Decision Diagrams and Large Neighbourhood Search for Earliness Tardiness Single Machine Scheduling with Sequence Dependent Setups

by

Victor Lo

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Graduate Department of Mechanical and Industrial Engineering
University of Toronto

Decision Diagrams and Large Neighbourhood Search for Earliness Tardiness Single Machine Scheduling with Sequence Dependent Setups

Victor Lo
Master of Applied Science
Graduate Department of Mechanical and Industrial Engineering
University of Toronto
2022

Abstract

Industrial manufacturing environments often penalize job earliness and tardiness or the sum of setup times, yet little work has addressed the combination of these complexities. This thesis formalizes and investigates the Earliness Tardiness Scheduling with Setups (ETSS) problem, which is a single machine scheduling problem minimizing weighted earliness/tardiness and setup costs. We develop several solution methodologies utilizing the following optimization techniques: Integer Programming (MIP), Constraint Programming (CP), Decisions Diagrams (DD), and Large Neighbourhood Search (LNS). Our computational studies demonstrate scalability issues with the complex objective function and find that LNS is the best technique developed, suggesting that LNS is an appropriate method for solving the ETSS problem.

Acknowledgments

First and foremost, I want to express my gratitude to my supervisor, Professor J. Christopher Beck, for his invaluable support, encouragement, and guidance throughout the past years. His dedication and thoughtfulness pushed me to always strive for better and really taught me to think critically.

I would also like to thank my committee member, Professor Andre Cire, for providing invaluable feedback on my thesis. Thank you to our industry client, Visual8, for providing the data used in this thesis and for their ongoing support.

Thank you, Professor Merve Bodur, for really challenging me in your courses and offering your support when I struggled. Thank you to all of the TIDEL members for being a family away from home. To Eldan, Kyle, and Margarita, thank you for being incredibly welcoming and helpful with getting me situated in a new environment. I appreciate the technical help, especially when all else seemed to fail, and our casual banter. To Jason, thank you for always offering a much-needed laugh and for being a great course partner. Lastly, I want to thank Alex, Anton, Arik, Arnoosh, Evghenii, Giovanni, Jasper, Litong, Louis, Luke, Minori, Ryo, and Tanya. I wish you all the best in your endeavors.

A most sincere thank you to my family and friends. To my parents and sister, for their ongoing love, support, and encouragement. To Basil, Catherine, Douglas, Graham, Keyur, Linda, Liz, and Sasha, for keeping me positive and offering a fresh perspective. For crazy adventures and balancing the academics with the non-academics.

Finally, I would like to thank my students for showing me the joys of teaching and reigniting my passion of learning.

Just keep pedalling...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main focus of this thesis is the investigation of optimization techniques for better solving the Earliness Tardiness Scheduling with Setups (ETSS) problem. The penalization of job earliness/tardiness arises from the Just-In-Time (JIT) methodology, which aims at increasing production efficiency by penalizing both early and tardy delivery [18, 60]. JIT was first used in manufacturing environments, however, it has recently seen uses in other industries including call centres, healthcare, and software development [22, 64, 35]. Similarly, minimizing setup times and costs in scheduling is also motivated by practical considerations. Machines cannot be used for the duration of a setup time between jobs, causing increased manufacturing times, which inflates production costs. Several sectors address this problem such as manufacturing, transportation, and logistics [68]. Together, these two complexities form the ETSS problem. The particular ETSS problem that we study is based on the heat treatment of rolled metals within manufacturing sectors that our industry client is involved with. The scheduling problem is typically solved multiple times a day, possibly every few hours as conditions change on the shop floor. Optimal or near-optimal schedules are required to be found within 10 minutes.

The ETSS problem requires scheduling jobs on a single machine with unary capacity. We are concerned with minimizing the total cost incurred from the scheduled jobs, which is a combination of earliness/tardiness costs for each job and the associated setup costs between each job. A job incurs an earliness cost if it finishes processing before its due date. Conversely, a tardiness cost is paid if a job finishes after its due date. Earliness/tardiness costs are multiplied by each job's quantity, resulting in a weighted earliness/tardiness cost. A setup cost occurs between each job and represents a cost associated with switching jobs on a manufacturing line. Setup costs are asymmetric. Between each pair of jobs is a setup time, also asymmetric, during which the machine cannot process any job. Further, jobs have release dates which dictate the earliest time a job can start. Thus, the ETSS problem combines a traditional sequencing problem via setup costs and times and a scheduling problem via determining job completion times for earliness/tardiness costs. These combined intricacies

bolster the motivation for studying this problem.

Removing the minimization of earliness/tardiness costs from the ETSS problem yields a single machine scheduling problem minimizing setup costs. This simplified problem is NP-hard as it is analogous to the well-known Travelling Salesman Problem which has been proven NP-hard [25]. Similarly, removing the consideration of setup costs results in a single machine scheduling problem minimizing job earliness/tardiness costs which is strongly NP-complete [61]. Thus, the ETSS problem with its shared complexities is NP-complete. We utilize four different methodologies including Mixed Integer Programming (MIP), Constraint Programming (CP), Decision Diagrams (DD), and Large Neighbourhood Search (LNS) to solve the ETSS problem. Empirical analyses are conducted to understand each approaches' benefits and drawbacks.

The experimental results demonstrated that MIP and CP faced scalability issues, attributed to poor lower bounds and weak solution quality improvement over time, and CP had weak objective function propagation. DDs faced similar scalability issues but are promising for smaller instances. However, LNS methods consistently performed better than the other techniques developed, suggesting that LNS is a suitable, scalable approach for solving the ETSS problem.

1.1 Dissertation Outline

Chapter 2 formally defines and provides the notation for the ETSS problem. It describes a heuristic-based approach that the industry client currently uses to solve the problem. This chapter discusses the real-world data provided by our industry client and provides an explanation of the instance generation used for experiments.

Chapter 3 provides a review of recent work on the ETSS problem, alongside the related optimization problems: single machine scheduling with earliness/tardiness costs and single machine scheduling with sequence dependent setups. We also describe the solution techniques used in this work including MIP, CP, DD, and LNS, noting where these methods have been previously applied to solve the ETSS problem.

Chapter 4 investigates MIP and CP models to solve the ETSS problem. Two state-of-the-art MIP models for single machine makespan minimization were modified for the ETSS problem. One additional MIP model was developed and one CP model. These models were easy to implement and provided a baseline for comparisons against more sophisticated techniques. Numerical results for the presented methods, a previous MIP model for ETSS, and client heuristic are discussed and analyzed. Our CP model outperforms the MIP models, but none of the proposed models are competitive with the client heuristic. All of the models exhibit solution quality stagnation over time.

Chapter 5 develops a Multivalued Decision Diagram (MDD) and CP-based

approach for the ETSS problem. The MDD method works over the sequence of jobs, providing an alternative technique for generating earliness/tardiness and setup cost lower bounds. The underlying constraint programming optimization engine works to propagate changes in decision variable domains to the decision diagram and vice versa. Under certain conditions, the decision diagram lower bound can be unsatisfactory. Thus, we develop a second, novel approach that combines the MDD model with a timing algorithm to address this shortcoming. Similar to Chapter 4, we show numerical results for the new methods. Unfortunately, neither MDD method was competitive with the previously developed models, including the client heuristic, as they failed to find feasible solutions to the larger instances. However, for smaller instances both MDD methods found stronger lower bounds than the base CP model.

Chapter 6 presents MIP and CP based LNS methods for the ETSS problem. We develop 90 unique MIP and 90 unique CP LNS approaches, parameterized by their neighbourhood function, neighbourhood solving criteria, and diversification procedure. We compare the effectiveness of MIP LNS and CP LNS configurations and find that the CP-based LNS approaches tend to surpass MIP-based LNS. The selected MIP LNS and CP LNS models outperform all previous models, except for the client heuristic, which they remain closely competitive with. The LNS methods greatly improve solution quality over time, which previous techniques struggled with.

The thesis concludes with Chapter 7, which summarizes the results and conclusions made throughout the chapters and provides possible future research directions.

1.2 Summary of Contributions

The following list describes the main contributions of this thesis:

- We formally defined the Earliness Tardiness Scheduling with Setups (ETSS) problem, which is derived from real-world industrial scheduling applications.
- We provided an extensive review of the ETSS literature, demonstrating a shortage in comparison to the ETSS subproblems: earliness/tardiness scheduling and scheduling with setups.
- We modified two existing state-of-the-art single machine makespan minimization models for the ETSS problem and demonstrated the lack of interchangeability between the two problems.
- We developed 187 solution approaches to the ETSS problem, including a novel MDD method that uses a timing algorithm, drawing techniques from MIP, CP, MDD, and LNS.

- We performed numerical experiments and found that the overall best performing technique was the CP LNS method. The best approximate technique was the client heuristic.
- We demonstrated CP's poor objective function propagation and root lower bounds for the ETSS. The MDD method solved these issues, but faced scalability issues, similar to MIP.
- Adding the timing algorithm to MDD was ineffective at improving performance, whilst taking longer to run.
- We showed the scalability of LNS for the ETSS problem and its competitiveness with non-exact techniques.

Chapter 2

Problem Preliminaries

In this chapter, we formally define the Earliness Tardiness Scheduling with Setups (ETSS) problem. We provide an overview of our instance generation procedure and the real-world data provided by our industry client. Finally, we discuss the current solution technique, a heuristic, used by the industry client.

2.1 Problem Definition

We define the ETSS problem by explaining the data and parameters required to solve a given instance. We are given a set of jobs, $j \in \mathcal{J}$, that must be scheduled on a single machine with unary capacity. Each job has a processing time p_j, release date r_j, due date d_j, and quantity q_j. Each pair of distinct jobs, $j, j' \in \mathcal{J}$, have a sequence dependent setup time $s_{j,j'}$ and setup cost $S_{j,j'}$. Consequently, switching between jobs on the machine results in a setup time, during which no job is run, and a setup cost penalty. Setup time and costs are not assumed to follow the triangle inequality. Jobs are non-prememptive, meaning that once a job has started it must run for p_j time units. Jobs cannot start processing until the machine is free and it is past a job's release date r_j. The problem's objective function is to minimize total weighted earliness/tardiness and setup cost. The earliness/tardiness cost for job j is defined as $q_j \times \max\{\alpha\left(d_j - c_j\right), \beta\left(c_j - d_j\right)\}$, where $\alpha\left(\beta\right)$ is an overall earliness (tardiness) penalty and c_j is the completion time of job j. α and β are not assumed to be equal, thus leading to asymmetric earliness and tardiness costs. Then, the ETSS objective function is the sum of each job's earliness/tardiness costs and the sum of setup costs for the job sequence:

$$\sum_{j \in \mathcal{J}} q_j \max\{\alpha\left(d_j - c_j\right), \beta\left(c_j - d_j\right)\} + \sum_{i \in \mathcal{J}} \sum_{j \in \mathcal{J}} S_{ij} x_{ij}$$

Where x_{ij} is a binary decision variable, which is equal to 1 if job j directly follows job i; otherwise it takes a value of 0. A summary of the problem notation is presented in Table 2.1.

Table 2.1: ETSS Problem Notation

Symbol	Definition
$j \in \mathcal{J}$	Set of jobs
p_j	Processing time of job j
r_j	Release date of job j
d_j	Due date of job j
q_j	Quantity of job j
$s_{i,j}$	Setup time between job i and job j
$S_{i,j}$	Setup cost between job i and job j
α	Earliness penalty
β	Tardiness penalty

2.2 Data Overview

The ETSS problem data used to evaluate the techniques in this thesis is from a combination of client-provided real instances and randomly generated instances. The industry client provided two sample instances with 100 and 200 jobs.

2.2.1 Instance Generation

The remaining instances were randomly generated, following the characteristics of the client data. For each instance, 20% of the jobs, rounded to the nearest integer, were selected to be late; meaning these jobs have a negative due date, following a uniform distribution between -10000 and 0. Otherwise, a job's due date was uniformly distributed between 0 and 2000. Job release dates are the minimum between its due date and a random uniform number between 0 and 300. This formula ensures that release dates are before or at due dates. Processing times are distributed uniformly between 1 and 90. Job quantities are uniform between 1 and 3000. Setup costs were distributed uniformly between 1 and 800, while setup times were uniformly distributed between 1 and 60. All of instance data is discrete and the time unit is assumed to be minutes, as per the client data.

The following problem sizes were tested, denoted by the number of jobs: 10, 20, 50, 100, 150, 200, 250, 300, 350, and 400. For each problem size, 10 randomly generated instances were created, resulting in 100 generated instances and 2 client instances.

2.3 Client Heuristic

The industry client currently uses a heuristic to solve the ETSS problem, based on a paper by Boctor in 2016 [13]. The only change from the paper implementation is that the client heuristic uses a fixed time limit instead of a fixed number of iterations for termination criteria. The heuristic does not insert idle

time within the schedule. Informally, the heuristic generates an initial feasible solution following a greedy procedure. It then applies two improvement procedures, which are intended to find the local minima within the initial sequence neighbourhood. Then, a perturbation algorithm is run to promote diverse neighbourhood exploration. Each time the perturbation algorithm finishes, both improvement procedures are run. The perturbation and improvement procedures are cyclically repeated until the time limit is reached and the best solution is returned, as shown in Algorithm 1.

Algorithm 1 Client Heuristic

```
Generate initial solution
Run improvement procedures
while Time Remaining > 0 do
    Run perturbation procedure
    Run improvement procedures
return best solution
```

2.3.1 Initial Solution

An initial solution is generated following a simple construction rule. The procedure starts at time $t = a$, which is the earliest time the machine is available to run a job. Then, we calculate for each unscheduled, available job j the incremental cost C_j of scheduling j at t. A Restricted Candidate List (RCL) is built where only jobs meeting the following criteria are added: $C_j \leq C_{min} + \alpha\,(C_{max} - C_{min})$. Where C_j is the incremental cost of scheduling job j at time t, $C_{min}\,(C_{max})$ is the minimum (maximum) incremental cost of all unscheduled, available jobs, and α is drawn from a standard uniform distribution between 0 and 1. From the RCL, a random job is selected, with uniform probability, to schedule and the time is incremented to the end time of this job. The process is repeated until all jobs are scheduled. Algorithm 2 outlines this construction rule.

Algorithm 2 Initial schedule construction

```
t = 0
while Unscheduled jobs remain do
    Calculate C_j for each job at time t
    Calculate C_min and C_max
    Generate α from U_[0,1]
    Put jobs on RCL where C_j ≤ C_min + α (C_max − C_min)
    Randomly select and schedule a job from RCL
    Increment t to the end of scheduled job
return final sequence
```

2.3.2 Improvement Procedures

The improvement neighbourhood is a two-part local search technique. The first improvement procedure iterates through each job in the current sequence and inserts that job into the position which yields the largest cost reduction. It stops when no improvement can be found for any job. Thus, the neighbourhood is all the neighbour sequences where only one job is moved to a different position. See Algorithm 3 for details.

Algorithm 3 First Improvement Procedure

```
1:  procedure I1(sequence)
2:      while improvement = true do
3:          improvement = false
4:          for j = 1; j ≤ n do
5:              for p = 1; p ≤ n, p ≠ j do
6:                  Calculate D_jp, the cost if job j moved to position p
7:              if min_p{D_jp} < 0 then
8:                  improvement = true
9:                  Move job j to arg min_p{D_jp}
10:     return final sequence
```

The second procedure considers all possible pairs of jobs and looks for the pair that, when swapped, provides the maximal cost reduction. Again, the procedure stops when no job pair swap will decrease the overall cost. This neighbourhood is all sequences where only one pair of jobs exchange their position in the current sequence. See Algorithm 4 for details.

Algorithm 4 Second Improvement Procedure

```
1:  procedure I2(sequence)
2:      while improvement = true do
3:          improvement = false
4:          for j = 1; j ≤ n do
5:              for l = 1; l ≤ n, l ≠ j do
6:                  Calculate D_jl, the cost of swapping jobs j and l
7:          if min_{j,l}{D_jl} < 0 then
8:              improvement = true
9:              Swap jobs i and k, where i, k = arg min_{j,l}{D_jl}
10:     return final sequence
```

To calculate the cost reduction in I1 and I2, sequences must have their job completion times recalculated. For example, every time a job is inserted into a different position with I1, the job completion times need to be optimally recalculated to determine the new earliness/tardiness costs. A similar situation occurs with the second improvement procedure.

Setup costs are trivial to determine given the job sequence. For the earliness/tardiness cost, the client heuristic assumes that the earliness penalty (α)

is less than the tardiness penalty (β). As the client heuristic does not insert idle time, it solves the job timing problem by shifting all job completion times to their earliest possible for a given sequence. This would not yield a minimum cost in the ETSS problem where idle time is permitted.

2.3.3 Perturbation Procedure

The perturbation repeats n times, where n is the number of jobs. In each iteration it randomly selects x, a value between 0 and 1. If x is less than a set threshold ξ, the procedure randomly selects three jobs and circularly permutes them. If $x \geq \xi$, the procedure will not permute any jobs for that iteration. ξ is used to control the level of perturbation that will occur. A higher value of ξ increases the probability that a circular perturbation will occur each iteration. The client heuristic uses $\xi = 0.3$. Algorithm 5 outlines the procedure.

Algorithm 5 Heuristic Perturbation Procedure

```
1: procedure PERTURBATION(ξ)
2:     for j = 1; j ≤ n do
3:         Generate x from U_[0,1]
4:         if x < ξ then
5:             Generate p, q, r from U_(1,n)
6:             Move job in position p to position q
7:             Move job in position q to position r
8:             Move job in position r to position p
9:     return final sequence
```

2.4 Summary

In this chapter, we formally defined the ETSS problem and provided the problem notation used throughout this thesis. A data overview was provided, explaining how instances are randomly generated in addition to the client provided instances. Finally, this chapter provided a description of the industrial client's heuristic, which is the current solution technique used to solve the ETSS problem.

Chapter 3

Literature Review

In this chapter we explore the literature concerning the Earliness Tardiness Scheduling with Setups (ETSS) problem, alongside some related scheduling optimization problems, namely Earliness Tardiness (ET) single machine scheduling and single machine scheduling with sequence dependent setups. We explore these latter problems as there are few discussions of the ETSS problem in the literature. Next, we present a summary of four solution techniques used to solve scheduling optimization problems that are utilized in this work: Mixed Integer Programming (MIP), Constraint Programming (CP), Decision Diagrams (DD), and Large Neighbourhood Search (LNS).

3.1 Earliness Tardiness Scheduling with Setups

Unfortunately, the literature is sparse regarding earliness/tardiness single machine scheduling with sequence dependent setups. Kate, Wijngaard, and Zijm were the first to combine earliness and tardiness costs with setup times and costs in their 1995 paper [38]. Within their framework, jobs were assigned to families, such that families of jobs shared the same processing times, due dates, and setup times. Although, in this approach, one can assign each job to its own family, yielding a similar result to the ETSS problem structure. However, the paper does not use job-weighted earliness/tardiness as is the case in the ETSS framework. Additionally, the authors assume that tardiness costs are strictly larger than earliness costs for all jobs. Two exact optimization techniques were used: a MIP model and a customized branch and bound implementation, along with four heuristic methods: an Earliest Due Date (EDD) rule, a shortest processing time rule, a greedy insertion procedure, and a local search procedure. The authors found best results from the local search procedure using a starting sequence from EDD. The branch and bound implementation outperformed MIP due to poor scalability. Unfortunately, there are no comparisons between the exact and heuristic methods.

In 1997, Wang and Wang developed a hybrid algorithm that uses a heuristic within a genetic algorithm for the ETSS problem [65]. To the best of our

knowledge, these authors are the first to tackle the full ETSS problem, which includes job specific earliness, tardiness, and setup penalties. Their results showed that the hybrid approach beat the standalone heuristic and standalone genetic algorithm.

In their 2005 paper [60], Sourd proposes a new, time-indexed MIP formulation for the ETSS problem, deriving lower bounds from a new valid cut and a Lagrangian relaxation. A branch and bound algorithm, using a weaker but faster lower bound and new dominance rules is developed. Finally, Sourd proposes a novel multi-start heuristic procedure.

The time-indexed model was selected over a disjunctive model due to providing strong lower bounds: Sourd's experiments demonstrated a lower bound improvement by a factor of 400, while taking less time [60]. The MIP model then assigns jobs to time slots to minimize overall cost, using the costs of assigning any part of job i to time t: c_{it}. Overall, the model was only able to solve small instances with 20 jobs and short processing and setup times, likely due to the increase in model size when the time horizon increases.

Two lower bounds are generated from the time-indexed model: a linear relaxation and Lagrangian relaxation. The linear relaxation is improved with a new valid cut based on preventing preemptive jobs. The linear relaxation with the valid cut improves the lower bound by a few percent over strictly the linear relaxation; however, it increases computation time typically by a factor of 10.

The Lagrangian relaxation is based on the removal of the non-preemptive constraints, yielding a linear programming model, that can be regarded as a path in a graph structure. This graphical Lagrangian relaxation shares similarities with decision diagram based Lagrangian relaxations, such as in the multicommodity pickup-and-delivery Travelling Salesman Problem (TSP) [19]. Finding the shortest path from root to terminal node in this graph solves the Lagrangian relaxation and takes $O(nT)$ time, where n is the number of jobs and T is the length of the time horizon. The Lagrangian relaxation parameters were selected via a subgradient algorithm to maximize the lower bound. This lower bound provides a slightly weaker lower bound than the linear relaxation of the MIP but is much faster to compute.

Finally, Sourd's branch and bound approach branches on job sequence, such that a complete path in the search tree represents a valid job sequence assignment. The technique uses a lower bound at each node that considers the partial sequences of jobs. Further, the implementation considers new dominance rules for partial sequences.

Overall, Sourd's MIP formulations are limited to instances with less than 10 jobs and for the branch and bound, 20 jobs. However, the paper additionally proposes a multi-start heuristic which works well for instances between 100 and 500 jobs. This heuristic begins with a random job sequence and performs an iterative improvement procedure using the new partial sequence dominance rules. The heuristic is then restarted with a new random solution and the

Table 3.1: Categorization of ETSS variants.

Paper	Problem Characteristic	Approach
[38]	Total earliness/tardiness	MIP, branch and bound, heuristics
[65, 60, 59]	Weighted earliness/tardiness	Heuristics, genetic algorithm, branch and bound, LNS

improvement procedure is repeated. The best solution is returned.

In a later work, Sourd turns to a very large neighbourhood search implementation for the ETSS problem using a dynasearch neighbourhood, which runs in pseudo-polynomial time [59]. A dynasearch neighbourhood is a composition of an arbitrary number of independent operators, yielding an exponential neighbourhood size, but a dynamic programming algorithm can compute the optimal schedule in polynomial time. Given an initial sequence of jobs, $(J_1, \ldots, J_n)$, and a pair of indices $i < j$, the algorithm defines three neighbourhood operators that act on the sequence $\sigma = \alpha J_i \beta J_j \gamma$, where α is the sequence of jobs before J_i, J_i (J_j) is the job at position i (j), β is the sequence of jobs between jobs J_i and J_j, and γ is the sequence of jobs after job J_j. The neighbourhood operators are as follows:

- $\text{SWAP}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha J_j \beta J_i \gamma$. If β is an empty sequence, the swap is adjacent.
- $\text{EBSR}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha J_j J_i \beta \gamma$. Stands for extraction and backward shifted reinsertion.
- $\text{EFSR}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha \beta J_j J_i \gamma$. Stands for extraction and forward shifted reinsertion.

Using these neighbourhoods, the dynasearch problem starts with a job sequence and searches to find a neighbour that has minimal cost or cost less than some parameter. Sourd found that combining SWAP, EBSR, and EFSR was critically important for producing an efficient search algorithm. Their best solution technique involved running a simple descent procedure and then running dynasearch if the solution found was within 3% of the best-known solution. Otherwise, the descent procedure is run again. The simple descent procedure is based on the union of the three identified neighbourhoods, while the dynasearch neighbourhood is seen as a composition of these neighbourhoods. Thus, dynasearch is time consuming but generates good quality solutions. The largest instance run was 100 jobs with 10 job groups for setup costs/times with a four-minute time limit. Unfortunately, the results were only compared between dynasearch and the simple descent procedure.

We provide a categorization of these ETSS papers in Table 3.1.

3.2 Related Problems

Two closely related research areas to the ETSS problem are ET scheduling on a single machine and single machine scheduling with sequence dependent setup times. In the former problem, we are concerned with minimizing earliness and tardiness, whereas in the latter usually it is the minimization of total setup time. These two problems can be seen as subproblems within the ETSS problem.

3.2.1 Earliness Tardiness Scheduling

There exists a large body of literature on ET scheduling, first introduced by Kanet in 1981 [37]. ET single machine problems consist of a set of jobs to schedule on a single machine with the objective of minimizing earliness and tardiness costs which stem from job due dates. ET problems are typically characterized by:

- ET penalties: *equal* is where earliness costs are equal to tardiness costs per time unit, *unequal* earliness costs are not equal to tardiness costs per time unit, *weighted* each job can have its own specific earliness and tardiness cost.
- Due dates: *common* all jobs share the same due date, *distinct/general* jobs have their own due dates.
- Setups: *none* there are no setup times between jobs, *sequence independent* there are setup times between jobs, but it only depends on the current job, *sequence dependent* setup times between jobs depend on both the current job and the previous job, which is the same as the problems in Section 3.2.2.

There are some other ET problem configurations, such as preventing machine idle time, but these complexities are outside the scope of this discussion.

