Universiteit Utrecht

Master Thesis Computing Science

Recoverable Robustness in Scheduling Problems

Author:

J.M.J. Stoef (3470997)
J.M.J.Stoef@uu.nl

Supervisors:

dr. J.A. Hoogeveen
J.A.Hoogeveen@uu.nl

dr. ir. J.M. van den Akker
J.M.vandenAkker@uu.nl

August 17, 2015

Abstract

Solving optimization problems is normally done with deterministic data. These data are however not always known with certainty. Recoverable robustness handles this uncertainty by modeling it in different scenarios. An initial solution is found and can be made feasible in each scenario by applying a given and simple recovery algorithm to it.

In this thesis the concept of recoverable robustness will be applied to different machine scheduling problems. First it is applied to the problem of minimizing the number of late jobs on one machine where the processing times of the jobs are uncertain. This problem is solved to optimality with the Moore-Hodgson algorithm when all data are certain. Applying recoverable robustness to this problem, where recovery can only be done by making extra jobs late, is proven to be weakly NP-complete when there is only one scenario. To solve the problem, different exact algorithms are developed, including a dynamic programming algorithm with pseudo-polynomial running time for a given number of scenarios. Branch and price is implemented with the use of the separate recovery framework as a lower bound. This algorithm and a branch and bound algorithm are then tested on their performance. When increasing the number of jobs or scenarios, instances quickly fail to be solved within three minutes. Overall branch and bound performs best.

In the field of scheduling with rejection, not all jobs need to be scheduled and some may be rejected which results in some penalty cost. To some of these problems for one machine recoverable robustness is applied. This problem is solved when minimizing the sum of the makespan and the total rejection penalty. The dynamic programming algorithm for minimizing the makespan with rejection when release dates are available is extended to handle the different scenarios of the recoverable robustness problem. Recoverable robustness is also applied on scheduling problems with rejection while minimizing the maximum lateness or maximum tardiness.

Contents

1 Introduction **5**

2 Basic Techniques **7**
2.1 Machine scheduling . . . 7
2.2 Linear Programming . . . 8
2.2.1 The Linear Program . . . 8
2.2.2 Column Generation . . . 9
2.2.3 Branch and Bound . . . 9
2.2.4 Branch and Price . . . 10
2.3 Recoverable Robustness . . . 10
2.3.1 Separate Recovery Decomposition Model . . . 11
2.3.2 Combined Recovery Decomposition Model . . . 11
2.4 Dynamic Programming . . . 12

3 Minimize the Number of Late Jobs on a Single Machine **14**
3.1 The $1||\sum U_j$-Problem . . . 14
3.1.1 Solution Methods . . . 15
3.2 Recoverable Robustness for the $1||\sum U_j$-Problem . . . 17
3.2.1 NP-Completeness . . . 18
3.2.2 Instances with Polynomial Time Algorithms . . . 20
3.3 Exact Algorithms . . . 27
3.3.1 Branch and Bound . . . 27
3.3.2 Separate Recovery Decomposition Model . . . 29
3.3.3 Branch and Price . . . 30
3.3.4 Dynamic Programming . . . 31
3.3.5 Direct Integer Linear Program . . . 32
3.4 Dominance Rules . . . 33

4 Scheduling with Rejection **38**
4.1 Introduction . . . 38
4.2 Minimizing the Maximum Makespan with Rejection . . . 39
4.2.1 Minimizing the Maximum Makespan with Rejection and Release Dates . . . 40
4.2.2 Recoverable Robustness for the $1|rej, r_j|C_{max} + RC$-Problem . . . 41
4.3 Minimizing L_{max} or T_{max} with Rejection . . . 43
4.3.1 Recoverable Robustness for $1|rej|L_{max} + RC$ and $1|rej|L_{max} + RC$ . . . 45

5 Experiments **47**
5.1 Problem Instances 47
5.2 Branching Algorithms Compared 48
5.2.1 Optimal Settings 49
5.2.2 Difficulty of Instances 52
5.2.3 Increased Number of Jobs and Scenarios 58
5.3 Direct Linear Program 62
5.4 Dynamic Programming 65
5.5 Conclusion 67

6 Conclusion and Further Research **69**

Chapter 1

Introduction

Consider a paint factory, consisting of various mixing machines that have to be operated by different employees. These mixing machines can produce different colors of paint with different structures. The number of liters that need to be made may also change per order placed by a customer. During one day in the factory, there are many orders that have to be executed on the different machines. The customers also have given a time on which they would like to have their order ready. The employees decide on how long it takes to produce the order. Now the factory needs to determine in which order the orders are processed on the different machines.

The objective of the paint factory is to minimize the number of late orders, where an order is late when it is finished after the requested moment by the customer. A customer will not be pleased when his order is finished after this time and therefore minimizing the number of late orders, or jobs, will result in the highest number of satisfied customers.

It may happen however that a customer decides that he wants his order a little later or earlier than requested or that an employee makes a guess on the processing time but they are not absolutely sure it takes this time. This gives a lot of uncertainty in the planning of the sequence of the orders on one day. It may even happen that not all of the supplies to produce one order are available at the start of a day and therefore working on an order can only be started when the necessary supplies are delivered. Of course one can never be certain when a supplier delivers its goods; a truck can always get stuck in traffic.

Determining the order of jobs on machines while optimizing an objective is researched in the field of machine scheduling. Given the objective a planner has and the situation that is being described, there are different algorithms to determine the optimal sequence of the orders when all the data are certain.

However, as explained for the painting factory, determining this sequence may be done under a lot of uncertainties. These uncertainties can be modeled in scenarios, where each scenario contains one situation that may occur. One solution to this problem for the factory is to find a schedule that is feasible for all different scenarios, without making adaptations to this solution in any situation, being robust against all possible variation. Such a solution can be difficult to find or might be expensive to execute. This is researched in the field of robust optimization [1]. There are two other research fields for creating robust solutions for all different scenarios; stochastic programming [2] and recoverable robustness [18, 5].

For stochastic optimization the uncertainties are represented in random variables, which follow a probability distribution of some kind. Two-phase stochastic programming finds a solution that is feasible for almost all scenarios. Because the solution is not feasible for all scenarios, some recourse action needs to be taken when the information changes. This recourse action is not restricted in its size and thus might be large

changes. The objective in the first step is then to optimize the cost of the initial solution together with the cost of the recourse action.

Recoverable robustness creates a solution that can be made feasible for each scenario when a certain given recovery algorithm is used. An initial solution is created and from this initial solution the scenario solutions can be created with a small adjustment. So one scenario needs to be the main scenario, called the initial problem. The concept of recoverable robustness was introduced in [18] and applied to railway optimization problems. In [6] the recoverable robust version of the knapsack problem is studied. In this version of the knapsack problem all data is uncertain and different scenarios are created. Here is shown that because the knapsack problem is weakly NP-hard, it is easily proven that the recoverable robust knapsack problem is also weakly NP-hard for a fixed number of scenarios. This can be done for each NP-hard problem that recoverable robustness is applied to. For an unbounded number of scenarios the problem is proven to be strongly NP-hard.

In [5], two decomposition frameworks were developed to solve recoverable robustness for a given problem. The separate recovery decomposition model separates the parts for creating the initial solution and recovery part. The combined recovery decomposition model combines the initial problem and the recovery part for each scenario. These models are implemented with the use of column generation. The models are applied to the size robust knapsack problem, where only the size of the knapsack is uncertain, and results showed that the separate recovery model obtained the nice results.

Both these decomposition models are implemented for the size robust multiple knapsack problem in [24]. Next to the models, two greedy approaches were introduced.

In this thesis the recoverable robustness approach will be investigated to solve the problem for the painting machine in which employees are not doing a great job in estimating the processing times of the orders. The factory is simplified by assuming there is only one machine. Minimizing the number of late jobs for one machine is solved to optimality in polynomial time with the Moore-Hodgson algorithm [19] when all information is certain. The recoverable robustness approach is defined for this problem where recovery can be done by making extra jobs late. It is investigated whether there are polynomial time solutions for this recoverable robustness problem when the processing times are uncertain.

It turns out however that finding a recoverable robust solution for this problem is NP-hard when considering only one possible scenario other than the initial problem and the processing time of an order only increases compared to the initially assumed time. Therefore different exact algorithms, like applying the separate recovery algorithm of [5], are developed and their performance is tested on different types of instances.

When the factory does not want to process all orders and is willing to lose customers to other factories or outsource the work to other partners, some orders might be rejected. These orders are then not taken into account when sequencing the orders optimally. Machine scheduling problems involving rejection are almost always NP-hard and therefore solving the recoverable robustness problem is in these cases also NP-hard. For various machine scheduling problems with rejection the existing dynamic programming algorithms are extended to also be able to handle the uncertainties in the recoverable robustness problem.

In Chapter 2 some preliminary knowledge that is used in different parts of this thesis is discussed. Then in Chapter 3 the recoverable robustness approach for minimizing the number of late jobs is investigated. Different algorithms are developed for this problem. The extensions of the dynamic programming algorithms for the machine scheduling problems with rejection are covered in Chapter 4. The results of the experiments on the different exact algorithms for the recoverable robustness problem while minimizing the number of late jobs are presented in Chapter 5. In Chapter 6 conclusions are given on the different findings and developed algorithms and future research is discussed.

Chapter 2

Basic Techniques

In this chapter the basic techniques that are necessary to understand the rest of this thesis are discussed. First in Section 2.1 the field of machine scheduling is covered. Next linear programming is discussed in Section 2.2, together with various algorithms to solve a linear program. Next recoverable robustness is explained extensively together with the separate recovery model in Section 2.3. At last dynamic programming is covered in Section 2.4.

2.1 Machine scheduling

The field of machine scheduling concerns the problem of optimally allocating recourses to jobs over time, as stated in [16]. Generally, there are m machines that have to process n jobs $J_1, \ldots, J_n$. Each machine can only process one job at a time and each job can only be worked on by one machine at any moment in time. A solution of a machine scheduling problem is called a schedule. Here, one or more non-overlapping time intervals on one or more machines are allocated to each job. A schedule is feasible if the previously stated demands are met. The schedule is optimal if it optimizes a given optimality criterion.

There are many different characteristics for the machines, jobs and optimality criteria. The field of machine scheduling is deterministic: all the information that is necessary to describe a problem instance is known with certainty in advance. For a certain job J_j the following information may be known:

- A processing time p_{ij} which denotes the time it takes to process job J_j on machine i.
- A release date r_j which denotes when job J_j becomes available.
- A due date d_j which denotes when job J_j is preferably be completed.
- A weight w_j which denotes the importance of job J_j.
- A cost function f_j which calculates the cost when job J_j finishes at a certain time.

Not all of these properties are necessary for each problem instance. Which are needed depends on the optimality criterion and which situation is being described. The processing times are the only values that are always necessary. To denote which information of a job J_j is known and what the other characteristics of a problem instance are, Graham et al. described in [9] a three-field notation $\alpha|\beta|\gamma$. This notation is commonly used in the field of machine scheduling.

The first field consists of one or two symbols: one number specifies the number of machines m, and the other specifies the type of machines used. If the number is left out, there are m machines. When the field

contains a P, there are identical parallel machines which means that processing a job takes the same time on each machine. In the case of Q, there are uniform parallel machines which means that each machine executes the jobs at a given speed. When the value is R all the machines are unrelated. There are many more options, but these are not in the scope of this thesis. The reader is referred to [9].

The second field indicates the additional job characteristics. It may contain $\bar{d}_j$, which indicates that each job J_j has a strict deadline on which this job has to be finished. When it contains *pmtn*, this denotes that preemption is allowed. This means that the processing of a job may be stopped and continued at a later point. When the field contains *prec*, this means that there are precedence constraints between jobs which are specified by a directed graph. If the field contains *tree* then this graph is a tree. If it contains r_j, it means that the release dates of the jobs are specified, otherwise the release dates are assumed to be zero. Finally it can contain $p_{ij} = 1$, which means that each operation that has to be performed has unit processing times.

The last field defines the optimality criterion. Given a schedule the following values can be computed for each job J_j:

- Its completion time C_j, the time when every machine is done processing job J_j.
- Its lateness $L_j = C_j - d_j$ denotes the time difference between the completion time and due date. When the value is positive this job was late, when it is negative the job was on-time.
- The tardiness $T_j = \max\{0, L_j\}$.
- The penalty U_j which is one when $T_j > 0$ and zero otherwise.

With these values, different optimality criteria can be constructed. Most common are C_{max} and L_{max}, which denote the maximum completion time over all jobs and the maximum lateness of each job, respectively. Other commonly used criteria for example are $\sum C_j, \sum T_j, \sum U_j$ and $\sum w_j C_j$.

In [17] an extensive overview is given on the complexity of different scheduling problems. The authors also refer to the polynomially-bounded algorithms that were created for many problems.

2.2 Linear Programming

2.2.1 The Linear Program

In a linear programming problem a linear objective function $f(\mathbf{x}) = \mathbf{c}^\mathbf{T}\mathbf{x}$ has to be optimized given a set of linear constraints $\mathbf{Ax} \leq \mathbf{b}$. Here, $\mathbf{x}$ is the vector of decision variables, $\mathbf{c}$ and $\mathbf{b}$ are vectors and $\mathbf{A}$ is a matrix all with known coefficients. A linear program (LP) can always be written in the following standard form, where 'min' is short for minimize:

$$\min f(\mathbf{x}) = \mathbf{c}^\mathbf{T}\mathbf{x}$$

subject to

$$\begin{aligned} \mathbf{Ax} &\leq \mathbf{b} \\ \mathbf{x} &\geq 0 \end{aligned}$$

The last constraint states that the decision variables have to be nonnegative. Additionally, it can be demanded that the variables need to be integral. Then the problem is an integer linear program (ILP). A special case of an ILP is when $\mathbf{x} \in \{0,1\}$, which makes it a binary integer linear program, or a $0-1$ integer linear program. When only some of the decision variables need to be integral, the problem is called a mixed

integer linear program (MILP).

There are many commonly used algorithms to solve linear programming problems. The algorithms that will be used in this thesis are explained in the next sections. In Section 2.2.2, column generation is explained. Next, in Section 2.2.3, branch and bound is explained, and lastly branch and price is covered in Section 2.2.4, which solves ILPs.

2.2.2 Column Generation

A method for solving large LPs is column generation. When using column generation to solve an LP, the problem is first solved for only a subset of the decision variables. The solution that is found might be optimal for the entire problem, then you stop. If the solution is not optimal, more variables, columns, need to be added and the problem is solved again. This is repeated until the optimal solution is obtained.

To investigate whether the obtained solution is optimal, or which columns can be added, the reduced cost can be calculated of each variable that is not yet in the problem. The reduced cost of a variable x denotes the net gain per unit of adding this variable. When minimizing the linear program, adding a variable with negative reduced cost can decrease the value of your solution. When no variable with negative reduced cost exists the LP has reached its optimum. The reduced cost is calculated by $\mathbf{c} - \mathbf{A^T}\omega$ where ω is the shadow price vector. The shadow price of a constraint is the unit price someone would want to pay for a little more of this resource. The shadow prices are calculated when solving the LP.

There are two main problems that are needed to solve an LP with column generation, namely a pricing problem and a master problem. The pricing problem finds the variable for which the reduced cost is minimal in the case of minimizing the LP. This variable is then added to the master problem, which solves the LP for a limited amount of variables. After this the pricing problem can be solved again, until no variable is found that has negative reduced costs.

Column generation can also be used for finding a lower bound on the solution of an ILP. To obtain this lower bound the integrality constraints of the ILP need to be relaxed, so $\mathbf{x} \geq 0$. This results in a restricted master problem (RMP) which can be solved with column generation. If the solution of the RMP is non-integer, then this solution is a lower bound on the solution of the ILP when the objective is being minimized. When the solution of the RMP is integer, then this is the optimal solution of the ILP.

