'. Proof of the validity of this merge operator follows below.

Updating Scheduling Decisions in Relaxed Nodes

In the case of a relaxed decision diagram where there can be multiple non-equivalent paths that lead to a node as is the case with v'' above, it is not always clear what set of edges to add to the constraint graph in order to update start time vectors est and lst. In the case of a relaxation, we propose to ignore the sequence dependent edges so far and instead update the constraint graph at an edge $e = (u, v)$ by seeding starting times of the operations in $All^{\downarrow}_{v''}$ and $Some^{\downarrow}_{v''}$ as follows:

- For every operation, the start times of the corresponding node in the constraint graph is initialised as

$$\min\{est_e : e \in in(v'')\} \tag{5.12}$$
$$\max\{lst_e : e \in in(v'')\} \tag{5.13}$$

and then run the longest path computation as usual. After a merge, further transitions can continue to add explicit scheduling edges as in Section 5.4.1 until another merge requires this computation. An example merge is demonstrated in Step II of Figure 5.2.

Validity of Merge Operator

A valid merge operator is one that guarantees that the resulting decision diagram is indeed a relaxation. Hooker (2017) developed some general sufficient conditions for the validity of merge operators. In summary, a valid merge operation $v \boxplus v' = v''$ enforces that v'' is a relaxation of both v and v' where a state v'' relaxes v if

– (C1) All feasible transitions in v are also feasible in v''
– (C2) The cost of any transition from v'' is smaller or equal to the cost of the same transition in v,
– (C3) If v'' and v transition to states t'' and t respectively by scheduling the same operation j, t'' is also a relaxation of t.

Condition (C1) is trivially satisfied as we take the union of outgoing edges. Condition (C2) is equally satisfied because of the added condition that we take the minimum weight edge for any repeated edge labels when taking the union of outgoing edges. Condition (C3) is satisfied because all transitions from v'' satisfy (C1) and (C2) when compared to transitions from v. Take (C1) for instance, any transition from v'' satisfies the same precedence constraints as a transition from Sv and all new additions to the outgoing edges appear in both nodes. For (C2), since all outgoing edge properties in v'' are smaller or equal to outgoing edge properties in v, updating edge properties after similar transitions from v'' and v, maintains the same relationship between the values, therefore the new states after a transition also satisfy (C2).

Algorithm 6 adopts the top down construction algorithm from (Bergman, Cire, Van Hoeve, *et al.* 2016). Given a problem instance and a maximum diagram width, Line 2 creates the first layer containing only the root node we then begin layer by layer to construct the diagram. Lines 16-20 create child nodes and edge transitions

Algorithm 6 Top-Down Flat-Arc Construction

```
1: function TDCOMPILE(Maximum width W, multi-machine scheduling problem (J,M,O,C) )   ▷ returns decision diagram (V,E)
2:     create root and terminal nodes r, t
3:     l_1 ← {r}
4:     V ← l_1                                                                        ▷ set of diagram nodes
5:     E ← ∅                                                                          ▷ set of diagram edges
6:     for j = 1 to |O| do
7:         for v ∈ l_j do
8:             for val ∈ dom(v) do
9:                 create node v' and edge e
10:                update the state values of v' and e    ▷ as in Equations (5.3) to (5.7) and Section 5.4.1
11:                if v' is not infeasible then
12:                    connect v to v' via e
13:                    E ← E ∪ {e}
14:                    l_{j+1} ← l_{j+1} ∪ {v'}
15:        if l_{j+1} > W then
16:            l_{j+1} ← mergeNodes(l_{j+1}, W)                                       ▷ as in Equations (5.8) to (5.11)
17:        V ← V ∪ l_{j+1}
18:    Connect L_{|O|+1} to t
19:    return (V,E)
```

for each node in a layer according to the operations still feasible from the node. Nodes that are infeasible are pruned in Lines 13-15 and whenever we are on the last layer we connect all nodes to the terminal node t. Note that maximum width is set to *infinity* when not supplied and Algorithm 6 goes on to construct the exact DD.

5.5. Incremental Refinement

In this paradigm, we begin with a relaxed diagram, typically a single-width diagram where all transitions are possible from every layer and there is only one node per layer. We then progressively refine the diagram by splitting nodes and filtering infeasible edges.

In order to refine a relaxed diagram, there are two main steps necessary; a refinement step to undo relaxations and a filtering step to remove infeasible edges. The process of refining the decision diagram gradually unrolls the relaxation until each node is exact or another stopping condition is met.

In this section, we present filtering rules for the multi-machine scheduling problem. The edge and node representation is the same as in top-down construction with node update rules in Equations (5.1) to (5.4) and edge update rules in Equations (5.5) to (5.7) and Section 5.4.1 still valid.

Filtering Rules

We build on the work of Andersen *et al.* (2007) and Cire and Van Hoeve (2013) for defining filtering rules where we identify necessary feasibility conditions for an edge to exist in the diagram. These conditions are identified on a constraint by constraint basis, i.e., we define a filtering rule for each constraint in the problem. In the rest of this section, we define filtering rules for some popular scheduling constraints. Note that some of these constraints have been discussed by Cire and Van Hoeve (2013) for single resource scheduling and in the case where the rules directly apply to the multi resource case, we include a summary for completeness but point the reader to past work for details.

Ordering Constraints These are constraints pertaining to the order of operations in a sequence and not on any timing properties. There are two main constraints here namely precedence constraints and no-repetition constraints.

Precedence constraint filters follow exactly from the rule in (Cire and Van Hoeve 2013) and filter an edge $e = (u, v)$ if

$$\exists o \in (O \setminus Some_e^{\downarrow}) \quad s.t. \quad o \prec val_e, \tag{5.14}$$

which basically states that we filter out edge e if there $r \rightarrow u$ path that includes all its precedence requirements. This rule also applies to the multi resource case.

The *no repetition* constraint is very similar to the *AllDifferent* constraint (Van Hoeve 2001) and is typically the case that we only want to schedule an operation or a task once[1] and so ordering that include the same operation multiple times are invalid. We use the same rules as Andersen *et al.* (2007) for propagating the *AllDifferent* constraint and filter an edge $e = (u, v)$ if

$$val_e \in All_u^{\downarrow}, \tag{5.15}$$

$$|Some_u^{\downarrow}| = layer(u) - 1 \quad \wedge \quad val_e \in Some_u^{\downarrow}, \tag{5.16}$$

which states that edge e is infeasible if there is definitely another edge in every $r \rightarrow u$ path that includes the operations scheduled along e. This again also applies to the multi resource case and the proof is in (Cire and Van Hoeve 2013).

Timing Constraints There are 3 main timing constraints namely, *No overlap*, *Deadlines and maximum separation constraints*, and *Setup times*. The effect of each of these is contained in the earliest start time est_e and latest start time lst_e values of any edge e. However, because these values are largely bounds, we do not expect them to be correct unless the source node of an edge is *exact*. However, we can still filter when Equation (5.17) holds.

$$est_e > lst_e, \tag{5.17}$$

[1] There could be some edge cases in systems that include re-entrancy where one operation is processed multiple times but we can also recognise each re-entrance of the operation as a separate operation and the claims still hold.

The condition in Equation (5.17) is an obvious feasibility requirement as an edge that schedules an operation whose earliest and latest start times overlap violates a constraint (Cire and Van Hoeve 2013).

Additionally, we can also filter edges that do not improve the objective in any way. If known solutions already exist either as input or produced during the construction of the diagram, we can filter out any edges that are worse than the objective value of the best known solution.

5.6. The Role of Dominance

The decision diagram can grow exponentially, especially in problems with few precedence constraints. Apart from deliberately building a limited width diagram, the size of the decision diagram can also be reduced by pruning *dominated* nodes. A node is said to be dominated when there exists another node in the decision diagram whose future expansions will always result in a better objective. Whenever such an assertion can be made, it is safe to prune the dominated node and continue the decision diagram expansion, ignoring all future expansions of the dominated node.

Coppé (2024) provide a framework for modelling dominance in decision diagrams which our dominance conditions below fall under. We propose problem specific conditions for dominance below with one key difference: we assert that unlike (Coppé 2024), dominance does not only have to be computed between exact nodes but inexact nodes can also be dominated though they can never dominate.

5.6.1. Conditions for Dominance

In order to assert dominance, we require conditions under which nodes can be compared and rules to then compare nodes and edges. Definition 2 defines our dominance conditions. First, only exact nodes can be dominant but any other nodes can be dominated. This is because for all inexact nodes, state values held by their incoming and outgoing edges are only bounds and thus cannot produce definitive dominance relations. Secondly, we also assert that the set of operations that have definitely been scheduled in the path leading to the dominated node have also been scheduled by the dominant node (See Equation (5.18)). This allows us compare similar graphs as the remaining sub-problems from a dominated node are contained in the remaining sub-problems from a dominating node.

Thirdly, all future transitions from a dominated node must also be available from the dominant node as stated in Equation (5.19). This rule serves to preserve completeness of the diagram because we completely prune a dominated node, and do not want to exclude future possible solutions that are yet unseen. All the above conditions do not yet consider the timing. As a final step in the dominance check, we compare the timing properties of the outgoing edges.

Technically, nodes violating this Equation (5.18) but satisfying Equation (5.19) can still be checked for dominance and correctly pruned, i.e., $All^{\downarrow}_{v} \subset All^{\downarrow}_{v'}$. This is because even for a node v that has scheduled fewer operations than another node v', the set of sub-problems remaining from v' is contained in the set of sub-problems remaining from v provided the outgoing edges satisfy Equation (5.19). It is indeed expected that

node v is (i) higher up in diagram than v', and (ii) has better edge state variables and will likely dominate v'. However it is more algorithmically useful to prune nodes that are further up in the tree, as this reduces the size of the diagram more significantly. For instance, the root node in a 1-width diagram dominates all other nodes but pruning on this condition will result in re-expansion of all other nodes anyway. Thus, the utility of Equation (5.18).

Definition 2. *A node v dominates another node v' when v is* exact,

$$All_v^{\downarrow} \supseteq All_{v'}^{\downarrow}, \tag{5.18}$$

$$out(v) \supseteq out(v'), \tag{5.19}$$

and for every edge e with label $l \in out(v)$, the equivalent edge with label $l \in out(v')$ dominates e' for all constraints $c \in \mathcal{C}$.

We compare all outgoing edges of nodes that satisfy Equations (5.18) and (5.19). Edges are compared based on constraints thus for every constraint type $c \in \mathcal{C}$ in the problem, there should be a dominance rule that compares edges based on said constraint.

5.6.2. Dominance Rules for Scheduling Constraints

In this section, we define dominance rules for some popular scheduling constraints. Whenever these constraints are present, the set of rules pertaining to the available constraints can be applied one after the other and dominance is only asserted if all conditions hold. Note that we only define rules for constraints pertaining to the timing as ordering constraints, e.g., precedence constraints, do not contain information on the utility of a state towards the objective – they only directly affect feasibility.

No overlap constraints

Since we assume no overlap constraints and non-preemptive scheduling, machine availability serves as a lower bound for start times of any operation to be scheduled. We use this information to form a dominance rule based on the no overlap constraint. Dominance can be evaluated as in Definition 3.

Lemma 2. *For any nodes v and v' that satisfy Definition 2, all paths from $v' \rightarrow t$ are also available from $v \rightarrow t$.*

Proof. The proof is trivial from the observation that if nodes v and v' satisfy the conditions in Definition 2, then the same precedence constraints have been satisfied and that every scheduling decision available from v' is also available from v. ■

Definition 3. *An outgoing edge e' from node v' has been* dominated *according to the no overlap constraints if there exists another edge e from a node v such that*

$$\begin{aligned} a_e^{\mu_k} + \min\{\underline{\mathcal{S}}(o_v)\} \leq a_{e'}^{\mu_k} + \min\{\underline{\mathcal{S}}(o_v)\} \\ \forall o_v \in dom(destination(e)) \quad \forall \mu_k \in M, \end{aligned} \tag{5.20}$$

$$est_e^{o_a} \leq est_{e'}^{o_a} \quad \forall o_a \in O, \tag{5.21}$$

and the nodes v and v' satisfy Definition 2 where $\min\{\underline{\mathcal{S}}(o_v)\}$ *is the minimum setup time induced on operation o_v on machine μ_k .*

Theorem 2. *If edge e dominates another edge e' according to Definition 3, any sequence of edges forming a path from destination(e') to the terminal vertex t is also available from destination(e) to t with smaller or the same makespan.*

Proof. Let *destination*$(e') = v'$ and *destination*$(e) = v$. The first part of the proof follows from Lemma 2. If the set of operations completed is the same per Definition 2, then every sub-problem remaining to solve from v' is also in v. This means every path from v' to t is also available from v to t. The second part of the proof follows from the fact that though sub-problems in v' are also in v, v and v' present the sub-problems with different starting points. This effect is captured in machine availability because, if for instance v makes all machines available earlier than v', v gives every operation an earlier starting time than in v'. Since adding any operations to a schedule can only further increase the makespan or keep it the same, the sub-problem with the earlier starting point is thus guaranteed to result in a better makespan. ■

Setup Time Constraints

Setup times refer to minimum separation constraints between operations. They define a minimum time that must elapse between the completion of one operation and the start of another. A setup time between two operations o_a and o_b is defined as $\mathcal{S}(o_a, o_b)$. Setup times can be sequence independent – existing for every schedule – or sequence dependent – only defined for particular ordering of operations. The rules in Definition 3 also account for setup times and hold irrespective of sequence (in)dependence as all kinds of setup times can be considered in Equation (5.20).

Accommodating Varying Setup Time Constraints The dominance check in Definition 3 require us to compute $\min\{\underline{\mathcal{S}}(o_v)\}$, i.e., the minimum setup time induced on operation o_v. However, different kinds of setup times exist.

In general, we always want to use some lower bound $\underline{\mathcal{S}}$ on the setup time induced for an operation scheduled along an edge, i.e., we only add setup times we are sure have been incurred. Note that in an exact node, we actually know the exact setup time and the minimum values calculated below are the real setup times and not bounds. Below we address some popular setup time constraints and show how the minimum is calculated for an operation o.

- Sequence independent setup times between two operations that indirectly precede each other. These setup times occur irrespective of the exact sequence chosen. An example is the travelling time of an operation from one machine to the next. Such a setup time exists irrespective of the chosen sequence and as such is already included in the computations of earliest start times. Therefore, we do not need to consider these setup times in $\min\{\underline{\mathcal{S}}(o)\}$ as their effects are already captured in Equation (5.20) of Definition 3.

- Sequence independent setup times pertaining to a machine. It can also be that machines have fixed setup activities they must do before scheduling a particular operation. These are then easy to compute as each edge knows exactly the setup time it enforces on a machine (Equation (5.22)).

$$\min\{\underline{\mathcal{S}}(o)\} = \mathcal{S}(m(o), o) \tag{5.22}$$

- Sequence dependent setup times for sub-sequences on a machine. It is also more typical for sequence dependent setup times to exist for operations on the same machine. As our graphs flatten the sequences for all machines, we need a check to find the right setup time for this case. If we assume $Some_u^{\downarrow}$ is ordered by the layer in which an operation was added to it, we can find the right setup time as

$$\min\{\underline{\mathcal{S}}(o)\} = \mathcal{S}(o', o) : o' = \max\{a \in Some_u^{\downarrow} \quad s.t. m(o) = m(o')\}, \tag{5.23}$$

  where the $\max$ operator on a set finds the maximal element[2]. In the case of a tie, we break it by taking the minimum of all maximal elements[3].

- Purely sequence dependent setup times. These are a more natural extension of single resource scheduling and can occur in a case where all tasks irrespective of the machine it is assigned uses some other common resource, e.g., shared memory in multi processor or shared fuel tank in a manufacturing plant.

$$\min\{\underline{\mathcal{S}}(o)\} = \min\{\mathcal{S}(val_{e'}, o) : e' \in in(u)\}, \tag{5.24}$$

In the event that a problem has multiple of these setup time constraints, we then compute the maximum of all the minimum setup times incurred $\max\{\min\{\underline{\mathcal{S}}(o)\} \forall c_{setup} \in \mathcal{C}\}$ of all setup time constraints present – again assuming setup times are allowed to overlap.

Deadlines and Maximum Separation Constraints

Deadlines are constraints that enforce upper bounds on the start or completion times of operations. Deadlines can be absolute, when defined relative to time 0 or relative when defined relative to the start time of some other operations. Maximum separation constraints also enforce upper bounds on the start times of operations but define them relative to start times of other operations. Thus, maximum separation constraints can be thought of as relative deadlines and we handle them similarly.

[2] A maximal element of an ordered set is an element that is not smaller than any other element in the set according to the order. In this case, our $Some_u^{\downarrow}$ set is ordered by layer and so we pick the operation that was added on the layer closest to $layer(u)$.

[3] In a relaxed node, we can have multiple paths leading to the node as such an operation is not necessarily added to $Some_u^{\downarrow}$ in the same layer by all paths and multiple paths can add different operations to $Some_u^{\downarrow}$ in the same layer. We break this tie by picking the latest layer among all the paths whenever $Some_u^{\downarrow}$ is updated. Thus, we can have multiple maximal elements in $Some_u^{\downarrow}$.

Definition 4. *An edge e' with destination v' has been* dominated *according to deadline constraints if there exists another edge e with destination v such that such that*

$$lst_e^{o_a} - est_e^{o_a} \geq lst_{e'}^{o_a} - est_{e'}^{o_a} \quad \forall o_a \in O \tag{5.25}$$

and the nodes $source(e)$ and $source(e')$ satisfy Definition 2.

Theorem 3. *If edge e dominates another edge e' according to Definition 4, any sequence of edges forming a path from destination(e') to the terminal vertex t is also available from destination(e) to t with a larger or the same slack for every operation where slack is the difference between lst and est.*

Proof. Let $destination(e') = v'$ and $destination(e) = v$. The first part of the proof follows from Lemma 2 and Theorem 2 above. If the set of operations completed is the same per Definition 2, then every sub-problem remaining to solve from v' is also in v. This means every path from v' to t is also available from v to t.

The second part of the proof follows from the fact that though sub-problems in v' are also in v, v and v' present the sub-problems with different slack values and as such can overshoot the deadlines by different amounts. This effect is captured in Equation (5.25) because adding any operations to a schedule can only further decrease the slack or keep it the same. Therefore, the sub-problem with the larger slack is guaranteed to overshoot the deadline less. ■

Considerations

As seen from most of the rules defined above, information on starting time of all operations in the problem are required for quite a few dominance checks. It then means that a computation such as the constraint graph based state evaluation described in Section 5.4.1 is required. This computation, combined with the dominance comparison can be a significant expense when repeated often during the construction. However, when we actually encounter a dominated state, the time gains from pruning can also be significant.

Additionally, to properly situate these rules in the literature, we come back to the work of Coppé (2024) where two necessary operators are defined for dominance rules namely a *dominance key* operator and a *partial dominance utility* operator.

The *dominance key* operator maps a state to a reduced state value such that nodes with the same *dominance key* are comparable. For the rules defined above, we would want a dominance key that maps any two nodes that satisfy Equations (5.18) and (5.19) to fall under the same dominance key. If we remove the inequality and only compare nodes such that

$$All_v^{\downarrow} = All_{v'}^{\downarrow}, \tag{5.26}$$

$$out(v) = out(v'), \tag{5.27}$$

then we can define a dominance key such that every node is mapped to a binary sequence whose positions represent operations and values represent the presence of those operations. For example, recall the problem in Example 2. Assuming the chosen

order of the binary sequence is $o_{11}o_{21}o_{12}o_{22}$, a node given a node v with $All_v^{\downarrow} = \{o_{11}\}$ and $out(v) = \{o_{21}, o_{12}\}$ maps to the sequences $(1000, 0110)$.

The *partial dominance utility* operator characterises the utility of the node towards the objective. Two nodes are then compared for dominance based on their utility as defined by this operator. In this chapter, state values directly capture this utility as we can see that all timing related state values affect the makespan and are the basis of node comparisons in the earlier defined dominance rules. Therefore, our utility operator maps state values to themselves.

5.7. Decision Diagram Construction From Initial Solution

Motivated by the fact that heuristics still provide state of the art performance for many shop scheduling problems, we design an approach to start out construction of the DD from an initial solution. We do this by first constructing a path corresponding to the initial solution(s) before continuing the DD exploration. Other exact solvers such as constraint programming solvers and mixed integer solvers also have the ability to start from an initial solution — which could be a complete or partial assignment of variables — known as a *warm start*. The existence of such an initial path can also serve to guide the construction process as opposed to the strict layer by layer approach and recent work by Gillard and Schaus (2022) has shown that good performance can be obtained by combining Large Neighbourhood Search (LNS) with decision diagrams in a way that solutions in the neighbourhood of the initial solution are explored first.

We begin construction of the decision diagram as usual from the root node r; r has no in-degree edges as no sequencing decision is made for any machine. To expand the decision diagram, we begin with a path corresponding to an initial solution Ω. Recall that a schedule is defined as a sequence of operations; as such, a solution Ω represents an ordering of all operations. Therefore, we can create a path corresponding to Ω, i.e., one edge for every operation according to the given ordering. This is illustrated by Step 1 of Figure 5.4. Edges and nodes corresponding to the initial solution thus form the starting decision diagram.

We continue to expand the decision diagram and transition states as in Section 5.4. We can choose to construct layer by layer but in order to take advantage of the initial solution, we can also select the next node to expand based on the quality of the partial solution at that node as in (Horn, Maschler, *et al.* 2021) or the proximity of the node to our existing solution as in (Gillard and Schaus 2022). Hence, we compile our decision diagram using a top-down approach but not necessarily layer by layer. Note that the quality of the partial solution at a node also serves as a lower bound on the possible eventual solution from that node. This is because our constraint graph based start time update always computes the start time of all operations in the constraint graph including the terminal node (Section 5.4.1). Thus, we can prune a node when its bound is worse than a known solution.

Example 3. As an example to help explain our approach, let us take a small scheduling problem to optimally schedule four operations $\{a, b, c, d\}$ on a single machine. In this problem, there are no precedence constraints, we only have one

machine. We also have an initial solution $\{a, b, c, d\}$ provided by a heuristic. For simplicity, we ignore timing information.

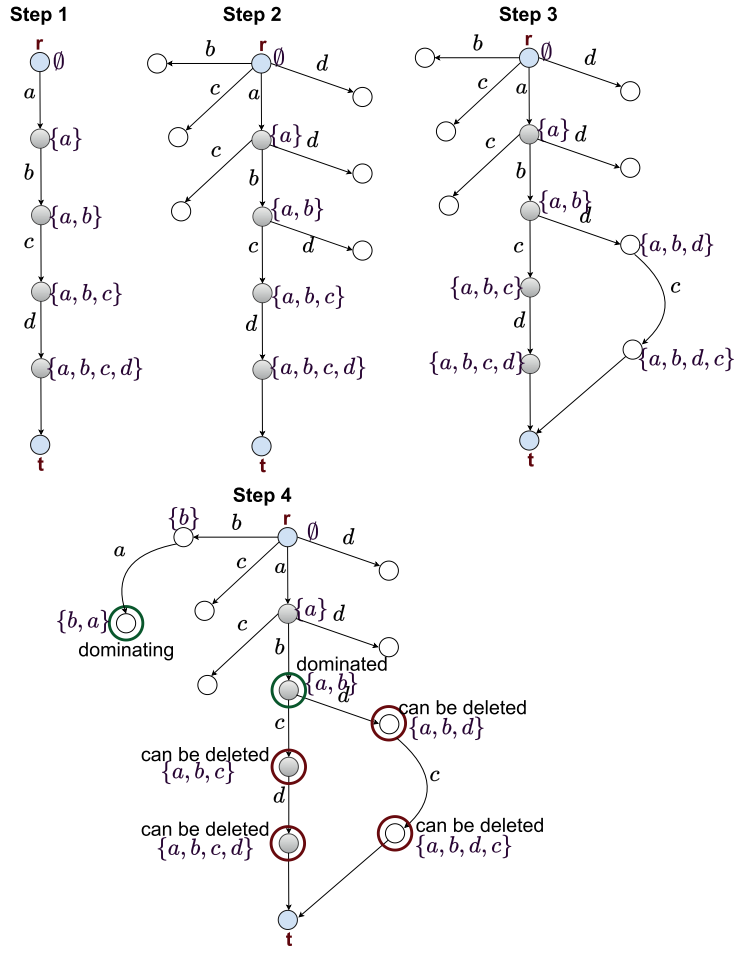

Figure 5.4.: Construction of a decision diagram from an initial solution for the problem in Example 3

The construction process for Example 3 is illustrated in Figure 5.4. In Step 1, we create the decision diagram initialising it with a path corresponding to the initial solution. Steps 2 and 3 show how we continue expanding the diagram and Step 4 shows a point where our expansion curbing techniques kick in to prune some parts of the diagram.

Algorithm 7 formally shows the construction process of a decision diagram from an initial solution. We start by constructing a path according to said solution (Lines 2–6). Typically, such an initial solution is derived from a heuristic. After we initialise the decision diagram with the path corresponding to the initial solution, we explore states

until a stopping criterion is met – in this case, until all states have been explored. Lines 8–11 handle the expansion of a node while lines 12–20 evaluate the newly formed nodes discarding those that are infeasible, dominated or will lead to poorer solutions than we already have. At the end of the algorithm, the best solution found is returned and lines 13–18 detect when a solution has been found and update the best solution accordingly.

Algorithm 7 Decision Diagram Construction From Initial Solution

1: **function** DDSEED(Multi-machine scheduling problem $(J, M, O, \mathcal{C})$, Seed heuristic solution Ω) ▷ **returns** Schedule Ω
2: $UpperBound \leftarrow Makespan(\Omega)$
3: create root and terminal nodes r, t
4: $L \leftarrow \{r\}$ ▷ Set of leaf nodes ordered by node selection scheme
5: $L_\Omega \leftarrow$ all nodes in $r-t$ path corresponding to Ω
6: $L \leftarrow L \cup L_\Omega$
7: **while** $L \neq \emptyset$ **do**
8: $v \leftarrow$ select node from L
9: **for** $o \in \text{dom}(v)$ **do**
10: Create new node v_o and edge e
11: ▷ as in Equations (5.3) to (5.7) and Section 5.4.1
12: **if** v_o is not *infeasible* **and** not *dominated* **then**
13: **if** all operations have been scheduled **then**
14: attach t to v
15: $\Omega_v \leftarrow r-t$ path passing through v
16: **if** $Makespan(\Omega_v) < UpperBound$ **then**
17: $UpperBound \leftarrow Makespan(\Omega_v)$
18: $\Omega \leftarrow \Omega_v$
19: **else if** $\min\{\max\{a_k \forall \mu_k \in M\} \forall i \in in(v_o)\} < UpperBound$ **then**
20: ▷ Discard nodes worse than known best
21: $L \leftarrow L \cup \{v_o\}$ ▷ Update leaf nodes
22: $L \leftarrow L \setminus \{v\}$ ▷ remove v from leaves left to explore
return Ω

5.8. Computational Results

In this section, we evaluate the empirical performance of our solution for the two problem classes described in Section 5.8.1. We compare our approach with other exact solution approaches in the literature, namely, integer and constraint programming[4].

[4]The integer and constraint programming models are in Appendix C.