Many approaches have been used to solve the ET problem. Sourd and Kedad-Sidhoum develop a new lower bound based on the decomposition of each job into unit time operations that are assigned to time slots. This lower bound, along with new dominance rules, is integrated into a branch and bound algorithm [62]. In later work, Sourd develops a new lower bound based on the Lagrangian relaxation of a time-indexed MIP model reinforced with valid cuts [61]. Then, this lower bound is used in a branch and bound algorithm to solve the ET problem with both common and distinct due dates. Kedad-Sidhoum and Sourd developed a LNS algorithm focusing on fast neighbourhoods for larger problem sizes, such that good solutions can be found in a couple of seconds [39]. For 50 job benchmark instances, for which the optimum is known, the algorithm's mean deviation is 0.01%. For instances with 100 to 500 jobs,

Table 3.2: Categorization of ET variants.

Paper	Problem Characteristic	Approach
[62, 61, 39, 44]	ET	Branch and bound, LNS
[61]	ET, common due dates	Branch and bound
[20, 57, 40, 42]	ET, setups, distinct due dates	MIP, branch and bound, heuristics, genetic algorithm
[54, 6, 66]	ET, setups, common due dates	Branch and bound, beam search, genetic algorithm

where the optimum is unknown, the authors compared against Lagrangian-based lower bounds and found a mean deviation of 5.5% from these lower bounds. Thus, the LNS algorithm found solutions close to the optimal values. Laborie and Godard proposed a Self-Adapting LNS algorithm that uses machine learning to converge on the most efficient neighbourhoods and neighbourhood solving strategies [44]. The approach is very general; however, it did not perform strongly on single machine earliness/tardiness problems.

ET with Sequence Dependent Setups

The first MIP formulation of an ET problem considering setup times was proposed in Coleman's technical note in 1992 [20]. This problem has been solved with branch and bound and heuristic techniques with equal ET penalties [57]. Similarly, it has been solved with weighted ET penalties using a time-indexed formulation and a Lagrangian-based branch and bound algorithm [40], in addition to a genetic algorithm [42].

ET with Sequence Dependent Setups and Common Due Dates

As well, work has been completed looking at ET problems with sequence dependent setups and common due dates. Rabadi, Mollaghasemi, and Anagnostopoulos [54] investigate an equally weighted ET problem with a customized branch and bound algorithm. The method uses a heuristic upper bound, branches by assigning a job to each possible position in the sequence, and uses a partial sequence lower bound calculation. Overall, the branch and bound method outperformed a basic MIP model [54]. Another branch and bound implementation and a beam search algorithm were developed by Azizoglu and Webster [6]. Webster and Gupta also examine the ET problem with setups and a common due date with a genetic algorithm [66]. Their model minimizes total weighted ET, whilst using the common job due date as a decision variable in their model.

Readers who are interested in more ET literature should refer to several surveys [7, 63]. For an in-depth discussion on ET lower bounds, refer to Schaller's paper [56]. We provide a categorization of these ET papers in Table 3.2.

3.2.2 Scheduling with Sequence Dependent Setups

Similar to ET scheduling, there is a plethora of research on single machine scheduling problems with sequence dependent setups. This problem is concerned with minimizing the total setup time occurred from the scheduling of all jobs on a machine. An analogous, well-known optimization problem is the Travelling Salesman Problem (TSP) [2]. Solving the TSP requires finding the shortest tour that traverses a set of locations, visits each location exactly once, and returns to the starting location. Mapping the two problems: jobs become locations, machine setup times between jobs become distances between locations, and the objective functions remain the same.

The TSP has complicated origins, rich with history, but one of the most influential early TSP researchers was Flood in the 1930s [4]. One of the first papers on scheduling with sequence dependent setups was by Gilmore and Gomory, who modelled and solved the problem as a TSP [27].

There are many variants to the TSP/scheduling with setups; however, we will briefly discuss the ones most pertinent to the ETSS problem:

- Symmetry: *symmetrical* - setups times do not depend on the job order within a pair, *asymmetrical* - setup times depend on job sequence within the job pair, i.e. $s_{ij} \neq s_{ji}$.
- Release dates: *none* - jobs to not have release dates, *present* - jobs have release dates.

A large number of techniques have been tried to solve general TSPs including customized optimization algorithms [53], branch and bound algorithms [14], dynamic programming [15], tabu search [45, 46], greedy randomized data adaptive search procedures [24], and other heuristic rules [67]. Additionally, Bianco, Salvatore, and Giovanni investigate a TSP with release dates via MIP and a heuristic algorithm with lower bounds and dominance criteria [10].

For more literature on TSPs, we refer the reader to two excellent books: *The Traveling Salesman Problem: A Computational Study* and *The Traveling Salesman Problem and Its Variants* [4, 29]. Likewise, approaching from the scheduling lens, we recommend an extensive survey by Allahverdi, Gupta, and Aldowaisan [2].

3.3 Mixed Integer Programming

Now, we shift our ETSS and related literature discussion to optimization techniques that are used to solve scheduling problems in this thesis. MIP is a well-known and successful exact technique for solving optimization problems involving continuous and discrete decision variables [36]. A generic MIP is expressed as follows:

$$\begin{aligned} \min\ & c^\top x + h^\top y \\ \text{s.t. } & Ax + Gy \leq b \\ & x \in \mathbb{Z}_+^n \\ & y \in \mathbb{R}_+^k \end{aligned}$$

Where c and h are objective function vector coefficients of size n and k, A is a $m \times n$ matrix and G is a $m \times k$ matrix, b is a vector of size m, x is a vector of integer decision variables, and y is a vector of continuous decision variables. If the model does not have any integer variables, it is a linear programming model. Conversely, a model without any continuous variables would be a pure integer programming model. The integer variable restrictions make MIPs challenging to solve, yielding an NP-hard problem [11], whereas solving a linear programming can be done in polynomial time with interior point methods [41].

There are two common MIP formulations for solving scheduling problems: disjunctive and time-indexed [43]. A disjunctive formulation focuses on the sequence of jobs, for example if job i is sequenced before or after job j. These models utilize two decision variables to model this disjunction: $C_j \geq 0$ are continuous decision variables that track the completion time of each job and $x_{ij} \in \{0, 1\}$ are binary decision variables that take a value of 1 *iff* job i comes before job j. The disjunction based on job sequence is then modelled as:

$$C_j \geq C_i + p_i - M\,(1 - x_{ij}) \qquad \forall i, j \in \mathcal{J}, i < j \qquad (3.1)$$

$$C_i \geq C_j + p_j - M\,(x_{ij}) \qquad \forall i, j \in \mathcal{J}, i < j \qquad (3.2)$$

Where $p_i\,(p_j)$ is the processing time of job $i\,(j)$ and M is a sufficiently large number. Constraint (3.1) ensures that the completion time of job j is after job i's completion time if $x_{ij} = 1$. Similarly, constraint (3.2) sets the completion time of job i to be greater than job j's completion time if $x_{ij} = 0$. These disjunctive constraints prevent jobs from overlapping on a machine and are commonly known as *big-M* constraints. One drawback with disjunctive formulations are that they tend to have weaker linear relaxations if a poor choice of M is selected [16].

The second formulation type is the time-indexed model. This model focuses on assigning jobs to time slots as opposed to reasoning about job sequence. This formulation uses a binary decision variable x_{jt} that is 1 *iff* job j is scheduled to start at time t. As jobs are assigned to timeslots, a set of time points, $\mathcal{T} = \{1, \ldots, T\}$, is required. A time-indexed MIP uses the following constraints to schedule jobs:

$$\sum_{t \in \mathcal{T}} x_{jt} = 1 \qquad \forall j \in \mathcal{J} \tag{3.3}$$

$$\sum_{j \in \mathcal{J}} \sum_{t'=t-p_j+1}^{t} x_{jt'} \leq 1 \qquad \forall t \in \mathcal{T} \tag{3.4}$$

Constraint (3.3) ensures each job starts exactly once and constraint (3.4) makes sure that at any time point, at most one job is executing. A drawback with time-indexed formulations are that the number of decision variables is proportional to the length of the time horizon. However, linear relaxations of time-indexed formulations for scheduling problems often provide strong lower bounds [1, 50].

3.4 Constraint Programming

Constraint programming (CP) is a technique for solving combinatorial constraint satisfaction and constraint optimization problems [55]. CP has been successfully applied to a wide range of optimization problems, including scheduling problems [8]. A constraint optimization problem consists of a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Z})$, where $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is the set of decision variables, $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ is the set of domains of the variables in $\mathcal{X}$, $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ is the set of constraints involving the variables in $\mathcal{X}$, and $\mathcal{Z}$ is the objective function. A solution to this problem is a complete assignment of the variables that satisfies all of the constraints and obtains a global minimum or maximum objective function value.

Similar to MIP, CP problems can be solved using branch and bound. However, at each node in the tree CP performs constraint propagation algorithms to reduce the variable domains and enforce consistency [32].

A set of constraints, $\mathcal{C}$, can have various types of consistency. The most basic, arc consistency, involves two decision variables and a binary constraint [12]. x_i is arc consistent with x_j if for every value a in the domain of x_i there exists a value b in the domain of x_j such that (a, b) satisfies a binary constraint between x_i and x_j. If the reverse is also true, then the constraint is arc consistent. If two variables are not arc consistent with a constraint, domain values that do not satisfy the constraint are pruned, resulting in either the constraint becoming arc consistent or infeasible if a variable domain becomes empty. A basic constraint propagation algorithm can make the entire problem arc consistent by repeating this process with all variable pairs. Basic arc consistency for binary constraints can be extended to all constraints, which is known as generalized arc consistency [12].

CP also utilizes more advanced constraint prorogation algorithms via global constraints [12]. These algorithms are designed to exploit structure in com-

binatorial optimization problems. A popular one, the `AllDifferent` global constraint, enforces that each of its variables must take on distinct values by ensuring generalized arc consistency over the given variables and their constraints.

Within CP search, inference techniques used at a given node reduce variable domains. Branching can occur from setting variable values heuristically or by branching on a constraint. If, during the search, a variable's domain becomes empty, the current node is infeasible and pruned, leading to a backtrack in the search [12].

Interval variables, which represent an interval of time during which a job is processed, are often used in a CP model for scheduling problems. An interval variable a is a variable whose domain $dom(a)$ is a subset of $\{\perp\} \cup \{[s, e) \mid s, e \in \mathbb{Z}, s \leq e\}$, where s (e) is the start time (end time) of the interval variable [34]. An interval variable is absent when $a = \perp$ and the variable is present when $a = [s, e)$. An absent interval variable is not considered by any constraint or expression involving it. For example, in a single machine scheduling problem, permitting an interval variable to be absent allows the interval variable to not be scheduled. We can model complex scheduling problems using interval variables in CP. For instance, we can set constraints on interval variable start times and end times without additional variables. Additional logic-based constraints involving interval variables are easy to implement [34].

For scheduling problems, we are often concerned with the sequence of jobs. CP offers a sequence variable that is defined on a set of interval variables A. The value of sequence variable is a total ordering of the variables in A. More formally, a sequence variable p on a set of interval variables A represents a decision variable whose possible values are all the permutations of the intervals of A [33]. It is easy to model constraints on the position of interval variables in the sequence variable using constraints such as `First`, `Last`, `Before`, and `Prev`. `First` (`Last`) specify what interval variable must be in the first (last) position in the sequence. Similarly, `Before` and `Prev` enforce the ordering of two interval variables within a sequence variable [33].

CP has many global constraints that further ease the modelling and solving of scheduling problems [55]. For example, the `NoOverlap` constraint on a sequence variable ensures that the sequence does not have any interval variables that overlap temporally. This constraint could be used to ensure a machine does not process more than one job at a time, i.e. any interval variable in the sequence is constrained to end before the start of the next interval variable in the sequence [33]. Additionally, the `NoOverlap` constraint can be configured to account for setup times between intervals variables, using a matrix of setup times for the different interval variables in the given sequence variable. That is, `NoOverlap` will enforce the minimal time that must separate two consecutive intervals in the sequence.

3.5 Decision Diagrams

Decision Diagrams (DD) are a graphical structure that encodes a set of solutions to a discrete optimization problem and they have been recently used for sequencing and scheduling problems [9]. DDs can either be exact or relaxed. An exact DD has a one-to-one encoding of the feasible solution set of its discrete optimization problem. In the case of a minimization (maximization) objective function, an exact DD reduces the optimization problem to a shortest (longest) path problem in its graph. Solving this path problem yields the optimal solution and objective value to the original discrete optimization problem. Conversely, a relaxed DD overapproximates the feasible set and objective function of its underlying problem. When solved, a relaxed DD provides bounds on its problem's objective function. Additionally, DDs provide important inferences when used in conjunction with a base optimization engine, such as with constraint programming, where the constraint programming optimization engine propagates changes in decision variable domains to the decision diagram and vice versa.

More formally, a DD is a layered directed acyclic multigraph, $G = (V, A)$, with node set N and arc set A. The out-degree of the nodes in N further classify the DD: a Binary Decision Diagram (BDD) strictly permits an out degree of less than or equal to 2 and a Multivalued Decision Diagram (MDD) allows an out-degree larger than 2. The set of nodes, V, is partitioned into $n+1$ layers $L_1, \ldots, L_{n+1}$. Layers L_1 and L_{n+1} are singletons representing the root $\mathbf{r}$ and the terminal $\mathbf{t}$ respectively. An arc $a = (u, v)$ of G is always directed from a source node u in some layer L_i to a target node v in the subsequent layer L_{i+1}, $i \in \{1, \ldots, n\}$. We create a value, $v_a \in \mathcal{J}$, for each arc $a \in A$, which is used to map the meaning of an arc back to the discrete optimization problem. In the case of a scheduling problem, arc values may represent job indices. $\ell(a)$ indicates the layer of the source node u of the arc a and $u \in L_{\ell(a)}$. We denote $a' \in in(u)$ as the set of all arcs that are directed into node u and $a' \in out(v)$ as the set of all arcs that are directed out of node v. Finally, we define weighted decision diagrams, in which each arc a has an associated length l_a. The length of a directed path $p = \left(a^{(1)}, \ldots, a^{(k)}\right)$ rooted at $\mathbf{r}$ corresponds to $l_p = \sum_{j=1}^{k} l_a^{(j)}$.

A decision diagram is exact if the $\mathbf{r}$–$\mathbf{t}$ paths in G precisely encode the feasible solutions of the discrete optimization problem, and the length of a path is the objective function value of the corresponding solution. However, due to computational complexity, decision diagrams are often relaxed in practice [9]. A relaxed decision diagram represents a superset of the feasible solutions and therefore, provides a discrete relaxation of the problem. A key advantage of relaxed decision diagrams is that they can be much smaller than exact ones while still providing a useful relaxation, if they are properly constructed [9]. A relaxed DD's size is controlled by specifying an upper bound on the width of the diagram. The width is the maximum number of nodes in any layer.

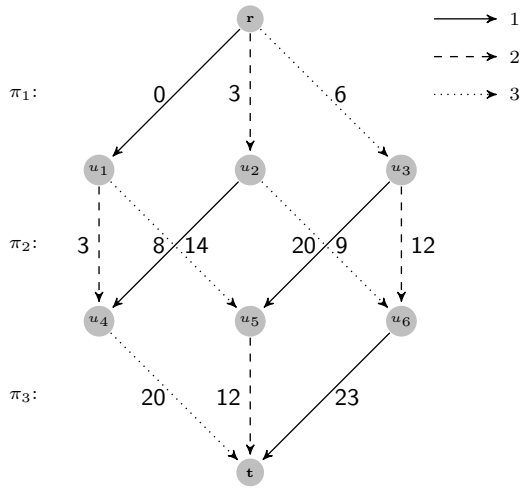


Figure 3.1: An example decision diagram.

We provide a simple DD example in Figure 3.1, where the decision diagram is an MDD, has 4 layers, and has a width of 3. Each arc can take the value of 1, 2 or 3, and the shortest path is $(\mathbf{r}, u_1), (u_1, u_5), (u_5, \mathbf{t})$ with length 20.

3.5.1 Decision Diagrams and Single Machine Scheduling

Decision diagrams have been used to great success for sequencing and single machine scheduling problems in combination with a constraint-based scheduling solver [9]. Various objective functions have been explored, including makespan, sum of setup times, and total tardiness. Further, more complex single machine scheduling aspects have been studied such as precedence constraints, job deadlines, and time windows within an asymmetric TSP [9].

We provide an example of a single machine scheduling problem minimizing makespan from Chapter 11 in Bergman, Cire, Van Hoeve, and Hooker [9]. Given a set of n jobs, $\mathcal{J}$, to schedule on a unary capacity machine, with release dates r_j, processing times p_j, deadlines d_j, and setup times s_{ij}, the following decision diagram formulation is used: each path in the DD represents a feasible ordering of jobs in $\mathcal{J}$. Job completion times must be before their deadline: $c_j \leq d_j$.

A three-job example instance is shown in Table 3.3 and the resultant DD in Figure 3.2. There are four possible paths from $\mathbf{r}$ to $\mathbf{t}$ in our DD, and the path traversing nodes $\mathbf{r}$, u_2, u_4, $\mathbf{t}$ represents a solution sequence of $j_3 \rightarrow j_2 \rightarrow j_1$. The jobs have the following complexion times: $c_1 = 15$, $c_2 = 9$, $c_3 = 3$. Note, j_1 cannot be first on the machine as this would violate the deadlines of j_2 or j_3. Thus, there is no arc with $v_a = j_1$ out of $\mathbf{r}$.

Computing the orderings to determine the optimal makespan or sum of setup times is polynomial in the size of the decision diagram [9]. For the case of makespan, we define the earliest completion time of an arc, ect_a, as the

Table 3.3: A three job example instance.

(a) Instance

Job	r_j	d_j	p_j
1	2	20	3
2	0	14	4
3	1	14	2

(b) Setup times

	1	2	3
1	0	3	2
2	3	0	1
3	1	2	0

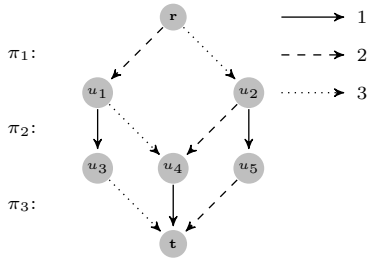


Figure 3.2: An exact decision diagram for Table 3.3.

earliest completion time of the job v_a, among all paths in the DD that contain arc a. If a is directed out of $\mathbf{r}$, then $ect_a = r_{v_a} + p_{v_a}$, where r_{v_a} is the release date of job v_a and p_{v_a} is the processing time of job v_a. The remaining arcs can be calculated as follows:

$$ect_a = max\{r_{v_a}, min\{ect_{a'} + s_{v_a,v_{a'}} : a' \in in\,(u)\}\} + p_{v_a}, \tag{3.5}$$

where $s_{v_a,v_{a'}}$ is the setup time between jobs v_a and $v_{a'}$ and $a' \in in\,(u)$ is the set of all arcs that are directed into node u. Then, the minimum makespan is $min_{a \in in(\mathbf{t})} ect_a$. An optimal job sequence is obtained by recursively retrieving the minimizer of arc $a' \in in\,(u)$ in Equation 3.5 [9].

For the DD in Figure 3.2, we obtain $ect_{\mathbf{r},u_1} = 4$, $ect_{\mathbf{r},u_2} = 3$, $ect_{u_1,u_3} = 10$, $ect_{u_1,u_4} = 7$, $ect_{u_2,u_4} = 9$, $ect_{u_2,u_5} = 7$, $ect_{u_3,\mathbf{t}} = 14$, $ect_{u_4,\mathbf{t}} = 11$, $ect_{u_5,\mathbf{t}} = 14$. The optimal makespan is $min\{ect_{u_3,\mathbf{t}}, ect_{u_4,\mathbf{t}}, ect_{u_5,\mathbf{t}}\} = ect_{u_4,\mathbf{t}} = 11$, corresponding to the path $(\mathbf{r}, u_1, u_4, \mathbf{t})$ or equivalently (j_2, j_3, j_1).

Alternatively, we can use a relaxed DD to obtain a lower bound on makespan. Figure 3.3 shows a 1-width and 2-width relaxation for this sequencing problem. Notice that both relaxed DDs contain all feasible orderings for our problem but they also include infeasible solutions, such as the path representing (j_3, j_2, j_2) in the 2-width relaxation. The makespan lower bound is calculated following the same approach as in the exact DD.

A relaxed DD can be trivially created using a 1-width relaxation. Then, the DD is incrementally modified to strengthen its relaxation, whilst respecting the maximum width specified. Two procedures are used to strengthen the DD relaxation: filtering and refinement [9].

- Filtering: the process of removing infeasible arcs in the decision diagram.

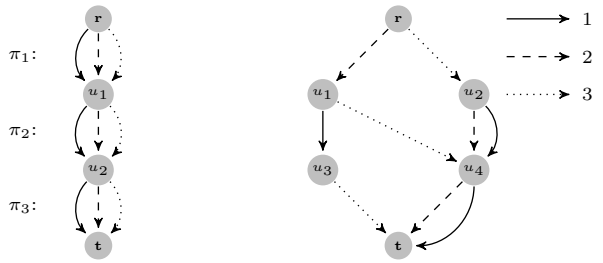


Figure 3.3: A 1-width relaxed DD and a 2-width relaxed DD for the Table 3.3 instance.

That is, an arc a is infeasible if all paths in the decision diagram containing a represent infeasible job sequences.

- Refinement: splitting nodes to represent the problem structure more accurately. For example, our single machine scheduling example, we wish to better represent the `AllDifferent` structure over the jobs.

If a relaxed DD does not have any infeasible arcs and no nodes require refinement, then the DD is exact [9]. For the single machine scheduling problem, there exist several filtering algorithms including: invalid permutations, precedence constraints, time window constraints, and objective function bounds [9]. Refinement focuses on the permutation of jobs, which is represented by the `AllDifferent` constraint.

For more information on decision diagrams and optimization, please refer to *Decision Diagrams for Optimization*, especially Chapter 11 for sequencing and single machine scheduling [9].

3.6 Large Neighbourhood Search

Large Neighbourhood Search (LNS) is a local search technique that defines and searches large neighbourhoods for solutions [21]. LNS was first proposed by Shaw in 1998 with their work *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems* [58]. Since then, LNS approaches have shown outstanding results in solving various transportation and scheduling problems [26].

In LNS, an initial solution is generated and is iteratively modified by relaxing parts of the solution and resolving [58]. Relaxing a solution entails fixing part of the existing solution, preventing it from changing during the resolve. In a scheduling context, one may choose to fix the start times of the jobs or the sequence in which jobs are processed. The rest of the variables are free to change values when the solution is resolved. Solution relaxation is dependent on a neighbourhood, which is defined as all the possible extensions of the fixed solution. The size of a neighbourhood is crucial in LNS: larger neighbourhoods provide possibilities of better-quality local optima but they take longer

to search through. Conversely, smaller neighbourhoods may offer poorer solutions but faster solving time [26]. Within LNS, neighbourhoods tend to be large, thus intelligent searching during the resolving phase is provided via tree search, constraint programming, or mixed integer programming [21].

Algorithm 6 [26] details a general LNS structure, where x^b is the best known solution, $N(x)$ is the neighbourhood function, which when solved returns the new neighbour x^t. Solving the neighbourhood can be done in various ways, such as using an optimization framework like constraint programming or using a heuristic. Further, considerations such as a time limit can be implemented. Line 5 determines if we accept the new neighbour following some acceptance criteria. The simplest choice is to only accept improving solutions. Line 7 is used to track the best solution where $c(x)$ denotes the objective function value of solution x. Typical termination criteria include a fixed number of iterations or a time limit.

Algorithm 6 Large Neighbourhood Search

1: Generate initial solution x
2: $x^b = x$
3: **while** Termination criteria not met **do**
4: $\quad x^t = N(x)$
5: $\quad$ **if** accept(x^t, x) **then**
6: $\quad\quad x = x^t$
7: $\quad$ **if** $c(x^t) < c(x^b)$ **then**
8: $\quad\quad x^b = x^t$
9: **return** x^b

Some LNS algorithms include a diversification procedure which moves the solution to a new part of the search space. This procedure is typically run if the LNS algorithm stalls, that is, repeatably finds neighbours with the same objective cost, for a selected number of iterations.

Several papers have used LNS techniques to solve a variety scheduling problems such as cumulative [28], job shop [17], and total weighted earliness/tardiness [39]. As well, Laborie and Godard provide an extensive study on various scheduling problems and LNS [44].

3.7 Summary

In this chapter, we presented the literature available for the ETSS problem. As the literature is sparse, we provided background information regarding ET single machine scheduling and single machine scheduling with sequence dependent setups, as these two problems are subproblems within the ETSS. We also provided a brief explanation and overview of the solution techniques used to solve scheduling problems similar to, and including, the ETSS problem: MIP, CP, DD, and LNS.

Chapter 4

Mixed Integer Programming and Constraint Programming for Earliness Tardiness Scheduling with Setups

This chapter investigates models to solve the Earliness Tardiness Single machine scheduling with sequence dependent Setups (ETSS) problem. These models are basic and easy to implement, thus providing a baseline foundation to compare against more sophisticated approaches. As well, we compare these initial models against the state-of-the-art for the related single machine scheduling problem with setups for minimizing makespan to determine if the state-of-the-art for that problem can efficiently solve the ETSS. The initial approaches include four Mixed Integer Programming (MIP) models and one Constraint Programming (CP) model. We introduce the experimental setup used for all of the experiments in this work. Finally, we present and discuss the numerical results for the MIP and CP models and compare against the client heuristic. The best performing exact technique is the CP model, followed by the Simplified MIP, although neither outperformed the client heuristic. The proposed methods face issues with improving solution quality over time. As well, the CP model struggles with variable domain propagation with our complex objective function.