In [13] a nice overview is given on applying column generation to various machine scheduling problems.

2.2.3 Branch and Bound

The branch and bound algorithm consists of two steps. In the branching step a set Q of possible solutions is split into two or more smaller sets $Q_1, Q_2, \ldots, Q_z$. In the bounding step a lower bound and an upper bound for the objective function $f(\mathbf{x})$ are calculated while only considering the solutions in a set Q_i.

The algorithm starts at the root node where the set Q consists of all possible solutions. For this node the bounding step is executed, so a lower and upper bound are obtained. For this node the branching step is then performed creating z child nodes. For these nodes the lower and upper bounds are also calculated. For each node the branching step can be performed again. Branching stops when $|Q| = 1$ in a node. These leaf nodes have only one solution and thus the optimal solution is obvious. This solution will be stored as the best solution if its value is better than the current best solution.

In certain situations it might not be useful to branch a node. If the lower bound of the node is greater than the upper bound of any of the other nodes, then branching this node will never result in a better solution than the node with the corresponding upper bound. If the lower bound in a node is equal to the upper bound in this node the optimal solution of this node is found. This solution is compared to the current best solution and stored as the new best when its value is lower than the current best value. When a node does not have a valid solution, this branch is stopped.

When there are no nodes left to branch on, the algorithm terminates. The optimal solution of the problem is then found. The order in which you branch the nodes can be done in a breadth first order or depth first order. When going through the tree in breadth first order, a newly created node is added at the end of the node queue. When it is done depth first, a node is added at the front of the queue. Another order is best first, where the node with the lowest lower bound is the first node to branch.

2.2.4 Branch and Price

Branch and price is a special case of the branch and bound algorithm, as it uses the branch and bound principle for solving an ILP. The lower bound calculated at each node is obtained by column generation. Additionally, the child nodes inherit the columns from their parent that are feasible columns in these nodes. The linear program is then solved including extra constraints to maintain the characteristics of the node, so only feasible columns for this node will be added.

Branching a node can now also be stopped when the lower bound obtained by the column generation is obtained by an integer solution. This solution is then the optimal solution for this branch and there is no need to continue. This solution is again stored when it has a better value than the current best value.

2.3 Recoverable Robustness

A way to deal with uncertain parameters in an optimization problem is to model them into a set of scenarios S. Each scenario handles a different situation that might occur. Recoverable robustness [18] is a method to find a solution for the initial problem and each scenario $s \in S$ that might occur. The solution that is calculated for the initial problem can be recovered to the solution for a scenario by the use of a given recovery algorithm. Unlike robust optimization [1] where such recovery is not allowed and the solution should also be feasible for each scenario $s \in S$.

Consider the following optimization problem

$$\min f(x) | x \in X$$

This is the initial problem, with X the set of feasible solutions and x the decision variables. This initial problem then has a set of scenarios S for which Y^s is the set of feasible solutions for scenario s and the decision variables are y^s. Let $\mathcal{A}$ be the set of admissible recovery algorithms. For a recovery algorithm $A \in \mathcal{A}$ it must hold that $A(x, s) \in Y^s$, i.e. $A(x, s)$ computes a feasible solution for scenario s from an initial solution x. Let in addition R_s be the feasible recovery set for scenario s, this set contains combinations of initial solutions and solutions to the scenario problem that satisfy the recovery algorithm as denoted below.

$$R_s = \{(x, y^s) | A(x, s) = y^s\} \quad \forall s \in S$$

The difference with two-phased stochastic programming [2] is that the recovery possibilities are limited. The recovery robust optimization problem ($RROP$) is now denoted as follows, as stated in [5].

$$\min f(x) + \sum_{s \in S} g(y^s, s)$$

subject to

$$\begin{aligned} x &\in X \\ y^s &\in Y^s \\ (x, y^s) &\in R^s \quad \forall s \in S \end{aligned}$$

This problem can then be solved by enumerating over all feasible combinations of solutions from X and Y^s for each scenario $s \in S$. The following sections explain how this method can be improved with the use of column generation.

2.3.1 Separate Recovery Decomposition Model

In [5] separate recovery is mentioned for the first time to solve the recoverable robustness problem. As mentioned earlier, when the full sets X and Y^s are enumerated to solve the *RROP* problem, a very large number of combinations has to be investigated to determine the optimal value. Therefore [5] suggests to only look at subsets of X and Y^s, thus $X^{'} \subseteq X$ and $Y^{'s} \subseteq Y^s$ are introduced. These sets start out small and new variables will be added to obtain a better solution, which thus makes this a column generation approach. A separate master problem (SMP) is introduced.

$$\min f(x) + \sum_{s \in S} g(y^s, s)$$

subject to

$$\begin{aligned} x &\in X^{'} \\ y^s &\in Y^{'s} \\ (x, y^s) &\in R^s \quad \forall s \in S \end{aligned}$$

To find new solutions for the sets $X^{'}$ and $Y^{'s}$ there are pricing problems for each set. A separate recovery initial pricing problem (SRIPP) for set $X^{'}$ and a separate recovery pricing problem (SRPP^s) for each scenario set $Y^{'s}$. These problems find the variables that have negative reduced cost, and these variables are added to the SMP. Once these problems do not find variables with negative reduced cost, the solution of the SMP is optimal. The solution of the column generation approach might be non-integral. If it is not required that the solution of the RROP is integral, the problem is solved to optimality. If the optimal solution of the SMP is integral, this solution is always the optimal solution of the RROP. When the solution of the SMP is non-integral and the RROP problem requires an integral solution, additional steps need to be taken. This can be done with branch and price for example, as explained in Section 2.2.4, where the separate recovery decomposition model is used to find the lower bound in each node.

The exact forms of the SRIPP and SRPP^s problems depend on for which type of optimization problem the recoverable robustness model is being solved.

2.3.2 Combined Recovery Decomposition Model

The separate recovery decomposition model makes solutions to the initial problem and the scenario problem independently of each other and then tries to combine them such that they satisfy the recovery constraint. The combined recovery decomposition model works in exactly the opposite way. For each scenario, a solution for the initial problem is made together with a solution for the scenario that satisfy the recovery constraints.

All these combinations are then tried to be combined in such way that the initial solutions are equal. This decomposition model is also developed in [5].

Here a combined master problem (CMP) is introduced. The RROP gave a feasible recovery set R^s for each scenario $s \in S$. In the CMP the restricted recovery sets $R^{'s} \subseteq R^s$ are considered. Again this set starts small and new variables will be added to obtain a better solution, following the column generation approach. With this the CMP is defined as follows.

$$\min f(x) + \sum_{s \in S} g(y^s, s)$$

subject to

$$\begin{aligned} (x^{'}_s, y^s) &\in R^s \quad \forall s \in S \\ x &= x^{'}_s \end{aligned}$$

To find new solutions for each set $R^{'s}$, there are pricing problems for each set, a combined pricing problem (CPP^s) for each scenario $s \in S$. From here on the combined recovery decomposition model works the same as the separate one, the CCP^s problems are solved to find variables with negative reduced cost until the solution of the CMP is optimal. The exact form of the CPP^s again depends on the type of optimization problem the model is applied to.

2.4 Dynamic Programming

Dynamic programming is used to solve many optimization problems. The algorithm solves a problem by dividing the problem into subproblems and combining their solutions to find the optimal solution for the main problem. When computing the solution of a subproblem the answer to this subproblem is stored such that it does not have to be computed again when it is needed later on.

In [8] dynamic programming is explained by showing the algorithm for the cutting stock problem. In this problem several rods of the same length need to be cut in smaller pieces that can be sold and the profit needs to be maximized. In this thesis this algorithm is demonstrated using it for the knapsack problem. In the knapsack problem there are n items that have a weight a_j and a revenue c_j. The items can be put into the knapsack as long as the total weight of the items does not exceed the knapsack size B. The goal is to maximize the revenue. This problem is a classic optimization problem.

To define the subproblems a recursive formula is created. $D_j(w)$ is defined as the best revenue that can be achieved when using a subset of the items in set $\{1, 2, \dots, j\}$ such that the weight of the knapsack is exactly w. When $w = 0$, no items can be in the knapsack thus the revenue will be zero, so $D_j(0) = 0 \quad \forall j$. It holds that $D_0(w) = -\infty \ \forall w \neq 0$ because this will not be a valid solution. Otherwise $D_j(w)$ is defined as

$$D_j(w) = \max\{D_{j-1}(w), D_{j-1}(w - a_j) + c_j\}.$$

For each item j it has to be decided whether it will be in the knapsack or not. If the item is not chosen the weight of the knapsack considering items $\{1, 2, \dots, j-1\}$ has to be w, so in this case $D_j(w) = D_{j-1}(w)$. If the item is taken then the value of the knapsack considering only items from $\{1, 2, \dots, j-1\}$ will be equal to $D_{j-1}(w - a_j)$. Adding the item increases the value with the revenue c_j of item j and thus in this case $D_j = D_{j-1}(w - a_j) + c_j$. The value of $D_j(w)$ is then equal to the maximum over these both cases.

The optimal value v_{max} of the problem is then equal to $\max_w D_n(w)$. The optimal solution can be found by backtracking for each item j whether it was put into the knapsack or not. Let $W = \sum a_j$, then at most $O(nW)$ values need to be calculated and calculating each value costs constant time, thus this algorithm takes $O(nW)$ time.

Chapter 3

Minimize the Number of Late Jobs on a Single Machine

In this chapter the single machine scheduling problem of minimizing the number of late jobs is discussed. In Section 3.1 the problem will be explained together with various solution methods. Next the recoverable robustness concept is applied to the problem in Section 3.2 and NP-hardness is proven here. In Section 3.3 the exact algorithms to solve the NP-hard problem are discussed. Lastly, in Section 3.4 some dominance rules are shown to simplify the recoverable robustness problem for minimizing the number of late jobs.

3.1 The $1||\sum U_j$-Problem

Consider n jobs $J_1, \ldots, J_n$ that have to be scheduled on one machine. For each job J_j a processing time p_j is known, this is the time it takes to process the job on the machine. Each job J_j also has a due date d_j. Each job is preferably finished before this time. The lateness of a job is defined as $L_j = C_j - d_j$, where C_j is the completion time of job J_j. The tardiness is then defined as $T_j = \max\{L_j, 0\}$. The penalty U_j is defined such that it equals one when $T_j > 0$ and zero otherwise. Preemption is not allowed, moreover, it would not result in a better schedule for this problem, so when the machine starts working on a job, it can only stop working on that job when the job is finished. Furthermore, each job is available from time zero.

With this information the 'minimize the number of late jobs' problem can be defined. For job $J_1, \ldots, J_n$ a schedule needs to be created that minimizes the number of late jobs. A job J_j is late when $T_j > 0$ and thus $U_j = 1$. An optimal schedule now needs to minimize $\sum U_j$. In the three-field notation of [9] this problem is defined as $1||\sum U_j$.

An example for this problem is given in Table 3.1. There are five jobs, $J_1, \ldots, J_5$ for which the processing times p_j and due dates d_j are given. The minimum number of late jobs is two and the corresponding schedule is made when J_2 is scheduled first then job J_3 and then job J_5. Jobs J_1 and J_4 are scheduled at the end and will be late.

In this problem all jobs are equally important. Sometimes however, there might be jobs for which it is more important to be on time than others. To accomplish this, a weight w_j may be assigned to each job J_j and the objective will be to minimize $\sum w_j U_j$, giving the $1||\sum w_j U_j$ problem. A job with a high weight is now more likely to be on time. To show this, weights are added to the jobs from the example in Table 3.1, see Table 3.2. Job J_1 will have a weight of three, the rest still has a weight of one.

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	4	3	2	5	6

(a) The problem instance.

	J_2	J_3	J_5	J_1	J_4
d_j	7	8	11	6	9
C_j	3	5	11	~~15~~	~~20~~

(b) The optimal solution.

Table 3.1: Schedule n jobs to minimize the number of late jobs.

	J_1	J_2	J_3	J_4	J_5
w_j	3	1	1	1	1
d_j	6	7	8	9	11
p_j	4	3	2	5	6

(a) The problem instance.

	J_1	J_4	J_2	J_3	J_5
w_j	3	1	1	1	1
d_j	6	9	7	8	11
C_j	4	9	~~12~~	~~14~~	~~20~~

(b) The optimal solution.

Table 3.2: Schedule n jobs to minimize the weighted number of late jobs.

The original solution when all the weights were equal had a solution value of two, using the same solution with the new weights would have a value of four. In Table 3.2 the optimal solution for the new problem is shown. By making three jobs late now, job J_1 can be on time and only gives a solution value of three. It is important to set the weights properly. If the weight of job J_1 would have been only two, the two solutions would have equal value and job J_1 would not necessarily be on time. To force a job J_j to be on time, the value w_j can always be set to $+\infty$.

For more variants on the $1||\sum U_j$ problem, the reader is referred to [12].

3.1.1 Solution Methods

This section explains how the optimal solutions of the previous section are obtained. In an optimal solution the jobs can always be divided into two sets. The set E of the jobs that are early and the set L of the jobs that are late. The jobs in L can always be scheduled after all the jobs from E because it does not matter how late the late jobs are, it just matters that they are late. In [19] it is proven that in an optimal solution the set E can always be scheduled in earliest due date order (EDD). This means that when job J_j and J_i are both in E and $d_j < d_i$, then job J_j is scheduled before job J_i. These jobs are scheduled without idle time, meaning that when a job is finished, the next job starts immediately. The job in E with the smallest due date starts at time zero. Therefore a schedule can always be determined when the sets E and L are known.

The problem of scheduling n jobs, $J_1, \ldots, J_n$ jobs with known processing times, $p_1, \ldots, p_n$ and due dates, $d_1, \ldots, d_n$ in such order that it minimizes the number of late jobs is known to be solved to optimality in polynomial time by the Moore-Hodgson algorithm [19].

Moore-Hodgson Algorithm The Moore-Hodgson algorithm divides the jobs in the two sets E and L from which the optimal schedule can easily be obtained as explained earlier. The algorithm consists of the following steps:

1. Set σ to be the schedule in which all the jobs are in EDD-order, thus $d_1 \leq d_2 \leq \ldots \leq d_n$. In the case of equal due dates the order does not matter.

2. Find the first job J_j in the order which is late. If there is no such job, this schedule is optimal.
3. Let job J_m be the longest job from the set of all the predecessors of J_j and J_j itself, in σ.
4. Remove job J_m from schedule σ and move all the jobs to make sure there is no idle time left. Repeat step 2 with the new schedule.

The jobs that are left in schedule σ are the jobs in set E. The deleted jobs will form set L together. The number of late jobs in the resulting schedule, i.e. the size of the set L, is called the *value* of the solution.

This algorithm can be made to run in $O(n \log n)$ if a maximum heap is used to find the longest job in step 3. An alternative algorithm that runs in $O(n^2)$ is the shortest processing time (SPT) algorithm, as explained in [10].

Shortest Processing Time Algorithm

1. Start with re-ordering the jobs in shortest processing time order, so $p_1 \leq \ldots \leq p_n$. In the case that processing times are equal, the job with the earliest due date comes first. The sets E and L start empty.
2. Find the first job in the order that is not in E or L. Put the jobs from $E \cup J_j$ in EDD-order. If each job is on time, job J_j is added to E. Otherwise, job J_j is added to L.
3. Continue until the last job is reached. Then set E contains all the on time jobs and the set L contains all the late jobs.

This algorithm runs slower than the Moore-Hodgson algorithm, so why would it be used? The advantage of this algorithm is that when a job is added to the set E it will never get removed from it. In the Moore-Hodgson algorithm a job is only surely in the set E when the algorithm terminates. This is be a useful property.