5.8.1. Problem Classes

Basic flow shops

Basic flow shop problems are common methods for modelling manufacturing systems where jobs have to be processed on a set of machines in a specific order (Emmons and Vairaktarakis 2012). A flow shop scheduling problem can be modelled using the same nomenclature as in Section 5.2 with jobs $J = \{j_1, ..., j_n\}$, machines $M = \{\mu_1, \ldots, \mu_m\}$ and operations O where operation $o_{ij} \in O$ is the jth operation of the ith job and has a processing time P_{ij}. In a *flow shop* problem, operations of each job visit the machines in the same order. In three-field notation (Graham *et al.* 1979), a flow shop can be expressed as $F||C_{\max}$ respectively where C_i is the completion time of J_i and subsequently, $C_{\max}$ is the makespan, i.e., the maximum completion time of any job in the flow shop. This basic version of the problem has no deadlines and no setup times. The only property of operations is the machines they belong to and their processing times on those machines. The problem can thus be modelled as a tuple as a tuple (M, J, O, P)

Large Scale Printing Use Case

We also discuss the operation of our DD formulation on the industrial use case of a large-scale printer (LSP) as described in Section 2.2. The LSP prints different types of duplex sheets that need to be processed twice by the same print head at a speed of 100 or more pages per minute. Jobs to be scheduled refer to sheets to be printed. The problem of scheduling sheets in the LSP can be modelled as a fixed-order flow shop with sequence-dependent set-up times and maximum separation constraints between operations where maximum separation constraints are relative deadlines between operations. The flow shop is such that there are three machines in total and one of them is a re-entrant machine that all operations have to visit twice. Formally, the problem is a tuple $(M, J, O, P, \mathcal{S}, T)$ where M, J, O, and P retain their definitions from the basic flow shop. Moreover, $\mathcal{S}(o_a, o_b)$ refers to setup times – which represent the required delay between the completion of an operation and the start of another operation while $T(o_a, o_b)$ refers to the maximum delay between the start times of two operations. Maximum separation constraints, T, can be thought of as relative sequence-independent deadlines. In three-field notation, this is the $F|s_i, s_{ij}, limited-wait|C_{\max}$ problem.

There is no special adjustment required to the specific constraints of this problem as maximum separation constraints can also be directly added as edges to the constraint graph representation of this problem, keeping our updates for est and lst values correct for this use case.

5.8.2. Experimental Setup

We experiment with the version of our decision diagram that uses a warm start as in Section 5.7. In each of these evaluations, "DD-BF", "DD-DF", and "DD-Bal" refer to the DD with best-first node selection – where nodes are ordered based on the lower bound at that node, depth-first node selection – where nodes are ordered based on their depth, and a balanced node selection – where nodes are ordered based on a

weighted sum of both the normalised lower bound and the depth, respectively. The balanced node selections uses weights 0.8 and 0.2 for the lower bound and depth respectively. Similarly, "MIP" and "CP" refer to the mixed integer and constraint programming solutions. In order to keep all the comparisons fair, the "MIP" and "CP" experiments are provided the same initial solutions given to the decision diagram solutions as a warm start.

All experiments are performed on a 16-core 1.9GHz AMD machine running Ubuntu 20.04 with 32GB RAM with a 15 minute time-out. Our algorithms are implemented in C++ and the MIP and CP models are solved by CPLEX version 22.1 and CP Optimizer version 22.1, respectively.

Numeric results are as shown in Tables 5.1 and 5.3 with the best solution of each instance in bold. We evaluate solution approaches based on two criteria, namely (i) the optimality gap (lower is better) and (ii) the percentage of instances solved in the given time budget (higher is better). The optimality gap is computed as the distance of the best solution found by a method to the highest lower bound on the objective reported by any of the methods.

5.8.3. Results

Basic flow shops

We base our evaluation on known benchmarks in literature (Demirkol, Mehta, and Uzsoy 1998). These benchmarks are a set of randomly generated instances. Each instance is picked such that the number of machines is in the set {15,20}, the number of jobs is in the set {20,30,40,50}, and processing times are uniformly distributed in the interval [1,200]. Here, the initial solution is provided by the Nawaz-Enscore-Ham (NEH) Heuristic (Nawaz, Enscore Jr, and Ham 1983).

Experiments show that our DD solution is competitive with the CP solution and MIP performs poorly. In the results shown in Table 5.1, our DD based solution provides the best optimality gap in three(3) of eight(8) instance sizes while the CP solution provides the best optimality gap in the other five(5). The MIP solver is not the best solver for any instance size and is also not able to solve all problems for any instance size. The kind of search approach used within the DD also has a huge effect on its performance. DD-BF scales the least and is only able to solve 100% of

M/Cs	Jobs	Optimality Gap(%)					Percentage Solved				
		CP	MIP	DD-Bal	DD-DF	DD-BF	CP	MIP	DD-Bal	DD-DF	DD-BF
15	20	**2.23**	3.52	3.76	4.24	4.24	100	70	100	100	100
	30	2.61	2.75	**2.29**	2.91	2.91	100	30	100	100	100
	40	**1.70**	1.72	2.77	3.04	2.88	100	20	100	100	90
	50	**1.93**	-	3.12	3.36	3.94	100	0	100	100	40
20	20	**4.48**	7.81	7.96	8.65	8.65	100	50	100	100	100
	30	**5.74**	9.11	7.53	8.01	8.01	100	10	100	100	100
	40	4.84	8.21	**4.65**	5.06	-	100	20	100	100	0
	50	6.85	-	**4.63**	4.84	-	100	0	100	100	0

Table 5.1.: Evaluation on basic flow shops.

problems for instance sizes up to 30 jobs. While CP provides the best solution in many instances, it is interesting to note that the average difference in optimality gap between the CP solution and the best performing DD solution (DD-Bal) is only 0.9%.

Large Scale Printing Use Case

Here, we generate benchmarks according to typical occurrences in the industrial use case of an LSP as in Section 5.8.1. Different types of jobs exist with properties as described in Table 5.2. We generate benchmarks with mixtures of job types such that jobs of a given type appear in repeated blocks. We randomise the length of the blocks and the number of times these blocks repeat to mimic arrival patterns of jobs in practice. This creates representative job sets because in the LSP, multiple copies of the same sheet (job) type often have to be printed at the same time. Initial solutions are provided by the Bounded Heuristic Constraint Scheduler (BHCS) (van Pinxten *et al.* 2017), the state-of-the-art heuristic for this problem.

From Table 5.3, an immediate observation is that neither MIP nor CP solvers scale to larger instances of this problem despite having a warm start. The most common reason was high memory requirements of the solvers leading to termination by the operating system and eventually no reported solution. Other unsolved instances simply timed out. This means that for this problem class, our solver was the only scalable one. All solvers are able to solve the smallest instance size to optimality while CP finds smaller makespans (indicated by smaller optimality gaps) than our DD solver until instance sizes of 200 jobs. MIP is able to solve much less instances, with the percentage of instances solved dropping to 90% for 50 jobs and 0% for all other instance sizes. The poor performance of MIP across all metrics is not surprising as previous work has shown the superiority of constraint programming for similar problems (Lunardi *et al.* 2020).

Type	$P(o_{i1})$	$P(o_{i2})$	$P(o_{i3})$	$P(o_{i4})$	$T(o_{i1}, o_{i2})$	$T(o_{i2}, o_{i3})$	$T(o_{i3}, o_{i4})$
0	0.25	0.30	0.30	0.21	0.85	12.30	1.00
1	0.35	0.42	0.42	0.30	0.95	12.42	1.12
2	0.50	0.59	0.59	0.42	1.10	12.59	1.29
3	0.70	0.84	0.84	0.60	1.30	12.84	1.54
4	0.99	1.19	1.19	0.85	1.59	13.19	1.89

(a) Job processing times and maximum separation values

Machine	Setup Time
μ_1	0.20
μ_2	0.05
μ_3	1.00

(b) Machine setup times

Path	Travelling Time
μ_1 to μ_1	0.60
μ_2 to μ_2	10.00
μ_2 to μ_3	0.70

(c) Job travelling times

Table 5.2.: Properties of jobs in the industrial use case. All timings are in seconds and job travelling times are treated as setup times between operations of the same job.

Jobs	Optimality Gap(%)					Percentage Solved				
	CP	MIP	DD-Bal	DD-DF	DD-BF	CP	MIP	DD-Bal	DD-DF	DD-BF
5	**0.00**	0.00	0.00	0.00	0.00	100	100	100	100	100
10	**0.00**	0.00	0.00	0.00	0.23	100	100	100	100	100
50	**0.13**	2.83	3.12	3.07	3.95	100	90	100	100	100
100	**0.90**	-	2.34	2.33	2.55	100	0	100	100	100
150	**2.24**	-	5.81	5.81	5.95	100	0	100	100	100
200	**2.61**	-	4.03	4.03	4.13	100	0	100	100	100
300	-	-	**7.53**	**7.53**	9.08	0	0	100	100	80
500	-	-	**6.57**	**6.57**	7.55	0	0	100	100	85
1000	-	-	**5.35**	**5.35**	6.07	0	0	100	100	88
2000	-	-	**4.84**	**4.84**	**4.84**	0	0	100	100	100

Table 5.3.: Evaluation on the industrial use case.

5.9. Conclusions and Future Work

This chapter introduced a generic decision diagram formulation for multi-machine scheduling. We proposed state representations for both top-down compilation and incremental refinement, defined dominance and filtering rules, and additionally proposed a scheme to warm-start the diagram construction with an initial solution.

This decision diagram, like many others, incorporates heuristic decisions in its operation. An interesting future direction would be to explore how much machine learning can improve these decisions. Additionally, in the methods presented in this chapter, every state is evaluated by running a longest path computation on the constraint graph. Faster but equally general ways to evaluate a state is an interesting future direction.

6

Comparative Study of Decision Diagram Compilation

Decision diagrams (DDs) have shown great promise for solving combinatorial optimisation problems, leading to many innovations in diagram compilation such as rough upper bounds, caching, dominance, and clustering. However, these innovations are often studied in isolation and the combined effects on a decision diagram are not quantified. In this chapter, we look into combinations of techniques for three DD tasks: merging nodes, using dominance rules and variable ordering. We provide insights on what classes of problems are better suited to what compilation techniques via empirical evaluations on a set of seven popular optimisation problems. We look at both top-down compilation and incremental refinement, and convert existing merge heuristics to perform as split heuristics. We find that the mode of compilation does not affect the effectiveness of a technique and that some merge heuristics significantly reduce the effectiveness of dominance rules. Our results provide an important understanding of decision diagram compilation both for compiling individual DDs and for integrating them in optimisation solvers.

6.1. Introduction

Initially introduced for representing boolean circuits, decision diagrams (DDs) are graphical structures capable of compactly representing the solution space of combinatorial optimisation problems. DDs are constructed via recursive dynamic programming formulations and bear very close similarities with Markov Decision Processes (MDP) (Bergman, Cire, Van Hoeve, *et al.* 2016). Additionally, they are tunable to represent approximations of the solution space as in *restricted* DDs, where the space is under-approximated, and *relaxed* DDs, where the space is over-approximated. The process of constructing a DD is also referred to as *compilation*.

DDs have been used to tackle optimisation problems as early as the 2000s (Andersen *et al.* 2007; Becker *et al.* 2005) where they were integrated into other solvers for *cut generation* in linear programming and *domain storage* in constraint programming, respectively. We point the reader to (van Hoeve 2024) for an in-depth explanation of the various applications of DDs to solving optimisation problems. Bergman et al. (Bergman, Cire, van Hoeve, *et al.* 2016) introduced a DD-based branch-and-bound (DD-BnB) algorithm which works by progressively creating restricted and relaxed diagrams from different nodes in the DD to provide solutions and bounds on the objective function, respectively.

Since the introduction of DD-BnB (Bergman, Cire, van Hoeve, *et al.* 2016), many improvements have been developed. Gillard et al. (Gillard, Coppé, *et al.* 2021) introduce two bounding techniques – rough upper bounds and local bounds – which both lead to better selection of which nodes to expand or branch on in DD-BnB. Coppé et al. (Coppé, Gillard, and Schaus 2024a) introduce a caching mechanism that addresses the fact that DD-BnB branches on nodes that represent overlapping sub-problems such that exactly the same solution can be reached in different branches. This is in stark contrast to other branch-and-bound solvers that split the solution space on every branching decision. The cache introduced by Coppé et al. (Coppé, Gillard, and Schaus 2024a) stores a threshold for expansion of a node leading to increased expansion efficiency of DDs.

Further, techniques have also been developed for improving the actual compilation of the DD, either during a relaxation or a restriction. For instance, dominance, a concept that has long existed for state-based search was recently formalised for DD solvers (Coppé, Gillard, and Schaus 2024b). Some other ideas such as using machine learning to improve certain aspects of DD compilation (Cappart, Goutierre, *et al.* 2019; Nafar and Römer 2024b), such as variable ordering or node selection, have also been explored. However, studies of the *comparative and combined impacts of these techniques* have, to the best of our knowledge, not been presented. The work presented in this chapter is a step towards closing this gap.

Specifically, this chapter contributes a portfolio study of compilation techniques for DDs for optimisation. We focus on three techniques, namely *merge heuristics, variable ordering* and, *dominance*. We evaluate these techniques empirically and also compare their combined effects. We find that (i) the effect of dominance is reduced when nodes in the DD are merged strictly based on their similarity, (ii) variable ordering is complementary to any merge heuristic or dominance rule, and, (iii) the compilation technique has no impact on the effectiveness of dominance rules or merge (split)

heuristics.

Such a portfolio study is important for at least two reasons. Firstly, when used for optimisation, DDs either involve continuously improving an already existing DD – as in their combination with constraint programming – or creating many relaxed and restricted diagrams for sub-problems – as in their use in branch-and-bound solvers. Thus, an understanding of compilation techniques is important both for stand-alone DDs and for their integration in other solvers. Secondly, with wider adoption of DD solvers as is evidenced by the existence of two open source solvers – DDO (Gillard, Schaus, and Coppé 2020) on which our experiments are based and CODD (Michel and van Hoeve 2024), users are faced with a suite of options to choose from when modelling and solving problems. This makes knowledge of the effects of modelling and heuristic choices important for users of such solvers.

The rest of the chapter is organized as follows: Section 6.2 explains the studied techniques, Section 6.3 introduces the problem classes used in our experimentation, while Section 6.4 discusses empirical evaluations of the techniques. In Section 6.5, we show how these techniques work for incremental refinement compilation. Section 6.6 discusses our findings and Section 6.7 concludes by highlighting opportunities for future work.

6.2. Compilation Techniques Studied

We next discuss the intelligent compilation techniques evaluated in this chapter. We look at *merge heuristics, variable ordering*, and *dominance* due to their proven impact on improving solutions and bounds derived from DDs (Bergman, Cire, Van Hoeve, *et al.* 2012; Cappart, Goutierre, *et al.* 2019; Coppé 2024; Frohner and Raidl 2019; Nafar and Römer 2024a). We refer strictly to top-down compilation here. In Section 6.4, we show their impact on top-down compilation and we extend the techniques to incremental refinement in Section 6.5.

Dominance

Dominance relations are comparative relations on sub-problems that conclude if a sub-problem can be excluded or pruned from the state space without compromising the objective. They have been studied as early as 1977 in the general context of branch-and-bound algorithms (Ibaraki 1977) and later for combinatorial optimisation in general (Jouglet and Carlier 2011). They can also be directly applied to DDs as each node in a DD represents the root of a sub-problem, i.e., nodes can dominate each other and dominated nodes can be pruned. Dominance rules have been formulated for different problems with dynamic programming and DD formulations (de Weerdt, Baart, and L. He 2021; Galand, Lesca, and Perny 2013) and recently, a general modelling language for integrating dominance rules in DDs was proposed (Coppé, Gillard, and Schaus 2024b). These kind of rules help to prune nodes before we ever expand them, thus shrinking the state space and leading to faster diagram compilation.

Variable Ordering

All edges leading to the nodes in a layer of a typical decision variable represent value assignments to a particular decision variable. The quality of solutions and bounds produced by a DD is tightly related to which variable is dealt with in which layer, i.e., the order in which variables are considered (Bergman, Cire, Van Hoeve, *et al.* 2012). However, finding the best variable ordering is itself NP-hard (Bollig and Wegener 1996), and state-of-the-art heuristics are often problem-specific (Nafar and Römer 2024a). One general proposal is to use reinforcement learning to create variable ordering heuristics (Cappart, Goutierre, *et al.* 2019). This eliminates the need to hand craft heuristics for each problem and has been successful for the maximum independent set problem (Cappart, Goutierre, *et al.* 2019).

Merge Heuristics

In building relaxed diagrams, the size of the diagram is limited by fixing the width W (number of nodes) of any layer in the diagram. Whenever a layer exceeds the desired width, some of the nodes are merged such that the target width is achieved. There are two main questions in merging nodes in relaxed diagrams are: (i) how to select which nodes to merge and (ii) how to combine these nodes. The selection is done by a merge heuristic and the merge itself is carried out by a merge operator. Different merging algorithms have been proposed with the most popular being *minLB* (sometimes referred to as *sortObj* in the literature) (Frohner and Raidl 2019; Nafar and Römer 2024b), which works by ordering all the nodes in a layer by their quality – where node quality is typically the quality of the bounds at that node but can be customized for a specific problem, keeping the $W-1$ best nodes and merging all the rest into a single node.

Merging nodes is based on a merge operator that combines the states represented by the nodes to be merged such that the resulting merged node is a relaxation of all the merged nodes, i.e., all feasible paths from each of the nodes still exist after merging and the value of the merged node is an over-approximation[1] of the value of any nodes that were merged. For this reason, whenever nodes are merged, some information is lost in the DD. It then follows that the similarity between the merged nodes matters and merging more similar nodes can reduce this information loss. Recognizing this, Frohner and Raidl (Frohner and Raidl 2020) introduced a measure of node (dis)similarity and designed a merge heuristic based on node similarity. This idea has been pushed even further by clustering all the nodes in a layer into W clusters with each cluster resulting in one merged node (Nafar and Römer 2024b).

Rough Upper Bounds

A rough upper bound on a node v_i in a DD gives a bound on the length of the best possible path originating from v_i. While the use of a rough upper bound is more a technique introduced for DD-BnB (Gillard, Coppé, *et al.* 2021) and is not studied in

[1] The value of merged nodes is an over-approximation in maximization problems and an under-approximation in minimization problems.

this chapter as a compilation technique, we introduce it here because it is used in some other ways such as computing the merge error (Section 6.4.1).

6.3. Problem Domains

We consider 7 optimisation problems of diverse nature covering both binary DDs and multi-values DDs. Below, we discuss each of these problems, describing the dynamic programming models and data sets used in our empirical evaluations. Not all problem classes have all modelling components available either due to the structure of the problem or due to them – to the best of our knowledge – not being developed yet. For instance, scheduling problems are not known to have variable ordering heuristics in the DD context and as such will be excluded from those experiments. Table 6.1 gives a summary of the modelling components available for each problem and detailed models used in our experiments are available in the appendix.

Our experiments are carried out on a generic solver, DDO (Gillard, Schaus, and Coppé 2020) on the problem domains below. DDO is publicly available[2] and is implemented in Rust. In all instances, we compare the optimality gap achieved with the width of the DD. The width is fixed at values in $\{20, 50, 100, 200, 500, 1000\}$ and the gap is computed by comparing the best solution found by DDO using branch-and-bound in 2 minutes. All experiments are run on a 16-core 1.9GHz AMD machine running Ubuntu 20.04 with 32GB RAM.

Talent Scheduling Problem (TSCHED) The talent scheduling problem has to do with determining the optimal sequence in which a set of scenes S should be shot

[2] https://github.com/xgillard/ddo

Acronym	Problem	Dominance	Rough Upper Bound	Variable Ordering	BDD	MDD
ALP	Aircraft Landing Problem	✓				✓
KNAPSACK	Knapsack	✓	✓	✓	✓	
LCS	Longest Common Subsequence	✓			✓	
MISP	Maximum Independent Set		✓	✓	✓	
SOP	Sequential Ordering		✓			✓
TSCHED	Talent Scheduling	✓	✓			✓
TSPTW	Travelling Salesperson with Time Windows	✓	✓			✓

Table 6.1.: Model Components Available per Problem

on a movie production set with a set of actors A. In this problem, each scene s_i has a duration d_i and requires a subset of actors $Q_i \subseteq A$. The goal is to order the scenes such that we minimize the total cost of actors bearing in mind that an actor is to be paid for the entire duration from the first to the last scene they participate in even if they are sometimes idle in between. This problem has been solved via constraint programming (B. M. Smith 2003), variable neighbourhood search (Ranjbar and Kazemi 2018), and dynamic programming (Garcia de la Banda, Stuckey, and Chu 2011) – which we use in our DD model. The state representation is such that a state $s = (\mathcal{Q}, \mathcal{A})$ where $\mathcal{Q}$ is the set of scenes still to be scheduled in that state, $\mathcal{A}$ is the set of actors on set in that state. Each transition chooses a scene to be scheduled and a path is terminal when all scenes have been scheduled. The instances used in our experiments are the base set of instances from (Garcia de la Banda, Stuckey, and Chu 2011) which are themselves derived from (Cheng, Diamond, and B. M. Lin 1993; B. M. Smith 2005).

Sequential Ordering Problem (SOP) The sequential ordering problem (Escudero 1988) involves finding a permutation of operations such that the total cost is minimized given a set of precedence constraints. The problem is often represented as a weighted graph $G = (V, E)$ with a set of precedence constraints between vertex pairs in V where the goal is to find the minimum cost Hamiltonian path[3] in the graph and the cost of a path is the sum of all the edge weights along the path. This problem has many applications in manufacturing, transportation, and even compiler design (Shobaki and Jamal 2015). A DD formulation for problems of this kind has been proposed in (Cire and Van Hoeve 2013). We use the state representation in (Cire and Van Hoeve 2013) for sequencing problems minimizing setup times where setup times are analogous to graph weights in the SOP. A state is thus represented as $s = (\mathcal{J}, \mathcal{X})$ where $\mathcal{J}$ is the last sequenced vertex (which can be a set in a relaxed node) and $\mathcal{X}$ is the set of vertices that still need to be placed. The instances used in our experiments are the set of SOP instances from TSPLIB (Reinelt 1991).

Maximum Independent Set Problem (MISP) Given a graph $G = (V, E)$ with set of vertices $V = \{v_1, \ldots, v_n\}$ and a set of edges E, the MISP aims to find the largest subset $X \subseteq V$ such that no two vertices in X share an edge. The problem can be weighted such that each vertex v_i is assigned a weight w_i and the goal is then to select the independent subset that maximizes the total weight. A state is represented as $s = \mathcal{V}$ where $\mathcal{V}$ is the set of vertices still available to be chosen. The instances used in our experiments are the complement graphs of the DIMACS clique instances (Johnson and Trick 1996).

Knapsack Problem (KNAPSACK) Given a knapsack with capacity C and a set of N items each with weight w_i and profit p_i, the aim is fill the knapsack with a subset of $X \subseteq Y$ items such that total profit is maximized without exceeding the capacity. We use the BDD formulation of this problem where the only branching decisions are

[3] A Hamiltonian path is a path in a graph that visits every vertex exactly once.

whether or not an item is selected to be in the knapsack. A state $s = \mathcal{K}$ is represented by a single variable $\mathcal{K}$, the remaining capacity of the knapsack. The instances used in our experiments are the set of low-dimensional and larger scale instances from (Pisinger 2005).

Aircraft Landing Problem (ALP) Given a set of aircraft N and a set of runways R, where the set of aircraft is partitioned into classes such that there is a minimum separation time required between landing aircraft in different classes; the goal of the ALP is to schedule the landing of aircraft on runways such that the total delay is minimized while respecting earliest and latest landing time constraints. We use the dynamic programming model and data sets presented in (Coppé, Gillard, and Schaus 2024b). The model is such that a state is a tuple $s = (\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ of three vectors: $\mathcal{X}$, which holds the number of unscheduled aircraft per class and $\mathcal{Y}$, which holds the earliest possible occupation times per runway and, $\mathcal{Z}$ which holds the set of classes of airplanes possibly scheduled last per runway. The problem instances are randomly generated, with number of aircraft in $\{25, 50, 75, 100\}$, runways in $\{1, 2, 3, 4\}$, and 4 aircraft classes.

Longest Common Subsequence Problem (LCS) Given a set of $M = \{m_1, \ldots, m_n\}$ input strings, the LCS problem aims to find the longest string that is a common subsequence of all input strings over an alphabet σ. DD formulations exist for this problem and we use the formulation in (Horn, Djukanovic, *et al.* 2020; Horn, Maschler, *et al.* 2021) where a state is defined as a position vector $\mathcal{P}$ such that the i^{th} position represents the current position in string m_i. The instances used in our experiments are the RAT, VIRUS, and RANDOM instances from (Shyu and Tsai 2009).

Travelling Salesperson with Time Windows Problem (TSPTW) Given a set of cities represented as a graph $G = (V, E)$ where V is a set of vertices representing the cities and E is the set of edges representing the cost of traveling between cities, the traveling salesman problem involves finding a minimum cost Hamiltonian cycle in G such that each vertex of the graph is reached within a specified time window. A state is represented as $s = (p, t, \mathcal{V})$ where p is the position of the salesman, t is the elapsed time since the salesman begun, and $\mathcal{V}$ is the set of vertices left to visit. The instances used in our experiments are the AFG set of instances from (Ascheuer 1996) and we use the TSPTW dynamic programming model in (Gillard 2022).

6.4. Top-Down Compilation

So far, we have introduced DDs and intelligent compilation techniques for them. We now explore the impact of individual techniques on the resulting DD in Section 6.4.1 and later in Section 6.4.2, we explore the effects of combining these techniques.

6.4.1. Impact of Individual Techniques

Merge Heuristics

In this section, we discuss the impact of merge heuristics on the DD. We compare the two extremes of *minLB* where keeping the best quality nodes exact is the only goal and *cluster* where merging the most similar nodes is the only goal. There are other heuristics in the literature that find a balance between the goals of preserving the best quality nodes and reducing node dissimilarity in merges (Frohner and Raidl 2020). These are not covered by our experiments but similar conclusions should hold as they are a mixture of *minLB* and *cluster*.

While (Nafar and Römer 2024b) shows that clustering is better technique for BDDs compared to *sortObj*, we find that the situation is more nuanced especially when extended to MDDs. In Figure 6.2, we compare the effects of *minLB* and *cluster* on 5 problem classes – KNAPSACK, MISP, SOP, TSCHED, and TSPTW – for which rough upper bounds – used in our estimation of merge error – have been defined in previous literature. In our experiments, the quality metric of a node is the length of the (shortest) longest path leading to that node and we set it up such that *cluster* is also based on this quality metric as the only feature[4]. The results show that the performance of a merge heuristic is dependent on the *merge error* – a measure of the information lost by combining nodes.

Furthermore, we find that the primary source of the merge error is not the merge heuristic but the merge operator itself, i.e., the operator used to combine the nodes, and the merge heuristic can mitigate the effect of errors induced by a weak merge operator by only merging already bad nodes. Example 4 shows how errors can be induced in merging depending on the choice of operator.

Merge Error The idea of quantifying the strength of a relaxation has been studied in different optimisation fields (Averkov, Hojny, and Schymura 2023; T. He and Tawarmalani 2024) and in the case of DDs, this is akin to quantifying the quality of a merge. A node merge quality metric exists for DDs (Frohner and Raidl 2019) and is extended in this chapter to estimate a merge error over a DD. This metric works as follows: say we merge n nodes $V = \{v_1, \dots, v_n\}$ into a new node v_ϕ. The error

$$\max_{v_i \in V} \{(v_\phi^{value} + v_\phi^{rub}) - (v_i^{value} + v_i^{rub})\}, \tag{6.1}$$

evaluates the maximum increase in length of any path going through v_ϕ. The value v^{value} of a node is the length of the longest path to that node and the rough upper bound v^{rub} is a bound on the length of any path originating from v to the terminal node. Including v^{rub} in Equation 6.1 increases the accuracy of the error measure by also roughly considering the future. However, Equation 6.1 is a measure for evaluating one merge. In order to take its effect over a layer, we again take the maximum error of all the merges in that layer. For the whole diagram, we average over every layer.