4.1 Mixed Integer Programming Formulations

A total of four MIP models were developed and are denoted as Base, AAA, MTZ_AM, and Simplified. All of the models share a main binary decision variable x_{ij}, which is equal to 1 if job j directly follows job i, otherwise it takes a value of 0.

4.1.1 Base Model

The Base MIP model was developed by Li in her undergraduate thesis in collaboration with the industry client [47]. This model uses the following decision variables:

- $g_j \geq 0$ as the start time of job j
- $e_j \geq 0$ as the earliness of job j
- $t_j \geq 0$ as the tardiness of job j

Using the problem notation as presented in Chapter 2, Table 2.1, the model is as follows:

$$\min \sum_{i \in \mathcal{J}} \sum_{j \in \mathcal{J}} S_{ij} x_{ij} + \sum_{j \in \mathcal{J}} \alpha q_j e_j + \beta q_j t_j \tag{4.1}$$

$$\text{s.t.} \sum_{i \in \mathcal{J}} x_{ij} = 1 \qquad \forall j \in \mathcal{J}, j \neq 0 \tag{4.2}$$

$$\sum_{j \in \mathcal{J}} x_{ij} \leq 1 \qquad \forall i \in \mathcal{J} \tag{4.3}$$

$$\sum_{j \in \mathcal{J}} x_{j0} = 0 \tag{4.4}$$

$$\sum_{j \in \mathcal{J}} x_{jj} = 0 \tag{4.5}$$

$$g_j - (s_{ij} + g_i + p_i) \geq M\,(x_{ij} - 1) \qquad \forall i, j \in \mathcal{J}, i \neq j \tag{4.6}$$

$$g_j \geq r_j \qquad \forall j \in \mathcal{J} \tag{4.7}$$

$$e_j \geq d_j - g_j - p_j \qquad \forall j \in \mathcal{J} \tag{4.8}$$

$$t_j \geq g_j + p_j - d_j \qquad \forall j \in \mathcal{J} \tag{4.9}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in \mathcal{J} \tag{4.10}$$

$$g_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.11}$$

$$e_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.12}$$

$$t_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.13}$$

Objective (4.1) minimizes the total setup cost plus earliness and tardiness costs. Constraints (4.2) - (4.3) are subtour elimination constraints; they ensure that the solution sequence executes each job exactly once and there is a path from start to finish. Constraints (4.4) and (4.5) lock in the first job and prevent a job from directly following itself (i.e., $x_{11} \neq 1$). Constraint (4.6) tracks the start time of each job and prevents overlap between the jobs using a disjunctive big-M constraint with processing and setup times. Constraint (4.7) enforces release dates on jobs. Constraints (4.8) and (4.9) track each job's earliness and tardiness. Finally, constraints (4.10) - (4.13) declare the decision variables.

One can observe that the base model can be simplified by modifying several of the constraints, which the next model takes into consideration.

4.1.2 AAA Model

The Avalos-Rosales et al. [5] (AAA) model was the state-of-the-art method, without using decomposition, for minimizing makespan on single machine scheduling problems with setup times and cost. It was recently outperformed by the MTZ_AM model, which is defined in Section 4.1.3. AAA appears similar to the base MIP model, however it removes the unnecessary constraints. We modify the objective function for our problem:

$$
\begin{aligned}
&\min && (4.1) && && \\
&\text{s.t.} && (4.6) - (4.13) && && \\
& && \sum_{i \in \mathcal{J}, i \neq j} x_{ij} = 1 && \forall j \in \mathcal{J}, j \neq 0 && (4.14) \\
& && \sum_{j \in \mathcal{J}, j \neq i} x_{ij} \leq 1 && \forall i \in \mathcal{J} && (4.15)
\end{aligned}
$$

The AAA model combines constraints (4.2) - (4.5) and simplifies them using constraints (4.14) and (4.15). In the base model, the subtour elimination constraints (4.2) and (4.3) do not prevent identical job transitions and do not lock in the first job. However, we can modify the original constraints to consider both, yielding constraints (4.14) and (4.15).

4.1.3 MTZ_AM Model

The Miller-Tucker-Zemlin AM (MTZ_AM) model is considered the state-of-the-art for minimizing makespan on single machine scheduling problems with setups without using a decomposition approach [23]. In their 2018 paper, Fanjul-Peyro, Ruiz, and Perea [23] demonstrated that the MTZ_AM model outperformed the previous state-of-the-art AAA model. This model is derived from the AAA model, except it uses the well-known Miller-Tucker-Zemlin Subtour Elimination Constraints (SEC) and two valid inequalities that the authors denote as AM.

MTZ_AM introduces a new variable, $U_j \in \mathbb{Z}_+$, which is the lower limit on the number of jobs processed before j on the machine where j is processed. Unfortunately, this model had to be modified to track job start times due to our complex objective function. The actual MTZ_AM model does not require $g_j, e_j,$ and t_j as, given a solution, one can calculate the makespan of the sequence without needing to time the jobs. However, in our problem, given a sequence of jobs without start times, we must determine the these timings in

order to minimize earliness/tardiness in our objective function. The modified MTZ_AM model is as follows:

$$\min \sum_{i\in\mathcal{J}}\sum_{j\in\mathcal{J}} S_{ij}x_{ij} + \sum_{j\in\mathcal{J}} \alpha q_j e_j + \beta q_j t_j \tag{4.16}$$

s.t. (4.6) – (4.15)

$$U_i - U_j + |\mathcal{J}|x_{ij} \leq |\mathcal{J}| - 1 \qquad \forall i,j \in \mathcal{J}, i \neq j \tag{4.17}$$

$$U_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.18}$$

Constraint (4.17) is a redundant subtour elimination constraint as our model eliminates subtours with job start times in (4.6). Additionally, due to constraint structure of (4.17), U_j is relaxed to be a continuous variable without permitting infeasible solutions. Constraint (4.17) is from the well-known Miller–Tucker–Zemlin (MTZ) constraints [48]. We wanted to investigate the addition of these redundant subtour elimination constraints in our work, especially with their success in the original MTZ_AM model. Further, we can directly measure the impact of these constraints as they are the only difference between the AAA and MTZ_AM models in our work.

4.1.4 Simplified Model

The Simplified MIP model is based on the previous MIP models; however, we utilize a variable f_j to track the earliness/tardiness cost of each job, instead of using one variable for earliness and another for tardiness. This cost tracking simplification greatly reduces the model size and is possible as each job is either early, on time or late; thus, we can track a job's cost with only one variable. As well, we use $c_j \geq 0$ to track job completion times. The model is then as follows:

$$\min \sum_{j\in\mathcal{J}} f_j + \sum_{i\in\mathcal{J}}\sum_{j\in\mathcal{J}} S_{ij}x_{ij} \tag{4.19}$$

s.t. (4.14) – (4.15)

$$c_j \geq c_i + s_{ij} + p_j - M\,(1 - x_{ij}) \qquad \forall i,j \in \mathcal{J}, i \neq j \tag{4.20}$$

$$c_j \geq r_j + p_j \qquad \forall j \in \mathcal{J} \tag{4.21}$$

$$f_j \geq \alpha q_j\,(d_j - c_j) \qquad \forall j \in \mathcal{J} \tag{4.22}$$

$$f_j \geq \beta q_j\,(c_j - d_j) \qquad \forall j \in \mathcal{J} \tag{4.23}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in \mathcal{J} \tag{4.24}$$

$$c_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.25}$$

$$f_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{4.26}$$

Objective (4.19) minimizes the total earliness/tardiness and setup costs. Constraint (4.20) prevents overlap between jobs. Constraint (4.21) sets the job completion time bounds. Constraints (4.22) and (4.23) track each job's earliness/tardiness cost. Finally, (4.24), (4.25), and (4.26) are the variable declarations.

4.2 A Constraint Programming Formulation

The base constraint programming model, denoted CP, is a basic, "off the shelf" model, that represents each job with an interval variable, $x_j \in \mathcal{J}$. To consider the sequence of jobs on the machine, we introduce a sequence variable u, which represents a sequence of interval variables. A formal definition of interval and sequence variables is found in Section 3.4. Informally, each interval variable has a start time, completion time, and duration. For the ETSS problem, our interval variables have a duration equivalent to the processing time of the job that it represents. Interval variable start and end times are determined by the constraint programming optimization engine, and thus are decision variables. The sequence variable is used to determine the order of our interval variables, or equivalently the sequence of jobs in the solution. Further, using a sequence variable allows us to enforce no overlap between interval variables by considering setup times.

We denote s as the setup time matrix from job i to j and S for the cost matrix. The model is then:

$$\min \sum_{j\in\mathcal{J}} \max\{\alpha\left(d_j - \texttt{End}\left(x_j\right)\right), \beta\left(\texttt{End}\left(x_j\right) - d_j\right)\} + \sum_{i\in\mathcal{J}} S\left(i, \texttt{TypeOfNext}\left(u, x_j, |\mathcal{J}|, |\mathcal{J}|\right)\right) \tag{4.27}$$

$$\text{s.t. } \texttt{Pres}\left(x_j\right) = 1 \qquad \forall j \in \mathcal{J} \tag{4.28}$$

$$\texttt{Length}\left(x_j\right) = p_j \qquad \forall j \in \mathcal{J} \tag{4.29}$$

$$\texttt{Start}\left(x_j\right) \geq r_j \qquad \forall j \in \mathcal{J} \tag{4.30}$$

$$\texttt{End}\left(x_j\right) \leq \mathbf{T} \qquad \forall j \in \mathcal{J} \tag{4.31}$$

$$\texttt{NoOverlap}\left(\{x_j : \forall j \in \mathcal{J}\}, s\right) \tag{4.32}$$

$$x_j \coloneqq \texttt{intervalVar} \qquad \forall j \in \mathcal{J} \tag{4.33}$$

$$u \coloneqq \texttt{sequenceVar}\left(\{x_0, \ldots, x_{|\mathcal{J}|}\}\right) \tag{4.34}$$

Objective (4.27) follows the same objective as the previous models. However, the setup costs are formulated using `TypeOfNext`, which determines the type of the next interval variable in our sequence. Each interval variable has its own type to model heterogeneous setup costs, meaning that each job has a unique set of transitions to other jobs. Using the max operator is valid in constraint programming. Constraint (4.28) ensures that each job is scheduled.

Constraint (4.29) sets each interval variable's length to its associated job's processing time. Constraint (4.30) enforces release dates. Constraint (4.31) is used to set an appropriate finite time horizon, following previous work on ETSS problems [60]. Constraint (4.32) enforces no overlap between jobs as explained in Chapter 3 Section 3.4. Finally, (4.33) and (4.34) declare our model variables.

4.3 Experimental Setup

All of the experiments in this thesis were conducted using the same experimental setup and were run on the SciNet Niagara Cluster. The MIP models were solved using CPLEX 20.1 and the CP model with CP Optimizer 20.1. All methods were coded in C++ except the client heuristic as mentioned in Section 2.3. The models were tested using 100 randomly generated instances and two industry client instances, following the discussion in Section 2.2.1. Each instance was run for a maximum of 30 minutes.

4.4 Numerical Results

In this section, we compare the proposed MIP and CP models using quantitative experiments. Graphs with error bars are located in Appendix A.

4.4.1 Mean Relative Error

As the MIP models and CP model use different optimization engines, Mean Relative Error (MRE) was used to compare the different methods. For each problem size, for example instances with 50 jobs, each model's MRE is calculated using the best-known solution for each instance across all of the methods in this thesis, using the formula below:

$$\text{MRE} = \frac{1}{N} \sum_{i=1}^{N} \frac{|(Z_i - Z_i^{best})|}{Z_i^{best}}$$

Where MRE is the Mean Relative Error for the selected method for the selected problem size, N is the number of instances in the selected problem size, Z_i is the selected method's solution for instance i in that problem size, and Z_i^{best} is the best-known solution for instance i across the all of the methods in this thesis.

From Figure 4.1, the client heuristic almost always finds the best solutions and dominates the other methods as the number of jobs increases. The base CP model remains competitive with the heuristic until it begins to struggle past 200 jobs. The MIP models have significantly worse performance that is

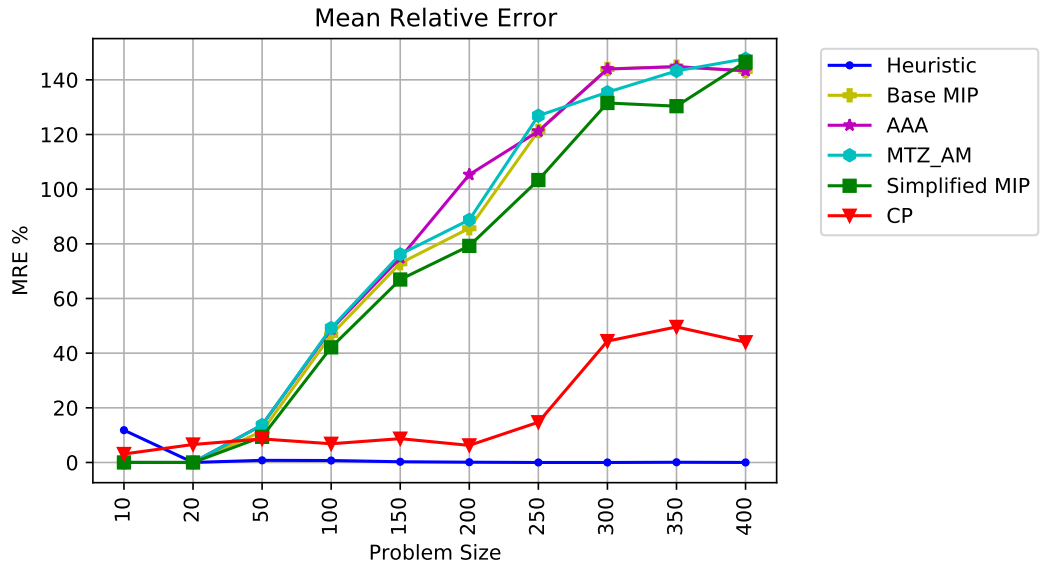


Figure 4.1: Mean Relative Error for MIP and CP models.

more impacted by job size. Of the MIP models, the Simplified MIP method performs the best.

The MRE suggests that the CP and Simplified MIP models are the most promising exact techniques thus far.

4.4.2 Average Run Time

As shown in Figure 4.2, all of the methods utilize the maximum time for medium sized instances. For smaller instances, the MIP models run much faster than CP, and both are faster than the heuristic.

4.4.3 Instances Solved

All of the MIP and CP models were able to find feasible solutions within the time limit.

4.4.4 MRE Over Time

To compare the solution quality over time, each method's objective was sampled every second for each instance. We then compute each method's MRE for every instance every second. The final MRE value at each second is the average of a method's MRE for each instance at that time point. For a selected method, MRE at time t (MRE_t) is:

$$\text{MRE}_t = \frac{1}{N}\sum_{i=1}^{N}\frac{|(Z_{i,t} - Z_i^{best})|}{Z_i^{best}}$$

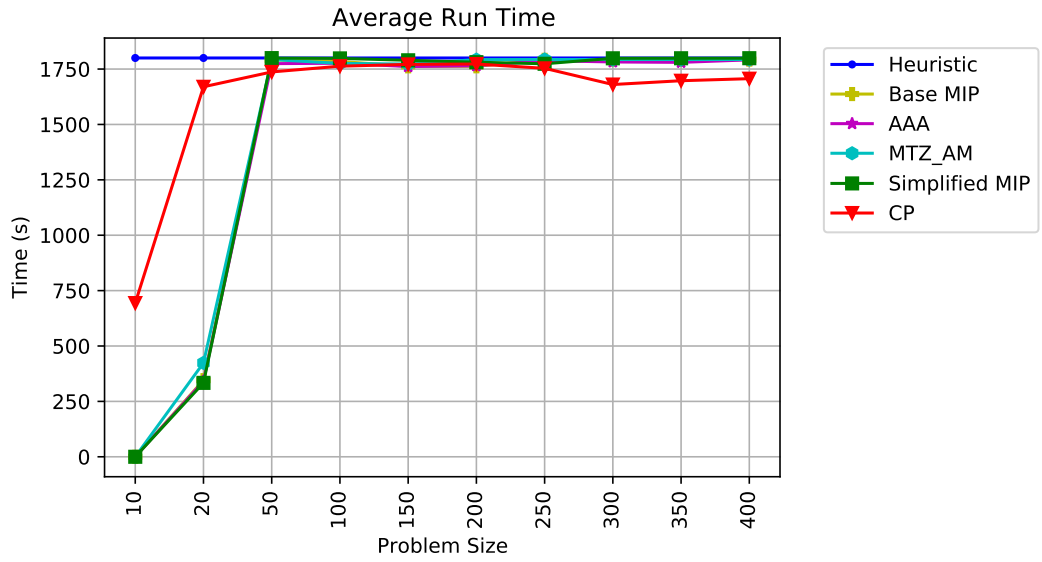


Figure 4.2: Average run time for MIP and CP models.

Where N is the number of instances with solutions at time t for the selected method, $Z_{i,t}$ is the selected method's objective value for instance i at time t, and Z_i^{best} is the best-known objective value for instance i across all of the thesis methods.

Figure 4.3 shows a rapid increase in solution quality early on, until MRE plateaus for the remaining time. Even with ample time remaining, none of the methods make significant improvements past the first quarter of the time limit. As expected from the MRE results, the heuristic and CP methods' plateaued quality is better than the MIP models.

4.4.5 MIP and CP Warm Start

As the Simplified MIP and CP model had difficulties improving solutions over time, both were tested with an initial solution from running the heuristic for 5 seconds. Warm starting these methods will demonstrate the search tree pruning effectiveness when given a strong starting upper bound. We denote the CP method with warm start as CP_warm and the Simplified MIP method with warm start as Simplified MIP_warm, and compare them to no warm start and the heuristic in Figure 4.4. The benefit of warm starting with the heuristic is evident in the large reduction of MRE for both warm start methods. Figure 4.5 shows that MIP_warm is not as competitive with CP_warm. These large reductions in MRE demonstrate the importance and impact of finding a good initial solution for both models. Further, it highlights MIP and CP's ability to be competitive with the client heuristic in this environment.

In Figure 4.6, we show the MRE over time for the warm start methods. CP_warm has better solution quality improvement over time compared to Sim-

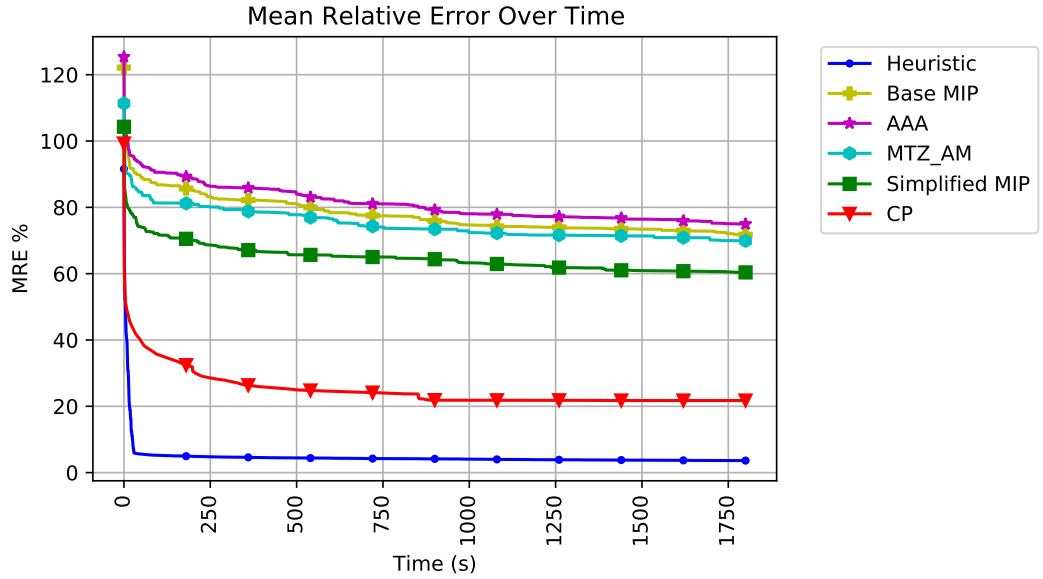


Figure 4.3: Mean Relative Error over time.

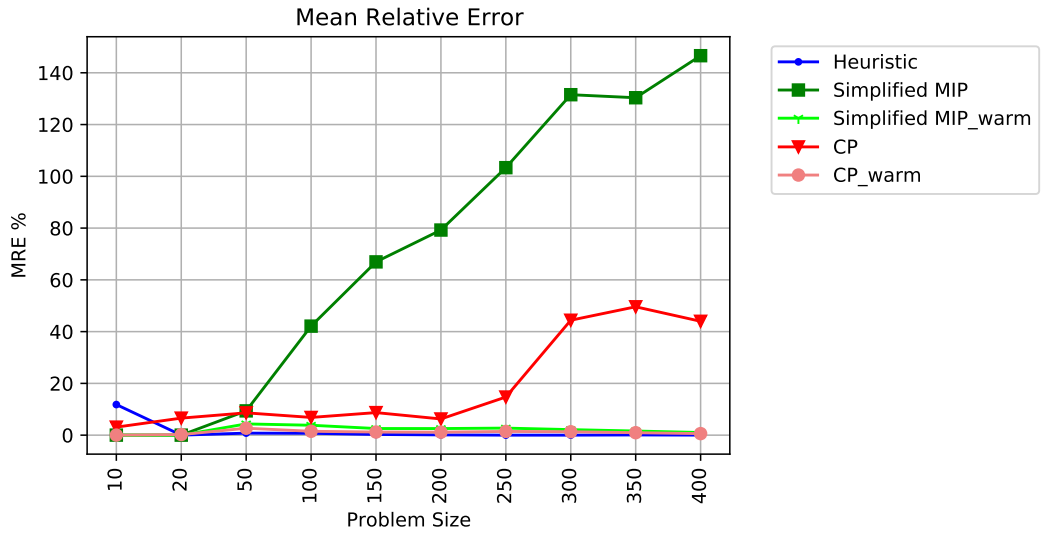


Figure 4.4: Mean Relative Error for Simplified MIP and CP with heuristic warm start and non-warm start.

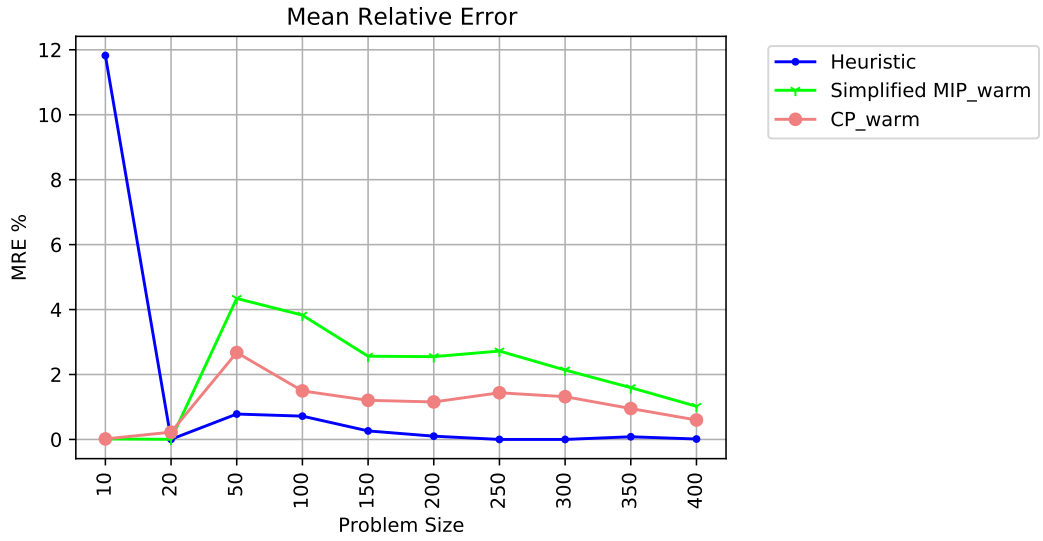


Figure 4.5: Mean Relative Error for Simplified MIP and CP with heuristic warm start.

plified MIP_warm, similar to CP and Simplified MIP. Interestingly, we observe little improve with time at all for Simplified MIP_warm, whereas CP_warm continues to have large improvements for the majority of the time limit.

4.4.6 CP Objective Function Propagation

One hypothesis with the CP model is that its complex objective function decreases the solver's ability to propagate changes in the objective function to changes in the interval variable domains. Constraint programming performance critically depends on this ability as it determines variable domain pruning whenever a new solution is found. To investigate this propagation, the CP model was compared against another CP model which only differed in its objective function. The latter model, denoted as CP_cmax, minimizes makespan subject to the same constraints as the base CP model. The CP_cmax objective function is: $max\{\text{End}\,(x_j)\}$, contrasted to the CP objective function in (4.27). Thus, the sole difference between these models is their objective function which would result in different propagation to the decision variables.