These algorithms can however not be used for the $1||\sum w_j U_j$-problem. This problem is shown to be NP-complete; for a proof the reader is referred to [17]. It is proven that there is no polynomial time algorithm, unless P = NP. The following dynamic programming algorithm, that can be found in [12], gives the optimal solution for the $1||\sum w_j U_j$-problem. To use the algorithm the jobs need to be ordered such that $d_1 \leq d_2 \leq \ldots \leq d_n$. The running time of this algorithm is $O(n \sum p_j)$. Naturally this algorithm can also be used for the $1||\sum U_j$-problem, here simply all the w_j's in the algorithm are one. In this case this algorithm still runs in pseudo-polynomial time.

Dynamic Programming Algorithm

$$f_j(t) = \begin{cases} \infty & \text{if } t > d_j \\ \min\{f_{j-1}(t) + w_j, f_{j-1}(t - p_j)\} & \text{otherwise} \end{cases}$$

It is known that the jobs that are on time are scheduled in EDD-order. Thus it can be decided in that order whether to schedule job J_j or not. Therefore this algorithm is performed after the jobs are ordered such that $d_1 \leq d_2 \leq \ldots \leq d_n$. Let $f_j(t)$ be the optimal value of the objective function when the total processing time of the on time jobs is t. Initially $f_j(t) = 0$ if $j = t = 0$ and ∞ otherwise. If job J_j is scheduled then the total processing time of the jobs that are scheduled from $J_1, \ldots, J_{j-1}$ is $t - p_j$, thus $f_j(t) = f_{j-1}(t - p_j)$. If job J_j is chosen to be late then the total processing time of the jobs that are scheduled from $J_1, \ldots, J_{j-1}$ is still t, but the total objective increases with w_j. Thus in this case $f_j(t) = f_{j-1}(t) + w_j$. The optimal value is then the minimum over these two cases. In the case that $t > d_j$ this solution will not be feasible and thus

the value will be ∞. After calculating all the values, the optimal solution value is equal to $\min_t f_n(t)$. The solution corresponding to this value can be found by backtracking.

Integer Linear Programming The last method to solve the problem is using linear programming. Although this may not be the most straightforward method, it will be used in the remainder of this thesis. In [14] assignment and positional date variables are used to make an ILP-formulation for the $1||\sum U_j$-problem. Here the variable x_{ij} is used with the following property.

$$x_{ij} = \begin{cases} 1 & \text{if job } J_i \text{ is assigned to position } j \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j$$

The first two constraints make sure that each job is on exactly one spot and one spot is filled by exactly one job. The variable l_k will denote the lateness of the job at spot k. The variable u_k denotes whether the job on spot k is late and is defined by the following piecewise linear function.

$$u_k = \begin{cases} 1 & \text{if } l_k > 0 \\ 0 & \text{otherwise} \end{cases} \quad \forall k$$

The linear program however requires a linear function, this equation can be rewritten to $l_k \leq (\sum p_j - d_{min})u_k$ with $d_{min} = \min_j d_j$ and $\sum p_j - d_{min}$ is the maximum possible lateness. These are all the necessary constraints. The objective is to minimize the number of late jobs, so this is to minimize $\sum u_j$. This gives the following integer linear program:

$$\min \sum_{j=1}^{n} u_j$$

subject to

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \tag{3.1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \tag{3.2}$$

$$\sum_{i=1}^{k} \sum_{j=1}^{n} p_j x_{ij} - \sum_{j=1}^{n} d_j x_{kj} = l_k \quad \forall k \tag{3.3}$$

$$l_k \leq (\sum p_j - d_{min})u_k \quad \forall k \tag{3.4}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \tag{3.5}$$

$$u_k \in \{0,1\} \tag{3.6}$$

$$\tag{3.7}$$

3.2 Recoverable Robustness for the $1||\sum U_j$-Problem

This section will be focused on solving the recoverable robustness model for the $1||\sum U_j$-problem when the processing times are uncertain but all other parameters are known for certain. This situation occurs in the

paint factory when the customers are not allowed to change their due date and the factory has to guess on the processing time on the given orders. This can be because the machine works a bit slower than expected or the order simply takes more time to produce than initially calculated.

The initial problem has n jobs $J_1, \ldots, J_n$ where each job J_j has a processing time p_j and a due date d_j. To model the uncertainties of the processing times there is a set S of scenarios where the n jobs have the same due dates as in the initial problem and each job has a processing time p_j^s for each scenario $s \in S$ which might be different from the processing time of job J_j in the initial problem. As explained in Section 2.3 there now can be found a set X with feasible solutions for the initial problem and a set Y^s with all the feasible solutions for each scenario $s \in S$.

As explained in the previous section a solution $x \in X$ or $y^s \in Y^s$ is defined by a set E with the on time jobs and a set L with the late jobs. This division of the jobs can also be denoted by a variable u_j, as used in the linear programming expression. This variable u_j is one if job J_j is late and thus is in set L and zero otherwise. For each $x \in X$ this gives variables u_{xj} $\forall j$ and for each $y^s \in Y^s$ this gives a $u_{y^s j}$ $\forall s \in S, j$.

In recoverable robustness now a recovery algorithm needs to be defined. As stated earlier not every possible recovery is allowed, as is the case in two-phased stochastic programming. In this thesis the recovery can be done by making additional jobs from the initial set E late, such that the resulting schedule is feasible in a scenario. The feasible recovery set for a scenario $s \in S$ is now defined as $R_s = \{(x, y^s) | u_{xj} \leq u_{y^s j} \quad \forall j\}$. A job that was late in the initial schedule cannot be on time in a scenario schedule, every other combination is allowed. Each scenario $s \in S$ occurs with a probability p_s and the initial problem occurs with a probability p_0. It must hold that

$$p_0 + \sum_{s \in S} p_s = 1.$$

The recoverable robustness problem for minimizing the number of late jobs (RRML) is now defined as

$$\min p_0 \sum_j u_{xj} + \sum_{s \in S} p_s \sum_j u_{y^s j}$$

subject to

$$\begin{aligned} x &\in X \\ y^s &\in Y^s \\ (x, y^s) &\in R^s \quad \forall s \in S \end{aligned}$$

A special case of the RRML problem is when in each scenario the processing times of the jobs can only increase. In this case it holds that $p_j \leq p_j^s$ for each job J_j and each scenario $s \in S$. This problem will be denoted as the recoverable robustness problem for minimizing the number of late jobs with only increase (RRML-I). In the previous section the $1||\sum w_j U_j$ was also discussed. For this problem the recoverable robustness problem (RRML-W) can also be defined. The only change is the objective which becomes $\min \sum_j w_j u_{xj} + \sum_{s \in S} \sum_j w_j u_{y^s j}^s$.

3.2.1 NP-Completeness

In a first attempt to solve the RRML problem it might be a good guess to use the Moore-Hodgson solution of the initial problem as the schedule for the initial problem in the optimal solution of the RRML problem. It is however not always optimal to do so, as is shown in the example in Tables 3.3. These tables show

a small instance of the RRML problem. In Table (a) the Moore-Hodgson solutions are shown. Here jobs J_2, J_3 and J_4 are on time when the Moore-Hodgson solution is calculated for the initial problem. These jobs can however not be on time in the scenario because job J_4 would violate its due date. Instead job J_5 will be processed on time to reach the optimal number of on time jobs. Using these two separate solutions as a solution to the RRML problem is not a valid solution because job J_5 violates the $u_j \leq u_j^s$ constraint and thus the recovery algorithm. In Table (b) the optimal solution is shown, where in the initial problem an other solution than the Moore-Hodgson solution is used.

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	~~4~~	③	②	④	~~6~~
p_j^s	~~4~~	③	②	~~8~~	①

(a) The Moore-Hodgson solutions

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	~~4~~	③	②	~~4~~	⑥
p_j^s	~~4~~	③	②	~~8~~	①

(b) An optimal solution for the RRML problem

Table 3.3: Counter example to the conjecture that using the Moore-Hodgson schedule of the initial problem as the inital schedule in RRML is optimal.

This example suggests that when there is only one scenario, using the Moore-Hodgson schedule of the scenario as the scenario solution in the RRML solution gives the optimal solution. But the example in Table 3.4 proves wrong again. Just increasing J_5 in the initial problem with one compared to the previous example, results in a counter example. Now job J_5 cannot be on time after job J_2 and J_3 and thus the previous solution is not valid anymore and there is no other possibility than making an additional job late.

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	~~4~~	③	②	④	~~7~~
p_j^s	~~4~~	③	②	~~8~~	①

(a) The Moore-Hodgson solutions

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	~~4~~	③	②	④	~~7~~
p_j^s	~~4~~	③	②	~~8~~	~~1~~

(b) An optimal solution for the RRML problem

Table 3.4: Counter example to the conjecture that using the Moore-Hodgson schedule of the scenario as the scenario schedule in RRML is optimal.

This means that the most straight forward solutions are not always valid and thus finding the optimal solution for an RRML problem might be difficult. It turns out that it is really difficult. Although the problem $1||\sum U_j$ can be solved in polynomial time, the RRML problem turns out to be weakly NP-complete, even when the due dates are all equal. It can be checked in polynomial time that a given solution to the RRML problem is valid and that the number of late jobs is not too high. Thus the problem is in NP. What remains is to prove that the RRML problem is NP-hard. The reduction is done from a variant of the partition problem, which is weakly NP-complete. The variant of partition is defined as follows.

Partition with cardinality constraint Given $2n$ positive integers $a_1, \ldots, a_{2n}$ with sum equal to $2A$, does there exist a subset R of size n of the index set $\{1, \ldots, 2n\}$ such that

$$\sum_{i \in R} a_i = A$$

Theorem 1. *The RRML problem is weakly NP-hard, even when* $|S| = 1$.

Proof. Given an instance of partition with $2n$ integers, the following instance of the decision variant of the RRML problem is created with $2n$ jobs and one scenario s. Let $M > 3A$, each integer a_i with $i \in \{1, \ldots, n\}$ corresponds to a job J_i with processing times $p_i = M - a_i$ and $p_i^s = M + a_i$. The due date for each job is $d = nM + A$. Special job J_0 is added with processing times $p_0 = 2A$ and $p_0^s = Q$, where $Q > M + A$. The due date of this job is also $d = nM + A$. The probability of the scenario, p_s, is equal to $\frac{1}{2}$ and the probability of the initial problem, p_0, is then of course as well $\frac{1}{2}$. The decision variant of the RRML problem now is: Does this instance have a feasible solution with a solution value not more than $n + \frac{1}{2}$, corresponding with exactly $2n + 1$ late jobs?

If the answer to the partition problem is "yes", then let R be the subset of $\{1, \ldots, 2n\}$ such that $\sum_{i \in R} a_j = A$. There now can be constructed a "yes" instance to the RRML problem by putting job J_i with $i \in R$ on time in both the initial schedule and the scenario schedule and if $i \notin R$ the job is late in both schedules. In the initial schedule job J_0 is also on-time. This gives a solution value of $n + \frac{1}{2}$.

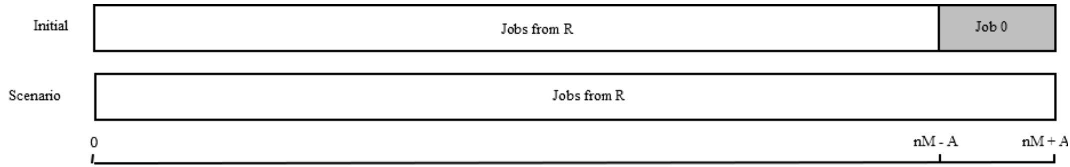

Figure 3.1: The Gantt chart of the optimal solution of the problem

If the answer to the RRML problem is "yes", let the set R contain the indices of the jobs that are on time in the initial schedule, except job J_0. For this schedule to be feasible, there need to be at least $n + 1$ jobs on time in the initial schedule. Because not more than n jobs can be on time from jobs $J_1, \ldots, J_{2n}$ job J_0 needs to be on time as well. It now holds that $2A + \sum_{i \in R}(M - a_i) = 2A + nM - \sum_{i \in S} a_i \leq nM + A$ and thus $\sum_{i \in R} a_i \geq A$. In the scenario schedule now n jobs need to be on time, this can only be the jobs that are on time in the initial schedule. So it must hold that $\sum_{i \in R}(M + a_i) = nM + \sum_{i \in R} a_i \leq nM + A$ and thus $\sum_{i \in R} a_i \leq A$. Thus it can be concluded that $\sum_{i \in R} a_i = A$.

The answer to partition is "yes" if and only if the answer to the RRML problem is "yes", so the problem is weakly NP-hard. □

This means that there is no polynomial time algorithm for the RRML problem unless $P = NP$. For a fixed number of scenarios a pseudo-polynomial-time algorithm can be obtained by using dynamic programming. More on the dynamic programming algorithm can be found in Section 3.3.4.

3.2.2 Instances with Polynomial Time Algorithms

Although the general RRML problem has now been proven to be NP-complete there might be instances of the problem that can be solved in polynomial time. Theorem 2 proves that although the Moore-Hodgson solutions cannot be used in the optimal solution of each problem directly, the solution values do give interesting information on the optimal solution value of the $RRML$ problem.

Theorem 2. *Solve the Moore-Hodgson algorithm for the initial problem and for each scenario $s \in S$ separately. Let v be the value of the Moore-Hodgson schedule of the initial problem and let w_s be the value of the Moore-Hodgson schedule of scenario $s \in S$. It holds that $v + \sum_{s \in S} p_s w_s$ is a lower bound on the optimal value of the RRML problem.*

Proof. Let the value of the optimal solution for RRML be smaller than $v + \sum_{s \in S} p_s w_s$. For this to be true there has to be at least one schedule that has fewer late jobs than the number of late jobs in its Moore-Hodgson schedule. This can not be true because the value of the Moore-Hodgson schedule is the minimum number of late jobs that is necessary to make all the other jobs be on time. So the value has to be at least $v + \sum_{s \in S} p_s w_s$. □

Thus the Moore-Hodgson schedules of the initial problem and all the scenarios together form a lower bound on the solution value of a RRML problem. This means that when a valid solution is found with a solution value equal to this lower bound, this solution is optimal. When considering the Moore-Hodgson solutions of all problems together as a solution for the RRML problem, this solution is only not valid if there is a job J_j for which holds that this is late in the Moore-Hodgson schedule of the initial problem and there is a scenario $s \in S$ in which job J_j is on time. This violates the recovery algorithm. This means however, that if such a job J_j does not exist, this solution is valid and optimal. This together gives the following lemma:

Lemma 1. *Consider an instance of the RRML problem. The Moore-Hodgson schedules of the initial problem and the scenarios form the optimal solution to the RRML problem, if there does not exist a job J_j that is late in the initial schedule and on time in a schedule of a scenario $s \in S$.* □

Whether such job J_j exists can easily be checked in $O(n|S|)$ time. This is a check that can be done quite quickly and if it has a positive result the optimal solution is found already. Only if such a job exists another algorithm needs to be used to find a valid and optimal solution.

If a RRML-I problem is considered more information about the lower bound can be obtained because more information is available on the value of the Moore-Hodgson solution of a scenario. When a set of jobs is on time in a scenario schedule with only increased processing times compared to the initial problem, these jobs can also be on time in the initial problem. With this information Theorem 3 is proven.

Theorem 3. *Let v be the value of the Moore-Hodgson schedule of the initial problem. Let s be a scenario where the processing times of the jobs are only increased. The value w of the Moore-Hodgson schedule of s will be bigger than or equal to v.*

Proof. Let $w < v$. It now holds that $n - w > n - v$ and thus the number of on time jobs in the scenario is larger than the number of on time jobs in the initial problem. The on time jobs of the scenario schedule are all contained in the set E_s. If we look at the jobs in E_s in the initial problem, they all have smaller or equal processing times in that problem. So the jobs in E_s can be on time in the initial problem as well. But this contradicts our assumption that $n - v$ is the maximum number of on time jobs in the initial problem. □

When considering an instance of the RRML-I with only one scenario, there now is an easy optimal solution when the Moore-Hodgson solutions of the initial problem and the scenario have the same value. Consider the set E_s from the Moore-Hodgson solution of the scenario. All these jobs have longer or equally long processing times as these jobs in the initial problem. Thus these jobs can also be on time in the initial problem. Thus making the jobs from E_s on time in both schedules of the solution of the RRML-I problem gives a valid solution. This solution is also optimal because the value is equal to the lower bound.