[4] It is also possible to cluster based on problem features in order to get even better measures of node similarity but the results in this chapter were not changed significantly as the bigger determinant of the merge error was the merge operator.

Our experiments (Figure 6.2) show that instances where the *merge error* is high are better served by *minLB* as opposed to *cluster*. The reasoning behind this is as follows: while merging the most similar nodes via clustering reduces the amount of dissimilarity, it also gives room for even high quality nodes to be merged and in instances where the merge operator itself induces a large error, information is lost from good nodes. This kind of information loss from good nodes is not an issue in *minLB* as only the worst nodes in a layer are merged.

Note that the merge error derived from Equation 6.1 have the weakness of not being normalised and such can vary largely from one problem to another as seen in Figure 6.2. However, comparisons can still be made as in any case, a higher error is always worse.

Example 4. Let us re-consider the running example of a KNAPSACK problem instance introduced in Example 1 with capacity 10 and 3 items of weights $6,4,2$ respectively and profits $3,5,7$ respectively. Recall the state representation is the tuple $(avail, pro)$ such that *avail* is the available capacity and *pro* is the highest profit achievable along any path to that state. Figure 6.1 shows two possible merge operators for the third layer of this diagram.

Say we want to merge the 2 highlighted states in Figure 6.1a with tuples $\{(6,5),(10,0)\}$, a possible merge (Figure 6.1b) operator assigns to the new merged node v_ϕ a value of $\{10,5\}$ assuming the maximum value can be attained by all paths leading to v_ϕ. However, any value larger than 5 for the *pro* value would also serve as a valid merge operator without violating any of the rules of merge validity (Hooker 2017). For instance, another operator (Figure 6.1c), assigns to the new merged node v_ϕ a value of $\{10,7\}$. Such an operator, while correct, introduces a larger error as can be seen in the eventual bounds achieved by both diagrams. While this error is completely unnecessary in the KNAPSACK problem, other problems may have variable interactions that do not accommodate tighter relaxations.

Dominance

Dominance rules either improve the performance of a DD or do nothing at all. They have no means to worsen performance as long as they are correct, as in the worst

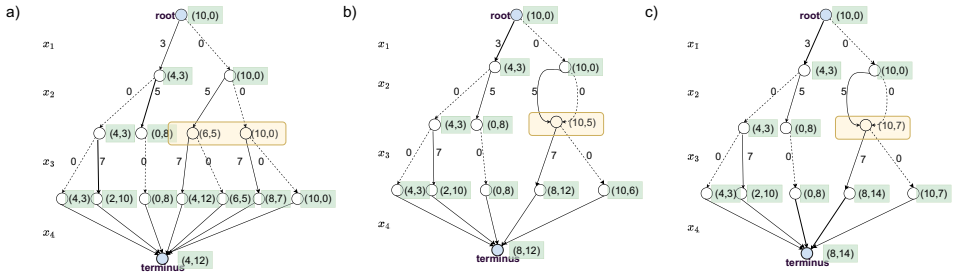

Figure 6.1.: Example compilation showing the influence of the merge operator on eventual bound derived.

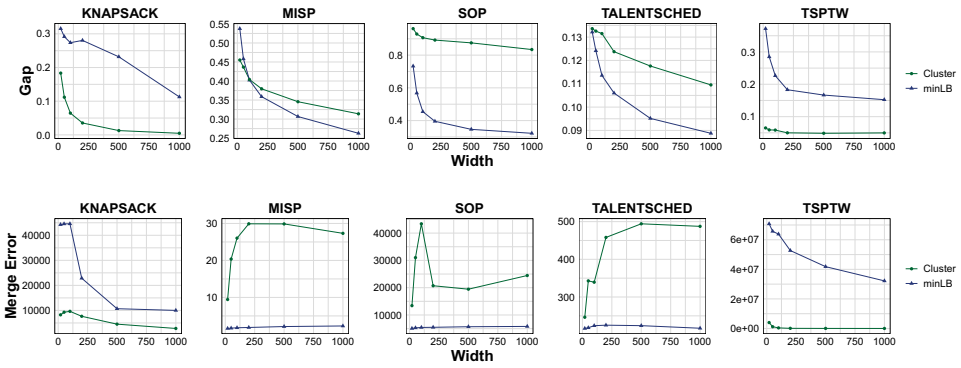

Figure 6.2.: *minLB* vs *cluster* comparisons based on relationship between the quality of the bound (represented by the optimality gap) and the merge error. The relative error between both methods matches the relative gap.

case, no dominance relations are established and no pruning is done. The direct effect of dominance can be seen in Figures 6.4 and 6.6 for the ALP, KNAPSACK, LCS, SOP and, TSPTW problems where dominance rules improve the performance of all but the LCS problem where it has no effect.

Variable Ordering

A good variable ordering improves the search as the final solution is more dependent on the values of earlier variables than the later ones. This is either because (i) assignments to the earlier variables shrink the possibilities for later variables or (ii) assignments to the earlier variables provide strong bounds for the remaining sub-problems. In general, a good variable ordering heuristic ensures that the diagram, as much as possible, branches on the backdoor variables first where backdoor variables are smaller subsets of decision variables that capture the overall problem structure (R. Williams, Gomes, and Selman 2003).

Thus, it is possible for a variable ordering heuristic to result in worsened performance and it is also possible that there exists no useful variable ordering heuristic for a particular class of problems. This is informally the case with scheduling problems where the variable ordering is prescribed by the sequential nature of the problems.

In Figure 6.3, we revisit the sample problem in Example 1 and show the effect of variable ordering on the diagram. While both orderings in Figure 6.3a and Figure 6.3b have the same eventual objective value of 12 – represented by the state with tuple (4,12) – this state is reached in layer 2 in Figure 6.3b, while it is only reached in layer 3 in Figure 6.3a. This means that the ordering in Figure 6.3b gives us tighter bounds earlier in the diagram.

Figure 6.5 shows the impact of variable ordering on the KNAPSACK problem – where variables are ordered in descending order of the profit of the item – and the

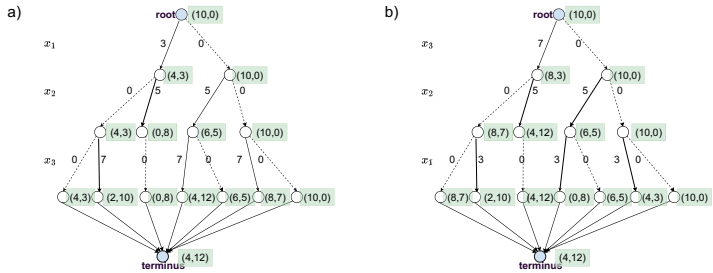

Figure 6.3.: Example compilation showing the effect of different variable orderings.

MISP problem – where we use a dynamic ordering to select the largest weighted vertex of the vertices still available to be selected in any state. In both problems, performance is improved by the addition of an informed variable order.

6.4.2. Impact of Technique Combinations

In this section, we empirically investigate how combinations of the studied techniques affect each other. We repeat experiments on the problem classes introduced in Section 6.3.

Merge Heuristics and Dominance

Compared to *minLB* where there is at most one merged node per layer, *cluster* can lead to a lot more merged nodes in a layer and this results in the unwanted effect of minimizing the effect of dominance when used together. The reason here is that dominance rules in DDs are often only defined between exact nodes (Coppé, Gillard, and Schaus 2024b) and even in the case when merged nodes can participate in dominance comparisons, they often limited to only being dominated. Thus, when we have a case where many nodes in a layer end up being merged, opportunities to assert dominance relations are also reduced.

Figure 6.4 shows this effect on five problem classes ALP, KNAPSACK, LCS, SOP and, TSPTW – all problem classes for which dominance rules already exist in the literature (Coppé, Gillard, and Schaus 2024b). In general, we see that dominance either does nothing or improves the bounds regardless of the merge heuristic used. When looking at the effects of adding dominance to *cluster* compared to *minLB*, the performance of the diagram is more or less reduced to that of clustering alone – as in LCS, SOP, TSPTW and KNAPSACK – except in ALP where although dominance improves the results of clustering alone, the performance is still much poorer than combining it with *minLB*.

An additional note on Figure 6.4 is on the difference in performance when compared to Figure 6.2 on problem classes that appear in both evaluations, e.g., for the KNAPSACK problem we see that optimality gaps in Figure 6.2 are in general lower than those in 6.4. The difference comes from the inclusion of the rough upper bound in the experiments in Figure 6.2 which leads to better node selection for merging.

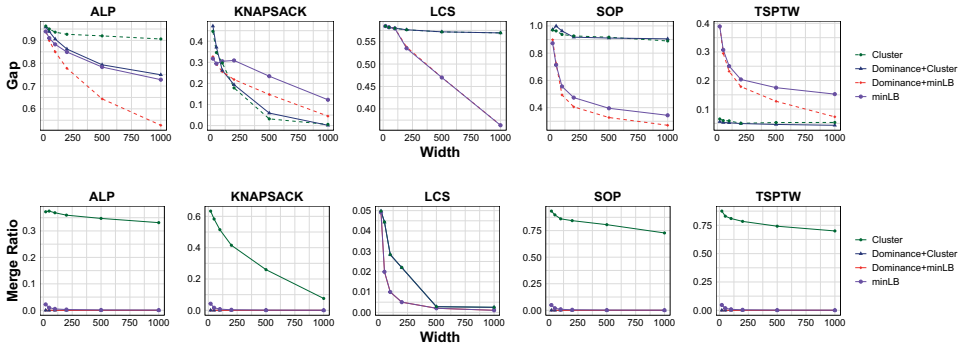

Figure 6.4.: The impact of merge heuristics on dominance.

Merge Heuristics and Variable Ordering

In general, variable ordering is complementary to any merge heuristic in use. A good variable ordering improves the results of the merge heuristic and a bad one worsens them. A good variable ordering improves merge results by generally reducing opportunities to introduce merge errors. This happens in one of two ways. In the first case, the variable order ensures that we branch first on variables that strongly constrain the rest of the problem thus, the number of nodes in subsequent layers is reduced and fewer nodes are merged resulting in reduced chance of error. In the second case, the variable order ensures we branch on variables whose value assignments have a stronger impact on the resulting objective function value; thus, the values of subsequent nodes are more representative of their contributions to the objective leading to better input values for merge heuristics. Figure 6.5 shows this effect on two problems with known variable ordering heuristics, KNAPSACK and MISP where the introduction of a variable ordering heuristic improves the results regardless of the merge heuristic used.

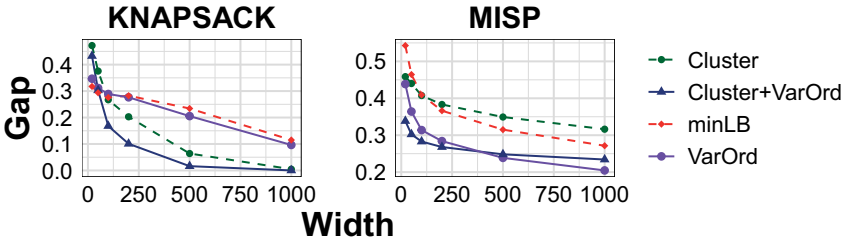

Figure 6.5.: The impact of merge heuristics on variable ordering.

Variable Ordering and Dominance

Our results show that variable ordering and dominance are also complementary. This is as expected due to conclusions already made on the impact of variable ordering in Section 6.4.2. For the two problems shown in Figure 6.6, we keep the merge heuristic fixed at *minLB* and in both cases, variable ordering improves the performance and layering dominance either further improves the performance compared to dominance alone or does nothing.

6.5. Extending Techniques to Incremental Refinement

Of the two main methods of compiling DDs, top-down construction is the most popular especially as it is the main method in DD-based branch-and-bound algorithms. Therefore, the techniques discussed in the previous section and as introduced in the referenced literature, have all been geared to the top-down compilation paradigm. However, incremental refinement has also proven effective in other application scenarios such as serving as constraint stores (Andersen *et al.* 2007), being integrated as propagators in constraint programming solvers (Cire and Van Hoeve 2013), and aiding column formulations of optimisation problems as in *column elimination* (Karahalios and van Hoeve 2023).

Incremental refinement (also called compilation by separation) is a means of compiling DDs that begins from a trivial diagram and systematically expands the diagram by creating more nodes. Refinement is typically used to build relaxed diagrams where the aim is to find good bounds on the objective function. Each step in the refinement procedure thus aims to tighten the relaxation. There are two major steps in refinement – (i) splitting nodes to expand the diagram and (ii) filtering infeasible edges to increase diagram accuracy where filtering is performed based on the problem constraints while splitting is based on heuristics.

In this section, we discuss the extension of the techniques discussed above to incremental refinement. Incremental refinement works with the same state transition model as top-down construction. As such, the only additional element is defining *constraint filters* and *split heuristics* where split heuristics can be thought of as performing the inverse operations of merge heuristics. In order to perform our evaluation, we implement these extensions to four problem classes; MISP, KNAPSACK,

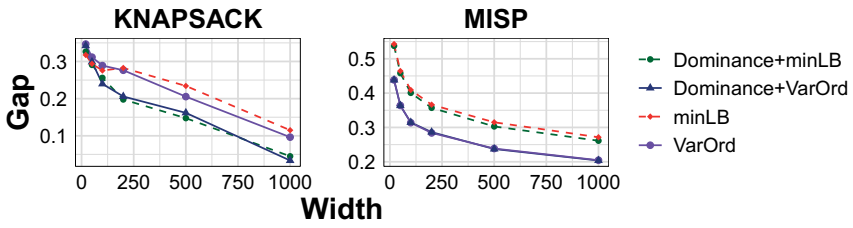

Figure 6.6.: The impact of variable ordering on dominance.

TSPTW, and SOP, providing a mix of binary and multi-valued DDs.

6.5.1. Variable Ordering

Variable ordering heuristics can either be static – deciding on a fixed order of variables decided before diagram expansion begins, or dynamic – selecting the next variable based on the state of the preceding layer. For example, ordering the items in the knapsack by their profit such that the most profitable items are decided upon earlier in the DD formulation of the KNAPSACK problem is a static ordering heuristic. On the other hand, ordering the nodes in a graph by their appearance in the states of a layer is a dynamic ordering heuristic for the MISP problem. Incremental refinement completely loses the gains of dynamic variable ordering heuristics because we start with a small dummy diagram and upon expansion do not reorder the nodes even though increasing the width gives us more information, which usually results in better dynamic orderings. In order to update the variable ordering after splitting, we would also have to reorder the diagram. Variable reordering is not impossible and has been considered (Awad, Hawash, and Abdalhaq 2022; Jiang *et al.* 2017) for BDDs where swapping adjacent layers has been shown to be an efficient and safe way to reorder a BDD. However, this has, to the best of our knowledge, not been done in the context of DDs for optimisation, likely because there are additional complexities. For instance, such reordering may fail for some optimisation problems where there are strict constraints that require us to propagate variable swapping decisions all through the diagram.

6.5.2. Split Heuristics

As stated, splitting a node can be thought of as the inverse of merging a node. As such almost all the merge heuristics in the literature can be modified to be split heuristics. For the merge heuristics that take into account all the nodes in a layer at once, the splitting equivalent will involve collecting all incoming edges in a layer and applying the merge heuristic on all the nodes resulting from transitioning along each edge. However, the more common way to perform node splitting is to select one node u and split it into nodes u' and u'' where the edge corresponding to the longest path goes to u' and all the other edges go to u'' (Cire and Van Hoeve 2013; Römer, Cire, and Rousseau 2018). We refer to this in the rest of the chapter as *pairSplit*. An additional step is required after splitting, which is to redirect the outgoing edges of

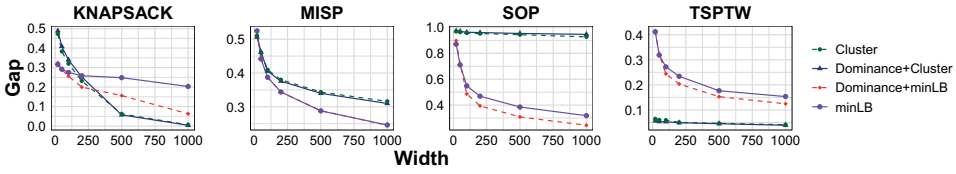

Figure 6.7.: Performance of split heuristics and dominance in incremental refinement.

each node since the later layers of the diagram already exist as opposed to top-down construction. We implement *pairSplit* and the split equivalent of both *minLB* and *cluster* as described above. *pairSplit* performs worse than implementing *minLB* as a split heuristic and this is confirmed by results in Figure 6.8. This is likely as a result of *pairSplit* having a more limited scope by splitting one node in every iteration as opposed to *minLB* re-ordering all the incoming edges to a layer. The relative performance of *minLB* and *cluster* as split heuristics maintain the same relationship as in top-down compilation.

6.5.3. Dominance

Dominance in incremental refinement works the same way as in top-down construction. As we refine the diagram, each newly created node can be checked against existing nodes for dominance and pruned if it is dominated. Pruning involves deleting all the incoming and outgoing edges of a node and can lead to nodes further down the diagram also being pruned in the next refinement step. Figure 6.7 (when compared with Figure 6.4) shows that the performance of dominance and the split heuristics is comparable to the performance of the corresponding merge heuristics in top-down compilation.

6.6. Discussion

Our results show that problem characteristics play a significant role on the performance of the techniques studied for DD compilation. However, some overall themes emerge leading us to make the following claims.

Claim 1. Dominance is incompatible with heuristics that create many merged nodes.

This claim comes from our observations in Section 6.4.2; the key issue being that dominance in DDs has so far been applied in such a way that only exact nodes can dominate others. As such, when many merged nodes are created, there is less opportunity to assert dominance. However, there is still some nuance here as there is no precise predictor of when a heuristic will create too many merged nodes for dominance to be effective.

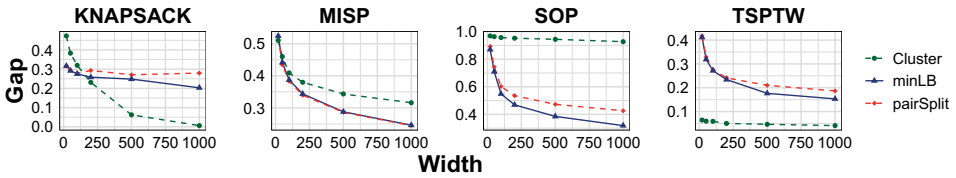

Figure 6.8.: Performance of merge heuristics repurposed as split heuristics in incremental refinement.

Claim 2. Variable ordering is complementary to any merge heuristic. A good ordering heuristics improves all results and a bad one worsens all results.

This follows directly from the discussion in Section 6.4.2 where the mechanics of variable ordering show that its performance is consistent across different merge heuristics and that it combines well with dominance.

Claim 3. The mode of compilation has no effect on the impact of dominance rules or merge (split) heuristics used.

In Section 6.5, we have extended the techniques studied in this chapter to also work on an incremental refinement solver and experiments showed that the performance of a technique in DD compilation is based on the problem characteristics and not the kind of compilation in use. Thus, the choice of compilation scheme should be application-based. For example, incremental refinement makes sense for combination with constraint programming solvers and propagation and shrinking of domains is analogous to filtering the DD.

6.7. Conclusions and Future Work

In this chapter, we have studied three techniques for DD compilation namely merge heuristics, dominance, and variable ordering. We have provided empirical evidence – on a set of 7 classic optimisation problems – of the effects of their combinations. Our results indicate that the performance of a merge heuristic for a particular problem is dependent on the strength of the relaxation provided by its merge operator – quantified by the merge error – and that the effect of dominance rules is significantly impacted by the merge heuristic in use.

While this chapter makes a step towards a better understanding of DD compilation, it still leaves some stones unturned. Specifically, the following aspects still need to be addressed:

- Variable reordering in the context of optimisation problems is also an interesting challenge that may yield benefits for compiling DDs by incremental refinement especially when they are embedded in other iterative solving methodologies such as in column elimination.
- The comparison done in this chapter can also be enriched by considering more problem classes and heuristics, considering the effect of using problem features in the heuristic decisions and possibly providing an algorithm selection method for DD compilation.

7

Learning to Build Decision Diagrams

The step most likely to produce solutions in decision diagram based branch and bound is the restriction step. The heuristic used to compute the restriction is thus very important for the quality of solutions provided. Typically, the DD literature uses restrictions that greedily discard nodes at every layer but this could be a good place to insert domain knowledge by using user defined heuristics to compute the diagram. We can go a step further and use a learned heuristic. In this chapter, we do just that by combining a reinforcement learning agent with the decision diagram so that anytime a restriction is built it is built by a trained RL agent. In this way, the agent performs a form of node selection. *However, as opposed to specifically training node selection, we train the agent to solve the problem as in the end, the agent effectively builds a solution everytime it is called. We demonstrate this scheme on the job shop scheduling problem and empirical results show that our learned DD shows improvements over a standard DD approach to job shop scheduling.*

Parts of this chapter have appeared at the 6th Data Science Meets Optimisation Workshop (DSO) (2024) (Eigbe, Schmidl, *et al.* 2024).

7.1. Introduction

Combinatorial optimisation problems have received a lot of attention over the years from different research domains including machine learning. There are different potential means to solve optimisation problems via machine learning. We could learn appropriate hyper-parameters and settings for an algorithm (Reijnen, Y. Zhang, Bukhsh, *et al.* 2022; Reijnen, Y. Zhang, Lau, *et al.* 2022), learn to solve the problem end-to-end (Song *et al.* 2022; C. Zhang *et al.* 2020), or learn a particular step in the solution process of another algorithm (Chalumeau *et al.* 2021).

We choose to go the route of integrating machine learning in the solution process of another solver, namely a decision diagram. This kind of amalgamation of methods gives us better chances to gain from both methodologies directly. We do not learn a particular solving step as this still keeps the learning and solving portions of the algorithm separate. We instead, train an RL agent to solve the whole problem. With both the agent and the solver having the capability to solve the problem, they are able to correct for each other's shortcomings when combined.

In summary, our work closes multiple gaps, namely: (i) while Song *et al.* (2022) and C. Zhang *et al.* (2020) already use reinforcement learning to solve job shop scheduling problems, we embed our agent in a solver such that we can still improve the solution provided by an agent and possibly be optimal, (ii) while (Chalumeau *et al.* 2021) already embed a reinforcement learning agent in an exact CP solver we give the agent much more freedom via our decision diagram structure so that we can get competitive solutions and finally, (iii) although Cappart, Goutierre, *et al.* (2019) also use reinforcement learning in a decision diagram, we focus on finding solutions as opposed to finding bounds.

The rest of the chapter is organised as follows: Section 7.2 introduces preliminary background information, Section 7.3 discusses related work, and Section 7.4 provides the details of our solution approach. We perform empirical evaluations in Section 7.5 and Section 7.6 concludes the chapter. Our code is available online[1].

7.2. Preliminaries

This section introduced some preliminary concepts, our terminology and notations on job-shop scheduling problems, decision diagrams, and reinforcement learning.

7.2.1. Job Shop Scheduling Problems

A job shop scheduling problem (JSSP) consists of a set of jobs $J = \{J_1, \dots, J_n\}$ to be scheduled on a set of machines $M = \{\mu_1, \dots, \mu_m\}$. Every job $J_i \in J$ has a set of operations $O = \{o_{i0}, \dots, o_{ik}\}$ where operation o_{ij} is the jth operation of the ith job and has processing time $P(o_{ij})$. Each operation is assigned to a machine.

Job shop scheduling is a well-known combinatorial optimisation problem (Applegate and Cook 1991) that is widely used in industry. Different versions of the JSSP exist depending on the particular industrial setup. It is known that most versions of the JSSP are NP-hard (Lenstra and Kan 1979). The solution to a JSSP is a schedule Ω,

[1] https://github.com/Eghonghonaye/NDDLSTM

i.e., a sequence of operations. We focus on the objective of makespan minimisation because minimising the makespan correlates with many other objectives, such a minimising tardiness and flow time. As such, we believe the design choices made in our method can easily be tweaked to fit a different objective.

7.2.2. Decision Diagram Representation

A decision diagram (DD) as introduced in Section 4.1 is a directed acyclic graph $G = (V, E)$ with a set of nodes V and edges E. For ease of the discussions in this chapter, this section, we additionally define a state s_v associated with every node $v \in V$. In the realm of scheduling, a state is a partial schedule, while an edge e is a transition based on a scheduling decision.

A decision diagram contains two special states, namely the root state s_r and the terminal state s_t. At the root state s_r, no decisions have been made, and at the terminal state s_t, all decisions have been made, i.e., all operations have been scheduled. Thus, every path from $s_r \rightarrow s_t$ represents a complete schedule.

Decision diagrams can be modelled fully recursively and as a Markov Decision Process (MDP) (Bergman, Cire, Van Hoeve, *et al.* 2016). As we combine our solver with a reinforcement learning (RL) agent, we design our solution such that both the DD and the RL setup use the same MDP formulation of the problem. This formulation will be discussed later in Section 7.4.2.

7.2.3. Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that sets up the learning task as a sequential decision-making process. The basic principle of RL is that an agent interacts with an environment by taking actions within that environment. The environment is Markovian such that every state-action pair provides sufficient information to transition to a new state. After every action, the agent gets feedback – in the form of rewards or penalties – and based on this feedback, the agent learns to take better actions within the environment (Wiering and Van Otterlo 2012).

Typically, the RL agent is controlled by some policy π, and the goal during training is to learn an optimal policy. This can be done by learning a policy directly or learning value functions in a state such that the optimal policy can be later retrieved from the learned value functions.

RL is attractive for scheduling problems because many of them have to do with deciding on sequences, and this ties in neatly with a sequential decision-making agent. We point the reader to (François-Lavet *et al.* 2018) for a more comprehensive understanding of RL.

7.3. Related Work

The concept of embedding RL agents in algorithms for combinatorial optimisation problems has been studied in recent literature. For instance, Reijnen, Y. Zhang, Bukhsh, *et al.* (2022) use RL to control parameter settings for evolutionary algorithms, while Reijnen, Y. Zhang, Lau, *et al.* (2022) use RL to select operators for adaptive

large neighbourhood search. However, in both of these methods, the agent is neither directly trained to solve the problem nor fully embedded in the solution, but instead learns a heuristic used within the algorithm.

A challenge in using machine learning methods to solve combinatorial optimisation problems – either directly or as a part of a larger solver – is that we often require our models to be size-agnostic and permutation-invariant. Although there are methods to mitigate the worsening effect of job permutations on the makespan in the context of RL (Schmidl, Simão, and Jansen 2024), they are mostly just applicable to policies represented as a feed-forward neural network. For this reason, many methods have turned towards *graph neural networks* (GNNs) as they are size- and permutation-invariant by design. These features enable a GNN to create fixed-size state embeddings out of graph structures that act as a state representation for an RL agent (Park *et al.* 2021). For more details on advances in this line, the recent survey by Cappart, Chételat, *et al.* (2023) provides an overview of how GNNs have been employed in solving combinatorial optimisation problems. Although GNNs can capture the relationships and interactions between nodes in the graph, they may lack the ability to fully capture temporal dynamics and sequences in a job shop scheduling problem. Given that a job shop has to adhere to precedence constraints that dictate operation sequences for jobs, autoregressive models may be a better fit than GNNs. Autoregressive models, such as recurrent LSTMs (Hochreiter and Schmidhuber 1997), have already been applied successfully to the JSSP (Iklassov *et al.* 2023) and can encode the operations of each job. A set2set (Vinyals, S. Bengio, and Kudlur 2015) network then uses these encodings to create a new equivariant representation. This leads to similar features of a GNN, mentioned before, with the ability to capture temporal dynamics.