To measure the level of back-propagation, or search space reduction, independent of the lower bound quality for CP and CP_cmax, we begin with an initial propagation. Then we post an upper bound constraint and propagate to measure the reduction in search space via the change in size of the variable domain store. The posted constraint is as follows:

$$obj\,(i) \leq opt\,(i) - \epsilon$$

Where $obj\,(i)$ is the objective function for instance i, $opt\,(i)$ is the optimal solution to instance i, and ϵ is a very small number. As we have two different

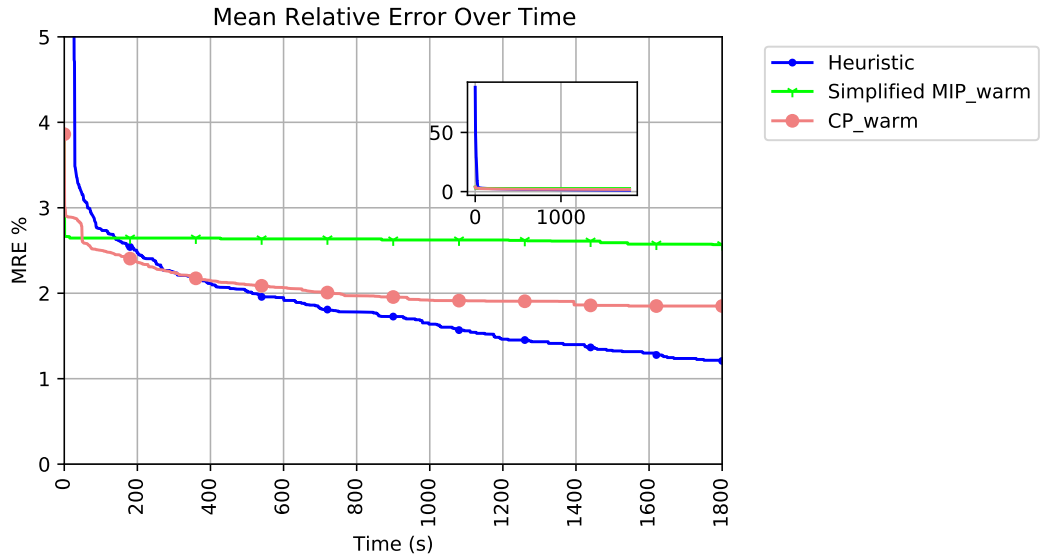


Figure 4.6: Mean Relative Error over time for Simplified MIP and CP with heuristic warm start and client heuristic.

objective functions, each model independently has this constraint posted and we use each model's respective objective function and the optimal value of the associated instance. The search space percentage reduction is calculated as $\frac{||dom|-|dom_p||}{|dom_p|}$, where $|dom|$ is the cardinality of the domain store after the upper bound propagation and $|dom_p|$ is the domain store cardinality prior.

We obtained the optimal solutions to all 10-job instances for both the ETSS and the makespan objective functions. On average, CP_cmax had a search space reduction of 88.3% after posting the optimal upper bound. Conversely, CP had an average search space reduction of 0.2%. Thus, the ETSS objective function propagates quite poorly, especially given that we provided optimal upper bounds to use for propagation.

4.4.7 Root Node Lower Bound Analysis

Here we compare the root node lower bound strength between the best MIP model, Simplified MIP, and CP. The CP model obtains its lower bound via a combination of inference through constraint propagation and a linear programming relaxation [52, 51]. This process reduces the domains of variables and thus the possible assignment combinations. At the root node, no branching has been done and thus no backtracking is utilized either. We measure the Mean Relative Error of the root node lower bound from the best-known solution:

Figure 4.7: Root node lower bound deviation from best-known solution.

$$\text{MRE}_{\text{root}} = \frac{1}{N} \sum_{i=1}^{N} \frac{|(\text{LB}_i - \text{Z}_i^{best})|}{\text{Z}_i^{best}}$$

Where MRE_{root} is the Mean Relative Error for the root Lower Bound (LB) for the selected method for the selected problem size, N is the number of instances for that problem size, LB_i is the selected method's root node LB for instance i in that problem size, and Z_i^{best} is the best-known solution for instance i across the various methods.

We investigate root node lower bounds as a strong lower bound is critical for effective pruning in the search tree for both constraint and mathematical programming. In Figure 4.4 we showed the impact of a strong upper bound via heuristic warm start and now look to the effectiveness of the current lower bounds. A method with a better root node lower bound will have a lower MRE score.

Interestingly, Figure 4.7 demonstrates that the Simplified MIP finds very good root node lower bounds for smaller instances but worsens as problem sizes increase. However, the Simplified MIP model always finds better root node lower bounds than CP. Although, the CP model is consistently faster at finding root bounds as shown in Figure 4.8, suggesting that the MIP root lower bound via linear relaxation is preferred for smaller problem sizes, while CP's slightly worse but faster root bound is preferred for larger instances. Consequently, these differences in root node bounds may explain the difference in performance between CP and MIP in Figure 4.1.

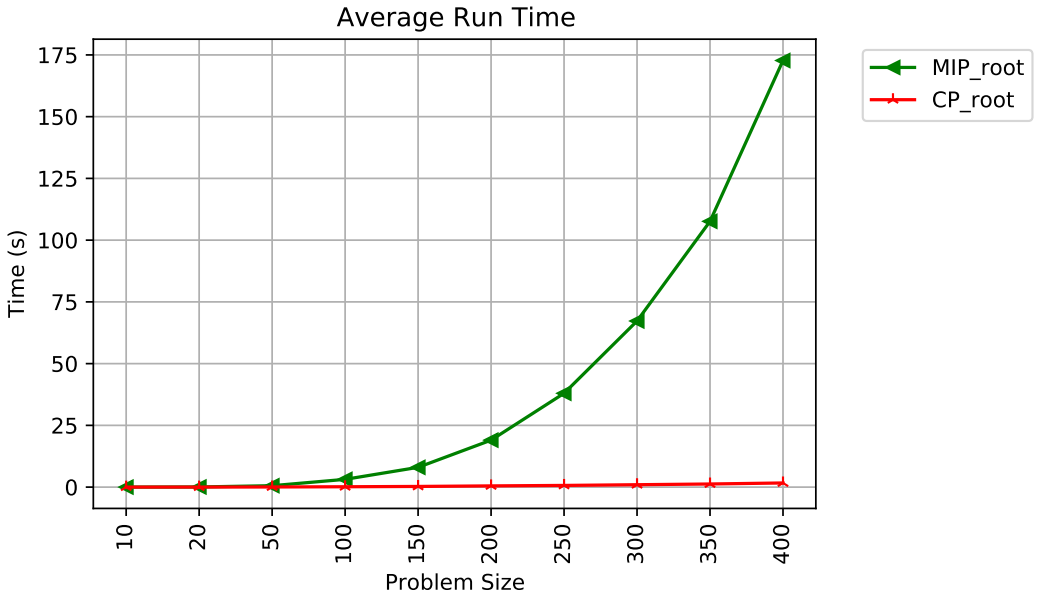


Figure 4.8: Time to obtain root node lower bound.

4.5 Discussion

Of the MIP and CP models presented in this chapter, the Simplified MIP and CP models perform the best with regards to solution quality. However, they have a significant performance gap with the client heuristic; although, the CP model is more competitive than the Simplified MIP. All of the methods struggle to improve solution quality over time after an initial strong performance. Thus, finding a good initial solution is important for improving search efficiency, as shown in the warm start experiments. Further, the proposed constraint programming model struggles with variable domain propagation compared to a model with a makespan minimization objective. Finally, the Simplified MIP model finds better root node lower bounds than CP, however it takes significantly longer as problem size increases. Thus, when considering time, CP provides a better root node bound trade off for larger instances.

Additionally, state-of-the-art MIP models for minimizing makespan on a single machine with setups are not competitive for this problem.

Chapter 5 addresses CP's weak propagation and looks at improving CP's lower bounds via decision diagrams. Chapter 6 then turns to Large Neighbourhood Search with diversification methods to reduce solution quality plateauing, along with the impact of warm starting via the client heuristic.

Chapter 5

DDs for Earliness Tardiness Scheduling with Setups

In this chapter we investigate using Multivalued Decision Diagrams (MDD) in combination with our base Constraint Programming (CP) model developed in Chapter 4. Recall that the base CP model was the best exact technique for solving the Earliness Tardiness Scheduling with Setups (ETSS) problem. However, the model struggled with finding strong root node lower bounds and propagating variable domain changes with its objective function. Integrating decision diagrams with our base CP model aims at improving these two short comings. As discussed in the literature review (see Section 3.5), decision diagrams have been successfully applied to scheduling problems, providing lower bounds through solving the shortest path problem over a decision diagram. Further, the constraint programming optimization engine is able to propagate changes in decision variable domains to the decision diagram and vice versa.

We develop a novel approach that combines the MDD approach with a timing algorithm, addressing the potential lower bound weakness of the decision diagram when it is used for our complex objective function. Given a sequence of jobs, this timing algorithm efficiently determines job completion times to minimize earliness/tardiness costs.

Finally, we compare and discuss the numerical results for our initial decision diagram approach and one with the timing algorithm. We include an analysis against the best performing methods from the previous chapter: the client heuristic, the Simplified Mixed Integer Programming (MIP) model, and the CP model. Neither MDD model was competitive with Simplified MIP, CP, and the heuristic, as they were unable to find feasible solutions to the larger instances in the time limit. However, for smaller instances the MDD methods were able to outperform CP in solution quality, time, and root node bound strength.

5.1 Decision Diagram Formulations

We formulate a relaxed MDD for the ETSS problem. By computing the shortest path in the relaxed decision diagram, we obtain a valid lower bound on its objective function. Informally, the decision diagram formulation focuses on the job sequence in our problem. Each layer of the decision diagram represents a position within the overall job sequence. That is, layer i represents position i in the job sequence. Thus, in each layer, arcs represent a feasible job assignment to the corresponding position in the sequence. Consequently, all of the paths from root node to terminal node in the decision diagram represent feasible job sequences. For each arc, we assign a label denoting the job that the arc represents. As well, each arc has a cost, stemming from the cost that would occur if that arc was used in its layer. Or equivalently, the cost that would occur if the job associated with that arc's label was in the layer's position within the job sequence.

Unfortunately, as the decision diagram focuses on job sequence, we do not have fixed completion times for the jobs on any given root-to-terminal path. As a result, the costs associated with each arc are lower bounds as opposed to true costs: we can track the setup times using parent arcs, but without completion times, we can only calculate earliness/tardiness cost bounds using earliest and latest completion times of the arc.

Recall that, generally, an exact decision diagram yields an optimal solution for its corresponding optimization problem [9]. However, for the ETSS problem, the decision diagram only determines the sequence of jobs and not the completion times, thus yielding only a lower bound, even if job sequence is represented exactly. However, we can propagate using the lower bound and infer the earliest and latest completion times of all jobs.

Using this mathematical framework, we then relax the decision diagram regarding job sequence, which provides a weaker lower bound and variable domain propagation. Relaxing the decision diagram provides a trade off between bound and inference strength and computational efficiency [9].

5.1.1 An Exact Decision Diagram For Job Sequence

For the ETSS problem, we utilize an MDD which represents all of the possible job permutations. Thus, the MDD is exact for job sequence; however, it is not exact for the objective function as we use earliness/tardiness and setup cost lower bounds. That is, a path from root to terminal will represent a feasible job sequence in an exact diagram. The explanation of the notation used for this MDD formulation is in Section 3.5. Formally, our MDD is represented as a layered directed acyclic multigraph $G = (V, A)$, where V is a set of nodes and A is a set of arcs. We utilize a Multivalued Decision Diagram to permit an out-degree larger than two for the nodes in G. Each layer of the MDD represents a position within the ETSS solution sequence. For example, the

first layer corresponds to the first sequence position. Then, each arc represents sequencing a job in the corresponding layer of that arc. To map jobs to arcs, we create a value, $v_a \in \mathcal{J}$, for each arc $a \in A$, where v_a is the job that arc a represents. Then, an arc-specified path $(\mathrm{a}_1, \ldots, \mathrm{a}_\mathrm{n})$ from the root node, $\mathbf{r}$, to terminal node, $\mathbf{t}$, identifies the ordering $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n)$, where $\pi_i = v_{a_i}$ for $i = 1, \ldots, n$, in other words a sequence of jobs: $\boldsymbol{\pi}$.

In an exact decision diagram, every feasible ordering is identified by some path from $\mathbf{r}$ to $\mathbf{t}$ in G, and conversely every path from $\mathbf{r}$ to $\mathbf{t}$ identifies a feasible job sequence. Sequence feasibility is ensured by using an `AllDifferent` global constraint over $\boldsymbol{\pi}$, which prevents any duplicate values within $\boldsymbol{\pi}$. Figure 5.1 shows a decision diagram for an instance with three jobs, where there are three layers representing each position in the job sequence. For example the path $(\mathbf{r}, u_1), (u_1, u_4), (u_4, \mathbf{t})$ represents the job sequence $1 \rightarrow 2 \rightarrow 3$.

We determine a lower bound on the cost associated with each arc $a = (u, v)$ in G, where u is the parent node and v is the child node for a, such that the sum of these costs along an $\mathbf{r} - \mathbf{t}$ path will be a valid lower bound for our problem. To do so, we define the following arc states: earliest completion time ect_a and latest completion time lct_a, $\forall a = (u, v) \in A$, where:

- $ect_a = max\{r_{v_a}, min\{ect_{a'} + s_{v_{a'}, v_a} : a' \in in(u)\}\} + p_{v_a}$, where r_{v_a} is the release time of job v_a, $s_{v_{a'}, v_a}$ is the setup time from job $v_{a'}$ to job v_a, $in(u)$ denotes the set of arcs that enter node u, and p_{v_a} is the processing time of job v_a. If $a \in out(\mathbf{r})$, $ect_a = r_{v_a} + p_{v_a}$, where $out(\mathbf{r})$ denotes the set of arcs that leave the root node.
- $lct_a = max\{lct_{a'} - p_{v_{a'}} - s_{v_a, v_{a'}} : a' \in out(v)\}$. If $a \in in(\mathbf{t}), lct_a = \mathbf{T}$, where $\mathbf{T}$ denotes the instance time horizon: $\mathbf{T} = \max_i d_i + \sum_i p_i + \sum_i \max_j s_{ij}$. The calculated time horizon is a valid upper bound on the makespan of an optimal schedule for the ETSS problem [60].

We define the length l_a or cost associated with each arc $a = (u, v) \in A$ using the arc states defined above, such that the shortest path in G from $\mathbf{r}$ to $\mathbf{t}$ represents a valid lower bound on our ETSS objective function:

$$l_a = q_{v_a}\, max\{0, \alpha(d_{v_a} - lct_a), \beta(ect_a - d_{v_a})\} + min\{S_{v_a, v_{a'}} : a' \in out(v)\}$$

where q_{v_a} is the quantity of job v_a, α is the earliness penalty coefficient, β is the tardiness penalty coefficient, d_{v_a} is the due date of job v_a, and $S_{v_a, v_{a'}}$ is the setup cost between job v_a and job $v_{a'}$.

Once our decision diagram is created, its lower bound will be the shortest $\mathbf{r} - \mathbf{t}$ path. Within the ETSS problem, the shortest path will represent a sequence of jobs in the given instance.

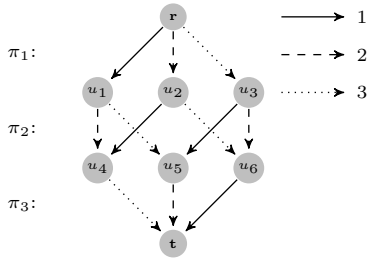


Figure 5.1: A decision diagram for a three job instance.

Table 5.1: A three job example instance.

(a) Instance

Job	r_j	d_j	p_j
1	0	3	1
2	4	7	6
3	3	2	5

(b) Setup times

	1	2	3
1	0	1	4
2	6	0	7
3	3	5	0

5.1.2 An Example

We introduce a simple, three job example in the decision diagram framework. For this example, we set the earliness (α) and tardiness (β) coefficients to one and set all job quantities (q_j) to one for simplicity. Further, we ignore setup costs as they are not required to demonstrate the formulation. Our three job instance is in Table 5.1.

Before formulating the decision diagram, the time horizon is calculated: $\mathbf{T} = \max_i d_i + \sum_i p_i + \sum_i \max_j s_{ij} = 7 + (1 + 6 + 5) + (4 + 7 + 5) = 35$. Next, we compute the decision diagram arc states which are earliest completion time and latest completion time. The earliest completion time is propagated from the root node down, whereas the latest completion time is propagated from the terminal node upwards. Once these two numbers are determined for each arc in the decision diagram, we can calculate each arc cost. We perform the calculations in Table 5.2 and show the associated decision diagram in Figure 5.2. We label each arc in the decision diagram with its cost and determine the shortest path in the decision diagram. The shortest path is $(\mathbf{r}, u_1), (u_1, u_5), (u_5, \mathbf{t})$ with length 20, which represents the job sequence $1 \rightarrow 3 \rightarrow 2$. Thus, our lower bound for this instance is 20. Optimally solving this instance yields a value of 24.

In addition to this example, we note that the decision diagram lower bound on arc cost can be zero for an arc if its *ect* and *lct* time window is not sufficiently tight. That is, if for job j: $ect_j \leq d_j \leq lct_j$ then the lower bound will be zero.

Table 5.2: Decision diagram calculations for two job instance in Table 5.1.

(a) Earliest Completion Times

Arc	*ect*
$(\mathbf{r}, u_1)$	$r_1 + p_1 = 0 + 1 = 1$
$(\mathbf{r}, u_2)$	$4 + 6 = 10$
$(\mathbf{r}, u_3)$	$3 + 5 = 8$
(u_1, u_4)	$max\{r_2, ect_{\mathbf{r},u_1} + s_{1,2}\} + p_2 = max\{4, 1 + 1\} + 6 = 10$
(u_1, u_5)	$max\{3, 1 + 4\} + 5 = 10$
(u_2, u_4)	$max\{4, 10 + 6\} + 1 = 17$
(u_2, u_6)	$max\{3, 10 + 7\} + 5 = 22$
(u_3, u_5)	$max\{0, 8 + 3\} + 1 = 12$
(u_3, u_6)	$max\{4, 8 + 5\} + 6 = 19$
$(u_4, \mathbf{t})$	$max\{r_3, min\{ect_{u_1,u_4} + s_{2,3}, ect_{u_2,u_4} + s_{1,3}\}\} + p_3 = 22$
$(u_5, \mathbf{t})$	$max\{4, min\{10 + 5, 12 + 1\}\} + 6 = 19$
$(u_6, \mathbf{t})$	$max\{0, min\{22 + 3, 19 + 6\}\} + 1 = 26$

(b) Latest Completion Times

Arc	*lct*
$(\mathbf{r}, u_1)$	$max\{lct_{u_1,u_4} - p_2 - s_{1,2}, lct_{u_1,u_5} - p_3 - s_{1,3}\} = max\{23 - 6 - 1, 24 - 5 - 4\} = 16$
$(\mathbf{r}, u_2)$	$max\{26 - 1 - 6, 31 - 5 - 7\} = 19$
$(\mathbf{r}, u_3)$	$max\{28 - 1 - 3, 28 - 6 - 5\} = 24$
(u_1, u_4)	$lct_{u_4,\mathbf{t}} - p_3 - s_{2,3} = 35 - 5 - 7 = 23$
(u_1, u_5)	$35 - 6 - 5 = 24$
(u_2, u_4)	$35 - 5 - 4 = 26$
(u_2, u_6)	$35 - 1 - 3 = 31$
(u_3, u_5)	$35 - 6 - 1 = 28$
(u_3, u_6)	$35 - 1 - 6 = 28$
$(u_4, \mathbf{t})$	$\mathbf{T} = 35$
$(u_5, \mathbf{t})$	35
$(u_6, \mathbf{t})$	35

(c) Arc Costs

Arc	*Cost*
$(\mathbf{r}, u_1)$	$max\{0, d_1 - lct_{\mathbf{r},u_1}, ect_{\mathbf{r},u_1} - d_1\} = max\{0, 3 - 16, 1 - 3\} = 0$
$(\mathbf{r}, u_2)$	3
$(\mathbf{r}, u_3)$	6
(u_1, u_4)	3
(u_1, u_5)	8
(u_2, u_4)	14
(u_2, u_6)	20
(u_3, u_5)	9
(u_3, u_6)	12
$(u_4, \mathbf{t})$	20
$(u_5, \mathbf{t})$	12
$(u_6, \mathbf{t})$	23

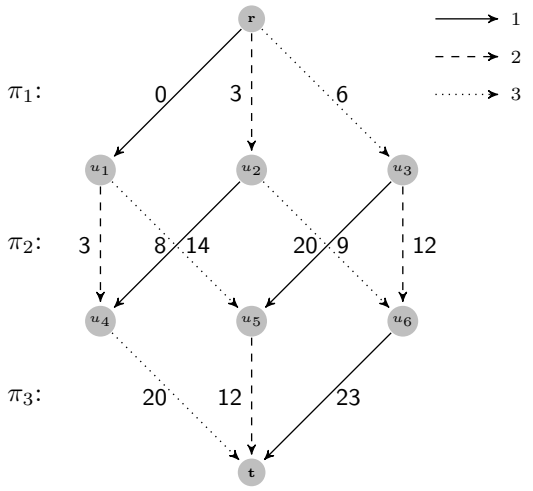


Figure 5.2: A decision diagram formulation for the instance in Table 5.1.

5.1.3 A Relaxed Decision Diagram

In practice, relaxed decision diagrams are often utilized due to computational resource limits [9]. Relaxed decision diagrams over-approximate the feasible solutions and objective function bound of the problem. In the case of the ETSS problem, our lower bound will be weaker when using a relaxed decision diagram. The relaxation is controlled by the decision diagram's width parameter: the maximum width of all its layers. Recall, that a layer's width is the number of nodes in it. Thus, a smaller width parameter results in a weaker relaxation but uses fewer resources.

To create a relaxed decision diagram for the ETSS problem, we adjust the creation procedure to prevent the model's width from exceeding our parameter. This process is denoted as incremental refinement and consists of two phases: filtering and refinement. Filtering removes arcs where all paths that cross them violate a constraint. For the ETSS MDD formulation, we are concerned with the `AllDifferent` constraint. Refinement consists of splitting nodes to strengthen the decision diagram formulation. The filtering and refinement procedures were developed in *Decision Diagrams for Optimization* [9].

5.1.4 Filtering

We wish to identify the paths crossing an arc that always assign some job more than once, that is, paths that violate the `AllDifferent` constraint. Then, this arc is infeasible. For each node in the decision diagram, $u_i, i \in V$, we define the node state as the jobs that were previously performed on the machine. For example, $u_{\mathbf{r}} = \{\emptyset\}$ and $u_{\mathbf{t}} = \{1, \ldots, n\}$. To strengthen checking arc feasibility, we introduce an additional redundant state that provides a sufficient condition to remove arcs [9]. This state tracks the jobs that might have been performed up to a node. We denote $\text{All}_u^{\downarrow} \subseteq \mathcal{J}$ as the set of arc labels that appear in

all paths from the root node $\mathbf{r}$ to node u and $\text{Some}_u^{\downarrow} \subseteq \mathcal{J}$ as the set of arc labels that appear in some path from the root node $\mathbf{r}$ to node u. Further, $\text{All}_{\mathbf{r}}^{\downarrow} = \text{Some}_{\mathbf{r}}^{\downarrow} = \emptyset$. Recall that $in(v)$ is the set of incoming arcs at a node v. Then, the states $\text{All}_v^{\downarrow}$ and $\text{Some}_v^{\downarrow}$ for some node $v \neq \mathbf{r}$ can be recursively calculated as follows:

- $\text{All}_v^{\downarrow} = \bigcap_{a=(u,v)\in in(v)} \left(\text{All}_u^{\downarrow} \cup \{v_a\}\right)$
- $\text{Some}_v^{\downarrow} = \bigcup_{a=(u,v)\in in(v)} \left(\text{Some}_u^{\downarrow} \cup \{v_a\}\right)$

Then, an arc $a = (u, v)$ is infeasible, via top down filtering, if either of the following conditions hold:

$$v_a \in \text{All}_u^{\downarrow} \tag{5.1}$$

$$|\text{Some}_u^{\downarrow}| = l(a) \text{ and } v_a \in \text{Some}_u^{\downarrow} \tag{5.2}$$

Where $l(a)$ denotes the layer arc a is in.

Proof ([3]). Let π' be any partial ordering identified by a path from $\mathbf{r}$ to node u that does not assign any job more than once. In condition 5.1, $v_a \in \text{All}_u^{\downarrow}$ indicates that v_a is already assigned to some position in π', thus adding the arc label v_a to π' will create a duplicate. For condition 5.2, the paths from $\mathbf{r}$ to u are comprised of $l(a)$ arcs, and as such π' represents an ordering with $l(a)$ positions. If $|\text{Some}_u^{\downarrow}| = l(a)$, then any $j \in \text{Some}_u^{\downarrow}$ is already assigned to some position in π': adding v_a to π' creates a repetition.