Now consider again an instance of the RRML-I problem with one scenario, but now the scenario has only one job J_j with an increased processing time, for all the other jobs the processing time in the scenario is equal to the initial problem. When the value of the Moore-Hodgson schedule of the initial problem is v, it is proven in theorem 4 that the value of the Moore-Hodgson schedule of the scenario now is v or $v + 1$.

Theorem 4. *Let v be the value of the Moore-Hodgson schedule of the initial problem. Let s be a scenario where only the processing time of one job j is increased. The value w of the Moore-Hodgson schedule of s will be v or $v + 1$.*

Proof. In Theorem 3 it is proven that w will not be smaller than v. So all that is left is to prove that the Moore-Hodgson schedule of s will never have a value bigger than $v+1$. There is always a schedule for scenario s with at most $v+1$ late jobs. Simply make the same jobs late as in the Moore-Hodgson schedule of the initial problem. If job J_j is late now, this schedule is valid. Otherwise make job J_j the extra late job in this schedule. The jobs that are left on time were on time in the initial schedule too, so they do not violate their due date. So we can always make a feasible schedule with $v+1$ late jobs, so the Moore-Hodgson value for scenario s will never be bigger than $v+1$. □

With the use of Theorems 2 and 4 an algorithm can be created for the RRML-I problem with one scenario in which only one job J_j has an increased processing time. The first step is to calculate the Moore-Hodgson solution for the initial problem and the scenario. The value of the initial schedule is v. From theorem 4 we know that the value of the Moore-Hodgson schedule of the scenario is v or $v+1$. We look at both cases.

Case 1, the value of the Moore-Hodgson schedule of the scenario is $v+1$: From Theorem 2 it follows that $2v+1$ is the lower bound of the value of the solution. An easy $2v+1$ solution can be constructed. Make all the jobs from the L set of the Moore-Hodgson solution of the initial problem late in both the initial schedule and the scenario and make job J_j late extra in the scenario schedule. This would not be possible if job J_j was already late in the Moore-Hodgson solution for the initial problem. But this can not be the case, otherwise the Moore-Hodgson value of the scenario solution would be v. So we have a solution for the lower bound value, giving the optimal solution.

Case 2, the value of the Moore-Hodgson schedule of the scenario is v: From Theorem 2 it follows that $2v$ is the lower bound of the value of the solution. An easy $2v$ solution can be constructed. Simply take the on time set E from the Moore-Hodgson solution of the scenario and make the same set on time in the initial problem. This is a valid solution as explained earlier and the solution is equal to the lower bound value, thus the solution is optimal.

Calculating the Moore-Hodgson solution for an instance takes $O(n \log n)$ time, this has to be done only twice. When these values are obtained the optimal solution is then easily obtained in $O(1)$ time. Thus finding the optimal solution to the RRML-I with one scenario in which only one job J_j has an increased processing time can be calculated in $O(n \log n)$ and thus in polynomial time.

A	J_1	J_2
p_j	○	○
p_j^s	○	○

B	J_1	J_2
p_j	○	○
p_j^s	○	/

C	J_1	J_2
p_j	○	/
p_j^s	○	○

D	J_1	J_2
p_j	○	/
p_j^s	○	/

E	J_1	J_2
p_j	○	○
p_j^s	/	○

F	J_1	J_2
p_j	○	○
p_j^s	/	/

G	J_1	J_2
p_j	○	/
p_j^s	/	○

H	J_1	J_2
p_j	○	/
p_j^s	/	/

K	J_1	J_2
p_j	/	○
p_j^s	○	○

L	J_1	J_2
p_j	/	○
p_j^s	○	/

M	J_1	J_2
p_j	/	/
p_j^s	○	○

N	J_1	J_2
p_j	/	/
p_j^s	○	/

P	J_1	J_2
p_j	/	○
p_j^s	/	○

Q	J_1	J_2
p_j	/	○
p_j^s	/	/

R	J_1	J_2
p_j	/	/
p_j^s	/	○

S	J_1	J_2
p_j	/	/
p_j^s	/	/

Table 3.8: The sixteen different possible Moore-Hodgson outcomes for the RRML-I when $n = 2$.

Small problem instances Sometimes when instances of difficult problems are small there is an easy solution to this problem. Therefore this will be investigated now for the RRML-I problem with one scenario, where both the scenario and the initial problem are equally important. When only considering one scenario a job J_j has four possible configurations when looking at the two different Moore-Hodgson schedules. It can be late or on time in both schedules and all combinations can occur. This means that when there are n jobs, there are 4^n possible outcomes for the Moore-Hodgson solutions. This exponential function gets big very quickly, but for $n = 2, 3$ and 4 it might be doable to take a look at these instances and maybe find small instances that are easily to solve.

When $n = 2$ there are sixteen different possible Moore-Hodgson outcomes, as all shown in Tables 3.8. A ○ denotes that this job is on time and / means that it is late. Not all of these outcomes can occur because it is proven in Theorem 3 that the Moore-Hodgson solution value of a scenario is bigger than or equal to the Moore-Hodgson solution value of the initial problem value. Considering all these possible configurations it holds that C, K, M, N, and R are not valid because of Theorem 3. When A, B, D, E, F, H, P, Q or S occurs, these Moore-Hodgson schedules form the optimal solution together because there is a valid schedule equal to the lower bound as proven in Theorem 2 and explained in Lemma 1. This leaves G and L and these will be investigated further.

G	J_1	J_2
p_j	(p_1)	~~p_2~~
p_j^s	~~p_1^s~~	(p_2^s)

L	J_1	J_2
p_j	~~p_1~~	(p_2)
p_j^s	(p_1^s)	~~p_2^s~~

Table 3.9: The two valid Moore-Hodgson outcomes with no direct optimal solution for $n = 2$

For both instances, given in Table 3.9, it holds that the lower bound of the optimal solution is equal to two. If there is a feasible schedule, which does not violate the recovery algorithm, with only two late jobs, this schedule is optimal. In G job J_2 can be finished before its due date in the scenario and thus $p_2^s \leq d_2$. It also holds that $p_2 \leq p_2^s$ because only RRML-I instances were considered, thus job J_2 can also be on time when job J_1 is not on time in the initial schedule. Thus there is an optimal solution with job J_2 as the only job on time in both schedules. The same argumentation can be constructed for L for job J_1.

This means that solving the RRML-I solution for only two jobs and one scenario will have straightforward solutions that can be concluded from the Moore-Hodgson solutions. When increasing n to three there are 64 possible configurations, which are too many to enumerate here all. From these 64 there are only fifteen that are not invalid, because the Moore-Hodgson value of the scenario is smaller than or equal to that of the initial problem, or are a direct solution, because of Lemma 1. These fifteen configurations are shown below in Table 3.13.

For all these possibly difficult problems, there are twelve instances that have an initial Moore-Hodgson solution value equal to the scenario Moore-Hodgson value. All these have easy optimal solutions with the

set E_s from the scenario Moore-Hodgson schedule on time in both schedules of the solution. This leaves G, N and Q that do not have a direct solution. However, it may be assumed that $p_i^s \leq d_i \; \forall i$. This means that the one job that is on time in the scenario in these examples can also be one of the other jobs in the scenario schedule. This creates a valid schedule with a value equal to the lower bound of the RRML-I problem for the three possible instances left. This means that when $n = 3$ the optimal solution of an instance can also easily be obtained by inspecting the Moore-Hodgson solutions of both problems.

A	J_1	J_2	J_3
p_j	○	/	○
p_j^s	○	○	/

B	J_1	J_2	J_3
p_j	○	○	/
p_j^s	○	/	○

C	J_1	J_2	J_3
p_j	/	/	○
p_j^s	/	○	/

D	J_1	J_2	J_3
p_j	/	○	/
p_j^s	/	/	○

E	J_1	J_2	J_3
p_j	/	○	○
p_j^s	○	○	/

F	J_1	J_2	J_3
p_j	/	○	○
p_j^s	○	/	○

G	J_1	J_2	J_3
p_j	/	○	○
p_j^s	○	/	/

H	J_1	J_2	J_3
p_j	/	○	/
p_j^s	○	/	/

K	J_1	J_2	J_3
p_j	/	/	○
p_j^s	○	/	/

L	J_1	J_2	J_3
p_j	○	○	/
p_j^s	/	○	○

M	J_1	J_2	J_3
p_j	○	/	○
p_j^s	/	○	○

N	J_1	J_2	J_3
p_j	○	/	○
p_j^s	/	○	/

P	J_1	J_2	J_3
p_j	○	/	/
p_j^s	/	○	/

Q	J_1	J_2	J_3
p_j	○	○	/
p_j^s	/	/	○

R	J_1	J_2	J_3
p_j	○	/	/
p_j^s	/	/	○

Table 3.13: The fifteen different possible Moore-Hodgson outcomes for the RRML-I when $n = 3$.

Adding another job, making $n = 4$, gives 256 possible configurations of which there are 74 feasible outcomes that do not have a directly visible optimal solution. Again many of them can be eliminated because the initial problem and the scenario Moore-Hodgson solution have an equal value. Other instances can be eliminated because they have only one job on time in the scenario Moore-Hodgson solution. This can easily be swapped to another job resulting in an optimal solution. What is left are the following twelve instances.

	J_1	J_2	J_3	J_4
p_j	○	/	○	○
p_j^s	○	○	/	/

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	(4)	~~5~~	(3)	(2)
p_j^s	(5)	(6)	~~7~~	~~8~~

	J_1	J_2	J_3	J_4
d_j	8	15	16	20
p_j	(6)	~~7~~	(5)	(8)
p_j^s	(7)	(8)	~~10~~	~~14~~

	J_1	J_2	J_3	J_4
p_j	○	○	/	○
p_j^s	○	/	○	/

	J_1	J_2	J_3	J_4
d_j	6	9	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	(6)	~~7~~	(8)	~~9~~

	J_1	J_2	J_3	J_4
d_j	6	9	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	(6)	~~7~~	(8)	~~11~~

	J_1	J_2	J_3	J_4
p_j	○	○	○	/
p_j^s	○	/	/	○

	J_1	J_2	J_3	J_4
d_j	7	9	16	17
p_j	(4)	(5)	(6)	~~7~~
p_j^s	(6)	~~7~~	~~10~~	(9)

	J_1	J_2	J_3	J_4
d_j	6	9	14	15
p_j	(3)	(5)	(6)	~~8~~
p_j^s	(6)	~~7~~	~~9~~	(8)

	J_1	J_2	J_3	J_4
p_j	/	○	○	○
p_j^s	○	○	/	/

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	~~5~~	(4)	(3)	(2)
p_j^s	(6)	(5)	~~8~~	~~9~~

	J_1	J_2	J_3	J_4
d_j	8	15	16	20
p_j	~~7~~	(6)	(5)	(8)
p_j^s	(8)	(7)	~~10~~	~~14~~

	J_1	J_2	J_3	J_4
p_j	/	○	○	○
p_j^s	○	/	○	/

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	~~5~~	(4)	(3)	(2)
p_j^s	(6)	~~8~~	(5)	~~9~~

	J_1	J_2	J_3	J_4
d_j	8	15	16	19
p_j	~~7~~	(6)	(5)	(8)
p_j^s	(8)	~~10~~	(7)	~~14~~

	J_1	J_2	J_3	J_4
p_j	/	○	○	○
p_j^s	○	/	/	○

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	~~5~~	(4)	(3)	(2)
p_j^s	(6)	~~8~~	~~9~~	(5)

	J_1	J_2	J_3	J_4
d_j	8	15	16	19
p_j	~~7~~	(6)	(5)	(8)
p_j^s	(8)	~~10~~	~~14~~	(11)

	J_1	J_2	J_3	J_4
p_j	○	○	/	○
p_j^s	/	○	○	/

	J_1	J_2	J_3	J_4
d_j	7	9	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	~~7~~	(6)	(8)	~~9~~

	J_1	J_2	J_3	J_4
d_j	7	9	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	~~7~~	(6)	(8)	~~11~~

	J_1	J_2	J_3	J_4
p_j	○	○	○	/
p_j^s	/	○	/	○

	J_1	J_2	J_3	J_4
d_j	7	9	16	17
p_j	(4)	(5)	(6)	~~7~~
p_j^s	~~7~~	(6)	~~10~~	(9)

	J_1	J_2	J_3	J_4
d_j	7	9	15	17
p_j	(4)	(5)	(6)	~~9~~
p_j^s	~~7~~	(6)	~~10~~	(11)

	J_1	J_2	J_3	J_4
p_j	○	/	○	○
p_j^s	/	○	○	/

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	(4)	~~5~~	(3)	(2)
p_j^s	~~7~~	(6)	(5)	~~8~~

	J_1	J_2	J_3	J_4
d_j	10	15	16	19
p_j	(6)	~~7~~	(5)	(8)
p_j^s	~~10~~	(8)	(7)	~~14~~

	J_1	J_2	J_3	J_4
p_j	○	/	○	○
p_j^s	/	○	/	○

	J_1	J_2	J_3	J_4
d_j	6	11	12	13
p_j	(4)	~~5~~	(3)	(2)
p_j^s	~~7~~	(6)	~~8~~	(5)

	J_1	J_2	J_3	J_4
d_j	13	15	16	19
p_j	(6)	~~7~~	(5)	(8)
p_j^s	~~13~~	(8)	~~14~~	(7)

	J_1	J_2	J_3	J_4
p_j	○	○	○	/
p_j^s	/	/	○	○

	J_1	J_2	J_3	J_4
d_j	9	10	16	17
p_j	(4)	(5)	(6)	~~7~~
p_j^s	~~9~~	~~10~~	(7)	(8)

	J_1	J_2	J_3	J_4
d_j	8	10	14	15
p_j	(3)	(5)	(6)	~~8~~
p_j^s	~~8~~	~~10~~	(7)	(9)

	J_1	J_2	J_3	J_4
p_j	○	○	/	○
p_j^s	/	/	○	○

	J_1	J_2	J_3	J_4
d_j	8	9	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	~~8~~	~~9~~	(7)	(8)

	J_1	J_2	J_3	J_4
d_j	8	11	14	16
p_j	(4)	(5)	~~6~~	(7)
p_j^s	~~8~~	~~11~~	(7)	(9)

Table 3.25: The twelve different possible Moore-Hodgson outcomes for the RRML-I when $n = 4$.

The twelve instances left are shown in Tables 3.25, in the left column the Moore-Hodgson schedules are shown. The middle and right column show concrete examples of instances that would result in these Moore-Hodgson solutions. In all these examples there is a job that violates the recovery algorithm but there is a difference between the instances in the middle and right column. For the problems in the middle column an optimal solution of three can be found, thus equal to the lower bound. These solutions can be created by replacing one of the on time jobs in the scenario with one of the late jobs in either the initial problem or the scenario. For the instances in the right column this is not possible. This would violate the due dates of some jobs. Thus for instances with these Moore-Hodgson schedules an optimal solution can not be found easily.

This means that for instances of the RRML-I problem with only one scenario the instances already get difficult to solve when $n = 4$. From this moment on there can be instances that have different optimal solutions when the processing times are different. Unfortunately there is no easy rule to check whether the optimal solution is equal to the lower bound or extra jobs have to made late in these cases. This makes these twelve instances difficult to solve and this is only the number when $n = 4$. This will only increase when n gets even larger.