With RL, especially where policy gradient methods are used, it is common to greedily sample from the trained policy by picking the highest probability action in every state. However, we know that for combinatorial optimisation problems, this is not always the best strategy (Bello *et al.* 2016; Kool, Van Hoof, and Welling 2018). Bello *et al.* (2016) show that allowing the trained agent to perform some kind of search in its solution space yields better performance and Iklassov *et al.* (2023) show that sampling, i.e., repeating the inference process by directly sampling from the policy distribution and then picking the best sample, performs well for job-shop scheduling. However, sampling is not necessarily aware of the position of a solution in the solution space and does not directly try to improve samples. Embedding the agent in a solver like we do, allows the agent to be guided to sample solutions from more promising parts of the solution space.

Another interesting related work is (Cappart, Goutierre, *et al.* 2019), where, similar to our work, RL is combined with decision diagrams. Cappart, Goutierre, *et al.* (2019) train an RL agent to decide the variable ordering in a decision diagram. The connection between decision diagrams, dynamic programming, and RL – each of these three solution methods can be reasoned about as MDPs – is exploited in their algorithm. Apart from the difference in the learning task as discussed above, our work focuses on finding solutions while (Cappart, Goutierre, *et al.* 2019) focuses on finding bounds.

7.4. Proposed Solution Method

Our method works by extending DD-BnB such that building a restricted diagram is treated as a sequential decision-making process. Every time a branching decision is made, a trained sequential decision-making agent is triggered to build a single width restricted diagram – leading to a new solution while relaxations are built as usual via merging nodes to create a limited width diagram. We make branching decisions based on several known rules in the literature, such as best- and depth-first expansion.

7.4.1. Decision Diagram Construction

A decision diagram $G = (V, E)$ consists of nodes V and edges E. Nodes V represent states of the job shop and edges E represent the scheduling decisions that transition the shop from one state to another. In this chapter, a scheduling decision is simply an operation to be scheduled.

A state s_v is directly represented as a feature vector with three entries (A, Φ, EST) where $A = \{A_m | \mu_m \in M\}$ represents the earliest availability of all machines, $\Phi = \{o_{ij} | J_i \in J\}$ represents the next ready operation for each job, and $EST = \{J_i^{\text{est}} | J_i \in J\}$ is the earliest start time for any unscheduled operation of a job. We define a function $H(\Omega)$ that returns the (partial) makespan of a schedule.

Transition. We transition from one state to the next by choosing an operation to be scheduled next. Every time, we transition from state s to s' by scheduling an operation o_{ab} on machine μ_k, we create a new state s' such that $A'_k = \max(A_k, J_a^{\text{est}}) + P(o_a)$, $o'_{ab} = o_{a(b+1)}$, and $J_a^{'\text{est}} = A'_k$. Whenever there are no more unscheduled operations in a state, we transition to the terminal state.

Algorithm 8 Decision Diagram Construction

```
1:  function DDC(Job Shop problem S, RL Policy π )        ▷ returns Schedule Ω
2:      Create root and terminal nodes r, t
3:      Create empty schedule Ω
4:      L ← {r}                          ▷ Set of leaf nodes, same as fringe in DD-BnB
5:      while not stopping criterion do
6:          Select v from L                                  ▷ Perform node selection
7:          DD ← Create relaxed diagram with root v
8:          while v not terminal ∨ infeasible ∨ dominated do
9:              a ← π(s_v)                          ▷ Select action based on RL Policy
10:             Create new node v' from (v, a)
11:             Evaluate v'      ▷ Update start times, check dominance and feasibility
12:             v ← v'
13:         if v is terminal ∧ H(Path(r, v)) < H(Ω) then
14:             Ω ← Path(r, v)                  ▷ Update Ω with path from root to v
15:         if DD* < H(Ω) then
16:             Let S be an exact cutset of DD
17:             L ← L ∪ S
        return Ω
```

Algorithm 8 describes the steps of constructing the DD. In Lines 1-4 we initialise an empty schedule and an empty DD. In Line 5, we select a node to expand according to a node selection heuristic. We create a relaxed diagram in Line 7 to give us a bound on the solution reachable from this node and then we hand over to the agent to continually choose actions in Lines 8-12. If the agent ends up in a state that is a solution, infeasible[2] or dominated, we go back to the DD in Line 6 to select another point to continue expansion from. We update the set of leaf nodes by adding the exact cutset from the relaxed diagram only if the relaxation has a better bound than the best known schedule in Line 15-17. We repeat this process until a stopping criterion is met which is primarily that there are no nodes left to expand although we can specify other criteria such as runtime or memory limitations.

7.4.2. Reinforcement Learning Agent

We train an RL agent to solve the problem independently, i.e., without the support of an external solver. The intuition is that an agent can learn a good sequencing policy, which produces solutions that can be quickly improved using the DD. Setting up such an agent requires a few definitions, which are stated below.

State. States carry the same representation and meaning between the DD and the RL agent, e.g., s_0, is both the root state of the DD and the initial state of the RL agent. This makes for a smooth interaction between the agent and solver.

Action. We define an action as an operation selection. This also carries the same meaning and representation as an edge in the DD. Therefore, the action space is discrete and contains indices of operations as integer values.

Reward. We keep the reward simple. For every transition from state s to s', an agent gets a reward that is the difference in makespans of the partial solutions where a makespan of a partial solution is the maximum machine availability i.e., $r = \max(A) - \max(A')$.

Policy. We use the same policy setup as (Iklassov *et al.* 2023), which takes as input the instance description x and the *internal state* during the problem resolution s_τ and outputs a policy $\pi(x, s_\tau)$, which is a distribution over the next pending operations at the current decision step τ. The underlying policy architecture is a combination of an LSTM (Hochreiter and Schmidhuber 1997) that encodes the operations on each job recurrently, a set2set (Vinyals, S. Bengio, and Kudlur 2015) network to achieve equivariance for job combinations, and actor-critic networks to finally compute distribution over actions. The LSTM can effectively capture temporal sequences between operations, while the set2set network can prevent makespan degradation that may be caused by job permutations (Schmidl, Simão, and Jansen 2024). The policy is

[2]While the actions are masked such that only feasible options are explored, some problems have constraints whose violations can only be assessed after a decision has been made, e.g., problems with relative deadlines.

trained using the Reinforce algorithm (R. J. Williams 1992). We make use of the policy implementation provided by (Iklassov *et al.* 2023), which is also available online[3].

Action masking In our approach, we use action masking during training and testing. In every state, infeasible actions are masked out according to the constraints of the problem. After training, when the agent is used within the DD, we further mask out actions that have already been taken in the past to prevent duplication of paths in the diagram.

7.5. Results

Benchmark. We evaluate our method on the popular Taillard instances introduced in (Taillard 1993). The benchmark consists of problem instances with the number of machines in the set {15,20}, the number of jobs in the set {15,20,30,50,100}, and the processing times uniformly distributed in the interval [1,99]. The benchmark contains 80 problem instances and we train our RL agent on 10 randomly selected instances and learn only one policy across all instance sizes. We use the full set of 80 instances in testing.

Baselines. We compare our method (referred to as DD-RL) with two classes of baselines. The first class is other RL methods to solve this problem. We compare our work with greedily sampling our trained agent (referred to as RL), the results of (C. Zhang *et al.* 2020) (referred to as Zhang *et al.*) and the results of (Iklassov *et al.* 2023) (referred to as Iklassov *et al.*). The second class of baselines is the decision diagram alone using the same scheme as in Algorithm 8 but with the RL agent replaced by a simple depth first search. We refer to this as DD in the results.

We compare all methods to the best known solutions in the literature (referred to as UB) and compute optimality gaps as the distance in percentage from these solutions.

Note that we only compare with the base model results in (Iklassov *et al.* 2023) and not the best results presented in the paper. The best results were achieved via curriculum learning (Y. Bengio *et al.* 2009) and always perform better across all instances. However we believe comparing with these results would not be fair as it takes significantly more training resources to achieve the curriculum learning results. In our future work, we plan to deploy more strategies that allow us to use a curriculum learning trained agent along with the DD on smaller resources.

Configurations. All experiments were run on a machine with an AMD Ryzen 9 7950X3D (5.7GHz), 64GB of RAM, and a single Nvidia Geforce RTX 4090. The operating system is Ubuntu 22.04 LTS. We train our model for 20 000 iterations using a batch size of 8 and a constant learning rate of 10^{-4}. All DD solutions are given a stopping criterion of 5 minutes run time.

[3] https://github.com/Optimization-and-Machine-Learning-Lab/Job-Shop/tree/main_nips

Discussion. Table 7.1 summarises our results, showing the achieved makespans and their optimality gaps with the best performing solution highlighted in bold. Overall, the DD-RL approach outperforms its competitors for all benchmark instances.

For both setups with 15 and 20 machines, DD-RL yields better makespans than other methods. We observe up to 5% improvement in optimality gap for DD-RL compared to Iklassov *et al.* For setups with 30 and 50 jobs, the DD-RL approach is still performing better than the rest, but we can observe that the approach by Iklassov *et al.* dips in performance and is worse than the pure RL approach. This could be attributed to some forgetting as we train our base model for 20 000 iterations while Iklassov *et al.* trains the same model for 45 000 iterations.

M/Cs	Jobs		RL	DD	DD-RL	Zhang *et al.*	Iklassov *et al.*	UB
15	15	Obj.	1538.30	2016.4	**1404.60**	1547.4	1413.0	1228.90
		Gap	*25.24%*	*64.07%*	***14.35%***	*25.96%*	*14.98%*	
	20	Obj.	1773.50	2425.7	**1635.60**	1774.7	1692.1	1364.80
		Gap	*29.38%*	*77.78%*	***19.86%***	*30.03%*	*23.97%*	
	30	Obj.	2255.90	3311.1	**2171.70**	2378.8	2275.3	1790.20
		Gap	*26.16%*	*85.36%*	***21.43%***	*33.0%*	*27.19%*	
	50	Obj.	3292.30	4863.0	**3195.10**	3393.8	3346.7	2773.80
		Gap	*18.76%*	*75.38%*	***15.22%***	*22.38%*	*20.69%*	
20	20	Obj.	2068.30	3008.2	**1932.40**	2128.1	1958.4	1617.50
		Gap	*27.85%*	*86.02%*	***19.49%***	*31.61%*	*21.11%*	
	30	Obj.	2569.20	3873.7	**2447.90**	2603.9	2540.2	1948.50
		Gap	*31.89%*	*98.83%*	***25.66%***	*33.62%*	*30.4%*	
	50	Obj.	3439.30	5654.8	**3363.10**	3593.9	3518.0	2843.90
		Gap	*20.97%*	*99.02%*	***18.27%***	*26.51%*	*23.73%*	
	100	Obj.	5949.40	9582.0	**5900.70**	6097.6	6145.5	5365.70
		Gap	*10.87%*	*78.61%*	***9.96%***	*13.61%*	*14.52%*	

Table 7.1.: Results on Taillard Instances.

7.6. Conclusions

In this chapter, we explore the combination of decision diagrams and reinforcement learning to solve job-shop scheduling problems. We make use of an existing RL model in the literature and show via an empirical study that the combination of DD and RL into one solver performs better than either solution method on its own.

However, we deploy a relatively simple training for our agent and results from more robust agents in the literature still outperform those presented here. We hypothesize that the performance of RL agents trained with extensive resources and more complex schemes can be replicated by simpler training and better inference such as the integration with a solver presented in this chapter. Should this be the case, RL could become more usable in industrial settings where extensive training resources may not be available. It will be fruitful for future research to explore and possibly validate this hypothesis.

8

Decision Diagrams for Chance Constrained Optimisation

Rational decision making under uncertainty is notoriously difficult for humans. Effectively solving mathematical formulations of decision problems under uncertainty is also computationally challenging. This chapter considers chance constrained optimisation problems with uncertainty in the objective function. For this important class of problems, we introduce a novel solution methodology based on decision diagrams, for cases with both continuous and discrete random variables. Second, we provide theoretical properties for bounds when the uncertain variables are normally distributed. Third, we show that our ideas extend to other models of uncertainty, particularly uncertainty theory.

8.1. Introduction

The dynamic nature of reality ensures that many real-world optimisation problems contain uncertainty. While the more common approach is to solve deterministic versions of these problems, there is also a rich literature on optimisation under uncertainty. Researchers have considered robust optimisation (Ben-Tal, Nemirovski, and El Ghaoui 2009) where the goal is to optimise for the worst possible outcome, and stochastic programming (Birge and Louveaux 2011) where the uncertainty is captured based on probabilities of occurrences. Solution approaches to stochastic programming problems range from optimising the expected value of the objective function, to distributionally robust optimisation (F. Lin, Fang, and Z. Gao 2022) where the underlying probability distributions of uncertain variables are themselves uncertain. In this chapter, we focus on chance constrained optimisation problems (Charnes and Cooper 1959; Miller and Wagner 1965) where the challenge is to find the best solution such that the probability that the resulting decisions comply with the specified constraints is higher than a given threshold.

Decision diagrams (DDs) as introduced in Chapter 4 are graphical structures capable of compactly representing the solution space of combinatorial optimisation problems. They are directed acyclic graphs (DAGs) and are constructed based on recursive formulations of problems such that each node represents a state and each edge represents a transition from state to state. While proven to be very good tools for solving discrete optimisation problems (Bergman, Cire, Van Hoeve, *et al.* 2016; Cire and Van Hoeve 2013; Coppé, Gillard, and Schaus 2022; González *et al.* 2022; O'Neil and Hoffman 2019; Tjandraatmadja and van Hoeve 2021), there is limited work on using DDs to solve problems with uncertainty.

The bulk of DD-based uncertainty optimisation involves integrating DDs into solution steps of other solvers such as propagating chance constraints in constraint programming (Perez, Malalel, *et al.* 2023) or generating cuts for stochastic programming with recourse (Lozano and C. Smith 2022; MacNeil and Bodur 2024). Hooker (2022) also introduces stochastic decision diagrams with an aim to develop a policy to select an optimal action to take in any state given that the resulting transition from that action is uncertain. In this chapter, we consider an alternative case where the DD directly produces bounds and/or solutions.

While any problem parameter can be uncertain, there is a common subclass of problems where uncertainty only appears in the objective function (Aissi, Bazgan, and Vanderpooten 2009; Vu *et al.* 2025). In this chapter, we consider this class of optimisation problems with uncertainty limited to the objective function, i.e., the transitions are certain but the costs of those transitions are not. We introduce the first DD-based solutions for chance constrained optimisation problems considering cases with continuous and discrete random variables. We further show that when variables are normally distributed, we can find bounds on the objective by selecting edge weights in the DD based on the cumulative distribution functions (CDFs) without affecting the time complexity.

Contributions. The key contributions of this chapter are:

- A DD formulation for problems with discrete random variables in the objective, the first such formulation in the literature.
- Demonstration that, in the case of normally distributed continuous variables:
  - standard DD formulations can be used to provide bounds by assigning edge weights based on the CDF, and
  - this setup only provides bounds on the objective even when the DD is exact.
- Investigation of alternatives to probability theory as a means to model uncertainty by using uncertainty theory instead of probabilities. With independent uncertain variables, assigning edge weights using the uncertainty distribution gives solutions and bounds in the traditional DD sense from exact and relaxed diagrams respectively.

8.2. Related Work

The seminal work by Hooker (2022) introduced the concept of stochastic decision diagrams where state transitions are uncertain. This is akin to stochastic dynamic programming and the resulting state after a transition is uncertain. The goal is to find a policy to select an optimal action to take in any state. This is different from the pure optimisation goal where we want to directly produce a set of decisions that reach a given goal.

More recently, Perez, Malalel, *et al.* (2023) continued the long history of integrating DDs with constraint solvers (Andersen *et al.* 2007; Hoda, Van Hoeve, and Hooker 2010; Perez and Régin 2015) by proposing DD-based propagators for confidence constraints in constraint programming (CP). Confidence constraints are the CP equivalent of chance constraints where constraint violations are tolerated provided the probability of violation is below a certain threshold. The propagators in Perez, Malalel, *et al.* (2023) construct DDs for variable assignments in the problem and eliminate values in the variable domain that do not reach the required confidence level.

DDs have also been integrated with stochastic linear programs, specifically 2-stage problems where optimal decisions are made in the first stage based on the available information and the decisions in the second stage are only optimal in expectation (Lozano and C. Smith 2022; MacNeil and Bodur 2024) with the key insight being that the second-stage problems can be modelled as binary decision diagrams with edge weights parametrized by first stage decisions.

There is also work on using DDs to solve chance constrained problems. DDs in this context are usually used as a means to represent specific decisions or scenarios with the aim being to reformulate the DD to be introduced as additional constraints into some master problem. Latour, Babaki, Dries, *et al.* (2017) and Latour, Babaki, and Nijssen (2019) reformulate their DDs into quadratic constraint models and Haus, Michini, and Laumanns (2017) reformulate their DDs into linear inequalities. In Casassus, Castro, and Angulo (2025), DDs are used to solve the chance constrained

parallel scheduling problem in a similar way, using DDs to generate cuts for the master problem in their decomposition approach.

8.3. Preliminaries

8.3.1. Dynamic Programming

Dynamic programming (DP) (Bellman 1966) is a divide-and-conquer approach that recursively solves nested sub-problems The main ingredient in DP is the reformulation of a given optimisation problem in terms of states and transitions. A DP model is solved in stages where each stage contains a set of states; there are at least as many stages as there are decision variables.

The DP model is made up of the following elements:

- a set of decision variables $X = \{X_1, X_2, \ldots, X_n\}$ each with domains in $D = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$. A sequence of decisions is the vector $x = (x_1, x_2, \ldots, x_n)$ with $x_i \in \mathcal{D}_i$.
- a state space S partitioned into $n+1$ stages where each stage S_i contains a set of states in the i^{th} stage of the DP model and S_0 contains only the root or initial state r.
- a transition function $\tau : S_i \times \mathcal{D}_i \to S_{i+1}$
- a cost function $h : S_i \times \mathcal{D}_i \to \mathbb{R}$

The DP formulation can then be written as

$$f_i^*(s_i) = \max_{x_i \in \mathcal{D}_i} \{c_{x_i}^{s_i} + f_{i+1}^*(\tau(s_i, x_i))\} \tag{8.1}$$

where $f_i^*(s_i)$ is the optimal objective for the sub-problem in state s_i, and $c_{x_i}^{s_i}$ is the cost of taking decision x_i in state s_i, i.e., $c_{x_i}^{s_i} = h(s_i, x_i)$. In the case that the problem has uncertain costs, we can rewrite Equation (8.1) as

$$f_i^*(s_i) = \max_{x_i \in \mathcal{D}_i} \{\xi_{x_i}^{s_i} + f^*(\tau(s_i, x_i))\} \tag{8.2}$$

such that the model has a stochastic objective function $f_i(s_i)$ from any state s_i, cost function $h : S_i \times \mathcal{D}_i \to \xi$ returns a random variable instead of a number, and $\xi_{x_i}^{s_i}$ is the random cost associated with taking decision x_i in state s_i.

A solution Ω is a sequence of decisions such that the total objective of any sequence of decisions is

$$\sum_{x_i \in \Omega} c_{x_i}^{s_i} \tag{8.3}$$

or

$$\sum_{x_i \in \Omega} \xi_{x_i}^{s_i} \tag{8.4}$$

in the uncertain case.

8.3.2. Decision Diagrams

A DD is a DAG $G = (V, E)$ in which every node $v \in V$ represents a state and every edge $e \in E$ is a transition between states based on some variable assignment. The source and sink of the DAG are two special vertices known as the *root* vertex r and *terminal* vertex t. A path then represents a sequence of variable assignments and any $r \to t$ path is a complete assignment of values.

Every layer, i.e., vertices at the same depth, in a typical DD represents a decision variable such that all edges leading to the nodes in that layer represent value assignments to the decision variable. DDs can either be binary (BDDs), i.e., with only two branching options per node, or multi-valued (MDDs), i.e., with an arbitraty number of branching options per node.

DDs are known to have exponentially many nodes, especially when they encode all possible solutions to a problem, i.e., when they are *exact*. Therefore, bounded-size approximations exist to manage exponential growth of DDs (Cire and Van Hoeve 2013). Typically, the size of a DD is bounded by limiting its width, i.e., the maximum number of nodes in any layer of the diagram. *Restricted* DDs encode a subset of possible solutions, while *relaxed* DDs encode a super-set of possible solutions by allowing some infeasible paths to exist (Bergman, Cire, Van Hoeve, *et al.* 2016).

8.3.3. Uncertainty Theory

While the traditional way of reasoning about uncertainty is via probabilities, there is a challenge with instances where there is not enough data available to estimate a probability distribution. Uncertainty theory (B. Liu and B. Liu 2010) presents an alternative method to deal with this based on expert knowledge. The basis is to model occurrences with belief degree functions with values between 0 and 1 where the belief degree represents the confidence with which we believe an event will occur.

Quantities with uncertainty are represented as uncertain variables which are themselves measurable functions from an uncertainty space. The uncertainty distribution of an uncertain variable ξ

$$\Phi(z) = \mathcal{M}\{\xi \leq z\} \tag{8.5}$$

gives the confidence with which we believe an uncertain quantity falls to the left of the current point. The inverse uncertainty distribution $\Phi^{-1}(m) = z$ provides the value z such that $\mathcal{M}\{\xi \leq z\} = m$.

Additionally, an uncertainty distribution Φ is said to be *regular* if it is a continuous and strictly increasing function with respect to z such that

$$0 < \Phi(z) < 1, \tag{8.6}$$

$$\lim_{z \to -\infty} \Phi(z) = 0, \text{ and} \tag{8.7}$$

$$\lim_{z \to +\infty} \Phi(z) = 1. \tag{8.8}$$

We point the reader to (B. Liu and B. Liu 2010) for a more complete background on uncertainty theory.

8.4. Problem Definition

Let us consider the following formulation for a generic chance constrained optimisation problem with uncertainty in the objective

$$\begin{aligned} \max \quad & T_0 && (8.9)\\ \text{s.t.} \quad & P(f(x,\xi) > T_0) \geq \alpha && (8.10)\\ & g(x) \geq 0 && (8.11)\\ & \xi \geq 0 && (8.12) \end{aligned}$$

where $f(.,.)$ is an objective function dependent on decision variables x, and independent non-negative random variables ξ, and $g(.)$ corresponds to constraints on the problem. The goal is to find the maximum value T_0 such that the chances of our objective function exceeding T_0 are at least α.

By recognising that $P(f(x,\xi) > T_0)$ is the reliability function $R_{f(x,\xi)}(T_0)$, we can re-write Equation (8.10) as [1]

$$R_{f(x,\xi)}(T_0) \geq \alpha \equiv R^{-1}_{f(x,\xi)}(\alpha) \geq T_0. \tag{8.13}$$

We can therefore re-write this problem as

$$\begin{aligned} \max \quad & R^{-1}_{f(x,\xi)}(\alpha) && (8.14)\\ \text{s.t.} \quad & g(x) \geq 0 && (8.15)\\ & \xi \geq 0 && (8.16) \end{aligned}$$

Similarly, we can consider a minimisation problem such that Equations (8.9), (8.11) and (8.12) become

$$\begin{aligned} \min \quad & F^{-1}_{f(x,\xi)}(\alpha) && (8.17)\\ \text{s.t.} \quad & g(x) \geq 0 && (8.18)\\ & \xi \geq 0 && (8.19) \end{aligned}$$

as

$$P(f(x,\xi) \leq T_0) \geq \alpha \equiv F_{f(x,\xi)}(T_0) \geq \alpha \equiv F^{-1}_{f(x,\xi)}(\alpha) \leq T_0. \tag{8.20}$$

Solutions to such problems are sometimes also referred to as α-reliable solutions alluding to the fact that the probability of the decision variables attaining the computed objective exceeds α, with α generally being sufficiently large for systems to be considered reliable (Beraldi and Guerriero 2008; Corredor-Montenegro *et al.* 2021).

[1] The reliability function $R_\xi(.)$ of a random variable ξ is complementary to its cumulative distribution function $F_\xi(.)$ as $R_\xi(.) = 1 - F_\xi(.)$.

α-Reliable DD

A decision diagram representation of an optimisation problem is constructed from a recursive formulation of the problem such that each node represents the root of a sub-problem. Consequently, the weight of each edge represents the contribution of the decision along that edge to the objective function. Therefore, every $r \rightarrow t$ path is a sequence of decisions and represents a solution Ω with an objective function value equal to the length of the path. Thus, the optimal objective function value corresponds to the longest $r \rightarrow t$ path in the DD for a maximization problem.

Each of the edge weights in a DD corresponds to costs in the DP model in Equation (8.2) and finding the longest path is then a matter of finding the α-reliable longest path in a DAG which in turn solves the optimisation problem in Equations (8.9), (8.11) and (8.12).

The α-reliable longest path problem for a DAG with nodes V and edges E such that each edge $(i, j) \in E$ connects two nodes $i, j \in V$ and has random weight ξ_{ij} is:

$$\max \quad T_0 \tag{8.21}$$

$$\text{s.t.} \quad P(l(p) > T_0) \geq \alpha \tag{8.22}$$

$$l(p) = \sum_{(i,j)\in E} x_{ij}\xi_{ij} \tag{8.23}$$

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = \begin{cases} 1, \text{if } i \text{ is a root node} \\ -1, \text{if } i \text{ is a terminal node} \\ 0, \quad \text{otherwise} \end{cases} \tag{8.24}$$

$$x_{ij} \in \{0, 1\} \ \ \forall (i, j) \in E \tag{8.25}$$

where $l(p)$ is the length of a path p in the DAG and x_{ij} is a decision variable indicating whether or not an edge is part of the path.

Similarly, for a minimisation problem, we solve for the α-reliable shortest path in a DAG.

8.5. Discrete Random Variables

In the case of optimisation problems with discrete random variables in the objective, we propose *DD-PMF*, an algorithm to find both the objective of each path in the DD as well as the joint probabilities of edges along the path.

We structure the DD such that each edge carries two labels: (i) its cost and (ii) the probability of realizing that cost. For every $r \rightarrow t$ path in the DD, we then have that its objective function value is the sum of the edge costs and the probability of realizing that path is the product of all edge probabilities. So for a path representing a solution Ω where a solution is a sequence of decisions, there is an associated cost-probability tuple (c_i, p_i) for each $x_i \in \Omega$ such that

$$f(\Omega) = \sum_{x_i \in \Omega} c_i \tag{8.26}$$

$$P(f(\Omega)) = \prod_{x_i \in \Omega} p_i. \tag{8.27}$$

Thus, the probability of reaching any node v in the DD can be computed recursively as

$$P(v) = p_{(u,v)} \cdot P(u) \tag{8.28}$$

where the incoming edge to node v from u has a probability $p_{(u,v)}$ and the probability of reaching the root node $P(r)$ is 1.

Therefore, in constructing the diagram, we can prune all nodes with $P(v) < \alpha$ and return the resulting shortest or longest feasible path as the optimal solution. We call this process DD-PMF and it is as shown in Algorithm 9.