In addition to top down filtering, bottom up filtering can be used to provide stronger results. The bottom up approach uses two additional states $\text{All}_u^{\uparrow} \subseteq \mathcal{J}$ and $\text{Some}_u^{\uparrow} \subseteq \mathcal{J}$ for each node. These states are computed with respect to paths from $\mathbf{t}$ to u instead of $\mathbf{r}$ to u. Again, they can be computed recursively:

- $\text{All}_u^{\uparrow} = \bigcap_{a=(u,v)\in out(u)} \left(\text{All}_v^{\uparrow} \cup \{v_a\}\right)$
- $\text{Some}_u^{\uparrow} = \bigcup_{a=(u,v)\in out(u)} \left(\text{Some}_v^{\uparrow} \cup \{v_a\}\right)$

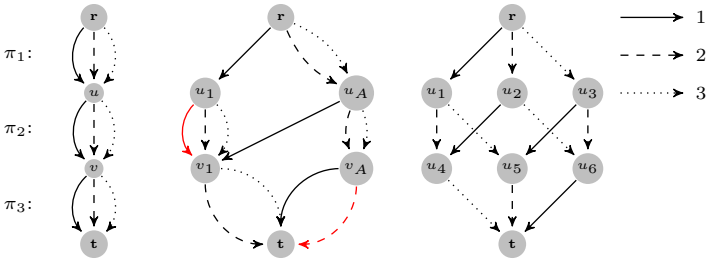
Then, an arc $a = (u, v)$ is infeasible if either of the following conditions holds:

$$v_a \in \text{All}_v^{\uparrow} \tag{5.3}$$

$$|\text{Some}_v^{\uparrow}| = n - l(a) \text{ and } v_a \in \text{Some}_v^{\uparrow} \tag{5.4}$$

$$|\text{Some}_u^{\downarrow} \cup \{v_a\} \cup \text{Some}_v^{\uparrow}| < n \tag{5.5}$$

Proof. The proofs for conditions 5.3 and 5.4 follow from [31] and are analogous to the proof for conditions 5.1 and 5.2. Condition 5.5 implies that any



(a) Initial relaxed MDD. (b) Job 1 is exact. (c) Exact MDD.

ordering identified by a path containing arc a will never assign all of the jobs in $\mathcal{J}$.

5.1.5 Refinement

The refinement procedure splits nodes to remove paths that have infeasible solutions, thus strengthening the relaxed decision diagram. For the `AllDifferent` constraint, the refinement procedure follows a heuristic approach that prioritizes job accuracy based on job priority. That is, a more accurately represented a job is closer to being exactly represented in the decision diagram, i.e. these jobs are not repeated in any sequence encoded by the decision diagram. For the ETSS problem, we define job priority as the job's quantity, as that feature is job dependent and directly impacts earliness/tardiness costs. We denote job priority by ranking the jobs in $\mathcal{J}^* = \{j_1^*, \ldots, j_n^*\}$, where jobs with a smaller index in $\mathcal{J}^*$ have a higher priority. Then, the job quantities in $\mathcal{J}^*$ would follow $q_{j_i^*} \geq q_{j_k^*}$ for $i < k,\ i, k \in \{1, \ldots, n\}$.

To start the refinement procedure, we require a relaxed decision diagram as input, which for the first iteration can be a trivial width-one decision diagram. For example, Figure 5.3a shows a width-one relaxation for a 3-job ETSS instance. Notice the violation of the `AllDifferent` constraint. For a given layer, L_i, the refinement procedure is as follows: for each job $j^* \in \mathcal{J}^*$, identify the nodes u such that $j^* \in \text{Some}_u^{\downarrow} \setminus \text{All}_u^{\downarrow}$. Then, split each such node into two nodes u_1 and u_2, where an incoming arc $a = (v, u)$ is redirected to u_1 if $j^* \in \left(\text{All}_v^{\downarrow} \cup \{v_a\}\right)$ or u_2 otherwise, and replicate all outgoing arcs for both nodes. This procedure is repeated for each layer in the decision diagram. If a given layer's width is at the maximum width of the decision diagram then no refinement occurs. Further, if a job is represented exactly in a layer, that is, there is no violation of the `AllDifferent` constraint, then no refinement is run for that job. Figure 5.3b shows three phases of refinement for a 3-job instance. Jobs are prioritized lexicographically, which is why job 1 becomes exact in the middle diagram. As well, the red arcs in Figure 5.3b are infeasible and thus can be filtered (removed). Finally, 5.3c is a width-3 decision diagram and it is exact.

5.2 Bound Strengthening Via Timing Algorithm

As previously mentioned, the decision diagram formulation only provides the earliest and latest completion times of jobs, resulting in an objective value bound. However, given a sequence of jobs from the decision diagram, we can efficiently calculate their optimal completion times using a timing algorithm following the work of Hendel and Sourd [30]. We run a $\mathcal{O}(m \log n)$ timing algorithm, where m denotes the sum of the number of piecewise linear costs segments and n is the number of jobs. In the case of weighted earliness and tardiness costs, $m = 2n$. Thus, Hendel and Sourd's timing algorithm is more computationally efficient than generating and solving a linear program for the timing problem.

Given a sequence of n jobs, $(J_1, \ldots, J_n)$, we wish to schedule them following their index order. Each job has a convex piecewise linear cost function, denoted as $f_i(c_i)$, which uses the completion time, c_i, of job J_i. For the ETSS problem, this cost function is earliness/tardiness. Hendel and Sourd's algorithm uses the variable x_i, which denotes the amount of cumulative idle time before starting job J_i, leading to an order among variables: $x_1 \leq x_2 \leq \ldots x_n$. In the paper, setup times are not considered, yielding a convex piecewise linear cost function: $g_i(x_i) = f_i\left(x_i + \sum_{j=1}^{i} p_j\right)$. To account for setup times, we modify the cost function to be: $g_i(x_i) = f_i\left(x_i + \sum_{j=1}^{i} p_j + \sum_{j=1, i>1}^{i} s_{j,j+1}\right)$.

For each cost function g_i, we store a list, b_i, of their breakpoints and corresponding slopes, which is dependent on the job's location in the sequence. To calculate b_i, we use Algorithm 7. This algorithm takes the parameters s and $\mathbf{T}$, where $s = \sum_{i=1}^{n} max_j\{\text{slope}_{ij}\}$ where slope_{ij} is the slope of the jth segment of f_i, and $\mathbf{T}$ is the time horizon for the given instance. Within the algorithm, a job will have either two or three breakpoints. The first breakpoint occurs at the job's earliest start time, the second only occurs if a job can be early, and the third breakpoint is at the time horizon. Additionally, we scale the data and round it to become integer.

Informally, at the ith iteration of the algorithm the job in position i is scheduled at time infinity. The job is then left-shifted until it is either at a time with minimum cost or it encounters a block of jobs with no idle time between them and is merged with them. The process continues with this block until it is timed with a minimum cost or it meets a preceding block. Thus, after the ith iteration, the schedule is optimal for the first i jobs.

More formally, the algorithm uses a reversed list per cost function, b_i; the list contains the breakpoints sorted from the last one to the first one. As well, we denote $g_i'(\infty)$ as the slope of the last segment in the i cost function. For the ETSS problem, $g_i'(\infty) = s, \forall i \in \{1, \ldots, n\}$. Sourd and Hendel use a unique heap to store all events, where an event is a job that can be adjusted in the current schedule in the current iteration. This heap is sorted so that the event with the maximal time, $t = t_j^e$ is at the top of the heap. An event

Algorithm 7 Initialize cost function breakpoints and slopes

procedure GENERATEBREAKPOINTANDSLOPE(s, $\mathbf{T}$)
  Initialize $b, min_{idle} = 0$
  for $i = 1; i \leq n$ **do**
    Calculate sum processing and setup times before $J_i \rightarrow y$
    if $y < r_i$ **then**
      $min_{idle} = max\{min_{idle}, r_i - y\}$
    b_i.push_back(min_{idle}, ∞)
    $\text{slope}_t = \beta q_i$
    if $min_{idle} + y + p_i < d_i$ **then**
      $\text{slope}_e = -\alpha q_i$
      b_i.push_back($d_i - p_i - y, \text{slope}_t - \text{slope}_e$)
    b_i.push_back($\mathbf{T}, s - \text{slope}_t$)
  return b

is a 4-tuple (t, j, k, m), where $t = t_j^e$ is typically the maximal of a breakpoint for cost function j, k is the index of the first job of the block, and m is the modifier for the associated breakpoint. Algorithm 8 shows how to insert J_i into the schedule and it is run n times for $J_1, \ldots, J_n$. After all of the jobs are inserted into the schedule, Algorithm 9 computes the idle time before each job by emptying the heap. Finally, with the amount of idle time between each job, x_i determined, we can compute the completion time of each job.

Algorithm 8 Insert J_i into the schedule

Initialize the current block: $\sigma \leftarrow g_i'(\infty)$ and $f \leftarrow i$
Let (t^e, m^e) be the last breakpoint of b_i
b_i.pop_back()
$j \leftarrow i$
while $\sigma \geq 0$ **do**
  Insert the event (t^e, j, f, m^e) in the heap
  Let (t, j, k, m) be the event removed from the top of the heap
  $\sigma \leftarrow \sigma - m$
  $f \leftarrow min\{f, k\}$
  Let (t^e, m^e) be the next event in the reverse list of b_j
  b_j.pop_back()
Insert the event $(t, j, f, -\sigma)$ in the heap

Hendel and Sourd's timing algorithm was used instead of the simple timing rule used in the client heuristic, Section 2.3, as it does not assume that a job's tardiness costs will be strictly larger than its earliness costs.

5.2.1 An Example With Strengthened Bounds

We return to our example in Section 5.1.2, instead utilizing a decision diagram formulation that includes using the timing algorithm on complete paths in the decision diagram. Previously, our lower bound was 20 and the optimal

Algorithm 9 Post Processing

```
1: i ← n
2: while i > 0 do
3:     Let (t, j, f, m) be the event removed from the top of the heap
4:     while i ≥ f do
5:         x_i = t
6:         Decrease i
```

objective was 24. If we consider the paths in our decision diagram, we can run the timing algorithm on them to optimally determine each job's completion time. For $1 \rightarrow 3 \rightarrow 2$, job 1 finishes at time 1, job 3 at 10, and job 2 at 21, with a cost of 24 (optimal). Now the shortest path in our decision diagram provides a lower bound that is the optimal objective value. In this example, we ran the timing algorithm on all paths in the decision diagram. However, running the algorithm on all paths in the implementation is not necessarily required. As the timing algorithm requires a full sequence of jobs, we only run it in the CP search tree if the sequence of interval variables is fixed, which occurs at the leaf nodes. In the worse-case scenario, assuming no node pruning, the timing algorithm would run once for every leaf node. For an instance with n jobs, it would run $n!$ times. However, with pruning we can expect the timing algorithm to run less often. Further, running the timing algorithm strengthens our overall objective lower bound as it provides the earliness/tardiness costs for a given path, which is used for additional pruning.

5.3 Implementation

The decision diagram formulation was integrated with the base CP model in C++ as a global constraint. Consequently, the constraint programming optimization engine is able to utilize the bounds from the decision diagram and propagate variable domain information between its domain store and the decision diagram. We use a relaxed decision diagram with a fixed width of 1000, referring to this method as MDD. Further, the decision diagram with timing algorithm was developed and is denoted as MDD_timing. For both models we set the constraint programming engine to use depth first search to reduce memory usage over other search strategies.

5.4 Numerical Results

We compare the proposed MDD and MDD_timing methods using quantitative experiments, following the experimental setup defined in Section 4.3. Additionally, we compare with the previous best performing methods: the client heuristic, Simplified MIP, and CP. Graphs with error bars are located in Appendix B.

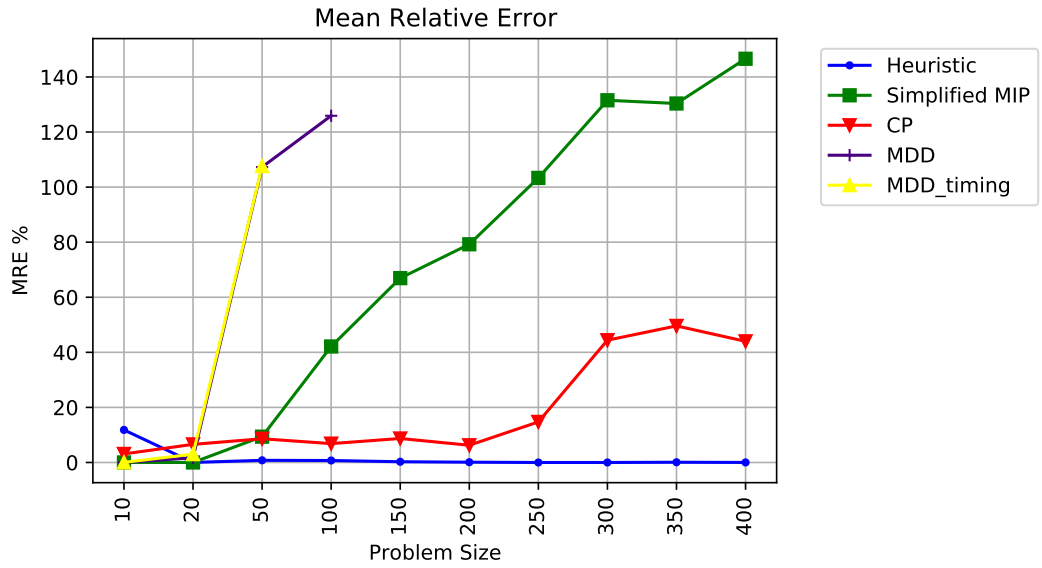


Figure 5.3: Mean Relative Error for MDD models and previous techniques.

5.4.1 Mean Relative Error

Unfortunately, neither MDD method was competitive in finding good quality solutions compared to the heuristic, MIP, and CP models as shown in Figure 5.3. Further, MDD was not able to find any feasible solutions in the given time limit for instances with more than 100 jobs and MDD_timing for more than 50 jobs. However, for 10 and 20 job instances both MDD methods find better solutions than CP and are competitive with the heuristic and MIP.

5.4.2 Instances Solved

The base MDD model was unable to solve any of the instances with more than 100 jobs. For instances with 150 jobs it was unable to find a feasible solution in the given time limit, whereas for 200+ job instances it ran out of memory. A similar situation occurred with MDD_timing, except at the 50 job threshold. Unfortunately, it appears that adding the timing algorithm does not lead to any significant benefit with regards to solution quality or the number of instances solved.

5.4.3 Average Run Time

As displayed in Figure 5.4, both MDD methods used the maximum amount of time for instances with 50 jobs. However, they are faster than the heuristic and CP for 10 and 20 job instances.

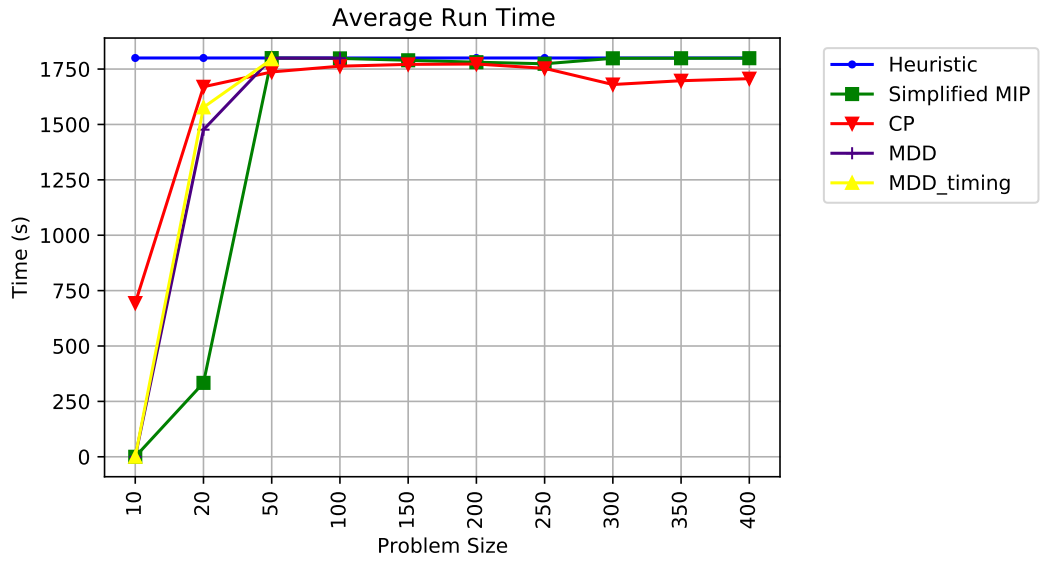


Figure 5.4: Average run time for MDD models and previous techniques.

5.4.4 MRE Over Time

As MDD was unable to solve instances with more than 100 jobs and MDD_timing unable to solve instances with more than 50 jobs, we only compare MRE over time using instances with 10, 20, and 50 jobs. Similar to the other methods, both MDD methods has difficulty improving solution quality over time, as shown in Figure 5.5. As well, MDD and MDD_timing plateau at a worse solution quality at the time limit than the previous methods.

5.4.5 MDD Warm Start

We test both MDD methods with an initial warm start solution provided by running the client heuristic for 5 seconds. We denote these methods as MDD_warm and MDD_timing_warm. Figure 5.6 shows that both MDD warm start versions greatly outperform their respective non-warm start versions and the heuristic for smaller instances. However, MDD_warm and MDD_timing_warm fail at finding solutions within the time limit for problem sizes with greater than 20 jobs. Thus, the heuristic warm start was unable to assist the MDD methods with finding solutions to larger instances.

In Figure 5.7, we compare the warm start MDD methods with the previous Simplified MIP_warm and CP_warm. MDD_warm and MDD_timing_warm achieve the best results for instances with 10 and 20 jobs, in fact they are equivalent in performance. However, they are not competitive with the other warm start methods given that they are unable to find feasible solutions.

Figure 5.8 compares the MDD warm start methods' MRE over time with previous warm start methods, using instances with 10 and 20 jobs. For these

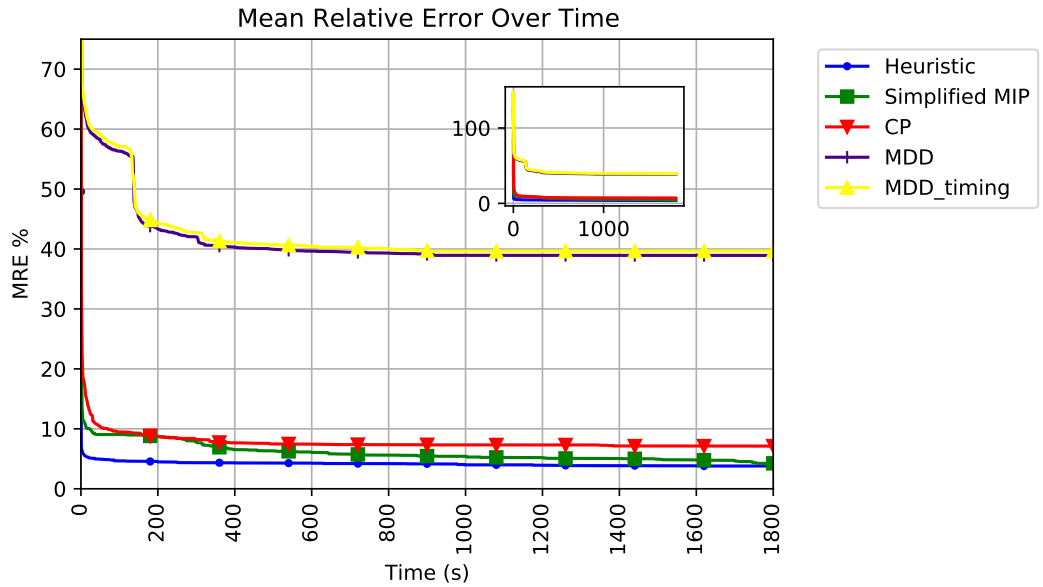


Figure 5.5: Mean Relative Error over time for MDD models and previous techniques, using 10, 20, and 50 job instances.

smaller instances, the Simplified MIP_warm and MDD_warm have the best solution quality at the time limit. Unfortunately the new MDD_timing_warm approach fares the worst, with CP_warm beating it.

5.4.6 Root Node Lower Bound Analysis

Although the MDD methods have difficulties finding competitive solutions, both find better root node lower bounds than CP for smaller instances as seen in Figure 5.9. Interestingly, for 10 job instances, both MDD approaches find better root node bounds than MIP. Although MDD and MDD_timing could not find feasible solutions in the given time limit for larger instances, both were able to find root node lower bounds for up to 300 job instances. Further, we observe that adding the timing algorithm to the MDD model does not improve the root node lower bound, as the timing algorithm does not run at the root node since it requires a candidate sequence.

We compared the best-known lower bounds that MDD and MDD_timing found for the instances they both did not solve optimally. Upon reaching the time limit, both methods had equivalent best-known lower bounds, suggesting that MDD_timing was unable to improve its lower bound over MDD.

However, there is a trade off between quality lower bounds and computation time: Figure 5.10 shows the rapid increase that the MDD approaches require to find root bounds. Thus, for 10 job instances an MDD lower bound is preferred and for larger instances CP is preferred.

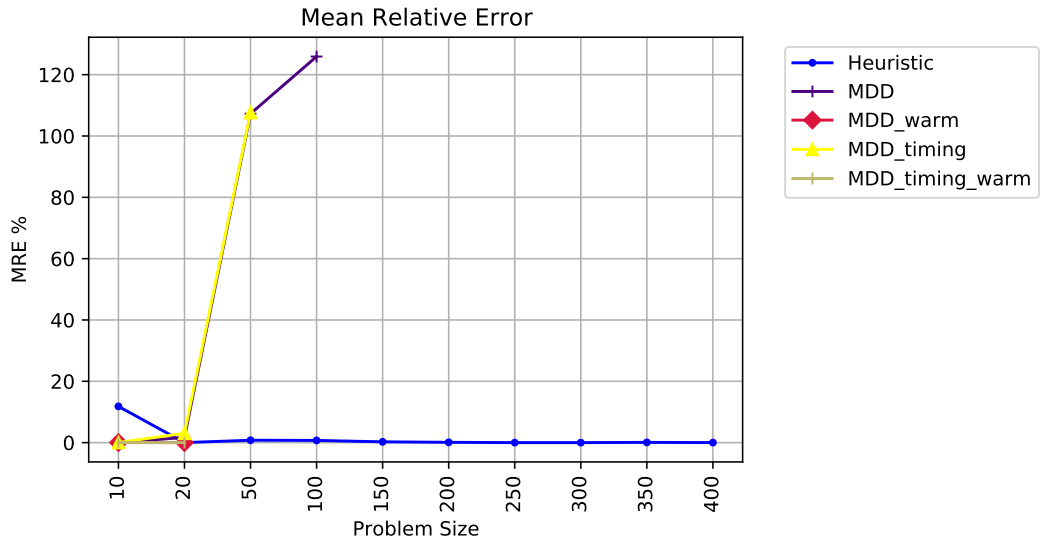


Figure 5.6: Mean Relative Error for MDD models with heuristic warm start and non-warm start. MDD_warm and MDD_timing_warm both have a MRE of 0% for 10 and 20 problem sizes and are represented by the line segment from $x = 10$ to $x = 20$ on the x-axis. Neither were able to solve instances larger than 20 jobs.

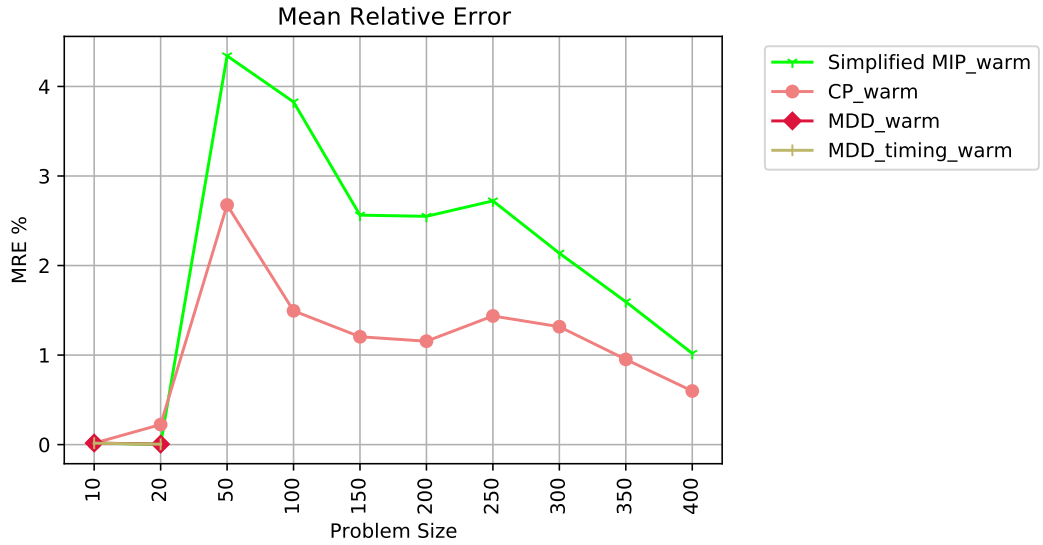


Figure 5.7: Mean Relative Error for MDD models with heuristic warm start and previous warm start techniques. MDD_warm and MDD_timing_warm both have a MRE of 0% for 10 and 20 problem sizes and are represented by the line segment from $x = 10$ to $x = 20$ on the x-axis. Neither were able to solve instances larger than 20 jobs.