3.3 Exact Algorithms

Solving an NP-complete problem cannot be done in polynomial time, unless P=NP. One way to find a good solution quickly is to use an approximation algorithm. Such an algorithm may run in polynomial time but will not always find an optimal solution. If the optimal solution value is v^*, an ϵ-approximation algorithm will then give a value v for which holds that $v \leq \epsilon v^*$ for $\epsilon > 1$. Other approximation algorithms exist that do not give an upper bound on the value found solution. If the exact optimal solution is needed an exact algorithm can be used. In this section four exact algorithms are explained for the RRML problem. First branch and bound is covered in Section 3.3.1, next the separate recovery framework from Section 2.3.1 is applied in Section 3.3.2, after branch and price is explained in Section 3.3.3, dynamic programming is done after that in Section 3.3.4 and last in Section 3.3.5 the direct integer linear program.

3.3.1 Branch and Bound

As explained in Section 2.2.3 branch and bound consists of two steps, the branching step and the bounding step. Many different choices can be made on how to define these steps. Below is explained which steps are used in the branch and bound algorithm for the RRML problem. Consider an instance of RRML with n jobs and a set $|S|$ of scenarios.

Branching step In the root node of the branching tree the set Q consists of all possible solutions of the problem. Now consider a job J_j, this job has $2^{|S|} + 1$ possible configurations in a solution. It can be late in all the schedules of the solution or on time in the initial schedule and then there are two options for each scenario thus giving the $2^{|S|}$ extra possibilities. In the first branching step the set Q is split into $2^{|S|} + 1$ subsets. In each subset only those solutions are allowed that satisfy one of the configurations of job J_j. For example, when $|S| = 1$ this means that three subsets are created. In set Q_1 job J_j must be on time in both schedules, in set Q_2 job J_j must be on time in the initial schedule and late in the scenario schedule and in the last set Q_3 job J_j must be late in both schedules.

After branching on job J_j each subset can be made into $2^{|S|} + 1$ smaller subsets again by allowing only one possible configuration for a next job J_i. This can be continued until the configuration of each job is determined and thus the size of the set will only be one.

All the jobs that are already branched on will be in the set B that is stored with the branch node so that it can easily be determined what jobs are already done. It might happen that a given configuration that is wanted for a job may violate a due date for some job because other jobs are also set to be on time. In this case the set Q will not contain any feasible solutions thus branching on this node is not longer necessary.

The last thing to decide on in each branching step is on which job to branch. The following orderings can be used:

- Increasing/decreasing due dates.
- Increasing/decreasing processing times.
- On a job that does not satisfy the recovery algorithm when looking at the Moore-Hodgson solutions. If no such job exists, the optimal solution is found in this branch node.

Bounding step In the bounding step a lower and upper bound need to be calculated. These only exist when the nodes contains a feasible solution. To verify if this exists, it needs to be checked that all the jobs that are determined to be on time can all fit within their due dates. This can easily be checked in $O(n)$ time.

If this is not the case this nodes contains no feasible solutions. If there are some feasible solutions, a lower bound on the optimal solution when considering all possible solutions can easily be found by calculating the Moore-Hodgson solutions as proven in Theorem 2. When the lower bound needs to be calculated in a non root node for some jobs the configuration is determined and this might not be equal to their configuration in the Moore-Hodgson solution. For these lower bounds the Moore-Hodgson algorithm needs to be adapted a bit to handle the possible configurations that are set for a certain job.

Consider the instance from Section 3.1, in the optimal solution jobs J_1 and J_4 were late. Now consider a set B in which it is set for this schedule that J_1 is on time and job J_5 is late. In the optimal solution now jobs J_1 and J_3 are on time.

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	(4)	~~3~~	(2)	~~5~~	~~6~~

Table 3.26: Schedule 5 jobs to minimize the number of late jobs when job J_1 is on time and job J_5 is late.

This can be determined by adapting the Moore-Hodgson algorithm from Section 3.1.1 in the following manner. In step one the jobs that are predetermined to be late are not even added in the initial schedule σ. So these jobs are not taken into consideration to be on time. In step three the longest job may be a job that must be on time. When this job needs to be removed the next longest job is taken instead. In the implementation with the maximum heap the jobs that must be on time are not inserted in heap. To make sure this algorithm gives a valid solution, it needs to be checked that all the jobs that are determined to be on time can all fit within their due dates. This was checked at the start of the bounding step. Thus a lower bound can be calculated for each set Q in the branching tree.

Any feasible solution that can be created is a valid upper bound. An upper bound can therefore be determined very easily. In the root node when all solutions are allowed an upper bound is made by putting all jobs late. This makes the upper bound value always equal to $p_0 n + \sum_{s \in S} p_s n$. Once the set B increases this value will get lower. It is easily counted how many jobs are set to be late in this set B. In addition to this all the jobs that do not have their configuration set are counted as late in each schedule. These numbers together will form the upper bound. The corresponding solution always gives feasible solutions because it is known that the set on time jobs fit. This upper bound is however not that tight.

A feasible solution in a branch node with set B that gives a stricter upper bound can be created as follows. Calculate the Moore-Hodgson solution of the initial problem and take into consideration the set configurations of the different jobs in B. This gives a set E with on time jobs and the set L of late jobs. In each schedule for a scenario now all the jobs from set L are set late. Consider now the jobs from set E and perform for each scenario $s \in S$ Moore-Hodgson on this set of jobs and take the branched jobs into consideration because otherwise it could violate the possible solutions of set B. The jobs that are on time in the Moore-Hodgson schedule of these smaller problems are on time in the scenario schedules of the complete problems. These schedules together form a valid solution for a branch node and therefore a valid upper bound for the RRML problem considering set B. That this upper bound does not give an optimal solution to the RRML problem was already shown in Section 3.2.

This makes that the bounding step now has a lower bound which is calculated by performing Moore-Hodgson on all nodes. For the upper bound there are two options. In the first, all jobs that are left to be determined are all set late, therefore this upper bound is called *everything late*. In the other upper bound

the Moore-Hodgson solution of the initial problem is used as a starting point. This upper bound is therefore called *Moore-Hodgson initial*.

3.3.2 Separate Recovery Decomposition Model

The separate recovery framework is implemented with column generation in [5] and in [4] for the size robust knapsack problem. The same can be done for the RRML problem. Let x_r and y_r^s be the binary decision variables for the problem.

$$x_r = \begin{cases} 1 & \text{if the } r\text{th schedule in } X \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

$$y_r^s = \begin{cases} 1 & \text{if the } r\text{th schedule in } Y^s \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

The schedules in X and Y^s are further defined by the following parameters:

$$u_{jr} = \begin{cases} 1 & \text{if job } j \text{ is late in the } r\text{th schedule in } X \\ 0 & \text{otherwise} \end{cases}$$

$$u_{jr}^s = \begin{cases} 1 & \text{if job } j \text{ is late in the } r\text{th schedule in } Y^s \\ 0 & \text{otherwise} \end{cases}$$

The integer linear program for the separate recovery model for the RRML problem is now formulated as:

$$\min p_0 \sum_r (\sum_{j=1}^{n} u_{jr}) x_r + \sum_{s \in S} p_s \sum_r (\sum_{j=1}^{n} u_{jr}^s) y_r^s$$

subject to

$$\sum_r x_r = 1 \quad (3.8)$$

$$\sum_r y_r^s = 1 \quad \forall s \in S \quad (3.9)$$

$$\sum_r u_{jr} x_r - \sum_r u_{jr}^s y_r^s \leq 0 \quad \forall j, s \in S \quad (3.10)$$

$$x_r \in \{0,1\} \quad \forall r \quad (3.11)$$

$$y_r^s \in \{0,1\} \quad \forall r, s \in S \quad (3.12)$$

Constraint (3.8) makes sure that from the set X with possible schedules for the initial problem only one is chosen. Constraint (3.9) does that for each scenario $s \in S$. Constraint (3.10) makes sure that each chosen scenario schedule can be created from the chosen initial schedule by making extra jobs late.

This model will then be solved using column generation. The constraints for x_r and y_r^s need to be relaxed to $x_r \geq 0$ and $y_r^s \geq 0$ to obtain the separate master problem (SMP). Recall that subsets $X^{'} \subset X$ and in $Y^{'s} \subset Y^s$ are used during the column generation. These sets contain the current solutions that are found for the initial problem and each scenario. Initial solutions are needed to put in $X^{'}$ and in $Y^{'s}$ for each $s \in S$ to solve the LP-relaxation for the first time. The first solution chosen here to put in each set is the schedule with all jobs late, so $u_{j0} = 1$ for each job J_j and $u_{j0}^s = 1$ for each job J_j and each scenario $s \in S$. Later new

solutions are put in these subsets.

When minimizing the SMP a better solution can be obtained when there is a solution for the initial problem or a scenario that has negative reduced costs. The dual variable for constraint (3.8) is λ, for constraints (3.9) this is μ_s for each scenario $s \in S$ and the last constraints have π_{sj} for each job J_j and scenario $s \in S$. The reduced costs for x_r and y_r^s are as follows:

$$\begin{aligned} \text{Reduced costs } x_r \quad &: \quad p_0 \sum_{j=1}^n u_{jr} - \sum_{j=1}^n \sum_{s \in S} u_{jr} \pi_{js} - \lambda \\ &= \quad \sum_{j=1}^n (u_{jr}(p_0 - \sum_{s \in S} \pi_{js})) - \lambda \end{aligned}$$

$$\begin{aligned} \text{Reduced costs } y_r^s \quad &: \quad p_s \sum_{j=1}^n u_{jr}^s + \sum_{j=1}^n u_{jr}^s \pi_{js} - \mu_s \\ &= \quad \sum_{j=1}^n (u_{jr}^s (p_s + \pi_{js})) - \mu_s \end{aligned}$$

For finding a new initial schedule to add to $X^{'}$ a valid schedule needs to be found which minimizes $\sum_{j=1}^n w_j u_j - \lambda$ where $w_j = p_0 - \sum_{s \in S} \pi_{js}$. The same needs to be done for each scenario $s \in S$. A valid schedule which minimizes $\sum_{j=1}^n w_j u_j^s - \mu_s$ where $w_j = p_s + \pi_{js}$ needs to be found. While one of these problems gives a negative value these solutions can be added to the SMP and repeat the procedure. When all the reduced costs are nonnegative the optimal solution for the SMP is found.

In these pricing problems the problem of finding a schedule while minimizing the number of late jobs with different weights for each job can be recognized. Finding a schedule for these pricing problems can thus be done with the dynamic programming algorithm described in Subsection 3.1.1.

The result of this column generation approach might however result in a non-integral solution. To obtain an integral solution this approach can be used as the lower bound in a branch and price algorithm, this will be covered in Subsection 3.3.3.

3.3.3 Branch and Price

As explained in Section 2.2.4 the branch and price algorithm is a special case of the branch and bound algorithm. Therefore all of the information explained on the branching steps in Section 3.3.1 holds also for this algorithm. Except for the calculation of the lower bound. This is not done here with the help of the Moore-Hodgson algorithm. Here a column generation approach is used to solve the LP-relaxation of the separate recovery model as explained in Section 3.3.2.

When calculating the lower bound in a non-root node, for some of the jobs their configuration is already determined. The column generation approach should be able to handle this, as was done in the branch and bound algorithm for the Moore-Hodgson algorithm that was used to calculate the lower bound in a node. The dynamic programming algorithm from Subsection 3.1.1 that is used in the pricing problem can be adapted as following. Below this algorithm is shown for the initial algorithm. With different processing times this is used for the pricing problems on each scenario as well.

$$f_j(t) = \begin{cases} \infty & \text{if } t > d_j \\ f_{j-1}(t) + w_{j+1} & \text{if job } J_j \text{ is configured late} \\ f_{j-1}(t - p_{j+1}) & \text{if job } J_j \text{ is configured on time} \\ \min\{f_{j-1}(t) + w_{j+1}, f_{j-1}(t - p_{j+1})\} & \text{otherwise} \end{cases}$$

One other difference is that the third branching strategy cannot be used because such job is not defined anymore in this case.

3.3.4 Dynamic Programming

The dynamic programming algorithm explained in Section 3.1.1 can easily be extended to solve the RRML problem. Of course it must still hold that the jobs are ordered such that $d_1 \leq \ldots \leq d_n$. First consider the following algorithm for the case with one scenario. Let $f_j(r,t)$ be equal to the value of the optimal solution considering only jobs $J_1, \ldots, J_j$ where the total processing time of the on time jobs of the initial schedule is r and the total processing time of the on time jobs of the scenario schedule is t. Initialize $f_j(r,t) = 0$ if $j = r = t = 0$ and ∞ otherwise. The values can now be calculated given the following recurrence relation

$$f_{j+1}(r,t) = \begin{cases} \infty & \text{if } r > d_{j+1} \quad \text{or} \quad t > d_{j+1} \\ \min\{f_j(r,t) + p_0 + p_1, f_j(r - p_{j+1}, t) + p_1, f_j(r - p_{j+1}, t - p^s_{j+1})\} & \text{otherwise} \end{cases}$$

If all these values are calculated correctly, the optimal solution can be determined by taking $\min_{r,t} f_n(r,t)$. There are $O(n \sum p_j \sum p^s_j)$ values that need to be calculated, each taking $O(1)$ time. This algorithm thus takes $O(n \sum p_j \sum p^s_j)$ time and space. If $P = \max\{\sum p_j, \sum p^s_j\}$, this can be written as $O(nP^2)$.

Theorem 5. *The values of $f_j(r,t)$, with $j \in \{0, \ldots, n\}$, $r \in \{0, \ldots, \sum_{i=1}^{j} p_i\}$ and $t \in \{0, \ldots, \sum_{i=1}^{j} p^s_i\}$ are calculated correctly.*

Proof. This theorem is proven by induction. Suppose the values of $f_j(r,t)$ are correct for $r \in \{0, \ldots, \sum_{i=1}^{j} p_i\}$ and $t \in \{0, \ldots, \sum_{i=1}^{j} p^s_i\}$. This is true when $j = 0$. Now is shown that the recurrence relation calculates the correct values for $f_{j+1}(r,t)$. If $r > d_{j+1}$ or $t > d_{j+1}$ the last job in one of the schedules completes after its deadline and thus this solution is infeasible. If $r \leq d_{j+1}$ and$t \leq d_{j+1}$ there are three possibilities. The first option is make job J_{j+1} late in both schedules, this gives the first minimand. The next option is to make job J_{j+1} on time in the initial schedule and therefore jobs $J_1, \ldots, J_n$ must fit in the interval $[0, r - p_{j+1}]$ in the initial schedule and is represented by the second minimand. The last option is to make job J_{j+1} on time in both schedules and thus jobs $J_1, \ldots, J_n$ must fit in the interval $[0, r - p_{j+1}]$ in the initial schedule and in the interval $[0, t - p^s_{j+1}]$ in the scenario schedule. Taking the smallest value of these three options gives the correct value of $f_j(r,t)$. □

This algorithm can easily be extended for more than one scenario. Change $f_j(r,t)$ into $f_j(r, t_1, \ldots, t_{|S|})$ where t_s is the total processing time of all the on time jobs from $J_1, \ldots, J_j$ in the schedule of scenario $s \in S$. The value of $f_j(r, t_1, \ldots, t_{|S|})$ will then be calculated by minimizing over the values for all the combinations of setting job J_j late and on time in the different schedules. There are $2^{|S|} + 1$ possible combinations. If P then is the maximum value of all the total processing time of each of the scenarios and the initial scenario. This algorithm then runs in $O(n2^{|S|}P^{|S|+1})$ time, which is not pseudo-polynomial for unknown $|S|$. Whether the RRML problem is strongly NP-hard is an open problem.