Example 5. Let us consider an uncertain knapsack problem, referred to as the bandit problem. We consider the traditional knapsack problem where the challenge is to select a subset of available items to place in a knapsack of given capacity such that the capacity is not exceeded by the total weight of the objects and the profit derived from the objects is maximized. Suppose we have the same setting but this time with a bandit at the site of a crime, who has a scale with him and can ascertain the weight of the objects but is uncertain about the profit he will derive from selling the items after the robbery. Let us consider an instance with capacity 10 and 3 items of weights $6, 4, 2$ respectively. Say the bandit is aware of a few possibilities for the profit of each item such that the profit distribution for the first item is $(5, 0.7), (3, 0.3)$, i.e., there is a 70% chance the item fetches a profit of 5 and a 30% chance it fetches a profit of 3. Similarly, the profit distribution for the second item is $(5, 1)$, and for the third item is $(7, 0.2), (2, 0.8)$.

For the running example in Example 5, we can construct a DD as in Figure 8.1 where the state representation is the tuple *(avail,profit,probability)* such that *avail* is the available capacity, *profit* is the highest profit achievable along any path to that state, and *probability* is the likelihood of reaching that state. Edges in this diagram represent the binary choice of putting an item in the knapsack or not where dotted edges represent the choice of not taking the item. Each edge has two labels, the weight which represents the added profit of the choice made along the edge, and the probability of achieving that profit for the given item.

Relaxed DD-PMF

Exact DDs are known to grow exponentially even in the deterministic case and the diagrams in DD-PMF are even larger because we have multiple edges for the same decision. Thus relaxations are very important for this formulation.

Merging Nodes A key operation in relaxing DDs is node merging, which leads to smaller DDs by reducing the number of nodes in the diagram. When we merge nodes, we target relaxations so we remain optimistic about the state of the resulting merged node. This remains the same in DD-PMF with the addition that we also remain optimistic about the likelihood of reaching the merged node.

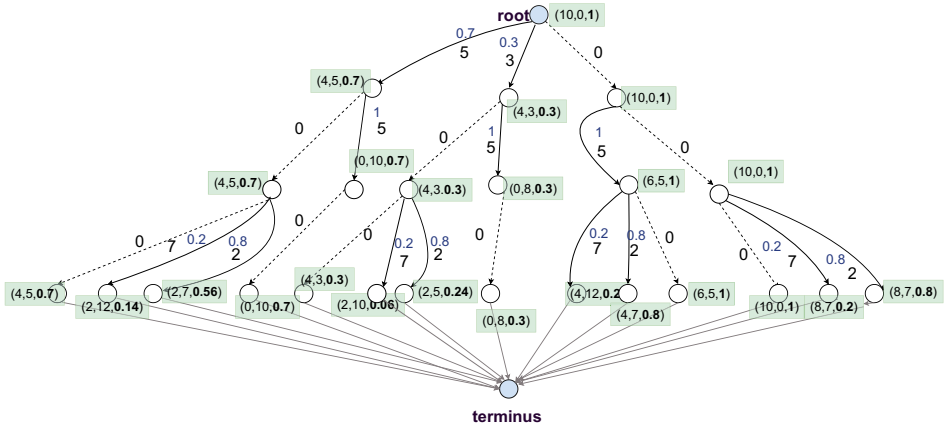

Figure 8.1.: Sample decision diagram for the problem in Example 5 with discrete random object profits.

Algorithm 9 Exact DD-PMF Compilation

function DD-PMF(dynamic programming model with n decision variables and corresponding domains $D = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$, transition function τ, and cost function h) ▷ **returns** decision diagram (V, E)
  create root node r and terminal node t
  $l_1 \leftarrow \{r\}$ ▷ create the first layer of the diagram
  $V \leftarrow l_1$ ▷ set of diagram nodes
  $E \leftarrow \emptyset$ ▷ set of diagram edges
  for $j = 1$ to n **do**
    $l_{j+1} \leftarrow \emptyset$ ▷ create a new layer
    for $u \in l_j$ **do**
      for $d \in \mathcal{D}_j$ **do** ▷ create all feasible transitions from previous layer
        for $(c, p) \in h_j(u, d)$ **do**
          ▷ (cost function h returns discrete distribution as *(value,probability)* tuples)
          $u' \leftarrow \tau_j(u, d)$
          $l_{j+1} \leftarrow l_{j+1} \cup \{u'\}$
          $e' \leftarrow (u, u', c, p)$
            ▷ create new edge as *(source,destination,weight,probability)*
          $E \leftarrow E \cup \{e'\}$
    $V \leftarrow V \cup l_{j+1}$
  Connect l_{n+1} to t
  return (V, E)

Thus, for a merged node v'', the probability of reaching v'' can be computed

recursively as

$$P(v'') = \max_{(u,v'')\in In(v)} (p_{(u,v'')}) \cdot \max_{(u,v'')\in In(v'')} (P(u)) \tag{8.29}$$

where $In(v'')$ returns all incoming edges to node v''.

8.6. Continuous Random Variables

In the case of optimisation problems with continuous random variables in the objective, we can no longer directly apply DD-PMF as there is an infinite number of values the variables can take, which would result in an infinite number of branches for every node. However, the problem in Equations (8.9), (8.11) and (8.12) can still be solved or bounded by a DD depending on the nature of the uncertain variables.

Normally Distributed Variables

In the section, we consider the case that the random variables are all normally distributed where a normally distributed random variable ξ_i is parametrised by its mean μ_i and standard deviation σ_i. We also assume a confidence level $\alpha > 0.5$. These are realistic assumptions as normally distributed parameters have been known to occur in real-world optimisation problems (Novak *et al.* 2022; Seo, Klein, and Jang 2005) and 0.5 is considered very low for reliable solutions.

Theorem 4. *Let $\{\xi_1, \xi_2, \ldots, \xi_n\}$ be n independent normally distributed random variables with cumulative distribution functions $F_1, F_2, \ldots, F_n$ respectively. Then the random uncertain variable $\xi_{sum} = \sum_{i=1}^{n} \xi_i$ has an inverse cumulative distribution function F_{sum}^{-1} such that $F_{sum}^{-1}(\alpha) \leq \sum_{i=1}^{n} F_i^{-1}(\alpha)$ for all $\alpha > 0.5$.*

Proof. First, we know that the sum ξ_{sum} of normally distributed random variables $\xi_1, \xi_2, ..., \xi_n$ is itself normally distributed with mean $\mu_{sum} = \sum_{i=1}^{n} \mu_i$ and variance $var_{sum} = \sum_{i=1}^{n} var_i$.

We also know that the inverse CDF of a normal distribution is defined by

$$F^{-1}(\alpha) = \mu + \sigma\sqrt{2}\,\mathrm{erf}^{-1}(2\alpha - 1) \tag{8.30}$$

where erf is the Gauss error function (Oldham *et al.* 2009).

Thus, the inverse CDF of ξ_{sum} is defined by

$$F_{sum}^{-1}(\alpha) = \sum_{i=1}^{n} \mu_i + \sqrt{\sum_{i=1}^{n} \sigma_i^2}\sqrt{2}\,\mathrm{erf}^{-1}(2\alpha - 1) \tag{8.31}$$

Meanwhile,

$$\sum_{i=1}^{n} F_i^{-1}(\alpha) = \sum_{i=1}^{n} \mu_i + \sum_{i=1}^{n} \sigma_i(\sqrt{2}\,\mathrm{erf}^{-1}(2\alpha - 1)) \tag{8.32}$$

Comparing the terms on the right hand side of Equations (8.31) and (8.32), we know that $\sqrt{\sum_{i=1}^{n} \sigma_i^2}\sqrt{2}\,\mathrm{erf}^{-1}(2\alpha - 1) \leq \sum_{i=1}^{n} \sigma_i(\sqrt{2}\,\mathrm{erf}^{-1}(2\alpha - 1))$ since

- $\sqrt{\sum_{i=1}^{n}\sigma_i^2} \leq \sum_{i=1}^{n}\sigma_i$ provided all σ_i are positive, which standard deviation values are by definition and,
- $\text{erf}^{-1}(2\alpha-1)$ is positive for all $\alpha > 0.5$ because by definition (Oldham *et al.* 2009)

$$\text{erf}^{-1}(x)\begin{cases} < 0, \text{ if } x < 0 \\ > 0, \text{ if } x > 0 \end{cases}. \tag{8.33}$$

We can therefore say that at any confidence level above 0.5 we have

$$F_{sum}^{-1}(\alpha) \leq \sum_{i=1}^{n} F_i^{-1}(\alpha) \tag{8.34}$$

■

Theorem 5. *Let $\{\xi_1, \xi_2, \ldots, \xi_n\}$ be n independent normally distributed random variables with reliability functions $R_1, R_2, \ldots, R_n$ respectively, then the random uncertain variable $\xi_{sum} = \sum_{i=1}^{n}\xi_i$ has an inverse reliability function R_{sum}^{-1} such that $R_{sum}^{-1}(\alpha) \geq \sum_{i=1}^{n} R_i^{-1}(\alpha)$ for all $\alpha > 0.5$.*

Proof. Recall that

$$R^{-1}(\alpha) = F^{-1}(1-\alpha). \tag{8.35}$$

Additionally,

$$F_{sum}^{-1}(1-\alpha) = \sum_{i=1}^{n}\mu_i + \sqrt{\sum_{i=1}^{n}\sigma_i^2}\sqrt{2}\,\text{erf}^{-1}(1-2\alpha) \tag{8.36}$$

and

$$\sum_{i=1}^{n} F_i^{-1}(1-\alpha) = \sum_{i=1}^{n}\mu_i + \sum_{i=1}^{n}\sigma_i(\sqrt{2}\,\text{erf}^{-1}(1-2\alpha)) \tag{8.37}$$

We also know that $\sqrt{\sum_{i=1}^{n}\sigma_i^2} \leq \sum_{i=1}^{n}\sigma_i$ and that for any confidence level above 0.5, $\text{erf}^{-1}(1-2\alpha) < 0$. Therefore,

$$F_{sum}^{-1}(1-\alpha) \geq \sum_{i=1}^{n} F_i^{-1}(1-\alpha) \tag{8.38}$$

Thus from Equation (8.35)

$$R_{sum}^{-1}(\alpha) \geq \sum_{i=1}^{n} R_i^{-1}(\alpha) \tag{8.39}$$

■

Corollary 1. *Let $\{\xi_1, \xi_2, \ldots, \xi_n\}$ be n independent normally distributed random variables with cumulative distribution functions $F_1, F_2, \ldots, F_n$ respectively. Then the random uncertain variable $\xi_{sum} = \sum_{i=1}^{n}\xi_i$ has an inverse cumulative distribution function F_{sum}^{-1} such that $F_{sum}^{-1}(\alpha) \geq \sum_{i=1}^{n} F_i^{-1}(1-\alpha)$ for all $\alpha > 0.5$.*

Proof. This follows from Theorems 4 and 5. Combining Equation (8.31) and Equation (8.36) with the fact that[2] $\text{erf}^{-1}(x) = -\text{erf}^{-1}(-x)$, we can say that

$$F^{-1}_{sum}(\alpha) = \sum_{i=1}^{n} \mu_i + \sqrt{\sum_{i=1}^{n} \sigma_i^2}\sqrt{2}\,\text{erf}^{-1}(2\alpha - 1) \tag{8.40}$$

$$F^{-1}_{sum}(1-\alpha) = \sum_{i=1}^{n} \mu_i - \sqrt{\sum_{i=1}^{n} \sigma_i^2}\sqrt{2}\,\text{erf}^{-1}(2\alpha - 1). \tag{8.41}$$

Thus,

$$F^{-1}_{sum}(\alpha) \geq F^{-1}_{sum}(1-\alpha). \tag{8.42}$$

From Theorem 5, $F^{-1}_{sum}(1-\alpha) \geq \sum_{i=1}^{n} F_i^{-1}(1-\alpha)$
Therefore

$$F^{-1}_{sum}(\alpha) \geq \sum_{i=1}^{n} F_i^{-1}(1-\alpha). \tag{8.43}$$

■

Corollary 2. *Let* $\{\xi_1, \xi_2, \ldots, \xi_n\}$ *be* n *independent normally distributed random variables with reliability functions* $R_1, R_2, \ldots, R_n$ *respectively, then the random uncertain variable* $\xi_{sum} = \sum_{i=1}^{n} \xi_i$ *has an inverse reliability function* R^{-1}_{sum} *such that* $R^{-1}_{sum}(\alpha) \leq \sum_{i=1}^{n} R_i^{-1}(1-\alpha)$ *for all* $\alpha > 0.5$.

Proof. We know from Corollary 1 that

$$F^{-1}_{sum}(1-\alpha) \leq F^{-1}_{sum}(\alpha), \tag{8.44}$$

and from Theorem 4

$$F^{-1}_{sum}(\alpha) \leq \sum_{i=1}^{n} F_i^{-1}(\alpha). \tag{8.45}$$

Thus,

$$F^{-1}_{sum}(1-\alpha) \leq \sum_{i=1}^{n} F_i^{-1}(\alpha), \tag{8.46}$$

and using $R^{-1}(\alpha) = F^{-1}(1-\alpha)$:

$$R^{-1}_{sum}(\alpha) \leq \sum_{i=1}^{n} R_i^{-1}(1-\alpha) \tag{8.47}$$

■

In constructing an α-reliable DD as defined in Section 8.4, the aim is to find the longest path where the length $l(p)$ of a path p in a DD is the sum of all the edge weights which are themselves uncertain variables. Using Theorems 4 and 5 and constructing a DD such that the edge weights are set to α-reliable values, i.e.,

[2] erf^{-1} is an odd function.

each uncertain edge weight w_e is set to $R^{-1}_{w_e}(\alpha)$, provides a bound on the α-reliable solution if all the variables are normally distributed.

From Theorem 4 and Corollary 1, the length of any path

$$l(p) = \sum_{(i,j)\in p} x_{ij} F^{-1}_{ij}(\alpha) \tag{8.48}$$

in the DD provides an upper bound on the α-reliable length of the path and

$$l(p) = \sum_{(i,j)\in p} x_{ij} F^{-1}_{ij}(1-\alpha) \tag{8.49}$$

provides a lower bound.

Thus, we can find a bound on the problem in Equation (8.2) to a confidence level α by constructing a DD with each random edge weight labeled as $F^{-1}_{ij}(\alpha)$ or $F^{-1}_{ij}(1-\alpha)$ and finding the longest or shortest path as usual.

We call this process *DD-CDF-Bound* and it is shown in Algorithm 10. Note that the resulting best path of *DD-CDF-Bound* is only a bound even if the DD is exact. A relaxed DD constructed this way also provides a bound while restricted DDs lose their meaning in this scenario.

Example 6. Let us re-consider the example in Example 5 but with normal distributions for the profit attainable per item.

For the running example in Example 6, we can construct a DD as in Figure 8.2 where the state representation is the same as in Figure 8.1. Each edge in this case has the same probability α and the weight of the edge is the weight w such that $P(\mathit{profit} \leq w) = \alpha$ as this is a maximisation problem.

Algorithm 10 α-Reliable DD-CDF-Bound Compilation

function DD-CDF-Bound(dynamic programming model with n decision variables and corresponding domains $D = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$, transition function τ, and cost function h) ▷ **returns** decision diagram (V, E)
  create root node r and terminal node t
  $l_1 \leftarrow \{r\}$ ▷ create the first layer of the diagram
  $V \leftarrow l_1$ ▷ set of diagram nodes
  $E \leftarrow \emptyset$ ▷ set of diagram edges
  for $j = 1$ to n **do** ▷ for every decision variable
    $l_{j+1} \leftarrow \emptyset$ ▷ create a new layer
    for $u \in l_j$ **do**
      for $d \in \mathcal{D}_j$ **do** ▷ create all feasible transitions from previous layer
        $u' \leftarrow \tau_j(u, d)$
        $l_{j+1} \leftarrow l_{j+1} \cup \{u'\}$
        $e' \leftarrow (u, u', F^{-1}_{h_j(u,d)}(\alpha))$ ▷ create new edge as *(source,destination,weight)*
        $E \leftarrow E \cup \{e'\}$
    $V \leftarrow V \cup l_{j+1}$
  Connect l_{n+1} to t
  return (V, E)

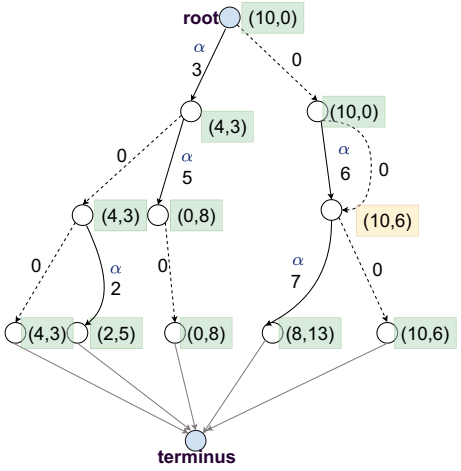

Figure 8.2.: Sample decision diagram for the problem in Example 6 with continuous uncertain object profits with one merged node highlighted in yellow.

Continuous Uncertain Variables

There are multiple ways to model uncertainity. In the case that we model them as uncertain variables (B. Liu and B. Liu 2010), we have the following result:

Theorem 6. *(Y. Gao 2011) Let* $\{\xi_1, \xi_2, \ldots, \xi_n\}$ *be n independent continuous uncertain variables with regular uncertainty distributions* $\Phi_1, \Phi_2, \ldots, \Phi_n$ *respectively and let* $f : \mathbb{R}^n \to \mathbb{R}$ *be a continuous and strictly increasing function, then the random uncertain variable* $\xi = f(\xi_1, \xi_2, \ldots, \xi_n)$ *has an inverse uncertainty distribution* Φ_ξ^{-1} *such that* $\Phi_\xi^{-1}(\alpha) = f(\Phi_1^{-1}(\alpha), \Phi_2^{-1}(\alpha), \ldots, \Phi_n^{-1}(\alpha))$

From Theorem 6, applying the same process as in Algorithm 10 to uncertain variables, provides the α-reliable solution in an exact DD and the bound in a relaxed or restricted DD. We call this process *DD-UD*.

Time Complexity

In both DD-CDF-Bound and DD-UD, we have only modified the edge weights and the rest of the DD construction is the same, thus the time complexity is unchanged compared to the deterministic case.

Considering other DD Dynamics

As our proposals only affect the edge weights, the rest of the DD compilation process largely remains the same for all three proposed algorithms. We discuss the main dynamics below.

Merging and Splitting Nodes Merge operators operate on state information with the point being to provide as tight a relaxation as possible. As such, the actual merge operation is not affected as none of our proposals affect state information. However, in DD-PMF, there is the chance to update merge heuristics to perform node selection based on the edge probability, for instance, the least likely occurrences can be merged. Splitting on the other hand, usually involves a redistribution of edges. We can select edges for redistribution in the same way as we have selected edges for merging.

Incremental Refinement So far in Algorithms 9 and 10, we have assumed top-down construction, but this is not the only existing DD compilation procedure. The alternative procedure, incremental refinement, is also unaffected by our proposals. An additional operation in incremental refinement is filtering infeasible edges and since we assume that none of the uncertain variables appear in constraints, we can still filter as usual.

8.7. Computational Experiments

In this section, we present computational experiments to quantify the performance of our proposed algorithms: DD-PMF (Algorithm 9), DD-CDF-Bound (Algorithm 10), and DD-UD based on the quality of the solutions and bounds they produce.

8.7.1. Benchmarks

We evaluate the three algorithms proposed above on the KNAPSACK problem as introduced in Examples 5 and 6. We create uncertain versions of the low-dimensional uncorrelated (LD-UC) knapsack problem benchmarks from Pisinger (2005). The original benchmarks are presented without uncertainty in the attainable profit per item, so we modify them to create new problems with uncertainty. Our modifications are as follows:

Discrete uncertain benchmarks

In these benchmarks, we take each profit value and randomly generate 1 to 5 profit scenarios where the number of scenarios is randomly selected assuming a uniform distribution, and the resulting profit for an item in each scenario is randomly chosen in the range of 0.1 to 2 times the profit in the standard benchmarks also assuming a uniform distribution. The probability distribution over the scenarios is then generated from the Dirichlet distribution (K. W. Ng, Tian, and Tang 2011) with a target sum of 1.

Normally distributed uncertain benchmarks

In order to create these benchmarks, we take each profit value and generate the parameters of a normal distribution such that the mean is the profit in the standard benchmarks and the standard deviation is chosen as a fraction of the mean. We select the standard deviation by uniformly sampling the fraction by which we multiply the mean from four ranges: [0.001, 0.002), [0.005, 0.004), [0.03, 0.4), [0.25, 0.4). These ranges vary the spread of the distribution, allowing us to evaluate scenarios with

different degrees of uncertainty – the smaller the standard deviations, the likelier it is that the profit is close to the mean.

Continuous uncertain benchmarks.

For continuous uncertainty benchmarks, we assume a zigzag distribution (B. Liu and B. Liu 2010) with parameters a, b, and c such that the uncertainty distribution function is piecewise linear between points (a, b) and points (b, c). The parameters a, b, and c are again generated relative to the profit in the standard benchmarks. The values are drawn by randomly sampling three values in the range 0.1 to 2 times the profit in the standard benchmarks and placing them in ascending order to get a, b, and c.

8.7.2. Baselines

In order to evaluate the performance of our proposed algorithms, we establish some baselines for the three classes of problems considered in this chapter. All baselines are based on MIP and CP models. As such, we define the nomenclature below that holds for all three models.

Input variables

I	Set of items available to be placed in the knapsack
C	Capacity of the knapsack
w_i	Weight of the i-th item
p_i^{μ}	Mean of the profit attainable from the i-th item
p_i^{σ}	Standard deviation of the profit attainable from the i-th item
p_i	Profit attainable from the i-th item
p_i^a	Point a of the zigzag uncertain distribution of the profit attainable from the i-th item
p_i^b	Point b of the zigzag uncertain distribution of the profit attainable from the i-th item
p_i^c	Point c of the zigzag uncertain distribution of the profit attainable from the i-th item
α	The required confidence level of the solution
K_α	Evaluation of the reliability function of the standard normal distribution at the required confidence
S	Set of scenarios considered
$p_{(i,j)}$	Profit attainable from the i-th item in the j-th scenario
$\mathcal{M}$	big M value

Discrete uncertain baseline

For this problem class, we compare our solution with a scenario-based MIP model using Sample Average Approximation (SAA) (Kleywegt, Shapiro, and Homem-de-Mello 2002). The model works by drawing multiple scenarios sampled from the discrete distribution of profits. Thus, a scenario is a set of possible profit values for each item

in the knapsack. We then solve for the choice of variables that maximises the profit provided that the ratio of scenarios in which that profit can actually be achieved is at least the confidence threshold α (Equation (8.51c)).

Equation (8.51a) below enforces that the chosen items do not exceed the knapsack capacity, Equation (8.51b) enforces that every scenario that reaches the target profit sets its satisfiability variable s_j^{sat} to 1 via a big-M formulation, and Equation (8.51c) enforces the confidence level constraint.

Note that the accuracy of the solution provided is dependent on the number of scenarios in S as SAA is an approximation and converges to the true optimal solution as $|S| \to \infty$. Thus too few samples could lead to (under)over-estimations.

Decision variables

c_i	Binary variable signifying if the i-th item is in the knapsack or not
$\mathscr{P}$	Variable for the total profit acquired
s_j^{sat}	Binary variable signifying if the j-th scenario achieves profit $\mathscr{P}$ or not

Objective

$$\max(\mathscr{P}) \tag{8.50}$$

Constraints

$$\sum_{i \in I} c_i w_i \leq C \tag{8.51a}$$

$$\mathscr{P} - \sum_{i \in I} (c_i p_{(i,j)}) \leq (1 - s_j^{sat}) \cdot \mathcal{M} \tag{8.51b}$$

$$\frac{\sum_{j \in S}(s_j^{sat})}{|S|} \geq \alpha \tag{8.51c}$$

This model is referred to as **MIP-PMF** in the rest of our experiments.

Normally distributed uncertain baseline

In this case, we compare our bounds with a chance-constrained CP model defined as follows:

Decision variables

c_i	Binary variable signifying if the i-th item is in the knapsack or not
$\mathscr{P}$	Variable for the total profit acquired
$\mathscr{P}^{\mu}$	Mean of the total acquired profit
$\mathscr{P}^{\sigma}$	Standard deviation of the total acquired profit

Objective

$$\max(\mathscr{P}) \tag{8.52}$$

Constraints

$$C1: \sum_{i \in I} c_i w_i \leq C \tag{C1}$$

$$C2: \mathscr{P}^{\mu} = \sum_{i \in I} c_i p_i^{\mu} \tag{C2}$$

$$C3: (\mathscr{P}^{\sigma})^2 = \sum_{i \in I} c_i (p_i^{\sigma})^2 \tag{C3}$$

$$C4: \mathscr{P} = \mathscr{P}^{\mu} + \mathscr{P}^{\sigma} K_{\alpha} \tag{C4}$$

The limit on the knapsack capacity is enforced by Constraint C1. To deal with normally distributed profits, this CP model directly uses the properties of a normal distribution, i.e., the mean and variance of the profit $\mathscr{P}$ are the sum of the means and variances of profits of all chosen items in the knapsack (Constraints C2 and C3). Furthermore, the actual achievable profit at the confidence level α is enforced by Constraint C4 by scaling the reliability function of the standard normal distribution.

Note that this model is referred to as **CP-Normal** in the rest of our experiments.

Continuous uncertain baseline

This is perhaps the most direct model of the baselines. It is the standard MIP model for the knapsack problem adjusted to get the profit at a certain confidence α from the inverse zigzag uncertainty distribution (Equation (8.55b)) below. The inverse zigzag uncertainty distribution is by nature a piecewise affine function but since we know the confidence level α as an input, it is affine in our model[3].

Decision variables

- c_i Binary variable signifying if the i-th item is in the knapsack or not
- $\mathscr{P}$ Variable for the total profit acquired

Objective

$$\max(\mathscr{P}) \tag{8.54}$$

Constraints

$$\sum_{i \in I} c_i w_i \leq C \tag{8.55a}$$

$$\mathscr{P} = \begin{cases} \sum_{i \in I} c_i ((2\alpha - 1) p_i^a + (2 - 2\alpha) p_i^b) & \text{if } (1-\alpha) < 0.5 \\ \sum_{i \in I} c_i (2\alpha p_i^b + (1 - 2\alpha) p_i^c) & \text{otherwise} \end{cases} \tag{8.55b}$$

This model is referred to as **MIP-UD** in the rest of our experiments.

8.7.3. Results

In this section, we present our numerical results. All experiments are performed on an 16-core 1.9GHz AMD machine running Ubuntu 22.04 with 32GB RAM. Algorithms are implemented in Rust and publicly available[4]. The MIP and CP models are solved by CPLEX version 22.1 and CP Optimizer version 22.1, respectively.

[3] We only trigger the segment that corresponds to α.

[4] https://github.com/Eghonghonaye/uncertain-ddo

How good are the bounds derived from DD-CDF-Bound in practice? In this set of experiments, we compare the bounds obtained from DD-CDF-Bound to (optimal) solutions obtained from the CP model. Figure 8.3 shows the gap between the bound and the best CP solution. We note that the quality of the bound is influenced by the spread of the variable quantified by its standard deviation. Of the 4 ranges of standard deviation we consider, we see significantly wider gaps between the bounds from DD-CDF-Bound and solutions from CP (up to 400%) for the highest standard deviations i.e., 0.25 to 0.5 times the mean. This is compared to the average of 30% for the smallest standard deviation range of 0.001 to 0.002 times the mean. This is reasonable as the standard deviation being relatively small compared to the mean reduces the discrepancy between $F^{-1}_{sum}(\alpha)$ and $\sum_{i=1}^{n} F_i^{-1}(\alpha)$, i.e., the actual cumulative distribution function of the objective and its approximation (see Equations (8.31) and (8.32)).