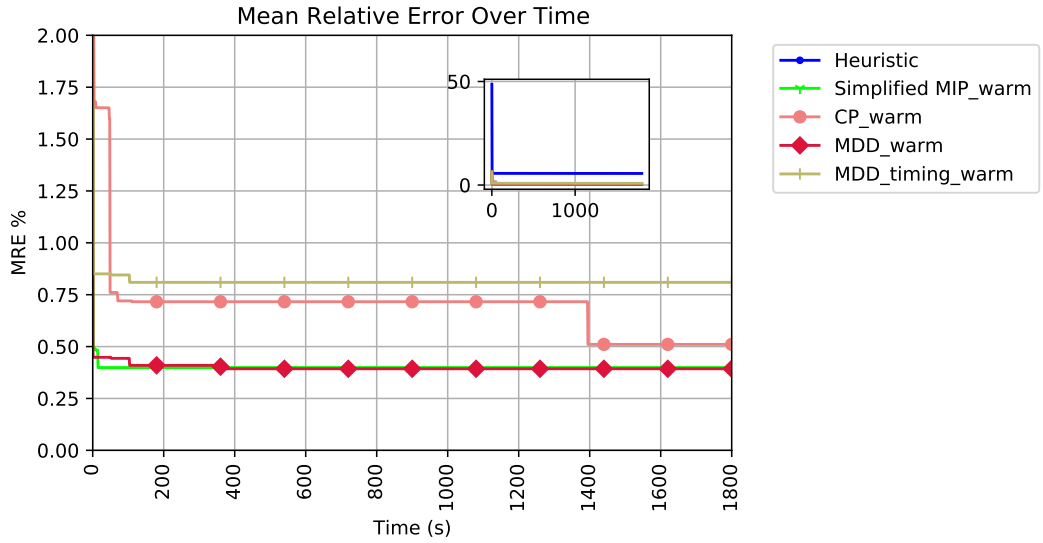


Figure 5.8: Mean Relative Error over time for MDD models with heuristic warm start and previous warm start techniques, using instances with 10 and 20 jobs.

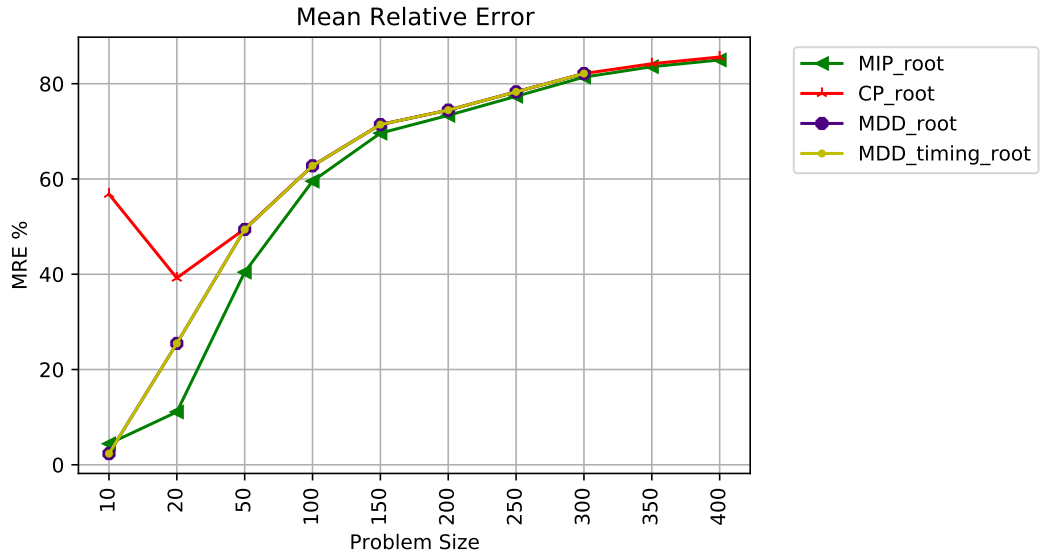


Figure 5.9: Root node lower bound deviation from best known solution for MDD models and previous techniques. Note, the two MDD methods have the same values.

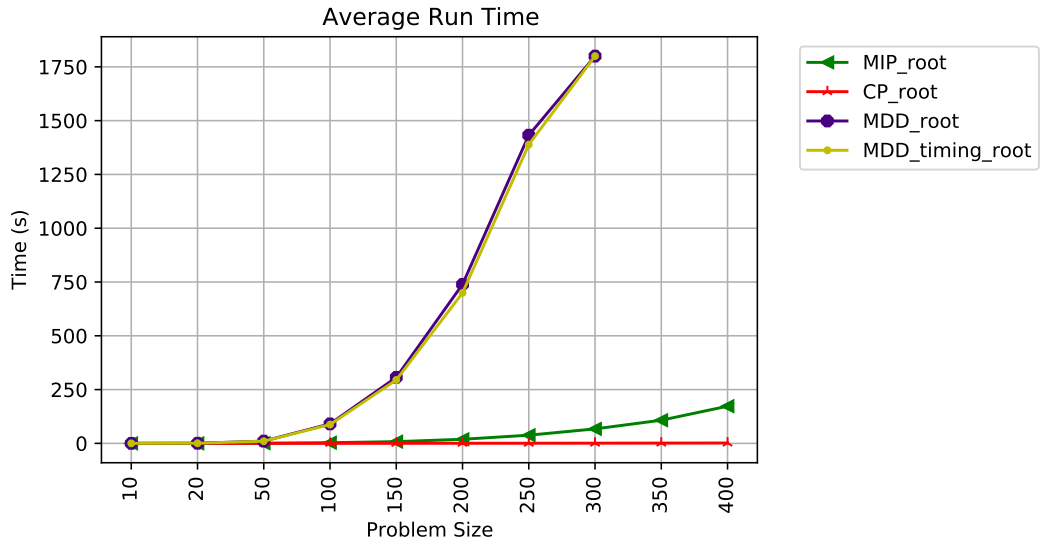


Figure 5.10: Average computation time for root node lower bounds.

5.5 Discussion

This chapter explored two decision diagram formulations to solve the ETSS problem. Decision diagrams were used in conjunction with CP to address CP's short comings regarding decision variable propagation with our complex objective function and to strengthen CP's lower bounds. We developed two decision diagram methods, both implemented as a global constraints with constraint programming, only MDD_timing differing by including a timing algorithm to provide better lower bounds on the decision diagram. Unfortunately, neither decision diagram method was competitive with the heuristic, Simplified MIP, and CP models. Their poor performance was due to the decision diagrams not finding good quality solutions in the given time limit for smaller instances and subsequently not finding feasible solutions in the time limit for larger instances. However, for smaller instances, both MDD and MDD_timing find better solutions over CP and run significantly faster. Further, both MDD methods find better root node lower bounds over CP for smaller instances. Thus, for smaller instances decision diagrams are effective over regular constraint programming; however, they fail to scale competitively.

Adding the timing algorithm to the decision diagram proved ineffective, as MDD_timing was equivalent to MDD, except that it took slightly longer to run on average and that it could not solve 100 job instances.

Chapter 6

Large Neighbourhood Search for Earliness Tardiness Scheduling with Setups

In this chapter we present Mixed Integer Programming (MIP) and Constraint Programming (CP) based Large Neighbourhood Search (LNS) methods for solving the Earliness Tardiness Scheduling with Setups (ETSS) problem. Each method relies on several LNS parameters, including the neighbourhood, neighbourhood solving criteria, and diversification procedure. We develop and investigate several options for each parameter, including a randomized approach, and select the best configurations: a CP-based LNS method with an improvement neighbourhood, optimal solving criteria, and random restart diversification procedure; and a MIP-based LNS method with a composite neighbourhood, time limit solving criteria, and composite diversification method. We compare the effectiveness of MIP LNS and CP LNS configurations and find that the CP-based LNS approaches tend to outperform MIP-based LNS. LNS targets the shortcomings of previous methods as they struggled to significantly improve solution quality over time. LNS experimental results demonstrate a large reduction in solution stagnation over time. Further, the LNS methods aim at rapidly finding solutions, similar to the client heuristic. Finally, we compare and discuss the numerical results for the developed LNS methods and compare against the previous competitive methods in this thesis: the client heuristic, Simplified MIP, CP, and Multivalued Decision Diagram (MDD). Overall, both LNS methods significantly outperformed the previous developed solution techniques and are competitive with the client heuristic.

6.1 Large Neighbourhood Search Formulations

For a review of LNS see Section 3.6. Both the MIP and CP LNS approaches follow the same framework:

- An initial solution is generated by running either the Simplified MIP or CP model until the first feasible solution is found.
- A selected neighbourhood function is applied to the solution which relaxes part of the solution thus generating candidate neighbour solutions. For example, one neighbourhood function is to relax a subsequence of jobs within in the current solution. Then, these jobs are free to change sequence and completion times.
- The neighbourhood is solved to find a new neighbour. The new neighbour is selected, and always accepted, by solving the neighbourhood problem using either MIP or CP, whilst following the selected solving criteria. The solving criteria dictates how MIP or CP solves the given neighbourhood problem, such as using a time limit or stopping when an optimal solution is found.
- If the LNS algorithm stalls, that is, repeatedly finds neighbours with the same objective cost, for a selected number of iterations, a diversification method is applied to move to a new part of the search space.

Neighbourhood generation, solving, and diversification, if required, is then repeated until the time limit is reached and the best solution found is returned. Algorithm 10 highlights the overall LNS structure. Line 5 tracks the number of stalls with nStalls and we set the stall threshold for running diversification, stallThreshold, as 100.

We provide a detailed overview of the neighbourhoods used in Section 6.1.1 and neighbourhood solving criteria in Section 6.1.2. Diversification methods are explained in Section 6.1.3.

Algorithm 10 Large Neighbourhood Search Framework

```
1: Generate initial solution
2: while Time Remaining > 0 do
3:     Apply neighbourhood function
4:     Solve neighbourhood using solving criteria → new solution
5:     if nStalls > stallThreshold then
6:         Run diversification → new solution
7: return best solution
```

6.1.1 Neighbourhoods

A total of six neighbourhood functions were used: random time window, sliding time window, greedy time window, rank based, improvement, and composite. All of the neighbourhoods produce feasible solutions.

Random Time Window

The random time window neighbourhood selects two random numbers from a uniform distribution between zero and the instance time horizon. The completion time and sequence of jobs scheduled to end between these two time points are then relaxed, allowing a search through all possible subsequences for the jobs in that time window. The jobs outside of the random time window are constrained in both completion time and sequence. For the jobs after the time window, this constraint means that their completion times do not shift even if there is space to shift earlier.

Sliding Time Window

The sliding time window divides the problem time horizon into k non-overlapping intervals. This neighbourhood relaxes the jobs in the current time interval, meaning these jobs can change sequence and completion times. Jobs outside this interval are not permitted to change sequence and completion times. Each time this neighbourhood is used, it advances to the next interval. For example, if the first time window is $[0, 3)$, then the next time this neighbourhood is selected, the time window will start from $t = 3$, and so on. Once the last time interval has been used the neighbourhood will begin again at its first interval. k was set to 10 for the numerical experiments.

Greedy Time Window

The greedy time window neighbourhood again divides up the problem time horizon into k non-overlapping intervals, similar to the sliding time window. It calculates each interval's cost using the earliness/tardiness and setup costs for the jobs scheduled in it. For each interval, say we have m jobs, then the interval cost is calculated as $\sum_{i=1}^{m} q_j \max\{\alpha\left(d_j - c_j\right), \beta\left(c_j - d_j\right)\} + \sum_{j=1}^{m-1} S_{j,j+1}$. The interval with the highest cost has all of its jobs relaxed, allowing their completion times and sequence to change, while locking in the jobs outside of this interval. The next time this neighbourhood is called, it will repeat and select a new interval with the highest cost. It is possible to select the same time interval again, depending on the cost structure of the new neighbour, which could yield the same solution. Again, k was set to 10 for the numerical experiments.

Rank Based

The rank-based neighbourhood is a basic variation where instead of relaxing some jobs and fixing others, we relax all jobs, but constrain them to be within r positions from their original position. That is, a job at rank (position) j in the solution is allowed to be in rank $j - r$ up to $j + r$. r was set to

$round\left(\frac{1}{\log_{10} n}\right) + 2$, where n is the number of jobs in an instance, such that the neighbourhood size is inversely proportional to the number of jobs.

Improvement

The improvement neighbourhood is a two-part local search technique inspired by the client heuristic (see Section 2.3.2 for details). The first improvement procedure iterates through each job in the current sequence and inserts that job into the position that provides the largest cost reduction. It terminates when no improvement can be found for any job. Thus, the neighbourhood is all the neighbour sequences where only one job is moved to a different position.

The second procedure considers all possible pairs of jobs and looks for the pair that, when swapped, provides the maximal cost reduction. The procedure finishes when no job pair swap will decrease overall cost. This neighbourhood is all sequences where only one pair of jobs exchange their position in the current sequence.

Recall, that within the two-part improvement procedure, job completion times must be recalculated when the job sequence is changed to determine sequence cost. The only change from the improvement procedure in the heuristic is that we recalculate job completion times using an efficient timing algorithm following the work of Hendel and Sourd [30], which is explained in detail in Section 5.2. Recall that, the client heuristic assumes earliness penalties are strictly less than tardiness penalties. Consequently, it assigns job completion times by scheduling jobs as early as possible in a given sequence. Our aim is to provide a more general approach through using the timing algorithm. After the improvement neighbourhood is run, we use the final sequence as a starting point in Line 4 of Algorithm 10. That is, when using the improvement neighbourhood, the new neighbour is generated by solving, following the given solving criteria, the ETSS problem using the improvement neighbourhood solution as a starting point in the solver.

Composite

Finally, the composite neighbourhood randomly selects one of the previous neighbourhoods with uniform probability each iteration in LNS.

6.1.2 Neighbourhood Solving Criteria

After applying the selected neighbourhood function, except for the improvement neighbourhood, the best neighbour is selected by solving the Simplified MIP or CP model. Anything that is relaxed is permitted to change during solving, for example the position and completion times of the jobs in a subsequence such as in the greedy time window neighbourhood. The optimization models have three options for solving criteria: optimality, first solution, and

time limit. The optimality solving criteria ends the CP or MIP optimization engine after it finds and proves the optimal solution. The first solution criteria terminates the search after the engine finds its first feasible solution. Finally, the time limit criteria forces the search to quit after the maximum time is reached, 5 seconds, returning the best-found solution. If no solution is found, the previous iteration's solution will be used as the next neighbour. Otherwise, once the optimization model has stopped, we have the next neighbour solution. All of the solving criteria will terminate if the overall LNS time limit is reached.

For the case with the improvement neighbourhood, we run the Simplified MIP or CP model using its solution as a starting point.

6.1.3 Diversification

The final component of our LNS is the diversification procedure. These methods focus on generating solutions that are far away from the current solution to escape local optima and thus are likely to generate a solution with a higher cost than the current solution. If the LNS algorithm stalls repeatedly, that is, finding solutions with the same objective cost, more than a set number of times, when applying neighbourhood functions and solving them, then we run the diversification procedure. This procedure yields a new solution which is used by the neighbourhood function in the next iteration, as shown in Line 6 of Algorithm 10. A total of five diversification methods were developed.

Random Restart

The random restart method randomly chooses a sequence of jobs such that each possible permutation of the jobs has uniform probability. Then it uses the timing algorithm to compute the completion times of each job following the new sequence.

Random Job Window

The random job window selects two jobs, without replacement, with uniform random probability. A random permutation of all jobs between and including these two jobs is chosen with uniform probability. The timing algorithm is used to re-time all of the jobs.

Ignore Setup Cost

The ignore setup cost method randomly selects $p = 25\%$ of jobs with uniform probability and ignores their setup costs with all other jobs. The selected jobs can be in any position in the sequence, as opposed to the subsequences used in the random job window method. Then the ETSS problem with adjusted

setup costs is solved from scratch using MIP or CP, depending on the current LNS method, for the first feasible solution.

Hybrid Swap Coefficient

The hybrid swap coefficient method selects one of two different procedures randomly with equal probability. This method was created to investigate the impact of having a two-part diversification method. The first procedure performs $round\left(\frac{n}{2}\right)$ iterations of selecting two jobs with uniform probability and swapping their positions. The resultant sequence is timed using the timing algorithm.

The second approach randomly modifies the earliness/tardiness weights for each job and the setup costs between all jobs. Each job's earliness/tardiness cost becomes $\delta \times q_j \times \max\{\alpha\left(d_j - c_j\right), \beta\left(c_j - d_j\right)\}$, where δ is uniformly distributed between 0 and 1. Similarly, all setup costs, S_{ij}, become $\delta \times S_{ij}$. δ is independently generated for each job and for each setup cost job pair. Then, the ETSS problem with adjusted coefficients is solved from scratch using MIP or CP for a feasible solution.

Composite

Finally, the composite diversification method randomly selects one of the previous diversification methods with uniform probability each time the diversification procedure is run.

6.2 Numerical Results

There are a total of 90 unique LNS parameter configurations as we have 6 neighbourhoods, 3 solving criteria, and 5 diversification techniques. Further, we either use the Simplified MIP model or CP model to generate new neighbours, resulting in 180 LNS methods. We denote each LNS method using the following naming convention: ⟨MIP or CP⟩ LNS ⟨Neighbourhood⟩ − ⟨Solving Criteria⟩ − ⟨Diversification⟩. Each method runs for an entirety of the 30-minute time limit, following the experimental setup defined in Section 4.3. All LNS methods generated an initial solution using their respective solver, MIP or CP, by finding the first feasible solution. We evaluate all 180 LNS methods and compare them, selecting the best performing MIP-based and CP-based LNS approach using Mean Relative Error (MRE), see Section 6.2.1. We then compare these LNS methods against the previously developed techniques: the client heuristic, Simplified MIP, CP, and MDD. Graphs with error bars are located in Appendix C.

Table 6.1: Top 5 CP and MIP LNS methods by MRE.

Method	MRE
CP LNS Improvement-Optimal-RandomRestart	3.56
CP LNS Improvement-Optimal-Composite	3.62
CP LNS Improvement-Optimal-HybridSwapCoeff	3.86
CP LNS Improvement-Optimal-RandomTimeWindow	3.86
CP LNS Improvement-Optimal-IgnoreSetupCost	3.95
MIP LNS Composite-TimeLimit-Composite	4.44
MIP LNS Composite-TimeLimit-HybridSwapCoeff	4.64
MIP LNS Improvement-TimeLimit-RandomTimeWindow	4.68
MIP LNS Improvement-TimeLimit-HybridSwapCoeff	4.80
MIP LNS Composite-TimeLimit-IgnoreSetupCost	4.83

6.2.1 LNS Selection

We evaluated the 180 LNS methods using all of the ETSS instances by comparing MRE values. That is, we calculate the error for each instance and average it for each method. Formally, we define MRE as:

$$\text{MRE} = \frac{1}{N}\sum_{i=1}^{N}\frac{|(Z_i - Z_i^{best})|}{Z_i^{best}}$$

Where MRE is the Mean Relative Error for the selected method, N is the total number of instances, Z_i is the selected method's solution for instance i, and Z_i^{best} is the best-known solution for instance i across the all of the methods in this thesis.

Table 6.1 displays the top five MIP-based and top five CP-based LNS approaches. We observe that CP-based LNS methods perform better than MIP-based. Additionally, all of the top five CP methods use the improvement neighbourhood and optimal solving criteria. However, the diversification methods are not consistent, suggesting that the diversification method used is less impactful for MRE performance.

Regarding the top five MIP-based models, while there is less consistency with the neighbourhoods and diversification methods, the time limit solving criteria is used by all top approaches.

Two LNS methods were selected to represent the best LNS approaches and to use for comparisons against previous solution techniques. One MIP-based and one CP-based LNS method were determined using the lowest MRE score. The chosen methods are CP LNS with the improvement neighbourhood, optimal solving criteria, and random restart diversification; and MIP LNS with the composite neighbourhood, time limit solving criteria, and composite diversification.

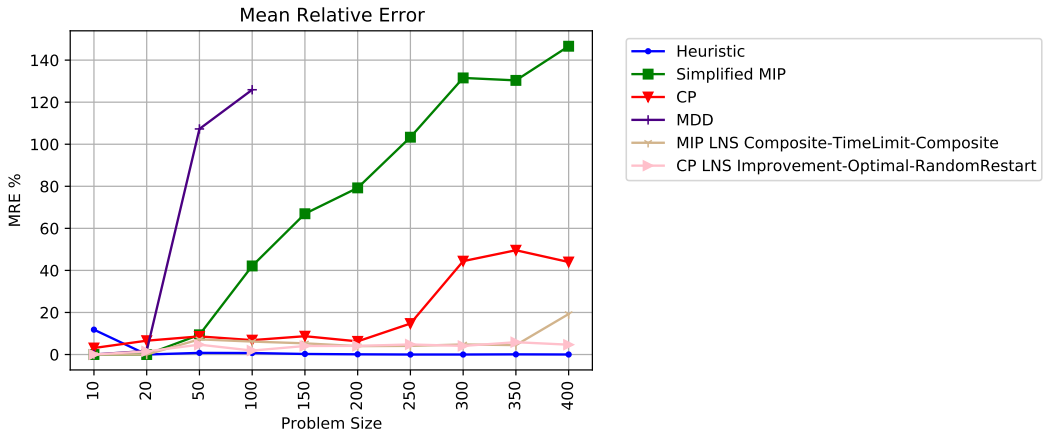


Figure 6.1: Mean Relative Error for LNS models and previous techniques.

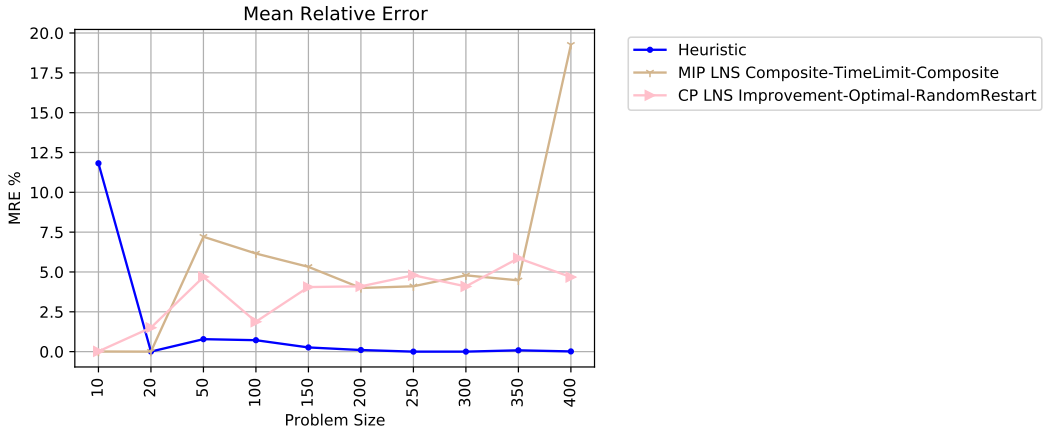


Figure 6.2: Mean Relative Error for LNS and client heuristic.

6.2.2 Mean Relative Error

We compare the mean relative error of the two LNS methods against the best-known solution for each instance. We include the previous methods investigated in this thesis for comparison in Figure 6.1. Both LNS methods outperform the Simplified MIP, CP, and MDD models. Figure 6.2 demonstrates the LNS competitiveness with the client heuristic in more detail. We observe that the MIP and CP LNS methods are comparable in performance, except for MIP LNS's poor performance with 400 job instances. However, the client heuristic still performs the best overall.

6.2.3 Instances Solved

All of the LNS models were able to find feasible solutions for all instances within the time limit.

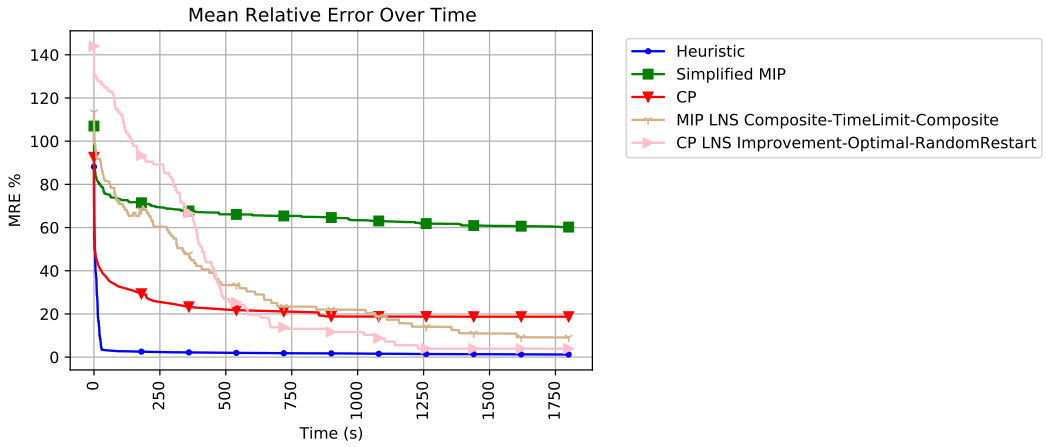


Figure 6.3: Mean Relative Error over time for best LNS methods and previous methods.

6.2.4 MRE Over Time

We also test the MRE over time for the LNS methods. Figure 6.3 demonstrates that both MIP and CP based LNS methods greatly reduce the stagnation in solution quality during the experiment compared to Simplified MIP and CP. Although the LNS methods are competitive with the client heuristic, we observe that the client heuristic finds very good solutions almost instantly whereas the LNS methods need the entire time limit for comparable quality. This performance pattern is likely due to the heuristic's strong initial solution, contrary to LNS starting with its first found feasible solution via MIP or CP.

We do not include the MDD method in this figure as its solution quality over time is misleading: it performed quite well on smaller instances but is missing data for the instances it did not find feasible solutions for, as explained in Section 5.4.4.