The RRML-W problem can easily be proven to be NP-hard, simply because the $1||\sum w_j U_j$ is already NP-hard. The dynamic programming algorithm for this problem is very similar to the algorithm for the RRML problem, except that a late job J_j does not contribute p_j to the objective but $w_j p_j$. This gives the following dynamic programming algorithm algorithm for one scenario, which can be extended to $|S|$ scenarios as well.

$$f_{j+1}(r,t) = \begin{cases} \infty & \text{if } r > d_{j+1} \text{ or } t > d_{j+1} \\ \min\{f_j(r,t) + p_0 w_0 + p_1 w_1, f_j(r - p_{j+1}, t) + p_1 w_1, f_j(r - p_{j+1}, t - p^s_{j+1})\} & \text{otherwise} \end{cases}$$

3.3.5 Direct Integer Linear Program

An other way to solve the *RRLM* problem is by adapting the linear program described in paragraph 3.1.1 to also handle the different scenarios. Let x_{ij}^s be the decision variable defined as

$$x_{ij}^s = \begin{cases} 1 & \text{if job } j \text{ is scheduled in spot } i \text{ in the schedule for scenario } s \\ 0 & \text{otherwise} \end{cases}$$

The constraints (3.1) and (3.2) can be added for each scenario $s \in S$. The same holds for constraints (3.3) and (3.4). The problem that now needs to be tackled is to make sure that each schedule for scenario $s \in S$ is created from the initial schedule by only making extra jobs late. Thus making sure that the recovery algorithm is being used. Therefore $u_j \leq u_j^s$ needs to be added for each job J_j and scenario $s \in S$.

With this we have the complete linear program

$$\min p_0 \sum_{j=1}^{n} u_j + \sum_{s \in S} p_s \sum_{j=1}^{n} u_j^s$$

subject to

$$\begin{aligned}
\sum_{i=1}^{n} x_{ij} &= 1 \quad \forall j \\
\sum_{i=1}^{n} x_{ij}^s &= 1 \quad \forall j, s \in S \\
\sum_{j=1}^{n} x_{ij} &= 1 \quad \forall i \\
\sum_{j=1}^{n} x_{ij}^s &= 1 \quad \forall i, s \in S \\
\sum_{i=1}^{k} \sum_{j=1}^{n} p_j x_{ij} - \sum_{j=1}^{n} d_j x_{kj} &= l_k \quad \forall k \\
\sum_{i=1}^{k} \sum_{j=1}^{n} p_j^s x_{ij}^s - \sum_{j=1}^{n} d_j^s x_{kj}^s &= l_k^s \quad \forall k, s \in S \\
l_k &\leq (\sum p_j - d_{min}) u_k \quad \forall k \\
l_k^s &\leq (\sum p_j^s - d_{min}^s) u_k^s \quad \forall k, s \in S \\
u_j &\leq u_j^s \quad \forall j, s \in S \\
x_{ij}, x_{ij}^s &\in \{0, 1\} \quad \forall i, j \\
u_j, u_j^s &\in \{0, 1\} \quad \forall i, j
\end{aligned}$$

This problem can then be solved with an algorithm that solves linear programs.

3.4 Dominance Rules

The performance of the algorithms developed in Section 3.3 may be improved when not all the jobs have to be considered. This might be the case if it is known that a job J_j is on time in all schedules in the optimal solution. Then J_j can already be set before the algorithms start. In this section it is investigated whether some jobs of a RRML problem have these properties.

Moore-Hodgson On Time for each Scenario

A first attempt to find a job that does not have to be included in the calculations, is to take a look at a job J_j that is on time in the Moore-Hodgson schedule of the initial problem and is on time for each Moore-Hodgson schedule of a scenario. It might be that this job will also be on time in the schedules of the optimal solution. In the example here below in Table 3.27, this is indeed the case. Here there are two jobs that are on time in the Moore-Hodgson solution of the initial problem and the scenario, namely job J_1 and J_4, and these are both on time in the optimal solution of the recoverable robustness problem.

	J_1	J_2	J_3	J_4
d_j	2	7	11	19
p_j	(1)	(6)	~~8~~	(7)
p_j^s	(2)	~~7~~	(9)	(8)

(a) The Moore-Hodgson solutions

	J_1	J_2	J_3	J_4
d_j	2	7	11	19
p_j	(1)	~~6~~	(8)	(7)
p_j^s	(2)	~~7~~	(9)	(8)

(b) An optimal solution for the RRML problem

Table 3.27: The jobs that are Moore-Hodgson on time in each Moore-Hodgson schedule seem also on time in the optimal solution.

This turns out to be true for many cases, but not in general. A counter example is given in Table 3.28, where job J_1 is the job that contradicts this idea. The optimal solution in Table (b) is the only possible solution with the solution value equal to the lower bound, no matter the values of p_0 and p_s.

	J_1	J_2	J_3	J_4	J_5
d_j	52	54	74	78	98
p_j	(27)	(20)	~~28~~	(25)	(25)
p_j^s	(27)	~~30~~	(28)	~~37~~	(37)

(a) The Moore-Hodgson solutions

	J_1	J_2	J_3	J_4	J_5
d_j	52	54	74	78	98
p_j	~~27~~	(20)	(28)	(25)	(25)
p_j^s	~~27~~	(30)	(28)	~~37~~	(37)

(b) An optimal solution for the RRML problem

Table 3.28: Counter example to the conjecture that a job that is on time in each Moore-Hodgson schedule is on time for each schedule in the optimal solution.

To determine which jobs that are on time in all Moore-Hodgson schedules will be on time in an optimal solution and which job will not be on time, the following two sets, $H^{'}$ and H with $H \subseteq H^{'}$, are defined.

Definition 1. *Set $H^{'}$ contains the indices of the jobs J_j with the following property:*

- *Job J_j is on time in the Moore-Hodgson schedule of the initial schedule and the Moore-Hodgson schedule of all the scenarios.*

Definition 2. *Set $H \subseteq H^{'}$ contains the indices from set $H^{'}$ for which the following additional property holds:*

- *Job J_j has a processing time shorter than all the jobs not in H in the initial schedule and in all the scenarios. And thus:*

$$p_j \leq p_i \quad \forall j \in H \text{ and } i \notin H$$
$$p_j^s \leq p_i^s \quad \forall j \in H, i \notin H \text{ and } s \in S$$

In the example from Table 3.27, the set $H^{'}$ contains jobs J_1 and J_4 and H contains only job J_1 and these jobs were on time in all schedules of the optimal solution. In the second example in Table 3.28 set $H^{'}$ contains J_1 and J_5 and set H is empty, however, J_1 is not on time in any of the two schedules of the optimal solution. Why can't this job be on time in an optimal solution?

Job J_1 is not in H because of job J_2, this violates the additional property. If the optimal solution in Tabel 3.27 (b) is adapted and job J_1 is made on time in both schedules, job J_4 becomes late in the initial schedule and job J_2 in the scenario schedule. Now in each schedule there is a job that was late in Moore-Hodgson, on time now. These are job J_3 for the initial problem and job J_2 for the scenario. Making these two jobs late, would solve the lateness created by making job J_1 on time in both schedules. The problem that arises now is that making these both late, creates the Moore-Hodgson schedules again, which is not a valid solution to the RRML problem. The other option would be to make job J_2 late in both schedules, this however does not resolve the lateness of job J_4 in the initial schedule. This problem is caused by the fact that it cannot be assumed that job J_2 is also larger than job J_1 in the initial schedule as is the case in the scenario schedule.

This problem indicates the intuition behind the definition of set H. When job J_1 from the previous example would have been from set H, then job J_2 would have been larger than job J_1 in both schedules of the solution and making job J_2 late would have resolved the lateness problem. With this, the following theorem can be proven about jobs in set H. To do so, Lemma 2 is needed.

Lemma 2. *Let OTI be the optimal set of jobs that are on time in the schedule of the initial problem in a RRML problem. The optimal solution of a scenario can be obtained by performing the Moore-Hodgson algorithm on the jobs in the set OTI.*

A consequence of this is the following lemma:

Lemma 3. *Consider an optimal solution O for an instance of the RRML problem with job J_j on time in the initial schedule of O. Job J_j was also on time in the Moore-Hodgson schedule of the initial problem and all scenarios. Job J_j will be on time in all schedules of solution O.*

Theorem 6. *Consider an instance of the RRML problem. The set H is defined as explained in definition 1. There exists an optimal solution with all the jobs from H on time in all schedules.*

Proof. Following Lemma 2and 3, an optimal solution exists where the jobs from H that are on time in this solution are on time in each scenario as well. Let O be such a solution. Each job from H that is not on time in all schedules of O, is late in all schedules. Consider job J_j the job with the smallest initial processing time from H that is not on time in all schedules of O. Now it is shown that an equally good solution $O^{'}$ can be created with job J_j on time in all schedules.

Because all jobs from H could be on time together in the Moore-Hodgson solution, the lateness of job J_j is caused by a job not in H that is on time in O. Let job J_l be such job with the smallest due date, smaller than job J_j that is not in H and is on time in the initial schedule of O. Now make job J_j on time in the

initial solution and job J_l late. This gives a valid initial solution with the same solution value, because by the property of set H it holds that $p_l \geq p_j$.

This however could have created a non valid solution, because job J_l could also have been on time in some of the scenario solutions. This can be solved by making job J_l late in these scenarios and job J_j on time, as was done for the initial problem. This also gives valid solutions for these scenarios because of the property of set H and the solution value is also still equal. Now job J_l is late in all schedules. Job J_j might however still have some scenarios where it is late. With the use of Lemma 2, it is shown that these problems can also be resolved. This all together results in a solution $O^{'}$ with job J_j on time in all schedules. The procedure can be repeated until the schedule $O^{'}$ is created with all jobs from H on time in all schedules. □

Algorithm 1 Obtain the set H

1: Calculate the Moore-Hodgson schedules of the initial problem and of each scenario $s \in S$.
2: $H \leftarrow$ all the jobs that are on time in all Moore-Hodgson schedules
3: $p_{min} \leftarrow$ the smallest processing time of the jobs not in H in the initial problem
4: $p^s_{min} \leftarrow$ the smallest processing time of the jobs not in H in scenario s
5: **for all** jobs J_j in H **do**
6: **if** $p_j \geq p_{min}$ or there is scenario $s \in S$ for which $p^s_j \geq p^s_{min}$ **then**
7: Remove job J_j from H
8: **end if**
9: **end for**
10: **return** H

Thus obtaining the set H before starting an algorithm can certainly reduce the number of jobs for which the configuration needs to be determined. How large this set can become is investigated in Section 5.2.3. Algorithm 1 shows the pseudo-code on how to obtain set H. Calculating all the Moore-Hodgson schedules takes $O(sn \log n)$ time, obtaining the initial set H takes $O(sn)$, calculating the minimum values of the jobs not in H takes $O(sn)$ again. Initializing the algorithm thus takes $O(sn \log n)$. The for-loop takes $O(s)$ for each job and thus $O(sn)$ in total. Therefore the total algorithm runs in $O(sn \log n)$.

Because set H can contain a lot of jobs that are surely on time, there can be jobs that can never be on time because of these jobs. This is covered in lemma 4. Before starting an algorithm, these jobs can be determined, reducing the number of jobs left even more.

Lemma 4. *Consider an instance of the RRML problem for which the set H is determined. Let job J_k be a job not in H. When considering the optimal solution with all jobs from set H on time, job J_k can never be on time in any of the schedules if holds that*

$$\sum_{J_j \in H, d_j < d_k} p_j + p_k \quad > \quad d_k$$

The job will not be on time in scenario s if holds that

$$\sum_{J_j \in H, d_j < d_k} p^s_j + p^s_k \quad > \quad d_k$$

So far it was shown that the jobs that are on time in all the Moore-Hodgson schedules, the jobs in $H^{'}$, will not all be on time in all schedules of the optimal solution of a $RRML$ problem. However, when an

extra property holds for the jobs, resulting in set H, they are proven to be on time in Theorem 6. There are however instances in which all jobs from $H^{'}$ are on time as shown in Table 3.27.

One could order the jobs of the initial problem not in earliest due date order, but in shortest processing time order (SPT). Now consider only those instances of the $RRML$-problem in which the scenarios have exactly the same SPT-order as the initial problem, these instances are called $RRML - SPT$ from now on. For these instances it does hold that all the jobs from $H^{'}$ are on time in all the schedules of an optimal solution. This can be easily proven by adapting the proof of Theorem 6 with the use of Theorem 7.

Theorem 7. *Consider an optimal solution O of $1||\sum_j u_j$ problem created with the shortest processing time algorithm from Section 3.1.1and let E be the set of on time jobs in this schedule and L the late jobs of this schedule. Now consider a different optimal solution $O^{'}$ with job $J_j \in E$ late and all the jobs in E with shorter processing times than J_j are still on time in $O^{'}$. In this solution there is now a job $J_l \in L$ on time that has a longer or equal processing time than job J_j.*

Proof. This is proven by contradiction, sort the jobs in SPT order and assume that $p_l < p_j$. Job J_l can be on time together with all the jobs with shorter processing time than job J_j but job J_j was on time in the SPT solution. Thus J_l must have been on time in the SPT algorithm as this job cannot be made late again once it is on time in the SPT algorithm. This forms a contradiction, thus $p_l > p_j$. □

Theorem 8. *Consider an instance of the RRML-SPT problem. The set $H^{'}$ is defined as explained in definition 2. There exists an optimal solution with all the jobs from $H^{'}$ on time in all schedules.*

Proof. Following Lemma 2 and 3, an optimal solution exists where the jobs from $H^{'}$ that are on time in this solution are on time in each scenario as well. Let O be such a solution. Each job from $H^{'}$ that is not on time in all schedules of O, is late in all schedules. Consider job J_j the job with the smallest initial processing time from $H^{'}$ that is not on time in all schedules of O. Now it is shown that an equally good solution $O^{'}$ can be created with job J_j on time in all schedules.

Because all jobs from $H^{'}$ could be on time together in the Moore-Hodgson solution, the lateness of job J_j is caused by a job not in $H^{'}$ that is on time in O. Proven in Theorem 7 there exists a job not in $H^{'}$ with longer processing time than J_j that is on time in O, call this job J_l. If multiple exist, take the one with the smallest due date. Now make job J_j on time in the initial solution and job J_l late. This gives a valid initial solution with the same solution value, because it holds that $p_l \geq p_j$.

This however could have created a non valid solution, because job J_l could also have been on time in some of the scenario solutions. This can be solved by making job J_l late in these scenarios and job J_j on time, as was done for the initial problem. This also gives valid solutions for these scenarios because of the property that all problems had the same SPT order and thus job J_l also has a larger processing time than job J_j in each scenario. The solution value is also still equal. Now job J_l is late in all schedules. Job J_j might however still have some scenarios where it is late. With the use of Lemma 2, it is shown that these problems can also be resolved. This all together results in a solution $O^{'}$ with job J_j on time in all schedules. The procedure can be repeated until the schedule $O^{'}$ is created with all jobs from $H^{'}$ on time in all schedules. □

Thus for an RRML-SPT instance, the set $H^{'}$ can be determined and these jobs can be set before starting an algorithm from Section 3.3. Another observation that can be made now is in the case of an instance of the RRML problem with a scenario that has total opposite SPT-order than in the initial problem the sets H and $H^{'}$ will be empty and thus for all jobs the configuration needs to be determined with an algorithm. This is also a property of the instance created for the NP-hardness proof in Theorem 1.

Moore-Hodgson Late for each Scenario

For a job that is late in every Moore-Hodgson schedule it might be expected that it is late in every schedule of the solution. What advantage can there be to have these jobs on time? This however turns out not to be true. The counter example can be found in Table 3.29. Here job J_5 contradicts the theorem. This example even contradicts that when the job that is the biggest in the initial problem and in the scenario is late in each Moore-Hodgson schedule, it will be late in all schedules of the optimal solution. So a proof like theorem 6 does not exist for Moore-Hodgson late jobs. The reason why job J_5 is on time in this schedule is simply because for job J_4 to be on time, jobs J_1 and J_2 cannot be on time and thus there is some space left in the initial schedule for job J_5. Thus these late jobs can be on time because other jobs have to be late to make the schedule feasible regarding the recovery algorithm.