Furthermore, the required confidence level also affects the bounds in this solution, with the bound becoming more pessimistic as confidence increases.

On the upside, the runtime of DD-CDF-Bound appears (i) relatively constant regardless of the spread of the variables, and (ii) significantly shorter than the runtime of the CP model (Figure 8.4). This is a desired property as finding bounds is one step within a solve and should be incorporated at limited cost to the process, especially when they are not guaranteed to be tight.

How does DD-UD perform compared to the baseline? In Figure 8.5, we present our results on DD-UD. We compare its performance with the MIP model defined in Equations (8.54), (8.55a) and (8.55b). We find that we are able to get the same solutions from both DD-UD and MIP-UD in similar amounts of time, validating our DD formulation.

How does DD-PMF perform compared to the baseline? In Figure 8.6, we show the performance of DD-PMF compared to the MIP model defined in Equations (8.50)

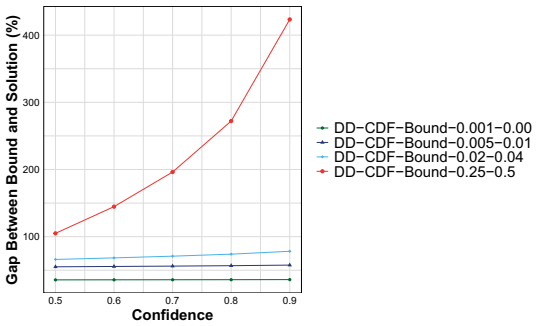

Figure 8.3.: Quality of bounds provided by DD-CDF-Bound for the knapsack problem with normally distributed profits.

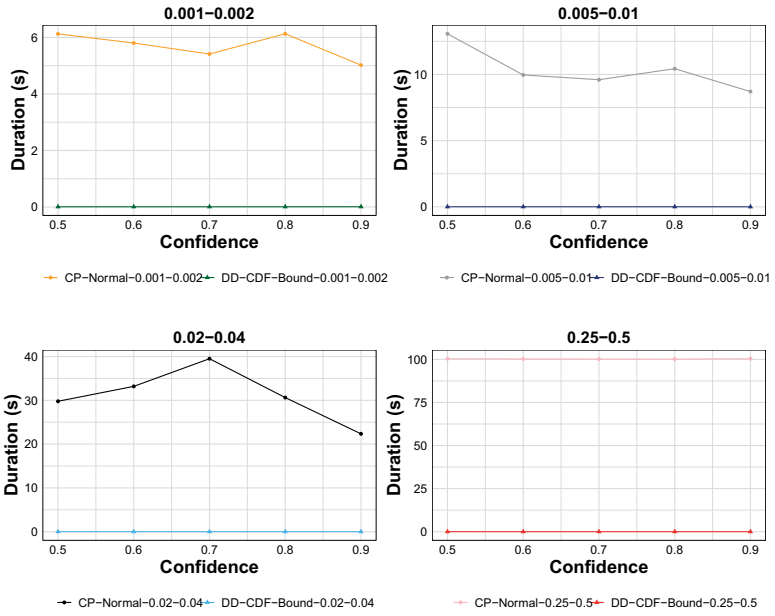

Figure 8.4.: Runtime of solutions provided for the knapsack problem with normally distributed profits.

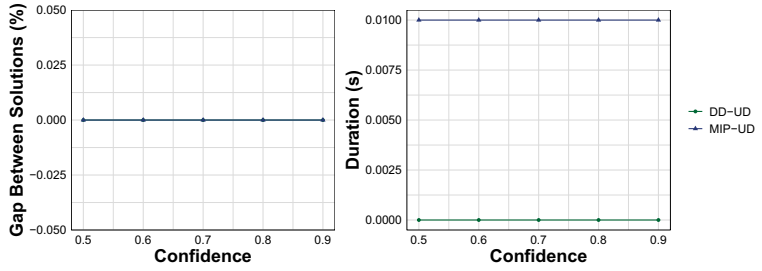

Figure 8.5.: Performance of solutions provided for the knapsack problem with zigzag uncertain profits.

and (8.51a) to (8.51c) across scenario sets with cardinality 1000, 5000, and 10000. We find DD-PMF to be superior in both runtime and solution quality metrics. Although MIP-PMF appears optimistic, reaching higher objective values, further inspection of the provided solutions shows that the confidence level is lower than required. This is a known issue with sampling based methods as they often require a high number of samples to represent the distribution accurately. This scenario size dependent performance variation is absent in DD-PMF.

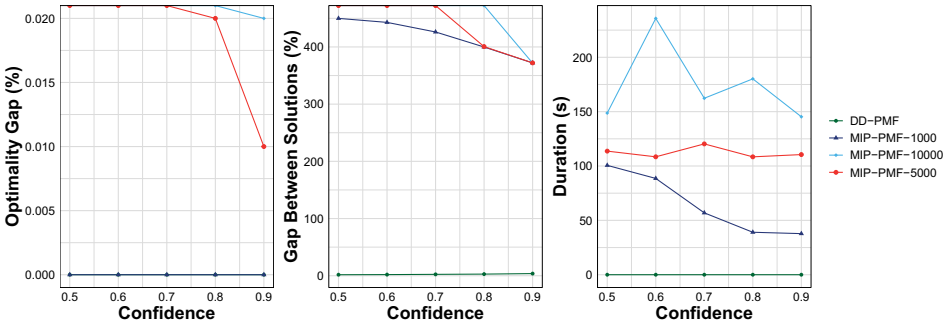

Figure 8.6.: Performance of solutions provided for the knapsack problem with discrete uncertain profits.

As a general note and a point for future work, we note that the benchmarks our instances are derived from in these experiments are limited in both variety and difficulty – there are only 10 instances and they have been solved to optimality in the deterministic case by many algorithms. While the point of these exploratory experiments in this chapter was to show the potential of each solution method proposed, it is a strong recommendation for future work to explore more benchmarks across different problem classes to more concretely quantify performance.

8.8. Conclusions

We have considered decision diagrams for optimising problems with uncertainty in their objective and provided three algorithms for different models of uncertain variables. We have looked into problems with discrete random variables, continuous random variables and continuous uncertain variables and proved specific results on finding bounds when the variables are normally distributed. Our computational experiments indicate that decision diagrams are significantly superior to mixed integer programming for handling discrete uncertainties in the objective function. Additionally, the quality of bounds derived from our proposed algorithm for normally distributed variables in the objective function are dependent on the spread of the underlying distribution. Recommended topics for future work are (i) extensions of the proposed solutions to cases with uncertainty also in the constraints, and (ii) generalized theoretical guarantees of bounds derivable from decision diagrams independent of specific distributions.

9

Conclusions

In this chapter, we revisit the research questions posed in both parts of this thesis and re-examine our answers. We further reflect on the implications and limitations of our results and end with a discussion on possible future research directions.

9.1. Answers to the Research Questions

In this section, we discuss our answers to the six research questions posed in this thesis. Section 9.1.1 discusses the two research questions addressed in Part I, while Section 9.1.2 discusses the four questions addressed in Part II.

9.1.1. Context Aware Heuristic Scheduling for Flexible Manufacturing Systems

In Part I our overall goal was to increase context awareness of heuristic algorithms for scheduling. As there is a lot of contextual information available in scheduling in general and the effects can vary wildly from application to application, we narrowed this goal down to two research questions as posed in the introduction. Below, we explain how these questions were addressed.

Research Question 1. *How to design a general method to incorporate maintenance decisions in heuristic scheduling algorithms for our driver case?* To answer this question, we first looked at the kinds of maintenance actions present noting that some maintenance actions are closely tied with the schedule of operations as they are (i) executed at the same time scale as operations themselves and (ii) triggered based on sequence-dependent deterioration. While solution methods for sequence dependent maintenance scheduling exist in the literature, the driver case presented an additional challenge in that the maintenance actions depended not only on the sequence of operations, but also on the timing of the operations. We thus answered this question by presenting three solution methods namely, mixed integer programming, constraint programming, and a heuristic solution. We further note that even with planning, some maintenance actions could still violate constraints of

already made schedules especially in the heuristic solution and therefore we designed a schedule repair method to enable recovery from such scenarios.

9.1.2. Decision Diagrams for Scheduling and Beyond

In Part II we considered the research question 'How to leverage decision diagrams to effectively solve real-world problems?'. This broad goal was further refined to four research questions. We looked at different aspects including modelling more problems with decision diagrams, increasing our understanding of how solutions and bounds derived from DDs are affected by the compilation techniques used, and adding more capabilities to DDs such as handling uncertainty. Each of the four questions and our answers to them contribute towards expanding both the understanding of DDs and their applicability to solving real-world optimisation problems.

Research Question 2. *How to formulate a general decision diagram model for optimising manufacturing systems as introduced in Part I of this thesis?* We answered this question in Chapter 5 by providing a generic DD formulation for scheduling manufacturing systems with an arbitrary number of machines. Existing methods for scheduling with DDs either assume there is a single resource on which operations are to be scheduled or that there is an arbitrary number of identical resources or machines. However, real manufacturing systems often have different machines performing different tasks. Our formulation provides a means to model this. In addition to the available machines, there are also new constraints in manufacturing problems including maximum separation constraints and all kinds of convoluted setup time interactions. We provided transition, filtering, and dominance functions for four kinds of setup time interactions and relative deadline constraints. Our experiments showed that our DD formulation is competitive with mixed-integer and constraint programming methods.

Research Question 3. *What is the impact of combining decision diagram compilation techniques on the eventual bounds derived?* To answer this question, we performed a empirical study based on seven classical optimisation problems in Chapter 6. We explored three DD compilation techniques and quantified their effects on the bounds and solutions derived. We also explored the effect of combinations of techniques. The results and guidelines we presented in the study can be used to provide guidance for modellers and users of DD-based solvers as to which heuristics and options perform best for particular problems.

Research Question 4. *How to combine learned policies with decision diagrams?* We answered this question in Chapter 7 by replacing the restriction step in decision diagram based branch-and-bound with a trained reinforcement learning agent. This approach takes advantage of the modelling similarities between reinforcement learning and decision diagrams to enable a seamless handover between the agent and the diagram. We demonstrated the operation of this solution method on the job

shop scheduling problem. Our results showed that the combination of reinforcement learning and decision diagrams outperformed the individual methods.

Research Question 5. *How can we handle uncertainty in optimisation problems solved by decision diagrams?* We answered this question in Chapter 8 by developing DD formulations for problems with uncertainty only in the objective. We considered three types of uncertainty representations namely discrete random variables, continuous random variables, and continuous uncertain variables. We created transitions for every branching possibility for discrete distributions and took advantage of cumulative distribution functions for continuous distributions. Our answers to this question, combined with recent work on stochastic decision diagrams (Hooker 2022) and DD based propagators for chance constraints in CP (Perez, Malalel, *et al.* 2023) provide the ground work for optimisation under uncertainty with DDs.

9.2. Contributions to the State of the Art

Below, we summarise this thesis' main contributions and highlight advances made to the state of the art. Our contributions are primarily algorithms and solution methods that tie back to the two overall goals of this thesis which were (i) to increase context awareness of heuristic algorithms for scheduling, and (ii) to leverage decision diagrams to effectively solve real-world problems.

Chapter 3 from Part I and Appendix A address the first goal. In Chapter 3, we propose a scheme to extend list scheduling heuristics to incorporate maintenance planning. Given a set of operations to be scheduled and a maintenance policy, our algorithms provide a schedule that already includes the necessary maintenance activities triggered by the chosen operation sequence. We further propose a small modification to the BHCS algorithm to improve its anytime behaviour (Appendix A).

Chapters 5 to 8 from Part II address the second goal of the thesis. We provide DD models for new problems and direct improvements to DD based algorithms. Specifically, in Chapter 5, we propose a decision diagram model for multi-machine scheduling and show via empirical evaluations that the proposed DD model is competitive with MILP and CP models. In Chapter 6, we perform an empirical evaluation of DD compilation techniques and provide insights on the effects of combining multiple techniques. In Chapter 7, we integrate leaning with DD based branch-and-bound by building restrictions with a reinforcement learning agent trained to solve the problem. This proposal was shown to significantly improve the results of using the agent alone. Finally, in Chapter 8, we provide new theoretical results on using DDs for chance constrained optimisation. We take advantage of the cumulative distribution function to design the first method that can derive bounds on chance constrained optimisation problems from traditional DDs.

9.3. Reflections

In this section, we reflect on the implications and limitations of the work presented in this thesis.

The algorithm proposed in Chapter 3 of this thesis has been integrated into the scheduler of a Canon (*Canon Production Printing* 2023) machine. Results of this integration presented by Farboud (2025) show some interesting considerations missing in the theoretical analysis.

The first is that a scheduler in a flexible manufacturing systems is one module among very many and execution of scheduling decisions is dependent on other components. Thus, the results achieved in this thesis do not transfer exactly to the implementation. We experience both unexpected losses and gains. Losses in that the makespans achieved were often higher than expected from this thesis and gains in that context awareness led to a significant reduction in frequency of stalling and interruptions from other processes involved in maintenance execution. This leads to the second observation which is that there is room to expand the notion of context awareness to not only include direct constraints and effects on the scheduling problem but also to consider second-order effects such as the response time of the scheduler.

Moving to the world of decision diagrams, we find quite surprisingly that DDs provide competitive results for many classical optimisation problems when compared to both MILP and CP solvers. However, there is still limited adoption partly due to popularity and age as DDs are a more recent means of directly solving optimisation problems. Ease of modelling also plays a role in this as decision diagram models are usually based on dynamic programming, which can be more involved to set up compared to declarative constraints in the CP world for instance.

9.4. Future Work

In this section, we highlight opportunities for future work.

Generalised Chance Constrained Bounds from DDs The current landscape of chance constrained programming with DDs is limited to tight assumptions on the underlying distribution of the uncertain variables. As such, there are still many problems that are not yet solvable by DDs. More research into generalising DDs for uncertainty to function irrespective of the distribution can bring tremendous gains to some difficult problems. We could begin by extending the proposed *DD-CDF-Bound* algorithm in Section 8.6 to work with other distributions with known properties of their sums such as exponential random variables.

Context-awareness Beyond List Scheduling A lot of the work present in this thesis assumes that the heuristic scheduler is a list scheduler. However, many other forms of heuristic schedulers exist. More generalised meta-heuristic frameworks for context aware heuristics would increase the applicability of these algorithms. For example, by integrating the context incorporation step into the construction phase of iterated greedy algorithms.

Better Merge Operators for DDs For Multi-Machine Scheduling In the multi-machine scheduling landscape the merge operator is very coarse. In any merging of

states, the state-of-the-art operators assume each machine is independent and is available as soon as it is available in any of the states to be merged. Many of these interactions are infeasible and constraint propagation techniques could be adopted to strengthen the merge operator. One possibility is to use the propagator for the *no-overlap* constraint to derive earliest machine availabilities. The downside here is the possibility of increased runtime if the merge operator gets expensive to compute as an advantage of building smaller diagrams by merging is speed.

Explainable DDs for Optimisation Small decision diagrams are inherently interpretable and explainable as they can be easily visualised and decisions traced intuitively. However, decision diagrams often grow exponentially in the context of using them to solve optimisation problems and operations like node merging, edge filtering, and pruning are then not always intuitive to follow. Thus, though the underlying structure lends itself to interpretability, the size and complexity means this important feature is missing. Formal verification techniques can be applied to provide explanations and certifications for claims made by DDs. Specifically, proof logging as has been in explored in the field of constraint programming (Flippo *et al.* 2024) could also be very beneficial for this field.

Bibliography

Abumaizar, R. J. and J. A. Svestka (1997). “Rescheduling job shops under random disruptions”. In: *International Journal of Production Research* 35.7, pp. 2065–2082.

Aissi, H., C. Bazgan, and D. Vanderpooten (2009). “Min–max and min–max regret versions of combinatorial optimization problems: A survey”. In: *European Journal of Operational Research* 197.2, pp. 427–438.

Akers (1978). “Binary decision diagrams”. In: *IEEE Transactions on Computers* 100.6, pp. 509–516.

Allahverdi, A. (1996). “Two-machine proportionate flowshop scheduling with breakdowns to minimize maximum lateness”. In: *Computers & Operations Research* 23.10, pp. 909–916.

Andersen, H. R., T. Hadzic, J. Hooker, and P. Tiedemann (2007). “A constraint store based on multivalued decision diagrams”. In: *13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*. Springer, pp. 118–132.

Applegate, D. and W. Cook (1991). “A computational study of the job-shop scheduling problem”. In: *ORSA Journal on Computing* 3.2, pp. 149–156.

Ascheuer, N. (1996). *Hamiltonian path problems in the on-line optimization of flexible manufacturing systems*. Tech. rep. Konrad-Zuse-Zentrum für Informationstechnik, Berlin.

Averkov, G., C. Hojny, and M. Schymura (2023). “Efficient MIP techniques for computing the relaxation complexity”. In: *Mathematical Programming Computation* 15.3, pp. 549–580.

Avilés, F. N., R. M. Etchepare, M. M. Aguayo, and M. Valenzuela (2023). “A mixed-integer programming model for an integrated production planning problem with preventive maintenance in the pulp and paper industry”. In: *Engineering Optimization* 55.8, pp. 1352–1369.

Awad, A., A. Hawash, and B. Abdalhaq (2022). “A genetic algorithm (GA) and swarm-based binary decision diagram (BDD) reordering optimizer reinforced with recent operators”. In: *IEEE Transactions on Evolutionary Computation* 27.3, pp. 535–549.

Becker, B., M. Behle, F. Eisenbrand, and R. Wimmer (2005). “BDDs in a branch and cut framework”. In: *International Workshop on Experimental and Efficient Algorithms*. Springer, pp. 452–463.

Bellman, R. (1966). “Dynamic programming”. In: *Science* 153.3731, pp. 34–37.

Bello, I., H. Pham, Q. V. Le, M. Norouzi, and S. Bengio (2016). “Neural combinatorial optimization with reinforcement learning”. In: *arXiv preprint arXiv:1611.09940*.

Ben-Tal, A., A. Nemirovski, and L. El Ghaoui (2009). *Robust Optimization*. Princeton University Press, pp. 3–25.

Bengio, Y., J. Louradour, R. Collobert, and J. Weston (2009). "Curriculum learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 41–48.

Beraldi, P. and F. Guerriero (2008). "The alpha-reliable shortest path problem". In: *Algorithmic Operations Research* 3.1, pp. 59–66.

Bergman, D., A. A. Cire, W.-J. Van Hoeve, and J. Hooker (2012). "Variable ordering for the application of BDDs to the maximum independent set problem". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 34–49.

– (2016). *Decision Diagrams for Optimization*. Springer, pp. 11–122.

Bergman, D., A. A. Cire, W.-J. van Hoeve, and J. Hooker (2016). "Branch-and-Bound based on decision diagrams". In: *Decision Diagrams for Optimization*, pp. 95–122.

Birge, J. R. and F. Louveaux (2011). *Introduction to Stochastic Programming*. Springer Science & Business Media, pp. 181–338.

Błażewicz, J., E. Pesch, and M. Sterna (2000). "The disjunctive graph machine representation of the job shop scheduling problem". In: *European Journal of Operational Research* 127.2, pp. 317–331.

Bollig, B. and I. Wegener (1996). "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on computers* 45.9, pp. 993–1002.

Canon Production Printing (2023). `https://cpp.canon`. Accessed: 2025-07-07.

Cappart, Q., D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković (2023). "Combinatorial optimization and reasoning with graph neural networks". In: *Journal of Machine Learning Research* 24.130, pp. 1–61.

Cappart, Q., E. Goutierre, D. Bergman, and L.-M. Rousseau (2019). "Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33, pp. 1443–1451.

Caricato, P. and A. Grieco (2008). "An online approach to dynamic rescheduling for production planning applications". In: *International Journal of Production Research* 46.16, pp. 4597–4617.

Casassus, N., M. Castro, and G. Angulo (2025). "A decision diagram approach for the parallel machine scheduling problem with chance constraints". In: *arXiv preprint arXiv:2504.20889*.

Chalumeau, F., I. Coulon, Q. Cappart, and L.-M. Rousseau (2021). "Seapearl: A constraint programming solver guided by reinforcement learning". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 392–409.

Chan, W. T. and T. H. Wee (2003). "A multi-heuristic GA for schedule repair in precast plant production." In: *13th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 236–245.

Chansombat, S., P. Pongcharoen, and C. Hicks (2019). "A mixed-integer linear programming model for integrated production and preventive maintenance scheduling in the capital goods industry". In: *International Journal of Production Research* 57.1, pp. 61–82.

Charnes, A. and W. W. Cooper (1959). “Chance constrained programming”. In: *Management Science* 6.1, pp. 73–79.

Cheng, T. E., J. Diamond, and B. M. Lin (1993). “Optimal scheduling in film production to minimize talent hold cost”. In: *Journal of Optimization Theory and Applications* 79.3, pp. 479–492.

Cire, A. A. and W.-J. Van Hoeve (2013). “Multivalued decision diagrams for sequencing problems”. In: *Operations Research* 61.6, pp. 1411–1428.

Cire, A. A. and W.-J. Van Hoeve (2012). “MDD propagation for disjunctive scheduling”. In: *Twenty-Second International Conference on Automated Planning and Scheduling*, pp. 11–19.

Coppé, V. (2024). “Advances in Discrete Optimization with Decision Diagrams: Dominance, Caching and Aggregation-Based Heuristics”. PhD thesis. Carnegie Mellon University, USA.

Coppé, V., X. Gillard, and P. Schaus (2022). “Solving the constrained single-row facility layout problem with decision diagrams”. In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 14:1–14:18.

– (2024a). “Decision diagram-based branch-and-bound with caching for dominance and suboptimality detection”. In: *INFORMS Journal on Computing*, pp. 1522–1542.

– (2024b). “Modeling and Exploiting Dominance Rules for Discrete Optimization with Decision Diagrams”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 226–242.

Corredor-Montenegro, D., N. Cabrera, R. Akhavan-Tabatabaei, and A. L. Medaglia (2021). “On the shortest α-reliable path problem”. In: *Transactions in Operations Research (TOP)* 29.1, pp. 287–318.

Dechter, R., I. Meiri, and J. Pearl (1991). “Temporal constraint networks”. In: *Artificial Intelligence* 49.1-3, pp. 61–95.

Delorme, M., M. Iori, and N. F. Mendes (2021). “Solution methods for scheduling problems with sequence-dependent deterioration and maintenance events”. In: *European Journal of Operational Research* 295.3, pp. 823–837.

Demirkol, E., S. Mehta, and R. Uzsoy (1998). “Benchmarks for shop scheduling problems”. In: *European Journal of Operational Research* 109.1, pp. 137–141.

De Weerdt, M., R. Baart, and L. He (2021). “Single-machine scheduling with release times, deadlines, setup times, and rejection”. In: *European Journal of Operational Research* 291.2, pp. 629–639.

Ding, J., L. Shen, Z. Lü, and B. Peng (2019). “Parallel machine scheduling with completion-time-based criteria and sequence-dependent deterioration”. In: *Computers & Operations Research* 103, pp. 35–45.

Efthymiou, N. and N. Yorke-Smith (2023). “Predicting the optimal period for cyclic hoist scheduling problems”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 238–253.

Eigbe, E.-a., B. De Schutter, M. Nasri, and N. Yorke-Smith (2022). “Predictive maintenance scheduling in twice re-entrant flow shops with relative due dates”.

Presented at: *The 15th Workshop on Scheduling and Planning Applications (SPARK) co-located with the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*.

Eigbe, E.-a., B. De Schutter, M. Nasri, and N. Yorke-Smith (2023). "Sequence-and time-dependent maintenance scheduling in twice re-entrant flow shops". In: *IEEE Access* 11, pp. 103461–103475. DOI: 10.1109/ACCESS.2023.3317533.

Eigbe, E.-a., C. Schmidl, B. De Schutter, N. Jansen, M. Nasri, and N. Yorke-Smith (2024). "Neural decision diagrams". Presented at: *The 6th Data Science Meets Optimisation Workshop (DSO) co-located with the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*.

Emmons, H. and G. Vairaktarakis (2012). *Flow Shop Scheduling: Theoretical Results, Algorithms, and Applications*. Vol. 182. Springer Science & Business Media, pp. 97–160.

Emmons, H., G. Vairaktarakis, H. Emmons, and G. Vairaktarakis (2013). "Reentrant flow shops". In: *Flow Shop Scheduling: Theoretical Results, Algorithms, and Applications*, pp. 269–289.

Escudero, L. F. (1988). "An inexact algorithm for the sequential ordering problem". In: *European Journal of Operational Research* 37.2, pp. 236–249.

Farboud, P. (2025). "Context-Aware Scheduling in Production Printing". EngD Thesis. Eindhoven University of Technology, The Netherlands.

Flippo, M., K. Sidorov, I. Marijnissen, J. Smits, and E. Demirović (2024). "A multi-stage proof logging framework to certify the correctness of CP solvers". In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 11–1.

Floudas, C. A. (1995). *Nonlinear and mixed-integer optimization: fundamentals and applications*. Oxford University Press, pp. 95–107.

François-Lavet, V., P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, *et al.* (2018). "An introduction to deep reinforcement learning". In: *Foundations and Trends in Machine Learning* 11.3-4, pp. 219–354.

Frohner, N. and G. R. Raidl (2019). "Merging quality estimation for binary decision diagrams with binary classifiers". In: *International Conference on Machine Learning, Optimization, and Data Science*. Springer, pp. 445–457.

– (2020). "Towards improving merging heuristics for binary decision diagrams". In: *Learning and Intelligent Optimization: 13th International Conference, LION 13, Chania, Crete, Greece, May 27–31, 2019, Revised Selected Papers 13*. Springer, pp. 30–45.

Galand, L., J. Lesca, and P. Perny (2013). "Dominance rules for the Choquet integral in multiobjective dynamic programming". In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI*, pp. 538–544.

Gao, Y. (2011). "Shortest path problem with uncertain arc lengths". In: *Computers & Mathematics with Applications* 62.6, pp. 2591–2600.

Garcia de la Banda, M., P. J. Stuckey, and G. Chu (2011). "Solving talent scheduling with dynamic programming". In: *INFORMS Journal on Computing* 23.1, pp. 120–137.

Gawiejnowicz, S. (2020). “A review of four decades of time-dependent scheduling: Main results, new topics, and open problems”. In: *Journal of Scheduling* 23.1, pp. 3–47.

Gharoun, H., M. Hamid, and S. A. Torabi (2022). “An integrated approach to joint production planning and reliability-based multi-level preventive maintenance scheduling optimisation for a deteriorating system considering due-date satisfaction”. In: *International Journal of Systems Science: Operations & Logistics* 9.4, pp. 489–511.

Gillard, X. (2022). “Discrete Optimization with Decision Diagrams: Design of a Generic Solver, Improved Bounding Techniques, and Discovery of Good Feasible Solutions with Large Neighborhood Search”. PhD thesis. UCL-Université Catholique de Louvain.

Gillard, X., V. Coppé, P. Schaus, and A. A. Cire (2021). “Improving the filtering of branch-and-bound MDD solver”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 231–247.

Gillard, X. and P. Schaus (2022). “Large neighborhood search with decision diagrams.” In: *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI*, pp. 4754–4760.