6.2.5 LNS Warm Start

We tested both LNS methods starting with an initial solution from the client heuristic after it was run for five seconds. Both the MIP and CP based LNS methods improved with the warm start as shown in Figure 6.4. The CP LNS with warm start outperformed MIP LNS warm, but neither were able to beat the heuristic when run for the entire time limit. In Figure 6.5 we include comparisons against the previous warm start methods, demonstrating that the MIP and CP LNS methods outperform the previous warm start techniques.

Further, we include a comparison of the warm start methods' MRE over time in Figure 6.6.

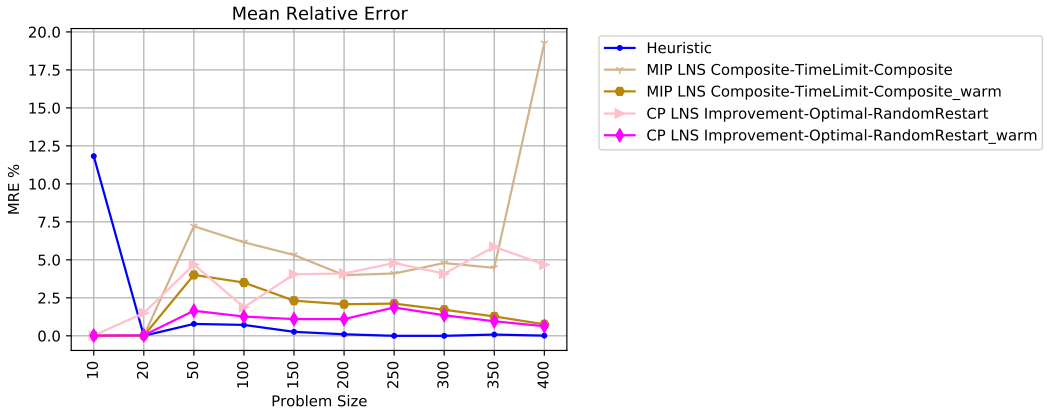


Figure 6.4: Mean Relative Error for best LNS methods with heuristic warm start.

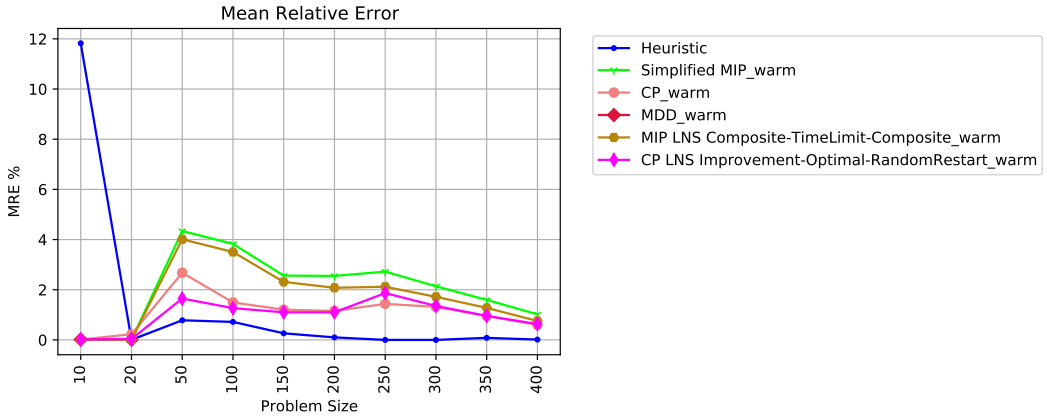


Figure 6.5: Mean Relative Error for best LNS methods with heuristic warm start and previous warm start methods.

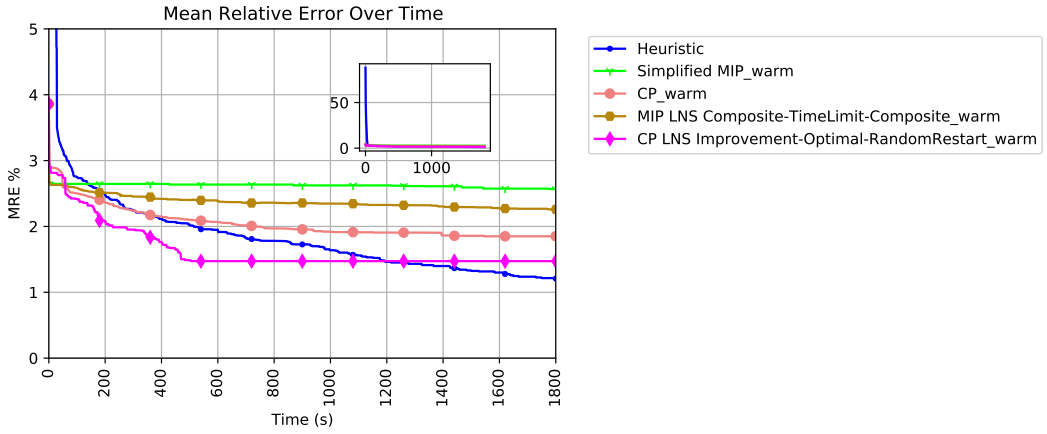


Figure 6.6: Mean Relative Error over time for best LNS methods with heuristic warm start and previous warm start methods and heuristic.

6.3 Discussion

This chapter formulated and tested 90 LNS models, using either MIP or CP as the underlying optimization engine, resulting in 180 total models. The best performing CP LNS method used the improvement neighbourhood, optimal solving criteria, and random restart diversification procedure. The best performing MIP LNS method used the composite neighbourhood, time limit solving criteria, and composite diversification method. Overall, the CP LNS approach performed slightly better than MIP LNS. Both LNS methods outperformed the previously developed solution techniques by a large margin. CP LNS and MIP LNS remained competitive with the client heuristic, with approximately a 2% worse overall MRE. As shown in the MRE over time experiments, the LNS methods continue to improve their solution quality overtime in comparison to CP and Simplified MIP. Thus, the application of large neighbourhood search was effective at targeting CP and Simplified MIP's shortcoming of finding improved solutions over time. The CP-based LNS performs the best across all methods in this thesis except for the client heuristic. The MIP-based LNS remains close in second place.

Chapter 7

Conclusion

This chapter concludes this thesis by presenting a summary of the previous chapters and providing the contributions of this work. We conclude by providing some insight on future research directions.

7.1 Summary and Contributions

In this thesis, we formally defined the Earliness Tardiness Scheduling with Setups (ETSS) problem. The ETSS problem is derived from real-world industrial scheduling applications and is comprised of two well-known optimization subproblems: Earliness tardiness scheduling and the Travelling Salesman Problem (TSP). Using the instances provided by our industry client, we generated additional instances for our experimental testing.

We presented four Mixed Integer Programming (MIP) and one Constraint Programming (CP) models for solving the ETSS problem, finding that none of the methods are competitive with the client heuristic. We demonstrated that the CP model has poor objective function propagation with our complex objective function. As well, both MIP and CP struggle with finding good root node lower bounds. Finally, we showed that the current state-of-the-art MIP models for minimizing makespan on a single machine with setups are not competitive for the ETSS problem.

Another approach using Multivalued Decision Diagrams (MDD) integrated with CP was created to improve root node bounds and objective function propagation. Further, we novelly overlaid the MDD method with a timing algorithm, considering job sequence and completion times, in pursuit of improved lower bounds over MDD. For small instances, both MDD methods outperformed CP in solution quality, time, and root node bound strength; however, neither MDD method was competitive against the other methods for larger instances, indicating a scalability issue. Adding the timing algorithm to the MDD proved ineffective as it performed similar to MDD whilst taking longer to run.

MIP and CP based Large Neighbourhood Search (LNS) techniques were investigated to target previous solution quality stagnation shortcomings. Ninety MIP LNS and ninety CP LNS variations were tested and demonstrated a large reduction in solution stagnation over time. The best MIP LNS and CP LNS methods significantly outperformed the previously developed solution techniques and are competitive with the client heuristic. The best CP LNS method outperforms all of the methods in this thesis except for the client heuristic. The best MIP LNS remains close in second place.

The various model performances against the client heuristic would likely be improved with altered problem characteristics. For example, changing the earliness α and tardiness β coefficients such that $\alpha > \beta$ would penalize the heuristic's shifting of jobs to their earliest competition times, whereas the methods developed in this thesis generalize better. As well, the thesis methods could outperform the heuristic in certain instances where the optimal schedules rely on large idle time periods, which the client heuristic does not consider.

7.2 Future Work

Improving the lower bounds for the CP model in Chapter 4 is a promising future research area. One could investigate the application of Sourd's [60] partial sequence lower bound and dominance criteria for the ETSS problem within a constraint programming context. These techniques proved quite competitive within a branch and bound implementation. Another direction is to address CP's poor objective function propagation by developing a global constraint to aid it. For example, Monette, Deville, and Van Hentenryck developed a novel global constraint for earliness/tardiness within job shop scheduling problems, including filtering and dedicated heuristics [49]. A first step is to integrate this constraint into the ETSS CP model, thus providing additional earliness/tardiness propagation. Then, the constraint could be modified to also consider setup costs between jobs.

Alternatively, a decomposition approach with the Simplified MIP is interesting to study, as the ETSS problem is comprised of two well-studied subproblems: earliness/tardiness scheduling and the TSP. The master problem could sequence the jobs to minimize setup costs, while the subproblem provides bounding information regarding earliness/tardiness costs for a given sequence. Sequencing is done with the binary decision variable x_{ij}, which is 1 if job i immediately precedes job j. The continuous decision variable η tracks the lower bound on earliness/tardiness costs. To solve the master problem, we add a dummy job to $\mathcal{J}$, which will represent the start and end job, with 0 setup time and cost to all other jobs. The master problem is a binary integer program, and as such can be solved using branch and cut by lazily adding subtour elimination constraints (7.4) and Benders optimality cuts (7.5). As well, the master problem does not require feasibility cuts as the subproblems are

always feasible for a given job sequence. The Benders reformulation's master problem would be:

$$\min \sum_{i\in\mathcal{J}}\sum_{j\in\mathcal{J}} S_{ij}x_{ij} + \eta \tag{7.1}$$

$$\text{s.t.} \sum_{i\in\mathcal{J}:i\neq j} x_{ij} = 1 \qquad \forall j \in \mathcal{J} \tag{7.2}$$

$$\sum_{j\in\mathcal{J}:j\neq i} x_{ij} = 1 \qquad \forall i \in \mathcal{J} \tag{7.3}$$

$$\sum_{i\in\mathcal{S}}\sum_{j\notin\mathcal{S}} x_{ij} \geq 1 \qquad \forall \mathcal{S} \subseteq \mathcal{J}, 2 \leq |\mathcal{S}| \leq |\mathcal{J}| - 2 \tag{7.4}$$

$$\eta \geq \sum_{i,j\in\mathcal{J}:i\neq j} (s_{ij} + p_j)\, x_{ij}\hat{\pi}_{ij} + \sum_{j\in\mathcal{J}} (r_j + p_j)\, \hat{\mu}_j + \alpha q_j d_j \hat{\theta}_j - \beta q_j d_j \hat{\tau}_j \qquad \forall \pi, \mu, \theta, \tau \in V(\Pi) \tag{7.5}$$

$$x_{it} \in \{0, 1\} \qquad \forall i, j \in \mathcal{J} \tag{7.6}$$

$$\eta \geq 0 \tag{7.7}$$

Objective (7.1) minimizes total setup costs, while considering a lower bound on its associated earliness/tardiness costs. Constraint (7.2) forces each job to have a predecessor. Constraint (7.3) forces each job to have a successor. Constraint (7.4) eliminates subtours. Constraint (7.5) represents the Benders optimality cuts, using the dual objective of the subproblem. Note that $\pi, \mu, \theta,$ and τ are the dual variables associated with the nonempty set of extreme points, $V(\Pi)$, from the subproblem dual polyhedron, Π. Finally, constraint (7.6) and constraint (7.7) set the variable bounds. Perhaps this subproblem can be solved using the timing algorithm discussed in Section 5.2; however, it is not immediately clear if it is possible to obtain cuts for the master problem.

The supbroblem takes a fixed sequence of jobs from the master problem, using $\hat{x}_{ij}$, and schedules them to minimize the total earliness/tardiness costs:

$$\min \sum_{j\in\mathcal{J}} f_j \tag{7.8}$$

$$\text{s.t. } c_j \geq (c_i + s_{ij} + p_j)\, \hat{x}_{ij} \qquad \forall i, j \in \mathcal{J}, i \neq j \tag{7.9}$$

$$c_j \geq r_j + p_j \qquad \forall j \in \mathcal{J} \tag{7.10}$$

$$f_j \geq \alpha q_j (d_j - c_j) \qquad \forall j \in \mathcal{J} \tag{7.11}$$

$$f_j \geq \beta q_j (c_j - d_j) \qquad \forall j \in \mathcal{J} \tag{7.12}$$

$$c_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{7.13}$$

$$f_j \geq 0 \qquad \forall j \in \mathcal{J} \tag{7.14}$$

Objective (7.8) minimizes the total earliness and tardiness of all jobs. Constraint (7.9) ensures jobs are scheduled following the master problem sequence and prevents overlap. Constraint (7.10) enforces job release dates. Constraint (7.11) tracks job earliness, while constraint (7.12) tracks job tardiness. Constraint (7.13) and constraint (7.14) set the variable bounds.

Regarding decision diagrams, in Chapter 5 we showed scalability issues which is an important research topic to further investigate. As the MDD formulation uses job sequence, again, a partial sequence earliness/tardiness lower bound could be utilized at each node in the diagram. Researching methods to better integrate temporal bounding could increase scalability via assisting with arc filtering using objective bounds and start time propagation. Alternatively, another direction is to investigate a network representation of the MDD, which has performed quite well as a subproblem to a Lagrangian relaxation technique for the Multicommodity Pickup-and-Delivery TSP [19]. Considering an MDD formulation that integrates time and job sequence would provide an exact formulation for the ETSS problem. Then, a relaxation of this MDD would yield stronger bounds than our formulation. As well, specific node merging strategies may increase MDD performance; more experimentation is required.

In Chapter 6, we showed the effectiveness of LNS for the ETSS problem, although there are a couple research possibilities to increase its competitiveness against the client heuristic. Firstly, exploring an adaptive LNS, where the neighbourhood size is chosen in each iteration. Perhaps earlier in the search or if stalling is occurring, the neighbourhood size is increased to promote more diverse solutions. It would also be interesting to test a Self-Adapting LNS algorithm that uses machine learning to converge on efficient neighbourhoods and neighbourhood solving strategies, following the work of Laborie and Godard [44]. However, one could use neighbourhoods targeted towards the ETSS problem and neighbourhoods that have proven successful in this thesis.

7.3 Concluding Remarks

The goal of this thesis is to formally analyze and better solve the ETSS problem. This problem is quite complex due to its unique assortment of objective function criteria. It does not have a large presence in the literature, and we believe that the ETSS problem has very interesting problem characteristics that can be further studied. We presented 187 solution approaches to the ETSS problem, including a novel MDD method that uses a timing algorithm, drawing techniques from MIP, CP, MDD, and LNS. Empirical results demonstrated that CP-based LNS is a good approach for solving the ETSS problem.

Appendix A

Mixed Integer Programming and Constraint Programming Graphs With Error Bars

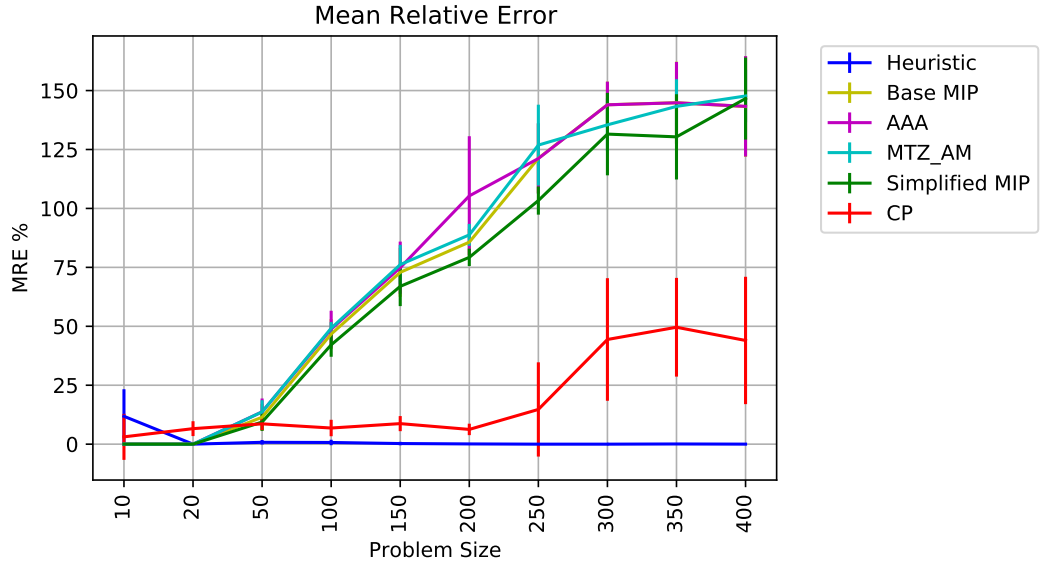


Figure A.1: Mean Relative Error for MIP and CP models with standard deviation error bars.

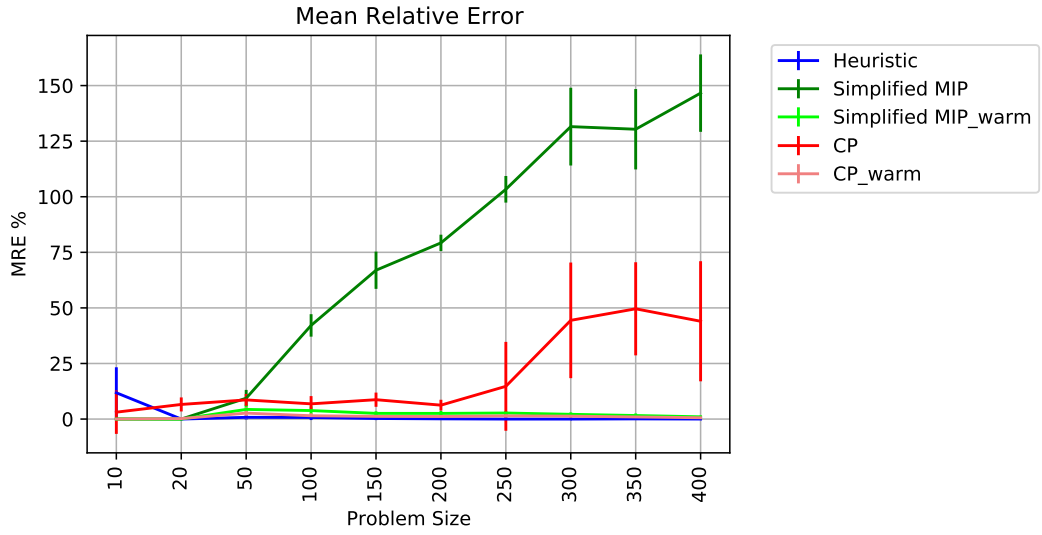


Figure A.2: Mean Relative Error for Simplified MIP and CP with heuristic warm start and non-warm start with standard deviation error bars.

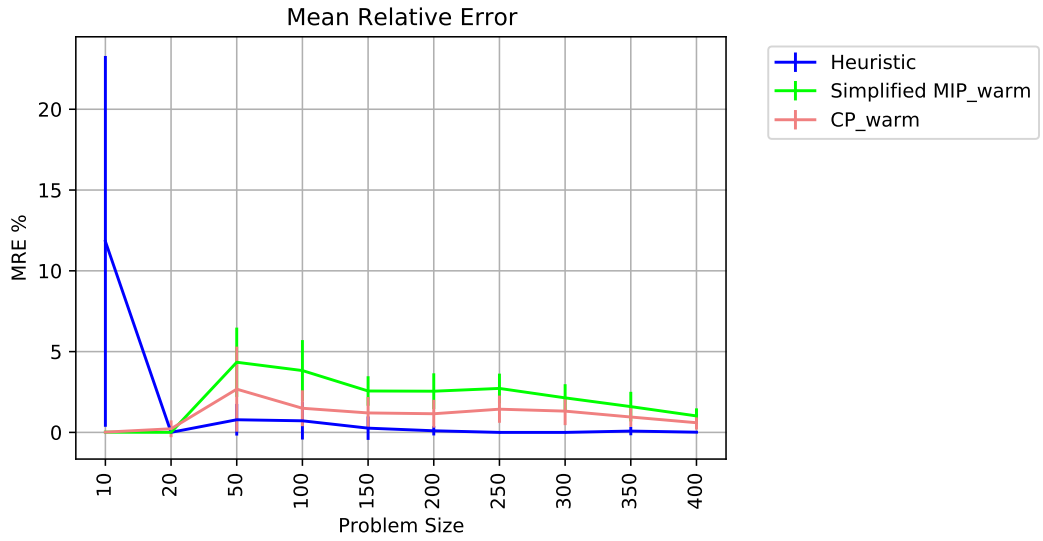


Figure A.3: Mean Relative Error for Simplified MIP and CP with heuristic warm start with standard deviation error bars.

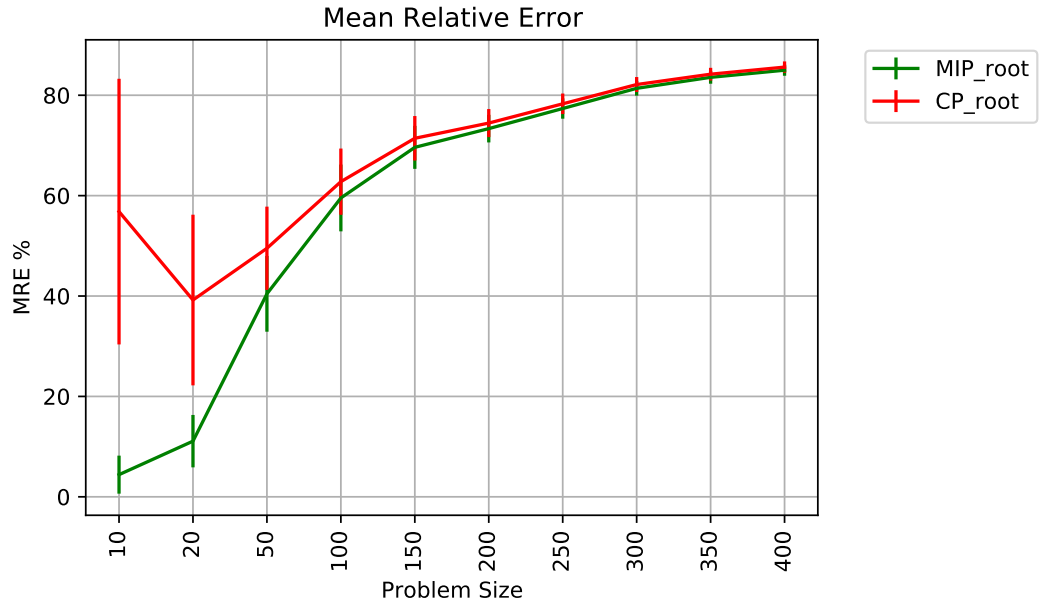


Figure A.4: Root node lower bound deviation from best-known solution with standard deviation error bars.

Appendix B

Decision Diagram Graphs With Error Bars

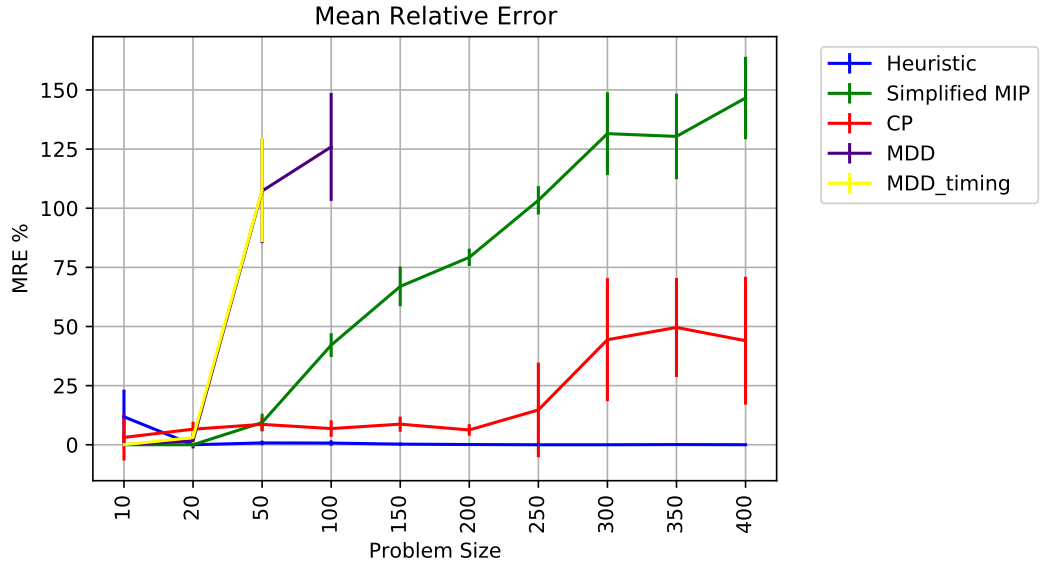


Figure B.1: Mean Relative Error for MDD models and previous techniques with standard deviation error bars.