	J_1	J_2	J_3	J_4	J_5
d_j	6	6	7	7	19
p_j	(2)	(4)	(1)	~~5~~	~~13~~
p_j^s	~~6~~	~~6~~	(2)	(5)	~~17~~

(a) The Moore-Hodgson solutions

	J_1	J_2	J_3	J_4	J_5
d_j	6	6	7	7	19
p_j	~~2~~	~~4~~	(1)	(5)	(13)
p_j^s	~~6~~	~~6~~	(2)	(5)	~~17~~

(b) An optimal solution for the RRML problem

Table 3.29: Counter example to the conjecture that a job that is late in each Moore-Hodgson schedule is late for each schedule in the optimal solution.

Chapter 4

Scheduling with Rejection

4.1 Introduction

In the previous section it was assumed that all jobs have to be processed. The jobs that are late are still processed at the end of the schedule in a random order, because the magnitude of the lateness does not matter. Instead of processing all jobs, the paint factory could also reject a job when the capacity cannot handle this job or this job cannot be finished before its due date. The customer could then offer the job to another company. Rejecting a job might come with a penalty, for example because rejecting a regular client is not good for receiving future jobs from this customer. Another kind of rejection cost might be caused by outsourcing the job. The factory outsources the job to a partner company and this will not be without cost.

The general definition of scheduling with rejection is: there are n jobs $J_1, \dots, J_n$ that have to be processed on m machines $M_1, \dots, M_m$. The jobs might have some additional information depending on the scheduling criterion of the factory. When rejection is allowed, e_j denotes the rejection penalty for job J_j. For these problems the three-field notation introduced by [9] as explained in Section 2.1 can still be used. When scheduling with rejection is used, the entry rej is added to the second field β.

When creating a solution, the set of jobs needs to be divided into two sets. The jobs that get accepted are in set A and the jobs that get rejected are in $\bar{A}$. The jobs in set A then need a schedule on how these jobs are processed on the different machines. The objective value is then measured by two different criteria. Criteria one ($F1$) is the scheduling criterion that depends on the ordering of the jobs that are accepted and the second criterion ($F2$) is minimizing the rejection cost $RC = \sum_{J_j \in \bar{A}} e_j$. This makes this problem a bi-criteria scheduling problem.

The field of multicriteria scheduling is extensively covered in [23]. From this paper it follows that there are four different objective functions that can be created with $F1$ and $F2$. Here $\epsilon(F1/F2)$ denotes that criterion $F1$ is minimized subject to $F2$ being upper bounded by a known value. For $\epsilon(F2/F1)$ this is exactly the other way around.

(P1) $\min F1 + F2$

(P2) $\epsilon(F1/F2)$

(P3) $\epsilon(F2/F1)$

(P4) The set of Pareto-optimal points for $(F1, F2)$

The definition of a Pareto-optimal point is given as follows in [22].

Definition 3. *A solution S is Pareto-optimal with respect to criteria $F1$ and $F2$ if there does not exist another solution S' such that $F1(S') \leq F1(S)$ and $F2(S') \leq F2(S)$ with at least one of these inequalities being strict.*

In this thesis the focus will be on objective $(P1)$, this is also the most studied objective of the four problems. Most scheduling problems with rejection are NP-hard and an overview of this can be found in [22]. The complexity and references of algorithms of the non NP-hard problems can also be found in this survey. In this chapter the concept of recoverable robustness will be applied to different scheduling problems with rejection. In the first section this problem will be minimizing the makespan of one machine, which is an easy problem with polynomial running time. In the next section the jobs will have release dates and this problem then already becomes NP-hard. The chapter is concluded with applying the concept to the problem of minimizing the maximum lateness or tardiness of a schedule.

4.2 Minimizing the Maximum Makespan with Rejection

The problem of minimizing the makespan, $1||C_{max}$, without considering rejection is very easy. The jobs can be simply scheduled in a random sequence without idle time. The total makespan will then be equal to $\sum_j p_j$ and this cannot be done in shorter time. When rejection is allowed, $1|rej|C_{max} + RC$, an easy $O(n)$ time algorithm can be constructed. An observation that can be made is that a job J_j either contributes p_j to the objective or e_j. For each job J_j it can easily be checked which of p_j and e_j is the smallest. If p_j is the smallest the job is scheduled on the machine, if e_j is the smallest the job will be rejected. The order in which the jobs will be scheduled on the machine has no influence.

In this problem there might be uncertainties about the values of the processing time of a job or the rejection cost of a job. The uncertainties can again be modeled with the recoverable robustness model explained in Section 2.3. Each scenario s will cover a different situation that might occur and the solution consists of a set A of the accepted jobs in the initial problem and A^s is the set of accepted jobs in each scenario s. Each of these sets has a complement, $\bar{A}$ or $\bar{A}^s$, that consists of the jobs that are rejected. The chosen recovery algorithm is about the same as in the previous chapter. A job that was rejected in the initial schedule, cannot be processed on the machine in a scenario. This is fair because the customer probably went to another factory to process its job and this job will not be available when the scenario occurs. A job that was initially processed might however be rejected in a scenario. Thus $\bar{A} \subseteq \bar{A}^s$. The objective function now becomes

$$p_0(C_{max} + \sum_{J_j \in \bar{A}} e_j) + \sum_{s \in S} p_s(C^s_{max} + \sum_{J_j \in \bar{A}^s} e^s_j).$$

Consider an instance with scenario set S and n jobs $J_1, \ldots, J_n$. Each job J_j has an initial processing time p_j, a scenario processing time p^s_j for each scenario $s \in S$, an initial rejection cost e_j and a scenario rejection cost e^s_j for each scenario $s \in S$. In this case every schedule contributes equally to the objective and thus $p_0 = p_s \; \forall s$. An important observation that can be made is that whether a job J_j is accepted or rejected in each scenario only depends on p_j, p^s_j, e_j and e^s_j. It does not depend on the values of the other jobs or on which job is scheduled before or after job J_j on the machine. A second observation that can be made is that a job will never be scheduled in scenario s if $p^s_j \geq e^s_j$. Now two cases can be distinguished for each job in this problem.

Case 1: $p_j < e_j$**.** In this case it would be better for the initial schedule to schedule job J_j. This means that it then can simply be determined for each scenario individually whether $p^s_j \geq e_j$. In this case this job will be scheduled in scenario s, otherwise it will be rejected.

Case 2: $p_j \geq e_j$. In this case the job J_j would not be scheduled in the initial problem if there were no scenarios. However there might be scenarios for which it holds that $p_j^s < e_j^s$ and these would be scheduled if this was the only schedule under consideration. Let the set $S_j^{'}$ be the set that contains all these scenarios for which hold that $p_j^s < e_j^s$. These jobs will be accepted for all scenarios in $S_j^{'}$ and in the initial problem if holds that $p_j + \sum_{s \in S_j^{'}} p_j^s \leq e_j + \sum_{s \in S_j^{'}} e_j^s$, otherwise job J_j will be rejected in all scenarios and the initial schedule.

Both cases cost $O(|S|)$ time and this must be done for all jobs $J_1, \ldots, J_n$. Thus the problem of $1|rej|C_{max} + RC$ with recoverable robustness can be solved in $O(n|S|)$. Therefore this algorithm runs in polynomial time no matter how many scenarios the problems has.

4.2.1 Minimizing the Maximum Makespan with Rejection and Release Dates

It may happen that not each job is directly available. This may happen because not all the material for this job has arrived at the factory. To model this each job J_j will now also have a release date r_j. Without rejection allowed this problem is defined as $1|r_j|C_{max}$. This problem is solved to optimality by using the earliest release date rule (ERD-rule) [3]. The jobs are ordered such that $r_1 \leq r_2 \leq \ldots \leq r_n$ and scheduled in that order. The starting time of job J_j is then equal to $\max\{r_j, C_{j-1}\}$.

When rejection is allowed the problem $1|rej, r_j|C_{max} + RC$ becomes NP-hard. This is proven in [15] by a reduction from the partition problem. In [7] it is proven by reduction from the 0/1-knapsack problem. An important lemma is the following.

Lemma 5. *There exists an optimal schedule for the $1|rej, r_j|C_{max} + RC$-problem in which the scheduled jobs from set A are scheduled by the ERD-rule.*

In [15] these dynamic programming algorithms were proposed to solve the $1|rej, r_j|C_{max} + RC$ problem and will be presented here. The second algorithm is slightly adapted with the use of the algorithm proposed in [25]. For both of these algorithms the jobs are ordered such that $r_1 \leq r_2 \leq \ldots \leq r_n$ and $r_{max} = \max_j r_j$, $P = \sum_j p_j$ and $W = \sum_j e_j$. When the optimal value is found with one of the algorithms, the jobs that belong to the set A of accepted jobs and the jobs that belong to $\bar{A}$ can be found by backtracking. For each job it is decided whether to accept it or not in ERD order, because of lemma 5.

Dynamic programming algorithm one (DP1)

Let $f_j(V)$ be the optimal value of the objective function when the jobs under consideration are $J_1, \ldots, J_j$ and the total rejection penalty is exactly V. In an optimal schedule, job J_j is either rejected or it will be processed on the machine. If job J_j is rejected, the total rejection penalty considering jobs $J_1, \ldots, J_{j-1}$ will be $V - e_j$. Thus in this case $f_j(V) = f_{j-1}(V - e_j) + e_j$. When job J_j is rejected the total rejection penalty considering the jobs $J_1, \ldots, J_{j-1}$ will still be V and the optimal objective value considering only these jobs is $f_{j-1}(V)$. The makespan of these jobs will then be equal to $f_{j-1}(V) - V$. Now as stated earlier, the starting time of job J_j will then be equal to $\max\{f_{j-1}(V) - V, r_j\}$ and thus the completion time of job J_j is equal to $C_j = \max\{f_{j-1}(V) - V, r_j\} + p_j$. It now holds that $f_j(V) = C_j + V = \max\{f_{j-1}(V) - V, r_j\} + p_j + V$. These cases give the following recursive formula for this dynamic programming algorithm.

$$f_j(V) = \min\{f_{j-1}(V - e_j) + e_j, \max\{f_{j-1}(V) - V, r_j\} + p_j + V\}$$

Where the initial values when only job J_1 is considered are $f_1(0) = r_1 + p_1$, $f_1(e_1) = e_1$ and $f_1(V) = \infty$ for any $v \neq 0, e_1$. The other values can be calculated using the previous formula. The optimal value is equal

to $\min_{0 \le V \le W} f_n(V)$.

The recursive function calculates at most $O(nW)$ state values and the calculation of each value costs constant time. Thus this algorithm runs in $O(nW)$ time. When all the rejection costs are equal, $e_j = e$ for all jobs J_j, then $W = ne$ and the running time of this algorithm becomes $O(n^2)$ as only the values $f_j(V)$ with V a multiple of e have interesting values. This means that in the special case that the rejection costs are equal, the problem $1|rej, r_j, e_j = e|C_{max} + RC$ is not NP-hard.

Dynamic programming algorithm two (DP2)

Let $f_j(t)$ be the minimum value of the total rejection penalty when the jobs under consideration are $J_1, \ldots, J_j$ and the makespan of the accepted jobs among $J_1, \ldots, J_j$ is exactly t. Again there are two cases for job J_j, it is rejected or it is accepted and processed. The case that job J_j gets rejected is easy. The makespan of jobs $J_1, \ldots, J_{j-1}$ is still t and thus $f_j(t) = f_{j-1}(t) + e_j$. When job J_j gets accepted it must hold that $t \ge r_j + p_j$. If $t > r_j + p_j$, the makespan of the scheduled jobs among $J_1, \ldots, J_{j-1}$ must be exactly $t - p_j$ and thus $f_j(t) = f_{j-1}(t - p_j)$. If $t = r_j + p_j$ then the makespan of the scheduled jobs among $J_1, \ldots, J_{j-1}$ is at most r_j. Thus it holds that $f_j(t) = \min_{0 \le t' \le r_j} f_{j-1}(t')$. This gives the following recursive formula for this dynamic algorithm algorithm.

$$f_j(t) = \begin{cases} f_{j-1}(t) + e_j & \text{if } t < r_j + p_j \\ \min\{f_{j-1}(t) + e_j, \min_{0 \le t' \le r_j} f_{j-1}(t')\} & \text{if } t = r_j + p_j \\ \min\{f_{j-1}(t) + e_j, f_{j-1}(t - p_j)\} & \text{if } t > r_j + p_j \end{cases}$$

The initial values when only job J_1 considered are $f_1(0) = e_1$, $f_1(r_1 + p_1) = 0$ and $f_1(t) = \infty$ for all values $t \ne 0, r_1 + p_1$. The optimal value is then equal to $\min_{0 \le t \le r_n + P}(f_n(t) + t)$.

The recursive formulation calculates at most $O(n(r_{max} + P))$ state values. In the first and third case this calculation takes constant time. In the case that $t = r_j + p_j$, each iteration costs $O(r_{max} + P)$. This case however happens only once for each job J_j. Thus the total running time is $O(n(r_{max} + P))$.

4.2.2 Recoverable Robustness for the $1|rej, r_j|C_{max} + RC$-Problem

For the $1|rej, r_j|C_{max} + RC$-problem, the recoverable robustness problem can be used to model possible uncertainties. The release dates now could come as an extra uncertainty on top of the processing times and the rejection cost. These release dates may be uncertain when it is not clear when the needed goods to process a certain job arrive at the factory. Because the $1|rej, r_j|C_{max} + RC$-problem is already NP-hard on its own, the recoverable robustness method will also be NP-hard for this problem. This means there will be no algorithm for the general problem that has a polynomial running time. The same recovery algorithm will be used as for the problem without release dates, thus it must hold that $\bar{A} \subseteq \bar{A}^s$.

In a first attempt to formulate an algorithm for the recoverable robust $1|rej, r_j|C_{max} + RC$-problem, DP1 can be extended. The state variable could be extended to $f_j(V_0, V_1, \ldots, V_S)$ which corresponds to the optimal value of the objective function when the jobs under consideration are $J_1, \ldots, J_j$ and the total rejection penalty in the initial schedule is V_0 and the total rejection penalty in scenario s is V_s. When there is one scenario this is simply $f_j(V_0, V_1)$ and each job has three possible configurations. When job J_j is rejected in both the initial problem and the scenario, $f_j(V_0, V_1) = f_j(V_0 - e_j, V_1 - e_j^s) + e_j + e_j^s$. In the second case, job J_j is accepted in the initial problem and rejected in the scenario, but then a problem arises. The completion time of the last accepted job from jobs $J_1, \ldots, J_{j-1}$ is needed from both the initial schedule and the scenario schedule. This value however cannot be obtained from f_{j-1} for any value of V_0 and V_1. Thus DP1 cannot

be extended to find a solution for the problem of $1|rej, r_j|C_{max} + RC$.

In a second attempt DP2 is extended. The state variable $f_j(t_0, t_1, \dots, t_{|S|})$ is the optimal value of the total rejection penalty when the jobs under consideration are $J_1, \dots, J_j$ and the makespan of the initial schedule is t_0 and the makespan of scenario $s \in S$ is t_s. This algorithm will first be explained when there is only one scenario. The state variable now is equal to $f_j(t_0, t_1)$ and each job J_j has three possible cases: it is accepted in both the initial problem and the scenario, it is rejected in both schedules or it is rejected in the scenario and accepted in the initial schedule. In DP2 the calculation depended on the value of t, resulting in three different minimands. Here the calculation depends on values of t_0 and t_1.