Gillard, X., P. Schaus, and V. Coppé (2020). “Ddo, a generic and efficient framework for MDD-based optimization”. In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 5243–5245.

González, J. E., A. A. Cire, A. Lodi, and L.-M. Rousseau (2022). “BDD-based optimization for the quadratic stable set problem”. In: *Discrete Optimization* 44, p. 100610.

Graham, R. L., E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan (1979). “Optimization and approximation in deterministic sequencing and scheduling: A survey”. In: *Annals of Discrete Mathematics*. Vol. 5. Elsevier, pp. 287–326.

Graves, S. C., H. C. Meal, D. Stefek, and A. H. Zeghmi (1983). “Scheduling of re-entrant flow shops”. In: *Journal of Operations Management* 3.4, pp. 197–207.

Gupta, J. N. and E. F. Stafford Jr (2006). “Flowshop scheduling research after five decades”. In: *European Journal of Operational Research* 169.3, pp. 699–711.

Haus, U.-U., C. Michini, and M. Laumanns (2017). “Scenario aggregation using binary decision diagrams for stochastic programs with endogenous uncertainty”. In: *arXiv preprint arXiv:1701.04055*.

He, T. and M. Tawarmalani (2024). “MIP relaxations in factorable programming”. In: *SIAM Journal on Optimization* 34.3, pp. 2856–2882.

Hnaien, F., F. Yalaoui, A. Mhadhbi, and M. Nourelfath (2016). “A mixed-integer programming model for integrated production and maintenance”. In: *IFAC-PapersOnLine* 49.12, pp. 556–561.

Hochreiter, S. and J. Schmidhuber (1997). “Long short-term memory”. In: *Neural Computation* 9.8, pp. 1735–1780.

Hoda, S., W.-J. Van Hoeve, and J. Hooker (2010). “A systematic approach to MDD-based constraint programming”. In: *16th International Conference on Principles and Practice of Constraint Programming (CP 2010)*. Springer, pp. 266–280.

Hooker, J. (2017). "Job sequencing bounds from decision diagrams". In: *23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*. Springer, pp. 565–578.

– (2022). "Stochastic decision diagrams". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 138–154.

Horn, M., M. Djukanovic, C. Blum, and G. R. Raidl (2020). "On the use of decision diagrams for finding repetition-free longest common subsequences". In: *Optimization and Applications: 11th International Conference, OPTIMA 2020, Moscow, Russia, September 28–October 2, 2020, Proceedings 11*. Springer, pp. 134–149.

Horn, M., J. Maschler, G. R. Raidl, and E. Rönnberg (2021). "A*-based construction of decision diagrams for a prize-collecting scheduling problem". In: *Computers & Operations Research* 126, p. 105125.

Ibaraki, T. (1977). "The power of dominance relations in branch-and-bound algorithms". In: *Journal of the ACM (JACM)* 24.2, pp. 264–279.

Iklassov, Z., D. Medvedev, R. S. O. De Retana, and M. Takac (2023). "On the study of curriculum learning for inferring dispatching policies on the job shop scheduling". In: *Proceedings of the 32nd International Joint Conference on Artificial Intelligence, IJCAI*, pp. 5350–5358.

Jeong, B. and Y.-D. Kim (2014). "Minimizing total tardiness in a two-machine re-entrant flowshop with sequence-dependent setup times". In: *Computers & Operations Research* 47, pp. 72–80.

Jiang, C., J. Babar, G. Ciardo, A. S. Miner, and B. Smith (2017). "Variable reordering in binary decision diagrams". In: *26th International Workshop on Logic & Synthesis*.

Jin, L., X. Yu, and Z. Dong (2018). "Single-machine scheduling with piece-rate maintenance, interval constrained processing times and rejection penalties". In: *2018 3rd Joint International Information Technology, Mechanical and Electronic Engineering Conference (JIMEC 2018)*. Atlantis Press, pp. 32–37.

Johnson, D. S. and M. A. Trick (1996). *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993*. Vol. 26. American Mathematical Soc., pp. 1–10.

Jouglet, A. and J. Carlier (2011). "Dominance rules in combinatorial optimization problems". In: *European Journal of Operational Research* 212.3, pp. 433–444.

Kang, J. and C. Guedes Soares (2020). "An opportunistic maintenance policy for offshore wind farms". In: *Ocean Engineering* 216, p. 108075.

Karahalios, A. and W.-J. van Hoeve (2023). "Column elimination for capacitated vehicle routing problems". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 35–51.

Katragjini, K., E. Vallada, and R. Ruiz (2015). "Rescheduling flowshops under simultaneous disruptions". In: *6th IEEE International Conference on Industrial Engineering and Systems Management (IESM)*, pp. 84–91.

El-Khalil, R. and Z. Darwish (2019). "Flexible manufacturing systems performance in US automotive manufacturing plants: A case study". In: *Production Planning & Control* 30.1, pp. 48–59.

Kleywegt, A. J., A. Shapiro, and T. Homem-de-Mello (2002). "The sample average approximation method for stochastic discrete optimization". In: *SIAM Journal on Optimization* 12.2, pp. 479–502.

Kool, W., H. Van Hoof, and M. Welling (2018). "Attention, learn to solve routing problems!" In: *arXiv preprint arXiv:1803.08475*.

Kutanoglu, E. and İ. Sabuncuoglu (2001). "Routing-based reactive scheduling policies for machine failures in dynamic job shops". In: *International Journal of Production Research* 39.14, pp. 3141–3158.

Laborie, P., J. Rogerie, P. Shaw, and P. Vilím (2018). "IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG". In: *Constraints* 23, pp. 210–250.

Lai, Y.-T., M. Pedram, and S. B. Vrudhula (1994). "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.8, pp. 959–975.

Land, A. H. and A. G. Doig (2009). "An automatic method for solving discrete programming problems". In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, pp. 105–132.

Latour, A., B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, and S. Nijssen (2017). "Combining stochastic constraint optimization and probabilistic programming: from knowledge compilation to constraint solving". In: *23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*. Springer, pp. 495–511.

Latour, A., B. Babaki, and S. Nijssen (2019). "Stochastic constraint propagation for mining probabilistic networks." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 1137–1145.

Lee, C.-Y. (1959). "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4, pp. 985–999.

Lenstra, J. K. and A. R. Kan (1979). "Computational complexity of discrete optimization problems". In: *Annals of Discrete Mathematics*. Vol. 4. Elsevier, pp. 121–140.

Lin, F., X. Fang, and Z. Gao (2022). "Distributionally robust optimization: A review on theory and applications". In: *Numerical Algebra, Control and Optimization* 12.1, pp. 159–212.

Liu, B. and B. Liu (2010). *Uncertainty Theory*. Springer, pp. 1–125.

Liu, W., Y. Jin, and M. Price (2017). "A new improved NEH heuristic for permutation flowshop scheduling problems". In: *International Journal of Production Economics* 193, pp. 21–30.

Liu, W., X. Wang, L. Li, and P. Zhao (2022). "A maintenance activity scheduling with time-and-position dependent deteriorating effects". In: *Mathematical Biosciences and Engineering* 19.11, pp. 11756–11767.

Lozano, L. and C. Smith (2022). "A binary decision diagram based algorithm for solving a class of binary two-stage stochastic programs". In: *Mathematical Programming* 191, pp. 1–24.

Lu, L. and J. Yuan (2007). "The single machine batching problem with identical family setup times to minimize maximum lateness is strongly NP-hard". In: *European Journal of Operational Research* 177.2, pp. 1302–1309.

Lunardi, W. T., E. G. Birgin, P. Laborie, D. P. Ronconi, and H. Voos (2020). “Mixed integer linear programming and constraint programming models for the online printing shop scheduling problem”. In: *Computers & Operations Research* 123, p. 105020.

MacNeil, M. and M. Bodur (2024). “Leveraging decision diagrams to solve two-stage stochastic programs with binary recourse and logical linking constraints”. In: *European Journal of Operational Research* 315.1, pp. 228–241.

Matsumoto, K., K. Hatano, and E. Takimoto (2018). “Decision diagrams for solving a job scheduling problem under precedence constraints”. In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 5:1–5:12.

Michel, L. and W.-J. van Hoeve (2024). “CODD: A decision diagram-based solver for combinatorial optimization”. In: *Proceedings of the 27th European Conference on Artificial Intelligence, ECAI*. IOS Press, pp. 4240–4247.

Miller, B. L. and H. M. Wagner (1965). “Chance constrained programming with joint constraints”. In: *Operations Research* 13.6, pp. 930–945.

Mor, B. and G. Mosheiov (2012). “Scheduling a maintenance activity and due-window assignment based on common flow allowance”. In: *International Journal of Production Economics* 135.1, pp. 222–230.

Naderi, B., R. Ruiz, and V. Roshanaei (2023). “Mixed-integer programming vs. constraint programming for shop scheduling problems: new results and outlook”. In: *INFORMS Journal on Computing* 35 (4), pp. 817–843.

Nafar, M. and M. Römer (2024a). “Strengthening relaxed decision diagrams for maximum independent set problem: Novel variable ordering and merge heuristics”. In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 21:1–21:17.

– (2024b). “Using clustering to strengthen decision diagram bounds for discrete optimization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38, pp. 8082–8089.

Nawaz, M., E. E. Enscore Jr, and I. Ham (1983). “A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem”. In: *Omega* 11.1, pp. 91–95.

Netto, R. J. K., E. de Freitas Rocha Loures, E. A. P. Santos, and C. F. dos Santos (2023). “Joint industrial preventive maintenance and production scheduling: A systematic literature review”. In: *International Conference on Flexible Automation and Intelligent Manufacturing*. Springer, pp. 614–621.

Ng, C. T., T. E. Cheng, and J. Yuan (2002). “Strong NP-hardness of the single machine multi-operation jobs total completion time scheduling problem”. In: *Information Processing Letters* 82.4, pp. 187–191.

Ng, K. W., G.-L. Tian, and M.-L. Tang (2011). “Dirichlet and related distributions: Theory, methods and applications”. In: pp. 37–96.

Novak, A., P. Sucha, M. Novotny, R. Stec, and Z. Hanzalek (2022). “Scheduling jobs with normally distributed processing times on parallel machines”. In: *European Journal of Operational Research* 297.2, pp. 422–441.

O'Neil, R. J. and K. Hoffman (2019). "Decision diagrams for solving traveling salesman problems with pickup and delivery in real time". In: *Operations Research Letters* 47.3, pp. 197–201.

Oldham, K., J. Myland, J. Spanier, K. B. Oldham, J. C. Myland, and J. Spanier (2009). "The error function erf (x) and its complement erfc (x)". In: *An Atlas of Functions: with Equator, the Atlas Function Calculator*, pp. 405–415.

Ouelhadj, D. and S. Petrovic (2009). "A survey of dynamic scheduling in manufacturing systems". In: *Journal of Scheduling* 12.4, pp. 417–431.

Park, J., J. Chun, S. H. Kim, Y. Kim, and J. Park (2021). "Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning". In: *International Journal of Production Research* 59.11, pp. 3360–3377.

Parker, R. G. and R. L. Rardin (2014). *Discrete Optimization*. Elsevier, pp. 1–27.

Perez, G., S. Malalel, G. Glorian, V. Jung, A. Papadopoulos, M. Pelleau, W. Suijlen, J.-C. Régin, and A. Lallouet (2023). "Generalized confidence constraints". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 4, pp. 4078–4086.

Perez, G. and J.-C. Régin (2015). "Efficient operations on MDDs for building constraint programming models". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 374–380.

Pinedo, M. L. (2012). *Scheduling*. Vol. 29. Springer, pp. 1–66.

Pisinger, D. (2005). "Where are the hard knapsack problems?" In: *Computers & Operations Research* 32.9, pp. 2271–2284.

Ranjbar, M. and A. Kazemi (2018). "A generalized variable neighborhood search algorithm for the talent scheduling problem". In: *Computers & Industrial Engineering* 126, pp. 673–680.

Reijnen, R., Y. Zhang, Z. Bukhsh, and M. Guzek (2022). "Deep reinforcement learning for adaptive parameter control in differential evolution for multi-objective optimization". In: *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 804–811.

Reijnen, R., Y. Zhang, H. C. Lau, and Z. Bukhsh (2022). "Operator selection in adaptive large neighborhood search using deep reinforcement learning". In: *arXiv preprint arXiv:2211.00759*.

Reinelt, G. (1991). "TSPLIB—A traveling salesman problem library". In: *ORSA Journal on Computing* 3.4, pp. 376–384.

Ren, Z., A. S. Verma, Y. Li, J. J. Teuwen, and Z. Jiang (2021). "Offshore wind turbine operations and maintenance: A state-of-the-art review". In: *Renewable and Sustainable Energy Reviews* 144, p. 110886.

Römer, M., A. A. Cire, and L.-M. Rousseau (2018). "A local search framework for compiling relaxed decision diagrams". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 512–520.

Rossi, F., P. Van Beek, and T. Walsh (2006). *Handbook of Constraint Programming*. Elsevier, pp. 11–25.

Roy, B. and B. Sussmann (1964). "Les problemes d'ordonnancement avec contraintes disjonctives". In: *Note ds* 9.

Rudich, I., Q. Cappart, and L.-M. Rousseau (2023). “Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams”. In: *Journal of Artificial Intelligence Research* 77, pp. 1489–1538.

Ruiz, R. (2015). “Scheduling heuristics”. In: *Handbook of Heuristics*. Springer, pp. 1–24.

Ruiz-Torres, A. J., G. Paletta, and R. M’Hallah (2017). “Makespan minimisation with sequence-dependent machine deterioration and maintenance events”. In: *International Journal of Production Research* 55.2, pp. 462–479.

Ruiz-Torres, A. J., G. Paletta, and E. Pérez (2013). “Parallel machine scheduling to minimize the makespan with sequence dependent deteriorating effects”. In: *Computers & Operations Research* 40.8, pp. 2051–2061.

Santos, V. L. A. and J. E. C. Arroyo (2017). “Iterated greedy with random variable neighborhood descent for scheduling jobs on parallel machines with deterioration effect”. In: *Electronic Notes in Discrete Mathematics* 58, pp. 55–62.

Schmidl, C., T. D. Simão, and N. Jansen (2024). “A supervised learning approach to robust reinforcement learning for job shop scheduling”. In: *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024, Volume 3, Rome, Italy, February 24-26, 2024*. Ed. by A. P. Rocha, L. Steels, and H. J. van den Herik. SCITEPRESS, pp. 1324–1335.

Schmidt, G. (1996). “Modelling production scheduling systems”. In: *International Journal of Production Economics* 46, pp. 109–118.

Seo, D. K., C. M. Klein, and W. Jang (2005). “Single machine stochastic scheduling to minimize the expected number of tardy jobs using mathematical programming models”. In: *Computers & Industrial Engineering* 48.2, pp. 153–161.

Shobaki, G. and J. Jamal (2015). “An exact algorithm for the sequential ordering problem and its application to switching energy minimization in compilers”. In: *Computational Optimization and Applications* 61, pp. 343–372.

Shyu, S. J. and C.-Y. Tsai (2009). “Finding the longest common subsequence for multiple biological sequences by ant colony optimization”. In: *Computers & Operations Research* 36.1, pp. 73–91.

Smith, B. M. (2003). “Constraint programming in practice: Scheduling a rehearsal”. In: *Research Report APES-67-2003, APES Group*.

– (2005). “Caching search states in permutation problems”. In: *11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*. Springer, pp. 637–651.

Song, W., X. Chen, Q. Li, and Z. Cao (2022). “Flexible job-shop scheduling via graph neural network and deep reinforcement learning”. In: *IEEE Transactions on Industrial Informatics* 19.2, pp. 1600–1610.

Stützle, T. and R. Ruiz (2018). “Iterated greedy”. In: *Handbook of Heuristics*, pp. 547–577.

Su, Y.-C., F.-T. Cheng, M.-H. Hung, and H.-C. Huang (2006). “Intelligent prognostics system design and implementation”. In: *IEEE Transactions on Semiconductor Manufacturing* 19.2, pp. 195–207.

Susto, G. A., A. Schirru, S. Pampuri, S. McLoone, and A. Beghi (2014). “Machine learning for predictive maintenance: A multiple classifier approach”. In: *IEEE Transactions on Industrial Informatics* 11.3, pp. 812–820.

Taillard, E. (1993). “Benchmarks for basic scheduling problems”. In: *European Journal of Operational Research* 64.2, pp. 278–285.

Tjandraatmadja, C. and W.-J. van Hoeve (2021). “Incorporating bounds from decision diagrams into integer programming”. In: *Mathematical Programming Computation* 13.2, pp. 225–256.

Valet, A., T. Altenmüller, B. Waschneck, M. C. May, A. Kuhnle, and G. Lanza (2022). “Opportunistic maintenance scheduling with deep reinforcement learning”. In: *Journal of Manufacturing Systems* 64, pp. 518–534.

Van den Bogaerdt, P. and M. de Weerdt (2018). “Multi-machine scheduling lower bounds using decision diagrams”. In: *Operations Research Letters* 46.6, pp. 616–621.

– (2019). “Lower bounds for uniform machine scheduling using decision diagrams”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR*. Springer, pp. 565–580.

Van der Tempel, R., J. van Pinxten, M. Geilen, and U. Waqas (2018). “A heuristic for variable re-entrant scheduling problems”. In: *21st IEEE Euromicro Conference on Digital System Design (DSD)*, pp. 336–341.

Van Hoeve, W.-J. (2001). “The alldifferent constraint: A survey”. In: *arXiv preprint cs/0105015*.

van Pinxten, J., U. Waqas, M. Geilen, T. Basten, and L. Somers (2017). “Online scheduling of 2-re-entrant flexible manufacturing systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s, pp. 1–20.

Van Hoeve, W.-J. (2024). “An introduction to decision diagrams for optimization”. In: *Tutorials in Operations Research: Smarter Decisions for a Better World*. INFORMS, pp. 117–145.

Vieira, G. E., J. W. Herrmann, and E. Lin (2003). “Rescheduling manufacturing systems: a framework of strategies, policies, and methods”. In: *Journal of Scheduling* 6.1, pp. 39–62.

Vinyals, O., S. Bengio, and M. Kudlur (2015). “Order matters: Sequence to sequence for sets”. In: *arXiv preprint arXiv:1511.06391*.

Vu, T.-A., S. Afifi, E. Lefèvre, and F. Pichon (2025). “Optimization problems with uncertain objective coefficients using capacities”. In: *Annals of Operations Research* 344.1, pp. 383–412.

Wang, M. Y., S. P. Sethi, and S. L. van de Velde (1997). “Minimizing makespan in a class of reentrant shops”. In: *Operations Research* 45.5, pp. 702–712.

Wang, Y., E. Elahi, and L. Xu (2019). “Selective maintenance optimization modelling for multi-state deterioration systems considering imperfect maintenance”. In: *IEEE Access* 7, pp. 62759–62768.

Waqas, U., M. Geilen, J. Kandelaars, L. Somers, T. Basten, S. Stuijk, P. Vestjens, and H. Corporaal (2015). “A re-entrant flowshop heuristic for online scheduling of the paper path in a large scale printer”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 573–578.

Wiering, M. A. and M. Van Otterlo (2012). “Reinforcement learning”. In: *Adaptation, Learning, and Optimization* 12.3, p. 729.

Williams, H. P. (2013). *Model Building in Mathematical Programming*. John Wiley & Sons, pp. 172–177.

Williams, R. J. (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8, pp. 229–256.

Williams, R., C. P. Gomes, and B. Selman (2003). “Backdoors to typical case complexity”. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI*. Vol. 3, pp. 1173–1178.

Wilson, J. M. (2003). “Gantt charts: A centenary appreciation”. In: *European Journal of Operational Research* 149.2, pp. 430–437.

Xanthopoulos, A., A. Kiatipis, D. E. Koulouriotis, and S. Stieger (2017). “Reinforcement learning-based and parametric production-maintenance control policies for a deteriorating manufacturing system”. In: *IEEE Access* 6, pp. 576–588.

Yang, S.-J. (2010). “Single-machine scheduling problems with both start-time dependent learning and position dependent aging effects under deteriorating maintenance consideration”. In: *Applied Mathematics and Computation* 217.7, pp. 3321–3329.

– (2011). “Parallel machines scheduling with simultaneous considerations of position-dependent deterioration effects and maintenance activities”. In: *Journal of the Chinese Institute of Industrial Engineers* 28.4, pp. 270–280.

Zhang, C., W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi (2020). “Learning to dispatch for job shop scheduling via deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 33, pp. 1621–1632.

Zhang, N., K. Cai, Y. Deng, and J. Zhang (2023). “Determining the optimal production–maintenance policy of a parallel production system with stochastically interacted yield and deterioration”. In: *Reliability Engineering & System Safety* 237, p. 109342.

Zhu, H., M. Li, Z. Zhou, and Y. You (2016). “Due-window assignment and scheduling with general position-dependent processing times involving a deteriorating and compressible maintenance activity”. In: *International Journal of Production Research* 54.12, pp. 3475–3490.

A

Improving the Anytime Behaviour of the Bounded Heuristic Constraint Scheduler (BHCS)

In this appendix, we present an anytime modification to the Bounded Heuristic Constraint Scheduler. Via empirical evaluation, we show that our modification, A-BHCS, indeed performs in an anytime manner and produces better results than state-of-the-art methods for re-entrant scheduling.

A.1. Introduction

Apart from providing good quality schedules, a major concern in many scheduling applications is how long it takes to produce schedules especially in online scheduling scenarios where the scheduler must make decisions within some limited time as new jobs come in on the fly[1]. On the other hand, the response time of scheduling algorithms is often not fixed as such there is no guarantee of how long it will take to produce a schedule for every instance. *Anytime scheduling* provides a variable time scheduling paradigm where the scheduler can still provide a valid solution even if it is stopped before completion. The desired anytime behaviour is that the scheduler is able to improve the solution given more time. In this appendix, we modify the BHCS algorithm as introduced in Section 2.2.2 to perform in any anytime manner.

A.2. Method

Recall the BHCS algorithm (Algorithm 11). The worst-case time complexity of this algorithm is $O(|J|L^3r^3)$ (van Pinxten *et al.* 2017) where $|J|$ is the number of jobs present in the scheduling problem, L is the maximum number of sequencing options present for each operation to be scheduled and r is the maximum number of

[1]The time budget to make a scheduling decision can either be fixed or dependent on some parameters of the job to be scheduled.

operations to be scheduled for a job – some operations have fixed positions and do not need to be decided on by a scheduler. This complexity can be seen in Algorithm 11. In Line 5, we choose an operation to schedule according to a given order and the total available operations to schedule are $|J|r$. The effect of the maximum number of sequencing options $|L|$ is reflected in Line 6, where we generate sequencing options for each operation. In order to evaluate the quality and feasibility of a sequencing option, we run a longest path computation on the constraint graph. For a graph with nodes V and edges E, the longest path has a time complexity $O(|V||E|)$, however, we do not always have to compute the whole graph and can instead work on a sub-graph (van Pinxten *et al.* 2017). There are at most $|L|r$ nodes that we have to include in the sub-graph and as $|V| \approx |E|$ in many problems, the time complexity of updating a scheduling option is $O(|L|^2 r^2)$.

From this analysis, an opportunity to get quicker solutions in BHCS is to reduce the number of sequencing options considered as every feasible sequencing option can lead to a valid solution. Thus, we can modify the algorithm for an anytime solution as in Algorithm 12. In Line 5, we add the option to return the partial schedule at any point that we run out of time to decide on a position for the operation currently being considered. The rest of the algorithm remains the same.

A.3. Computational Results

In this short set of experiments, we compare our proposed changes to a known anytime algorithm for the class of problems considered by BHCS. We aim to first empirically verify the anytime behaviour of A-BHCS and subsequently quantify its

Algorithm 11 Bounded Heuristic Constraint Scheduler (BHCS)

```
1: function BHCS(scheduling problem f)                ▷ returns schedule Ω
2:     Ω ←<>                                          ▷ empty schedule
3:     Ω′ ← ∅                                         ▷ empty set of schedules
4:     for o_c in order do
5:         Ω′ ← generateOptions(o_c, f, Ω)
6:         Ω ← selectHighestRanked(Ω′, rank)
       return Ω
```

Algorithm 12 Anytime Bounded Heuristic Constraint Scheduler (A-BHCS)

```
1: function BHCS(scheduling problem f)                ▷ returns schedule Ω
2:     Ω ←<>                                          ▷ empty schedule
3:     for o_c in order do
4:         Φ ← getAllOptions(o_c, f, Ω)
5:         while ≠ timeout ∧ Φ! = ∅ do
6:             Ω′ ← getNextOption(Φ)
7:             if Quality(Ω′) > Quality(Ω) then
8:                 Ω ← Ω′
9:         Φ ← Φ \ Ω′
       return Ω
```

Figure A.1.: Anytime performance of A-BHCS and Iterated Greedy.

relative performance to other known algorithms. We use the same experimental setup as in Section 3.7.

We confirm that A-BHCS works such that increasing the runtime leads to improvements in the solution. We note that for the size of jobsets considered, the algorithms are already able to consider all the scheduling options at $5ms$ leading to a flat curve up to $20ms$. It is also important to note that the anytime version of both algorithms is still greedy with no guarantee of reaching optimal in any length of time.

Compared with Iterated Greedy (Stützle and Ruiz 2018), we see that A-BHCS produces better solutions across all time-out values. This is due to the specialised nature of A-BHCS to the problem at hand.

B

Problem Classes

In this appendix, we provide more details of the models and data sources used for our experiments in Part II of this dissertation. For each problem, we define the dynamic programming model and the merge operator used.

The main ingredient in dynamic programming (DP) is the reformulation of the optimisation problem in terms of states. A DP model is solved in stages where each stage contains a set of states and there are at least as many stages as there are decision variables.

The DP model is made up of the following elements:

- a set of decision variables $X = \{X_1, X_2, \ldots, X_n\}$ each with domains in $D = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$. A sequence of decisions is the vector $x = (x_1, x_2, \ldots, x_n)$ with $x_i \in \mathcal{D}_i$.
- a state space S partitioned into $n+1$ stages where each stage S_i contains a set of states in the i^{th} stage of the DP model and S_0 contains only the root or initial state $\hat{r}$. We also define an infeasible state $\hat{0}$.
- a transition function $\tau : S_i \times \mathcal{D}_i \to S_{i+1}$
- a cost function $h : S_i \times S_{i+1} \times \mathcal{D}_i \to \mathbb{R}$

The DP formulation can then be written as

$$\min_x f(x) = \sum_{i=0}^{n-1} h(s^i, s^{i+1}, x_i) \tag{B.1}$$

$$\text{subject to} \quad s^{i+1} = \tau(s^i, x_i) \forall x_i \in \mathcal{D}_i \tag{B.2}$$

$$s^i \in S_i \tag{B.3}$$

$$s^i \neq \hat{0} \tag{B.4}$$

The merge operator $\oplus$ is structured such that given two states s and s', $s'' = s \oplus s'$ is a relaxation of both s and s'. A state s'' relaxes s if

– (C1) All feasible transitions in s are also feasible in s''

– (C2) The cost of any transition from s'' is smaller or equal to the cost of the same transition in s,

– (C3) If s'' and s transition to states t'' and t respectively by taking the same decision x_j, t'' is also a relaxation of t.

Each problem has a state representation which is a tuple of problem specific properties. We duplicate some state properties to keep track of the exact value of the property and the relaxed value. We denote this duplication with a $'$ symbol. For instance, the set of already scheduled operations at a state can be represented as Γ and Γ' where Γ is the set of operations that have definitely been scheduled and Γ' is the set of operations that may have been scheduled.