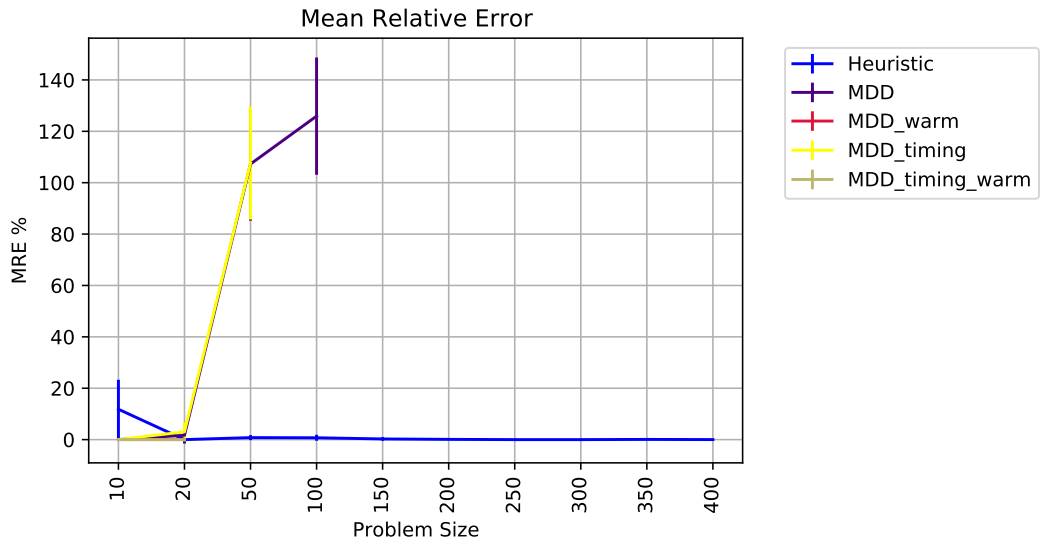


Figure B.2: Mean Relative Error for MDD models with heuristic warm start and non-warm start with standard deviation error bars. MDD_warm and MDD_timing_warm both have a MRE of 0% for 10 and 20 problem sizes and are represented by the line segment from $x = 10$ to $x = 20$ on the x-axis. Neither were able to solve instances larger than 20 jobs.

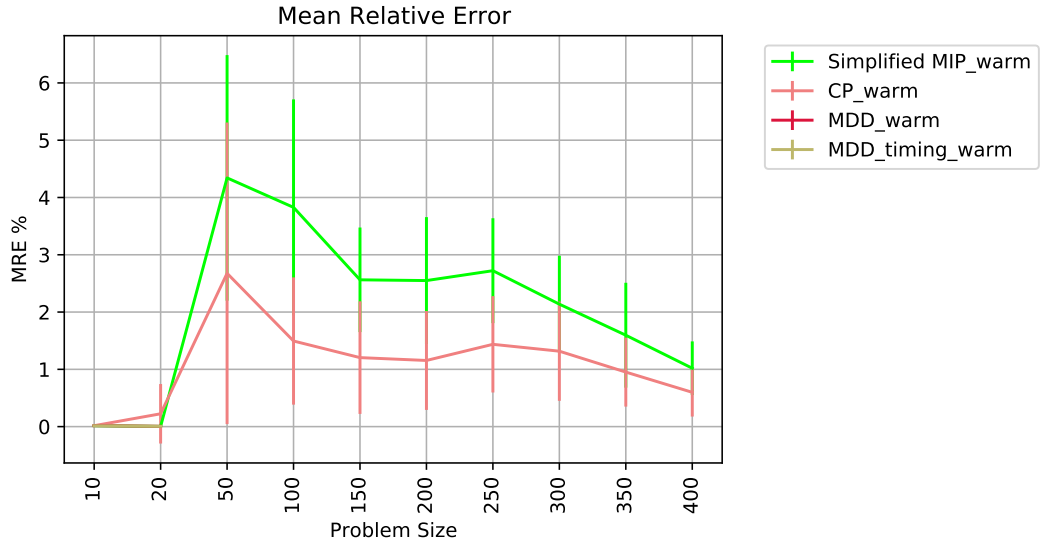


Figure B.3: Mean Relative Error for MDD models with heuristic warm start and previous warm start techniques with standard deviation error bars. MDD_warm and MDD_timing_warm both have a MRE of 0% for 10 and 20 problem sizes and are represented by the line segment from $x = 10$ to $x = 20$ on the x-axis. Neither were able to solve instances larger than 20 jobs.

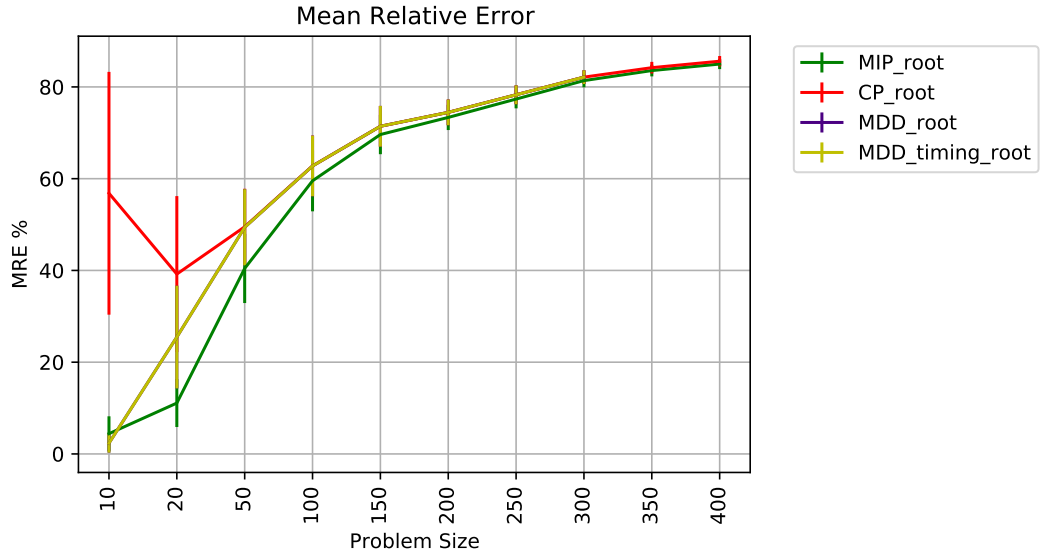


Figure B.4: Root node lower bound deviation from best known solution for MDD models and previous techniques with standard deviation error bars. Note, the two MDD methods have the same values.

Appendix C

Large Neighbourhood Search Graphs With Error Bars

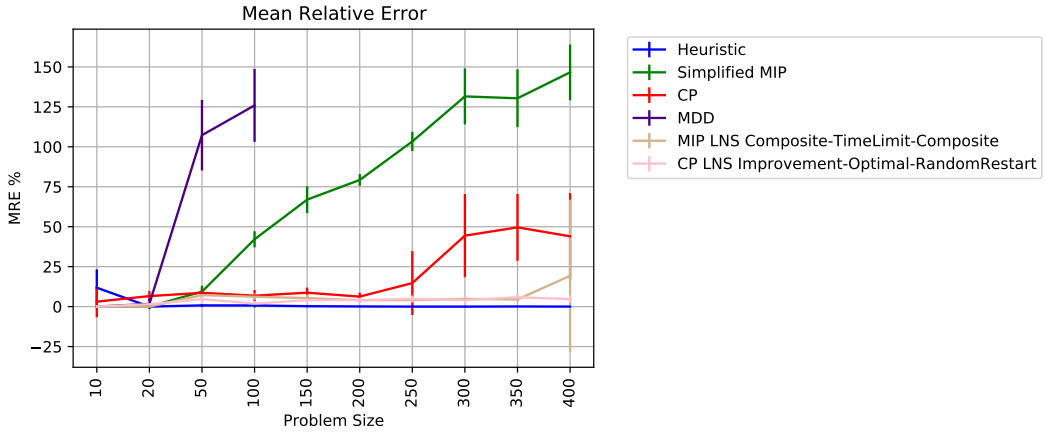


Figure C.1: Mean Relative Error for LNS models and previous techniques standard deviation error bars.

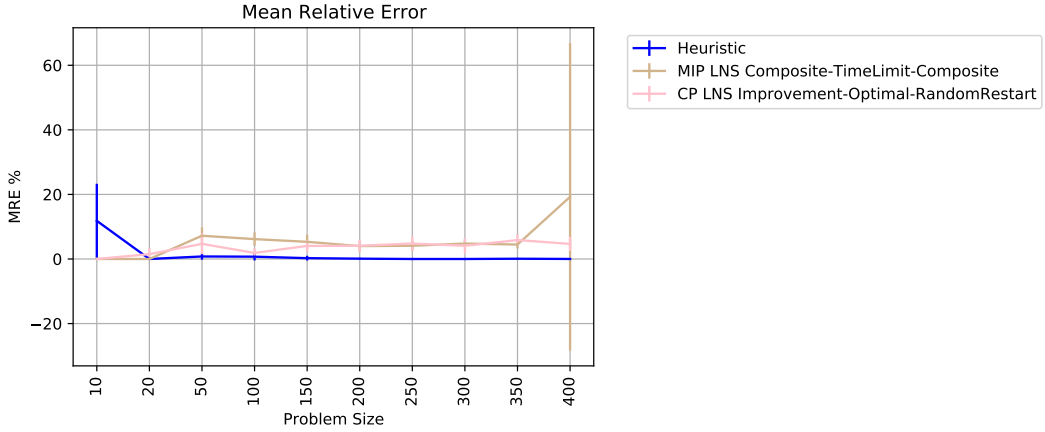


Figure C.2: Mean Relative Error for LNS and client heuristic standard deviation error bars.

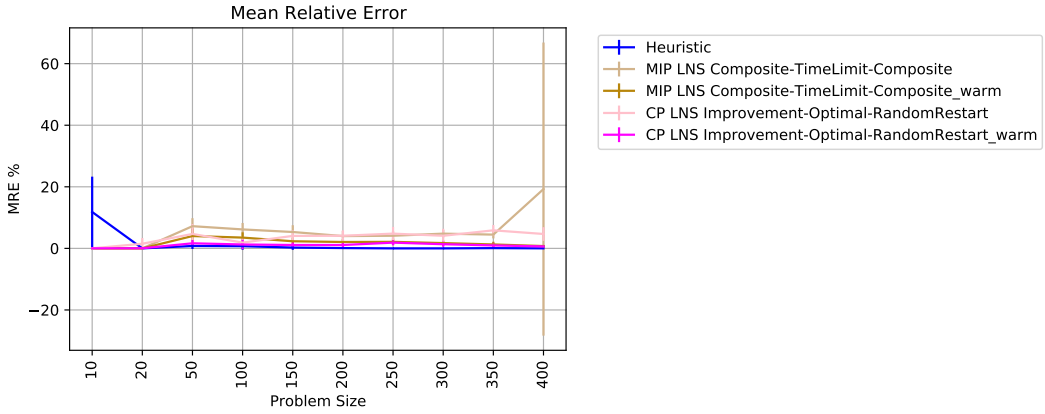


Figure C.3: Mean Relative Error for best LNS methods with heuristic warm start standard deviation error bars.

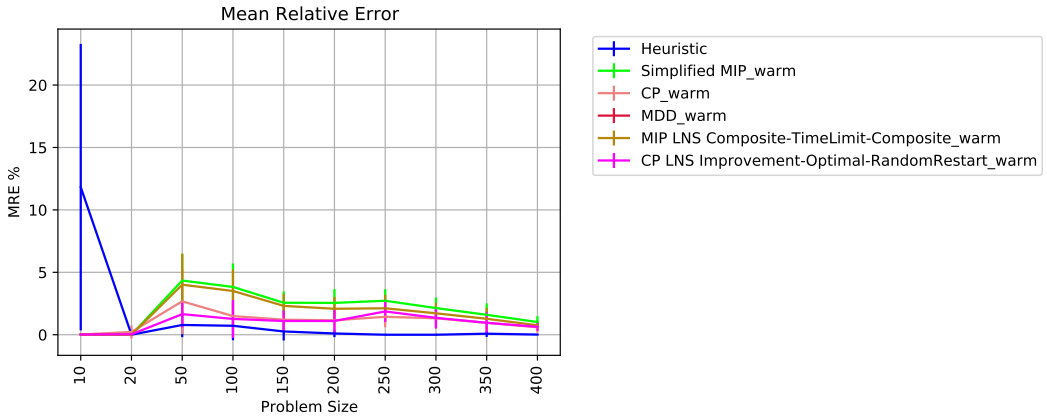


Figure C.4: Mean Relative Error for best LNS methods with heuristic warm start and previous warm start methods standard deviation error bars.

Bibliography

[1] JM Van den Akker, Cor AJ Hurkens, and Martin WP Savelsbergh. "Time-indexed formulations for machine scheduling problems: Column generation". In: *INFORMS Journal on Computing* 12.2 (2000), pp. 111–124.

[2] Ali Allahverdi, Jatinder ND Gupta, and Tariq Aldowaisan. "A review of scheduling research involving setup considerations". In: *Omega* 27.2 (1999), pp. 219–239.

[3] Henrik Reif Andersen, Tarik Hadzic, John N Hooker, and Peter Tiedemann. "A constraint store based on multivalued decision diagrams". In: *Principles and Practice of Constraint Programming (CP 2007)*. Vol. 4714. Springer, 2007, pp. 118–132.

[4] D.L Applegate, R.E Bixby, V Chvátal, and W.J Cook. *The traveling salesman problem: a computational study*. Princeton: Princeton University Press, 2006.

[5] Oliver Avalos-Rosales, Francisco Angel-Bello, and Ada Alvarez. "Efficient metaheuristic algorithm and re-formulations for the unrelated parallel machine scheduling problem with sequence and machine-dependent setup times". In: *The International Journal of Advanced Manufacturing Technology* 76.9 (2015), pp. 1705–1718. ISSN: 1433-3015. DOI: 10.1007/s00170-014-6390-6. URL: https://doi.org/10.1007/s00170-014-6390-6.

[6] Meral Azizoglu and Scott Webster. "Scheduling job families about an unrestricted common due date on a single machine". In: *International Journal of Production Research* 35.5 (1997), pp. 1321–1330.

[7] Kenneth R Baker and Gary D Scudder. "Sequencing with earliness and tardiness penalties: a review". In: *Operations research* 38.1 (1990), pp. 22–36.

[8] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. Springer Science & Business Media, 2001.

[9] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*. Vol. 1. Springer, 2016.

[10] Lucio Bianco, Salvatore Ricciardelli, Giovanni Rinaldi, and Antonio Sassano. "Scheduling tasks with sequence-dependent processing times". In: *Naval Research Logistics (NRL)* 35.2 (1988), pp. 177–184.

[11] E Robert Bixby, Mary Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. "MIP: Theory and practice — closing the gap". In: *System Modelling and Optimization*. Springer, 2000, pp. 19–49.

[12] Alexander Bockmayr and John N Hooker. "Constraint programming". In: *Handbooks in Operations Research and Management Science* 12 (2005), pp. 559–600.

[13] Fayez Fouad Boctor. *The Cold Rolling Mill Scheduling Problem: A New Formulation and a Perturbation Solution Approach*. Vol. 1. CIRRELT, 2016.

[14] Rodney Matineau Burstall. "A heuristic method for a job-scheduling problem". In: *Journal of the Operational Research Society* 17.3 (1966), pp. 291–304.

[15] John A Buzacott and Sujit K Dutta. "Sequencing many jobs on a multi-purpose facility". In: *Naval Research Logistics Quarterly* 18.1 (1971), pp. 75–82.

[16] Jeffrey D Camm, Amitabh S Raturi, and Shigeru Tsubakitani. "Cutting big M down to size". In: *Interfaces* 20.5 (1990), pp. 61–66.

[17] Tom Carchrae and J Christopher Beck. "Cost-based large neighborhood search". In: *Workshop on the combination of metaheuristic and local search with constraint programming techniques*. 2005.

[18] Caroline Banton. *Just-in-Time (JIT) Definition*. https://www.investopedia.com/terms/j/jit.asp. 2021.

[19] Margarita P Castro, Andre A Cire, and J Christopher Beck. "An MDD-based Lagrangian approach to the multicommodity pickup-and-delivery TSP". In: *INFORMS Journal on Computing* 32.2 (2020), pp. 263–278.

[20] B Jay Coleman. "A simple model for optimizing the single machine early/tardy problem with sequence-dependent setups". In: *Production and Operations Management* 1.2 (1992), pp. 225–228.

[21] Emilie Danna and Laurent Perron. "Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2003, pp. 817–821.

[22] Dennis Adsit. *Cutting-Edge Methods Help Target Real Call Center Waste*. https://www.isixsigma.com/implementation/case-studies/cutting-edge-methods-help-target-real-call-center-waste/. 2007.

[23] Luis Fanjul-Peyro, Rubén Ruiz, and Federico Perea. "Reformulations and an exact algorithm for unrelated parallel machine scheduling problems with setup times". In: *Computers & Operations Research* 101 (2019), pp. 173 –182. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2018.07.007. URL: http://www.sciencedirect.com/science/article/pii/S0305054818301916.

[24] Thomas A Feo, Kishore Sarathy, and John McGahan. "A GRASP for single machine scheduling with sequence dependent setup costs and linear delay penalties". In: *Computers & Operations Research* 23.9 (1996), pp. 881–895.

[25] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[26] Michel Gendreau, Jean-Yves Potvin, David Pisinger, and Stefan Ropke. "Large Neighborhood Search". In: *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[27] Paul C Gilmore and Ralph E Gomory. "Sequencing a one state-variable machine: A solvable case of the traveling salesman problem". In: *Operations research* 12.5 (1964), pp. 655–679.

[28] Daniel Godard, Philippe Laborie, and Wim Nuijten. "Randomized Large Neighborhood Search for Cumulative Scheduling." In: *ICAPS*. Vol. 5. 2005, pp. 81–89.

[29] Gregory Gutin and Abraham P Punnen. *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media, 2006.

[30] Yann Hendel and Francis Sourd. "An improved earliness–tardiness timing algorithm". In: *Computers & operations research* 34.10 (2007), pp. 2931–2938.

[31] Samid Hoda, Willem-Jan Van Hoeve, and John N Hooker. "A systematic approach to MDD-based constraint programming". In: *Principles and Practice of Constraint Programming (CP 2010)*. Vol. 6308. Springer, 2010, pp. 266–280.

[32] John N Hooker. "Logic, optimization, and constraint programming". In: *INFORMS Journal on Computing* 14.4 (2002), pp. 295–321.

[33] IBM ILOG. *Interval variable sequencing in CP Optimizer*. https://www.ibm.com/docs/en/icos/20.1.0?topic=c-interval-variable-sequencing-in-cp-optimizer. 2021.

[34] IBM ILOG. *Interval variables in CP Optimizer*. https://www.ibm.com/docs/en/icos/20.1.0?topic=concepts-interval-variables-in-cp-optimizer. 2021.

[35] Julia Hanna. *Bringing 'Lean' Principles to Service Industries*. https://hbswk.hbs.edu/item/5741.html. 2007.

[36] Michael Jünger, Thomas M Liebling, Denis Naddef, George L Nemhauser, William R Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A Wolsey. *50 Years of integer programming 1958-2008: From the early years to the state-of-the-art*. Springer, 2010.

[37] John J Kanet. "Minimizing the average deviation of job completion times about a common due date". In: *Naval Research Logistics Quarterly* 28.4 (1981), pp. 643–651.

[38] H.A.T Kate, J. Wijngaard, and W.H.M. Zijm. *Minimizing weighted total earliness, total tardiness and setup costs*. Research Report 95A37. University of Groningen, Research Institute SOM (Systems, Organisations and Management), 1995.

[39] Safia Kedad-Sidhoum and Francis Sourd. "Fast neighborhood search for the single machine earliness–tardiness scheduling problem". In: *Computers & Operations Research* 37.8 (2010), pp. 1464–1471.

[40] Taha Keshavarz, Martin Savelsbergh, and Nasser Salmasi. "A branch-and-bound algorithm for the single machine sequence-dependent group scheduling problem with earliness and tardiness penalties". In: *Applied Mathematical Modelling* 39.20 (2015), pp. 6410–6424.

[41] Leonid G Khachiyan. "Polynomial algorithms in linear programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (1980), pp. 53–72.

[42] Changseong Ko, Sooyong Kim, Byungnam Kim, and Shiegheun Koh. "Single Machine Scheduling for Minimizing Earliness/Tardiness Penalties with Sequence-Dependent Setup Times". In: *Toward Sustainable Operations of Supply Chain and Logistics Systems*. Springer, 2015, pp. 265–278.

[43] Wen-Yang Ku and J Christopher Beck. "Mixed integer programming models for job shop scheduling: A computational analysis". In: *Computers & Operations Research* 73 (2016), pp. 165–173.

[44] Philippe Laborie and Daniel Godard. "Self-adapting large neighborhood search: Application to single-mode scheduling problems". In: *Proceedings MISTA-07, Paris* 8 (2007).

[45] Manuel Laguna, J Wesley Barnes, and Fred W Glover. "Tabu search methods for a single machine scheduling problem". In: *Journal of Intelligent Manufacturing* 2.2 (1991), pp. 63–73.

[46] Manuel Laguna and Fred Glover. "Integrating target analysis and tabu search for improved scheduling systems". In: *Expert Systems with Applications* 6.3 (1993), pp. 287–297.

[47] Xiying Li. *Scheduling Jobs with Sequence-Dependent Setup Times and Setup Costs to Minimize Total Production Cost*. BASc Thesis, University of Toronto. Toronto ON, 2019.

[48] C. E. Miller, A. W. Tucker, and R. A. Zemlin. "Integer Programming Formulation of Traveling Salesman Problems". In: *J. ACM* 7.4 (Oct. 1960), pp. 326–329. ISSN: 0004-5411. DOI: 10.1145/321043.321046.

[49] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. "Just-in-time scheduling with constraint programming". In: *Nineteenth International Conference on Automated Planning and Scheduling*. 2009.

[50] Laurent Péridy, Eric Pinson, and David Rivreau. "Using short-term memory to minimize the weighted number of late jobs on a single machine". In: *European Journal of Operational Research* 148 (2003), pp. 591–603.

[51] Petr Vilím. *How to minimize the gap in CP solver*. https://community.ibm.com/community/user/datascience/communities/community-home/digestviewer/viewthread?GroupId=5557&MessageKey=ae89ea4e-71b0-42be-8ebb-c3ee3094846d. 2020.

[52] Philippe Laborie. *How is the lower bound that is displayed created?* https://stackoverflow.com/questions/56150730/how-is-the-lower-bound-that-is-displayed-created. 2019.

[53] JT Presby and ML Wolfson. "An algorithm for solving job sequencing problems". In: *Management Science* 13.8 (1967), B–454.

[54] Ghaith Rabadi, Mansooreh Mollaghasemi, and Georgios C Anagnostopoulos. "A branch-and-bound algorithm for the early/tardy machine scheduling problem with a common due-date and sequence-dependent setup time". In: *Computers & Operations Research* 31.10 (2004), pp. 1727–1751.

[55] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[56] Jeffrey Schaller. "A comparison of lower bounds for the single-machine early/tardy problem". In: *Computers & operations research* 34.8 (2007), pp. 2279–2292.

[57] Jeffrey E Schaller and Jatinder ND Gupta. "Single machine scheduling with family setups to minimize total earliness and tardiness". In: *European Journal of Operational Research* 187.3 (2008), pp. 1050–1068.

[58] Paul Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *International conference on principles and practice of constraint programming*. Springer. 1998, pp. 417–431.

[59] Francis Sourd. "Dynasearch for the earliness–tardiness scheduling problem with release dates and setup constraints". In: *Operations Research Letters* 34.5 (2006), pp. 591–598.

[60] Francis Sourd. "Earliness–tardiness scheduling with setup considerations". In: *Computers & operations research* 32.7 (2005), pp. 1849–1865.

[61] Francis Sourd. "New exact algorithms for one-machine earliness-tardiness scheduling". In: *INFORMS Journal on Computing* 21.1 (2009), pp. 167–175.

[62] Francis Sourd and Safia Kedad-Sidhoum. "The one-machine problem with earliness and tardiness penalties". In: *Journal of scheduling* 6.6 (2003), pp. 533–549.

[63] Vincent T'kindt and Jean-Charles Billaut. "Just-in-Time scheduling problems". In: *Multicriteria Scheduling: Theory, Models and Algorithms*. Springer, 2006, pp. 135–191.

[64] Daniel Walker. *The better hospital: Excellence through leadership and innovation*. MWV Medizinisch Wissenschaftliche Verlagsgesellschaft, 2015.

[65] Li Wang and Mengguang Wang. "A hybrid algorithm for earliness-tardiness scheduling problem with sequence dependent setup time". In: *Proceedings of the 36th IEEE Conference on Decision and Control*. Vol. 2. IEEE. 1997, pp. 1219–1222.

[66] S Webster, D Jog, and A Gupta. "A genetic algorithm for scheduling job families on a single machine with arbitrary earliness/tardiness penalties and an unrestricted common due date". In: *International Journal of Production Research* 36.9 (1998), pp. 2543–2551.

[67] Charles H White and Richard C Wilson. "Sequence dependent set-up times and job sequencing". In: *The International Journal of Production Research* 15.2 (1977), pp. 191–202.

[68] William Cook. *TSP Applications*. https://www.math.uwaterloo.ca/tsp/apps/. 2007.