B.1. Talent Scheduling Problem (TSCHED)

Given a set of scenes S to be shot on a movie production set with a set of actors A where each scene s_i has a duration d_i and requires a subset of actors $Q_i \subseteq A$, the goal is to order the scenes such that the total cost of actors is minimised bearing in mind that an actor is to be paid for the entire duration from the first to the last scene they participate in even if they are sometimes idle in between.

Dynamic programming model The state representation is $s = (\mathcal{Q}, \mathcal{Q}', \mathcal{A})$ where $\mathcal{Q}$ is the set of scenes that definitely still need to be scheduled, $\mathcal{Q}'$ is the set of scenes that may still need to be scheduled (i.e., the relaxation of $\mathcal{Q}$) and, $\mathcal{A}$ is the set of actors on the set. Each decision variable x_i has a domain Q as the choice of a transition is always which scene to schedule next. We define some helper functions namely $actors(Q)$ which returns the set of actors appearing in a set of scenes Q, $cost(a)$ which returns the payment rate of an actor a, and $duration(q)$ which returns the duration of a scene q.

The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, q) = (\mathcal{Q}_{i+1}, \mathcal{Q}'_{i+1}, \mathcal{A}_{i+1}) \tag{B.5}$$

$$\text{such that} \quad \mathcal{Q}_{i+1} = \mathcal{Q}_i \setminus \{q\} \tag{B.6}$$

$$\mathcal{Q}'_{i+1} = \mathcal{Q}'_i \setminus \{q\} \tag{B.7}$$

$$\mathcal{A}_{i+1} = (\mathcal{A} \cap actors(\mathcal{Q}' \setminus \{q\})) \cup actors(\{q\}) \tag{B.8}$$

$$h(s_i, s_{i+1}, q) = \sum_{a \in \mathcal{A}_{i+1}} cost(a) \cdot duration(q). \tag{B.9}$$

Merge Operator The merge operator is defined such that given $s = (\mathcal{Q}, \mathcal{Q}', \mathcal{A})$, and $\hat{s} = (\hat{\mathcal{Q}}, \hat{\mathcal{Q}}', \hat{\mathcal{A}})$,

$$s \oplus \hat{s} = \hat{\hat{s}} = (\hat{\hat{\mathcal{Q}}}, \hat{\hat{\mathcal{Q}}}', \hat{\hat{\mathcal{A}}}) \tag{B.10}$$

$$\text{with,} \hat{\hat{\mathcal{Q}}} = \mathcal{Q} \cap \hat{\mathcal{Q}} \tag{B.11}$$

$$\hat{\hat{\mathcal{Q}}}' = \mathcal{Q}' \cup \hat{\mathcal{Q}}' \tag{B.12}$$

$$\hat{\hat{\mathcal{A}}} = \mathcal{A} \cap \hat{\mathcal{A}} \tag{B.13}$$

B.2. Sequential Ordering Problem (SOP)

Given a weighted graph $G = (V, E)$ with vertices V and edges E, the SOP aims to find a path with minimum total cost given a set of precedence constraints between the vertices.

Dynamic programming model The state representation is $s = (\mathcal{J}, \mathcal{X}, \mathcal{X}')$ where $\mathcal{J}$ is the set of last sequenced vertices (when the state is not relaxed, $\mathcal{J}$ has a cardinality of 1), $\mathcal{X}$ is the set of vertices that definitely still need to be placed, and $\mathcal{X}'$ is the set of vertices that may still need to be placed. Each decision variable X_j has a domain V as the choice of a transition is always which vertex to visit next. We define a helper function $distance(u, v)$ which returns the distance between two vertices u and v. The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, v) = (\mathcal{J}_{i+1}, \mathcal{X}_{i+1}, \mathcal{X}'_{i+1}) \tag{B.14}$$

$$\text{such that} \quad \mathcal{X}_{i+1} = \mathcal{X}_i \setminus \{v\} \tag{B.15}$$

$$\mathcal{X}'_{i+1} = \mathcal{X}'_i \setminus \{v\} \tag{B.16}$$

$$\mathcal{J}_{i+1} = \{v\} \tag{B.17}$$

$$h(s_i, s_{i+1}, v) = \min_{u \in I_i}(distance(u, v)). \tag{B.18}$$

Merge Operator The merge operator is defined such that given $s = (\mathcal{J}, \mathcal{X}, \mathcal{X}')$, and $\hat{s} = (\hat{\mathcal{J}}, \hat{\mathcal{X}}, \hat{\mathcal{X}}')$

$$s \oplus \hat{s} = \hat{\hat{s}} = (\hat{\hat{\mathcal{J}}}, \hat{\hat{\mathcal{X}}}, \hat{\hat{\mathcal{X}}}') \tag{B.19}$$

$$\text{with,} \hat{\hat{\mathcal{X}}} = \mathcal{X} \cap \hat{\mathcal{X}} \tag{B.20}$$

$$\hat{\hat{\mathcal{X}}}' = \mathcal{X}' \cup \hat{\mathcal{X}}' \tag{B.21}$$

$$\hat{\hat{\mathcal{J}}} = \mathcal{J} \cup \hat{\mathcal{J}} \tag{B.22}$$

B.3. Maximum Independent Set Problem (MISP)

Given a graph $G = (V, E)$ with set of vertices V and a set of edges E, the MISP aims to find the maximum weighted subset $X \subseteq V$ such that no two vertices in X share an

edge. The problem is weighted such that each vertex v_i is assigned a weight w_i and the goal is then to select the independent subset that maximizes the total weight.

Dynamic programming model The state representation is $s = (\mathcal{V}, \mathcal{V}')$ where $\mathcal{V}$ is the set of vertices definitely still available to be chosen that and, $\mathcal{V}'$ is the set of vertices that may still be available (i.e., the relaxation of $\mathcal{V}$). Each decision variable X_j has a domain $\{0,1\}$ as the choice of picking vertex v_j as a transition is always whether or not to add a vertex to the independent set. We define two helper functions $neighbours(v)$, which returns all the vertices with which v shares an edge, and $weight(v)$, which returns the weight of v. The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, x_j) = (\mathcal{V}_{i+1}, \mathcal{V}'_{i+1}) \tag{B.23}$$

$$\text{such that} \quad \mathcal{V}_{i+1} = \begin{cases} \mathcal{V}_i \setminus \{\{v_j\} \cup neighbours(v_j)\} & \text{if } x_j = 1 \\ \mathcal{V}_i & \text{otherwise} \end{cases} \tag{B.24}$$

$$\mathcal{V}'_{i+1} = \begin{cases} \mathcal{V}'_i \setminus \{\{v_j\} \cup neighbours(v_j)\} & \text{if } x_j = 1 \\ \mathcal{V}'_i & \text{otherwise} \end{cases} \tag{B.25}$$

$$h(s_i, s_{i+1}, x_j) = x_j \cdot weight(j). \tag{B.26}$$

Merge Operator The merge operator is defined such that given $s = (\mathcal{V}, \mathcal{V}')$, and $\hat{s} = (\hat{\mathcal{V}}, \hat{\mathcal{V}}')$

$$s \oplus \hat{s} = \hat{\hat{s}} = (\hat{\hat{\mathcal{V}}}, \hat{\hat{\mathcal{V}}}') \tag{B.27}$$

$$\text{with,} \hat{\hat{\mathcal{V}}} = \mathcal{V} \cap \hat{\mathcal{V}} \tag{B.28}$$

$$\hat{\hat{\mathcal{V}}}' = \mathcal{V}' \cup \hat{\mathcal{V}}' \tag{B.29}$$

B.4. Knapsack Problem (KNAPSACK)

Given a knapsack with capacity C and a set Y of N items each with weight w_i and profit p_i, the aim is fill the knapsack with a subset of $X \subseteq Y$ items such that the total profit is maximized without exceeding the capacity.

Dynamic programming model The state representation is $s = \mathcal{K}$ where $\mathcal{K}$ is the remaining capacity of the knapsack. Each decision variable X_j has a domain $\{0,1\}$ as the choice of picking the j^{th} item as a transition is always whether or not to place a particular item in the knapsack. Functions $weight(j)$ and $profit(j)$ return the weight and profit of the j^{th} item respectively. The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, x_j) = \mathcal{K}_{i+1} \tag{B.30}$$

$$\text{such that} \quad \mathcal{K}_{i+1} = \mathcal{K}_i - (x_j \cdot weight(j)) \tag{B.31}$$

$$h(s_i, s_{i+1}, x_j) = x_j \cdot profit(j). \tag{B.32}$$

Merge Operator The merge operator is defined such that given $s = \mathcal{K}$, and $\hat{s} = \hat{\mathcal{K}}$

$$s \oplus \hat{s} = \hat{\hat{s}} = \hat{\hat{\mathcal{K}}} \tag{B.33}$$

$$\text{with,} \hat{\hat{\mathcal{K}}} = \max(\mathcal{K}, \hat{\mathcal{K}}) \tag{B.34}$$

B.5. Aircraft Landing Problem (ALP)

Given a set of aircraft N and a set of runways R, where N is partitioned into classes such that there is a minimum separation time required between landing aircraft in different classes; the goal of the ALP is to schedule the landing of aircraft on runways such that the total delay is minimized while respecting earliest and latest landing time constraints.

Dynamic programming model The state representation is $s = (\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ a tuple of three sets: $\mathcal{X}$, which holds the number of unscheduled aircraft per class and $\mathcal{Y}$, which holds the earliest possible occupation times per runway and, $\mathcal{Z}$ which holds the classes of airplanes possibly scheduled last per runway. Each decision variable X_j has a domain $N \times R$ as the choice of a transition is always which aircraft to place on which runway next. Therefore, a decision x_{uv} places aircraft n_u on runway r_v.

We define the following helper functions: (i) $class(n)$, which returns the class of an aircraft n, (ii) $separation(c_a, c_b)$, which returns the minimum separation between 2 aircraft classes, and (iii) $arrival(n)$ which returns the earliest possible landing time of aircraft n. We also refer to the set of all aircraft classes as C and the set of all runways as R.

The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, x_{uv}) = (\mathcal{X}_{i+1}, \mathcal{Y}_{i+1}, \mathcal{Z}_{i+1}) \tag{B.35}$$

$$\text{such that} \quad \mathcal{X}_{i+1}^{class(n_u)} = \mathcal{X}_i^{class(n_u)} - 1 \tag{B.36}$$

$$\mathcal{Y}_{i+1}^{r_v} = \max(\mathcal{Y}_i^{r_v}, \tag{B.37}$$

$$arrival(n_u) + \min_{x \in \mathcal{Z}_i^{r_v}} (separation(x, class(n_u)))) \tag{B.38}$$

$$\mathcal{Z}_{i+1}^{r_v} = \{class(n_u)\} \tag{B.39}$$

$$h(s_i, s_{i+1}, x_{uv}) = \mathcal{Y}_{i+1}^{r_v} - arrival(n_u). \tag{B.40}$$

Merge Operator The merge operator is defined such that given $s = (\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, and $\hat{s} = (\hat{\mathcal{X}}, \hat{\mathcal{Y}}, \hat{\mathcal{Z}})$

$$s \oplus \hat{s} = \hat{\hat{s}} = (\hat{\hat{\mathcal{X}}}, \hat{\hat{\mathcal{Y}}}, \hat{\hat{\mathcal{Z}}}) \tag{B.41}$$

$$\text{with,} \hat{\hat{\mathcal{X}}} = \{x \in \mathbb{R} | \exists j \in \{1,2,\ldots,|C|\} : x = \min(\mathcal{X}^j, \hat{\mathcal{X}}^j)\} \tag{B.42}$$

$$\hat{\hat{\mathcal{Y}}} = \{x \in \mathbb{R} | \exists j \in \{1,2,\ldots,|R|\} : x = \min(\mathcal{Y}^j, \hat{\mathcal{Y}}^j)\} \tag{B.43}$$

$$\hat{\hat{\mathcal{Z}}} = \{x \in \mathbb{U} | \exists j \in \{1,2,\ldots,|R|\} : x = \{\mathcal{Z}^j \cup \hat{\mathcal{Z}}^j\}\} \tag{B.44}$$

B.6. Longest Common Subsequence Problem (LCS)

Given a set of M strings, the goal is to find the longest string that is a common subsequence of all input strings over an alphabet σ.

Dynamic programming model The DD is constructed such that every path is a string. As we require a sub-sequence of all the strings in M, we also keep track of the how far we are into any of the strings in M. Thus, we represent a state as a position vector $\mathcal{P}$ such that the i^{th} element in $\mathcal{P}$ points to the index of string $m_i \in M$ at that state. The index can also be a $dummy$ position which signifies we have come to the end of a string. Each decision variable X_j is the next character being added to the subsequence where the domain of X_j is the alphabet σ. The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, x_j) = \mathcal{P}_{i+1} \tag{B.45}$$

$$\text{such that} \quad \mathcal{P}_{i+1} = (x \in \mathbb{N} | \exists j \in \{1,2,\ldots,|M|\} : x = next\_character\_position(\mathcal{P}_i^j)) \tag{B.46}$$

$$h(s_i, s_{i+1}, x_j) = \begin{cases} 1 & \text{if} \quad \mathcal{P}_{i+1} \neq dummy \forall i \in |M| \\ 0 & \text{otherwise} \end{cases} \tag{B.47}$$

Merge Operator The merge operator is defined such that given $s = \mathcal{P}$, and $\hat{s} = \hat{\mathcal{P}}$,

$$s \oplus \hat{s} = \hat{\hat{s}} = \hat{\hat{\mathcal{P}}} \tag{B.48}$$

$$\text{with,} \hat{\hat{\mathcal{P}}} = (x \in \mathbb{N} | \exists j \in \{1,2,\ldots,|M|\} : x = \min(\mathcal{P}^j, \hat{\mathcal{P}}^j)) \tag{B.49}$$

$$\tag{B.50}$$

B.7. Travelling Salesperson with Time Windows Problem (TSPTW)

Given a set of cities/positions represented as a graph $G = (V, E)$ where V is a set of vertices representing the cities and E is the set of edges with weights representing the cost of travelling between cities, the travelling salesman problem aims to find a minimum cost cycle such that each vertex of the graph is reached within a specified time window.

Dynamic programming model The state representation is $s = (p, t, \mathcal{V}, \mathcal{V}')$ where p is the set of possible positions the salesman is at (when the state is not relaxed, p has a cardinality of 1), $t = (k, l)$ is the interval of the minimum and maximum elapsed time since the salesman begun (when the state is not relaxed, $k = l$), $\mathcal{V}$ is the set of vertices definitely left to visit, and $\mathcal{V}'$ is the set of vertices that may still be left to visit. Each decision variable X_j has a domain V as the choice of a transition is always which vertex to visit next. The transition and cost functions are as follows:

$$s^{i+1} = \tau(s^i, v) = (p_{i+1}, t_{i+1}, \mathcal{V}_{i+1}, \mathcal{V}'_{i+1}) \tag{B.51}$$

$$\text{such that} \quad \mathcal{V}_{i+1} = \mathcal{V}_i \setminus \{v\} \tag{B.52}$$

$$\mathcal{V}'_{i+1} = \mathcal{V}'_i \setminus \{v\} \tag{B.53}$$

$$p_{i+1} = \{v\} \tag{B.54}$$

$$t_{i+1} = (\min_{u \in p_i}(distance(u, v)), \max_{u \in p_i}(distance(u, v))) \tag{B.55}$$

$$h(s_i, s_{i+1}, v) = \min_{u \in p_i}(distance(u, v)). \tag{B.56}$$

Merge Operator The merge operator is defined such that given $s = (p, t, \mathcal{V}, \mathcal{V}')$, and $\hat{s} = (\hat{p}, \hat{t}, \hat{\mathcal{V}}, \hat{\mathcal{V}}')$,

$$s \oplus \hat{s} = \hat{\hat{s}} = (\hat{\hat{p}}, \hat{\hat{t}}, \hat{\hat{\mathcal{V}}}, \hat{\hat{\mathcal{V}}}') \tag{B.57}$$

$$\text{with,} \hat{\hat{\mathcal{V}}} = \mathcal{V} \cap \hat{\mathcal{V}} \tag{B.58}$$

$$\hat{\hat{\mathcal{V}}}' = \mathcal{V}' \cup \hat{\mathcal{V}}' \tag{B.59}$$

$$\hat{\hat{p}} = p \cup \hat{p} \tag{B.60}$$

$$\hat{\hat{t}} = t \cup \hat{t} \tag{B.61}$$

C

Models

In this appendix, we present the CP and MILP models used in our computational experiments in Chapter 5.

C.1. Mixed Integer Programming Models

In this section we present a Mixed Inter Programming (MIP) model for the problems discussed in Chapter 5 (Section 5.8). The model is inspired by existing models for similar problems discussed in (Naderi, Ruiz, and Roshanaei 2023) among others. The model retains all variables defined in Section 5.2. Indices of variables corresponding to operations are either of the form x_{ij} when both the job and operation identifier are important or of the form x_a when it is only necessary to differentiate one operation from the other. Furthermore, a dummy operation o_{d} with processing time 0 is defined and constrained to be the first operation on each machine. The following additional variables are developed for the MIP model: ω_{ij} refers to the start time of operation $o_{ij} \in O$, B_{ab} is a binary variable relating to the precedence constraints between operations o_a and o_b. Note that B_{ab} refers only to direct precedence and not the general notion of o_a being scheduled sometime before o_b. For ease of modelling, we also define binary variables $X_{am} \quad \forall o_a \in O, \mu_m \in M$ to represent machine assignment, where X_{am} is set to 1 if operation o_a is assigned to machine μ_m. These are not decision variables and are part of the problem description.

$$\min(C_{\max}) \tag{C.1}$$

The objective of the model is to minimise makespan denoted by Equation 3.1. The constraints follow below. Basic constraints that exist for all the problem classes discussed in Chapter 5 are discussed first while other more specific constraints that only apply to a subset of the use cases follow after.

Basic Constraints Equation C.2a enforces setup time constraints between operations of the same job while also ensuring processing time separation is obeyed. Equation (C.2b) to Equation (C.2f) all serve to establish a precedence relationship between operations and ensure no-overlap of operations mapped to the same

machine. Particularly, Equation C.2b enforces that the dummy operation has no predecessors. Equation C.2c ensures that every operation has exactly one predecessor and Equation C.2d enforces that every operation has at most one successor. Equation C.2e enforces that operations only follow each other if they are mapped to the same machine and Equation C.2f enforces that there is no overlap between operations leaving room for maintenance operations. Finally, Equation C.2g calculates the makespan.

$$\omega_{i(j+1)} \geq \omega_{ij} + p(o_{ij}) + \mathcal{S}(o_{ij}, o_{i(j+1)}) \quad \forall o_{ij} \in O \tag{C.2a}$$

$$\sum_{o_a \in O} B_{ad} = 0 \tag{C.2b}$$

$$\sum_{o_a \in O \cup \{o_d\}} B_{ab} = 1 \quad \forall o_b \in O \tag{C.2c}$$

$$\sum_{o_b \in O} B_{ab} \leq 1 \quad \forall o_a \in O \tag{C.2d}$$

$$B_{ab} \leq \sum_{\mu_m \in \mu} X_{am} X_{bm} \quad \forall o_a, o_b \in O \tag{C.2e}$$

$$\omega_b \geq B_{ab}(\omega_a + p(o_a) + \mathcal{S}(o_a, o_b)) \quad \forall o_a, o_b \in O \tag{C.2f}$$

$$C_{\max} = \max_{o_a \in O}\{(\omega_a + p(o_a))\} \tag{C.2g}$$

Maximum separation constraints Equation C.3a enforces the maximum separation constraints between operations of the same job by directly constraining the start times of operations, where $T(o_a, o_b)$ represents the maximum separation time required between two operations o_a and o_b.

$$\omega_{i(j+1)} \leq \omega_{ij} + T(o_{ij}, o_{i(j+1)}) \quad \forall o_{ij} \in O \tag{C.3a}$$

Fixed order constraints Equation C.4a enforces the fixed order relationship between operations at the same level. The constraint works based on the assumption that the jobs are indexed in the order required.

$$\omega_{(i+1)j} \geq \omega_{ij} + p(o_{ij}) + \mathcal{S}(o_{ij}, o_{(i+1)j}) \quad \forall o_{ij} \in O \tag{C.4a}$$

No-overtaking constraints Here, we define the additional constraints that make the problem a permutation flow shop. In order to ensure no-overtaking constraints, we use the concept of job precedence denoted by the binary variable Γ_{ik} that defines the precedence between jobs j_i and j_k. Similar to the operation precedence constraints, we also define a dummy job j_d that is constrained to be the first job. Equation C.5a constrains the dummy job to have no predecessors, Equation C.5b constrains the dummy job to have exactly one successor. Equations C.5c and C.5d constrain all the jobs to have at most one successor and exactly one predecessor respectively. Finally

C.5e constrains relative precedence of all operations to obey the job precedence.

$$\sum_{j_i \in J} \Gamma_{id} = 0 \tag{C.5a}$$

$$\sum_{j_i \in J} \Gamma_{di} = 1 \tag{C.5b}$$

$$\sum_{j_j \in J} \Gamma_{ij} \leq 1 \quad \forall j_i \in J \tag{C.5c}$$

$$\sum_{j_j \in J} \Gamma_{ij} + \Gamma_{dj} = 1 \quad \forall j_i \in J \tag{C.5d}$$

$$\omega_{ij} \leq \Gamma_{iu}(\omega_{uj}) \quad \forall o_{ij}, o_{uj} \in O \tag{C.5e}$$

C.2. Constraint Programming Models

State-of-the-art constraint programming solvers provide us some scheduling specific modelling concepts (Laborie *et al.* 2018). Two such concepts used in our models are *interval variables* and *sequence variables*. Interval variables refer to an interval of time where an activity is carried out. They are analogous to operations in flow shop scheduling problems. The solution provided by a solver assigns start times to interval variables. Sequence variables on the other hand, represent orderings of interval variables. Sequence variables are declared as sets of interval variables and solutions provided by a solver include a sequencing of these interval variables. As interval variables, operations retain the representation of o_a. Sequence variables are defined per machine and referenced as $Sequence_m$ for corresponding machine μ_m where $Sequence_m$ contains all operations mapped to μ_m. Additionally, the sets P and S are collections of all processing and setup times respectively.

Equation C.6 defines the makespan minimisation objective. Like the MIP model in C.1, we also split our constraints into basic constraints that apply to all problems and additional constraints that only apply to specific problem classes.

$$\min(C_{\max}) \tag{C.6}$$

Basic Constraints Constraint C1 enforces the sequencing of operations in a job taking note of any setup times. Constraint C1 only applies to operations of the same job as is seen with the index of operations in the constraint. Sequence-dependent setup times and no overlap constraints are handled by Constraint C2, which ensures that both the separations required by processing times and sequence-dependent setup times are obeyed. The noOverlap constraint works such that the separation denoted by sequence-dependent setup times applies only between direct successors. We should state that there is a version of this constraint, noOverlapIndirect, which applies sequence-dependent setup time constraints to all successors. However, this is not in any of the use cases considered. Finally, C3 calculates the makespan as the

maximum finishing time of any operation.

$$C1 : \mathsf{startOf}(o_{i(j+1)}) \geq \mathsf{startOf}(o_{ij}) + p(o_{ij}) + \mathcal{S}(o_{ij}, o_{i(j+1)}) \quad \forall O_{ij} \in O \tag{C1}$$

$$C2 : \mathsf{noOverlap}(Sequence_m, P, S) \quad \forall \mu_m \in \mu \tag{C2}$$

$$C3 : C_{max} = \max_{o_{ij} \in O} \{\mathsf{endOf}(o_{ij})\} \tag{C3}$$

Maximum separation constraints Constraint C5 enforces the maximum separation constraints, which only apply to operations of the same job. Like in Appendix C.1 above, $T(o_a, o_b)$ represents the maximum separation time required between two operations o_a and o_b.

$$C5 : \mathsf{startOf}(o_{i(j+1)}) \leq \mathsf{startOf}(o_{ij}) + T(o_{ij}, o_{i(j+1)}) \quad \forall o_{ij} \in O \tag{C5}$$

Fixed order constraints Constraint C7 enforces an ordering between the first operations of each job. The before constraint defined over two operations, mandates one operation to precede the other in a sequence. Constraint C8 builds on C7 to then enforce that this ordering is respected across all other sequences using the sameSubsequence constraint.

$$C7 : \mathsf{before}(Sequence_1, o_{i1}, o_{(i+1)1}) \quad \forall o_{i1} \in O \tag{C7}$$

$$C8 : \mathsf{sameSubsequence}(Sequence_1, Sequence_m) \quad \forall \mu_m \in \mu \tag{C8}$$

No-overtaking constraints In this case, there is no fixed order prescribed but any order chosen is expected to be the same across all machines. Thus, Constraint C9 works exactly the same way as Constraint C8.

$$C9 : \mathsf{sameSubsequence}(Sequence_1, Sequence_m) \quad \forall \mu_m \in \mu \tag{C9}$$

Acknowledgements

Moyosore Orimoloye in the first volume of his Birthday Notes tells us that friendship cannot be had on the cheap and only he who wrestles his friend can hug him. It is my hope that my every tussle with all who are listed below has resulted in an eternal embrace.

Thank you to my supervisory team; Dr. Neil Yorke-Smith, Prof. dr. ir. Bart De Schutter, and Dr. Mitra Nasri.

Thank you to all the members of the Algorithmics Research Group with a special note for Noah Schutte, Kim van den Houten, Koos van der Linden, and Daniël Vos.

Thank you to my very best friends; Maxine Ankora, Rilwan Shittu, Dr. Chika Eneh, Oyinkansola Samuel, Selam Asefa, Hassan Yahaya Taiwo, Bezawit Zerayacob, Maimuna Bala Shehu, Tihitina Teshome, Daara Awe, Marius Faiß, Maia Rigot, and Mylène Brown-Coleman.

Thank you to fellow members of the SAM-FMS project; Prof. dr. ir. Twan Basten, Joan Marcè i Igual, and Christoph Schmidl.

And thank you to Akomu Eigbe; the one person who has, true to her name, shared the burden of my life with me everyday since I was born.

Curriculum Vitæ

Eghonghon-Aye Eigbe

25-09-1995 Born in Benin City, Nigeria.

Education

2011 – 2016 Bachelor of Science in Electrical and Electronics Engineering
University of Lagos, Akoka

2018 – 2020 Master of Science in Embedded Systems
Delft University of Technology, Delft

2020 – 2025 Doctor of Philosophy in Computer Science
Delft University of Technology, Delft
Thesis: Optimising Discrete Problems; Decision Diagrams and Context-Aware Heuristics
Promotors: Prof. dr. ir. B. De Schutter & Dr. N. Yorke-Smith

List of Publications

3. E.-a. Eigbe, C. Schmidl, B. De Schutter, N. Jansen, M. Nasri, and N. Yorke-Smith (2024). "Neural decision diagrams". Presented at: *The 6th Data Science Meets Optimisation Workshop (DSO) co-located with the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*

2. E.-a. Eigbe, B. De Schutter, M. Nasri, and N. Yorke-Smith (2023). "Sequence-and time-dependent maintenance scheduling in twice re-entrant flow shops". In: *IEEE Access* 11, pp. 103461–103475. DOI: 10.1109/ACCESS.2023.3317533

1. E.-a. Eigbe, B. De Schutter, M. Nasri, and N. Yorke-Smith (2022). "Predictive maintenance scheduling in twice re-entrant flow shops with relative due dates". Presented at: *The 15th Workshop on Scheduling and Planning Applications (SPARK) co-located with the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*