In the case job J_j gets rejected in both schedules, the makespan of the scheduled jobs among $J_1, \dots, J_{j-1}$ is still t_0 in the initial schedule and t_1 in the scenario schedule and thus $f_j(t_0, t_1) = f_{j-1}(t_0, t_1) + e_j + e^s_j$. This holds for every value of t_0 and t_1. Next consider the case where job J_j is accepted in the initial schedule and rejected in the scenario. The makespan of the jobs $J_1, \dots, J_{j-1}$ in the scenario schedule will still be t_1. For the initial schedule it must hold that $t_0 \geq r_j + p_j$. If $t_0 > r_j + p_j$, the makespan of the scheduled jobs among $J_1, \dots, J_{j-1}$ must be exactly $t_0 - p_j$ and thus $f_j(t_0, t_1) = f_{j-1}(t_0 - p_j, t_1) + e^s_j$. If $t_0 = r_j + p_j$, the makespan of the scheduled jobs among $J_1, \dots, J_{j-1}$ is at most r_j. Thus, $f_j(t_0, t_1) = \min_{0 \leq t' \leq r_j} f_{j-1}(t', t_1) + e^s_j$.

In the case job J_j gets accepted in both schedules, it must hold that $t_0 \geq r_j + p_j$ and $t_1 \geq r^s_j + p^s_j$. In case $t_0 > r_j + p_j$ and $t_1 > r^s_j + p^s_j$, the makespan of the scheduled jobs among $J_1, \dots, J_{j-1}$ must be $t_0 - p_j$ in the initial schedule and $t_1 - p^s_j$ in the scenario schedule, giving $f_j(t_0, t_1) = f_{j-1}(t_0 - p_j, t_1 - p^s_j)$. In case $t_0 = r_j + p_j$ and $t_1 = r^s_j + p^s_j$, the makespan of the scheduled jobs among $J_1, \dots, J_{j-1}$ is at most r_j in the initial schedule and r^s_j in the scenario, thus $f_j(t_0, t_1) = \min_{0 \leq t'_0 \leq r_j, 0 \leq t'_1 \leq r^s_j} f_{j-1}(t'_0, t'_1)\}$. The other two combinations left are a combination of both formulas and are easily obtained. This results in the following recursive formula for this dynamic programming algorithm.

$$f_j(t_0, t_1) = \begin{cases} f_{j-1}(t_0, t_1) + e_j + e^s_j & \text{if } t_0 < r_j + p_j \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, \min_{0 \leq t' \leq r_j} f_{j-1}(t', t_1) + e^s_j\} & \text{if } t_0 = r_j + p_j, t_1 < r^s_j + p^s_j \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, \min_{0 \leq t' \leq r_j} f_{j-1}(t', t_1) + e^s_j, & \text{if } t_0 = r_j + p_j, t_1 = r^s_j + p^s_j \\ \quad \min_{0 \leq t'_0 \leq r_j, 0 \leq t'_1 \leq r^s_j} f_{j-1}(t'_0, t'_1)\} & \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, \min_{0 \leq t' \leq r_j} f_{j-1}(t', t_1) + e^s_j, & \text{if } t_0 = r_j + p_j, t_1 > r^s_j + p^s_j \\ \quad \min_{0 \leq t' \leq r_j} f_{j-1}(t', t_1 - p^s_j)\} & \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, f_{j-1}(t_0 - p_j, t_1) + e^s_j\} & \text{if } t_0 > r_j + p_j, t_1 < r^s_j + p^s_j \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, f_{j-1}(t_0 - p_j, t_1) + e^s_j, & \text{if } t_0 > r_j + p_j, t_1 = r^s_j + p^s_j \\ \quad \min_{0 \leq t' \leq r^s_j} f_{j-1}(t_0 - p_j, t')\} & \\ \min\{f_{j-1}(t_0, t_1) + e_j + e^s_j, f_{j-1}(t_0 - p_j, t_1) + e^s_j, & \text{if } t_0 > r_j + p_j, t_1 > r^s_j + p^s_j \\ \quad f_{j-1}(t_0 - p_j, t_1 - p^s_j)\} & \end{cases}$$

The initial values when only job J_1 is considered are $f_1(0, 0) = e_1 + e^s_1$, $f_1(r_1 + p_1, 0) = e^s_1$ and $f_1(r_1 + p_1, r^s_1 + p^s_1) = e_1 + e^s_1$; for other values of t_0 and t_1 it holds that $f_1(t_0, t_1) = \infty$. The optimal value can be calculated as follows.

$$\min_{0 \leq t_0 \leq r_n + P, 0 \leq t_1 \leq r^s_n + P^s} f_n(t_0, t_1) + t_0 + t_1$$

Let $R = \max\{r_{max}, r^s_{max}\}$, then the recursive formulation calculates at most $O(n(R + P)^2)$ values. In the worst case, when $t_0 = r_j + p_j$ and $t_1 = r_j + p_j$, the calculation of a table entry takes $O(R^2)$ time. This however happens only n times. When $t_0 = r_j + p_j$ or $t_1 = r_j + p_j$ the calculation takes $O(R)$, this only happens $O(2nP)$ times. The other cases take at most $O(1)$ time. Thus the total running time of the algorithm is bounded by $O(n(R + P)^2)$.

The dynamic programming algorithm to solve the recoverable robustness problem for the $1|rej, r_j|C_{max} + RC$ problem with only one scenario is already quite extensive. This algorithm can however be extended to solve the problem for all sizes of $|S|$. As stated earlier this would be done with the state variable $f_j(t_0, t_1, \ldots, t_{|S|})$ which is equal to the optimal value of the total rejection penalty when the jobs under consideration are $J_1, \ldots, J_j$ and the makespan of the initial schedule is t_0 and the makespan of scenario $s \in S$ is t_s. If $R = \max\{r_{max}, \max_{s\in S} r^s_{max}\}$, the recursive formulation calculates at most $O(n(R+P)^{|S|})$ values. The number of minimands in one case will be $O(2^{|S|})$ and the worst minimand will take $O(R^{|S|})$ time and thus one case will take $O(2^{|S|}R^{|S|})$. Thus the algorithm will take $O(n2^{|S|}R^{|S|}(R+P)^{|S|})$ running time.

4.3 Minimizing L_{max} or T_{max} with Rejection

In the previous chapter it did not matter how late a job was; one time unit too late was equally good or bad as twenty time units too late. When it does matter how late a job is, the maximum lateness or maximum tardiness of a job can be measured. As mentioned earlier in Section 2.1 the lateness of a job is defined as $L_j = C_j - d_j$. The maximum lateness of a schedule is then defined as $L_{max} = \max_j L_j$. Negative lateness means that this job was on time, this is not taken into account when looking at tardiness. This means $T_j = \max\{L_j, 0\}$ and the maximum lateness of a job is then $T_{max} = \max_j T_j$.

The problems $1||L_{max}$ and $1||T_{max}$ are closely related and are both solvable using the same algorithm. In [11] it is proven that the optimal schedule can be created for these two problems by scheduling the jobs in earliest due date order (EDD-rule). The jobs are ordered such that $d_1 \leq d_2 \leq \ldots \leq d_n$ and the starting time of job J_j is simply C_{j-1}, where the start time of job $J_1 = 0$.

When rejection is allowed the problems $1|rej|L_{max} + RC$ and $1|rej|T_{max} + RC$ become NP-hard. The reduction from the partition problem can be found in [21]. For these problems two dynamic programming algorithms are constructed in [21], which are explained here below with a small adjustment such that the case of rejecting all jobs is covered better. They both depend on the following lemma, which can also be stated for $1|rej|T_{max} + RC$.

Lemma 6. *There exists an optimal schedule for the $1|rej|L_{max} + RC$ problem in which the scheduled jobs from set A are scheduled by the EDD-rule.*

For executing the following pseudo-polynomial time algorithms it is assumed that the jobs are ordered such that $d_1 \leq d_2 \leq \ldots \leq d_n$. Opposite to the previous dynamic programming algorithms these algorithms start by deciding on job J_n and then on job J_{n-1} and so on. In this way important information on the maximum lateness can be easily computed.

Dynamic programming on the rejection costs

Let $f_j(e)$ be the optimal value of the maximum lateness when the jobs in consideration are $J_j, J_{j+1}, \ldots, J_n$ and the total rejection penalty of the jobs is e. The boundary conditions on job J_n are the following, $f_n(e_n) = -d_{min}$, $f_n(0) = p_n - d_n$ and $f_n(e) = \infty$ if $e \neq e_n, 0$. When looking at the optimal schedule corresponding to the value of $f_j(e)$ there are two possible cases for job J_j, it is either rejected or accepted. Job J_j can only be rejected if $e \geq e_j$, in which case $f_j(e) = f_{j+1}(e - e_j)$ because the rejection penalty of the rejected jobs among $J_{j+1}, \ldots, J_n$ must be $e - e_j$. If job J_j is scheduled then the total rejection penalty is still e. The lateness of the jobs $J_{j+1}, \ldots, J_n$ will be increased by p_j and the lateness of job J_j is exactly $p_j - d_j$. Then $f_j(e) = \max\{f_{j+1}(e) + p_j, p_j - d_j\}$. This gives the following recursive formula.

$$f_j(e) = \begin{cases} \max\{f_{j+1}(e) + p_j, p_j - d_j\} & \text{if } e < e_j \\ \min\{f_{j+1}(e - e_j), \max\{f_{j+1}(e) + p_j, p_j - d_j\}\} & \text{otherwise} \end{cases}$$

The optimal value for the $1|rej|L_{max} + RC$-problem is equal to

$$\min_{0 \leq e \leq \sum_j e_j} f_1(e) + e$$

and for $1|rej|T_{max} + RC$ this is equal to

$$\min_{0 \leq e \leq \sum_j e_j} \max\{0, f_1(e) + e\}$$

. Recall that $W = \sum e_j$, at most nW values need to be calculated and each value requires $O(1)$ time, so the running time of the algorithm is $O(nW)$. This means again that when all the rejection costs are equal these problems, $1|rej, e_j = e|L_{max} + RC$ and $1|rej, e_j = e|T_{max} + RC$ can be solved in polynomial time.

Dynamic programming on the lateness of the jobs

Let $f_j(l)$ be the minimum value of the total rejection penalty when the jobs in consideration are $J_j, J_{j+1}, \ldots, J_n$ and the maximum lateness of the scheduled jobs is at most l. The boundary conditions on job J_n are the following.

$$f_n(l) = \begin{cases} e_n & \text{if } l = -d_{min} \\ 0 & \text{if } l \geq p_n - d_n \\ \infty & \text{otherwise} \end{cases}$$

In the schedule that corresponds to the optimal value of $f_{j+1}(l)$ there are two possible cases for job J_j. Either job J_j is accepted and scheduled or job J_j is rejected. In the case that job J_j is rejected the maximum lateness of the scheduled jobs of $J_{j+1}, \ldots, J_n$ is l and the rejection penalty will increase with e_j thus $f_j(l) = f_{j+1}(l) + e_j$. If job J_j gets scheduled its lateness is $p_j - d_j$ and thus there will be no feasible solution if $l < p_j - d_j$. The lateness of the scheduled jobs among $J_{j+1}, \ldots, J_n$ will increase with p_j and thus the lateness of these scheduled jobs can be at most $l - p_j$, that is, $f_j(l) = f_{j+1}(l - p_j)$. This gives the following recurrence relation.

$$f_j(l) = \begin{cases} f_{j+1}(l) + e_j & \text{if } l < p_j - d_j \\ \min\{f_{j+1}(l) + e_j, f_{j+1}(l - p_j)\} & \text{otherwise} \end{cases}$$

It is clear that $L_{max} \leq \sum_j p_j - d_{min}$, because C_j can never be larger than $\sum_j p_j$, and $L_{min} \geq -d_{min}$, thus there are only $\sum_j p_j$ possibilities of the value of L_{max}. The optimal value for $1|rej, e_j = e|L_{max} + RC$ can be found by

$$\min\{f_1(l) + l | -d_{min} \leq l \leq \sum_j p_j - d_{min}\}$$

and for the $1|rej, e_j = e|T_{max} + RC$ problem it is equal to

$$\min\{f_1(l) + \max\{0, l\} | -d_{min} \leq l \leq \sum_j p_j - d_{min}\}.$$

There are at most $n \sum_j p_j$ values that need to be calculated and each calculation takes $O(1)$ time so the running time of the algorithm is $O(nP)$.

4.3.1 Recoverable Robustness for $1|rej|L_{max} + RC$ and $1|rej|L_{max} + RC$

The problems $1|rej|L_{max} + RC$ and $1|rej|L_{max} + RC$ may have uncertain processing times, due dates or rejection penalties. These may again be modeled with the recoverable robustness approach from Section 2.3. As in the previous section one of the dynamic programming algorithms may be extended to also find solutions for each scenario that obey the recovery algorithm constraint.

Extending the dynamic programming algorithm on the rejection costs to have a state variable $f_j(e_0, e_1, \ldots, e_{|S|})$ that is the sum of the minimum lateness over all schedules given that the rejection penalty in the initial schedule is equal to e_0 and that in each scenario s this is equal to e_s raises a problem. This is because it cannot be determined what the exact lateness in one schedule is and thus it is not known how this value must be changed. Therefore a second attempt is made with the algorithm on the lateness of the jobs.

Let $f_j(l_0, l_1, \ldots, l_{|S|})$ be the minimum value of the total rejection penalty when the jobs in consideration are $J_j, \ldots, J_n$ and the maximum lateness of the initial schedule is at most l_0 and in scenario schedule s this is at most l_s. First consider the algorithm for only one scenario, then the state variable is equal to $f_j(l_0, l_1)$. There are three different configurations for job J_j. If job J_j is rejected in both schedules then the maximum lateness in the schedules considering only jobs $J_{j+1}, \ldots, J_n$ must still be l_0 and l_1. The penalty increases with $e_j + e_j^s$ and thus $f_j(l_0, l_1) = f_{j+1}(l_0, l_1) + e_j + e_j^s$. In the second case job J_j may be accepted in the initial schedule but rejected in the scenario. As in the previous algorithm it now must hold that $l_0 \geq p_j - d_j$, because this is the lateness of job J_j. The maximum lateness among the scheduled jobs among $J_{j+1}, \ldots, J_n$ can be at most $l - p_j$, thus $f_j(l_0, l_1) = f_{j+1}(l_0 - p_j, l_1) + e_j^s$. If job J_j gets accepted in both schedules, it must hold that $l_0 \geq p_j - d_j$ and $l_1 \geq p_j^s + d_j^s$. Now $f_j(l_0, l_1) = f_{j+1}(l_0 - p_j, l_1 - p_j^s)$. This gives the following recursive formula.

$$f_j(t) = \begin{cases} f_{j+1}(l_0, l_1) + e_j + e_j^s & \text{if } l_0 < p_j - d_j \\ \min\{f_{j+1}(l_0, l_1) + e_j + e_j^s, f_{j+1}(l_0 - p_j, l_1) + e_j^s\} & \text{if } l_0 \geq p_j - d_j, l_1 < p_j^s - d_j^s \\ \min\{f_{j+1}(l_0, l_1) + e_j + e_j^s, f_{j+1}(l_0 - p_j, l_1) + e_j^s, & \text{if } l_0 \geq p_j - d_j, l_1 \geq p_j^s - d_j^s \\ \quad f_{j+1}(l_0 - p_j, l_1 - p_j^s)\} & \end{cases}$$

The initial values when only the last job J_n is considered are

$$f_n(l_0, l_1) = \begin{cases} e_n + e_n^s & \text{if } l_0 = -d_{min}^0, l_1 = -d_{min}^1 \\ e_n^s & \text{if } l_0 \geq p_n - d_n, l_1 = -d_{min}^1 \\ 0 & \text{if } l_0 \geq p_n - d_n, l_1 \geq p_n^s - d_n^s \\ \infty & \text{otherwise} \end{cases}$$

The optimal value for the recoverable robust $1|rej|L_{max} + RC$ can be found by

$$\min\{f_1(l_0, l_1) + l_0 + l_1 | l_0^{min} \leq l \leq l_0^{max}, l_1^{min} \leq l \leq l_1^{max}\}$$

and for the recoverable robust $1|rej|T_{max} + RC$ problem it is equal to

$$\min\{f_1(l_0, l_1) + \max\{0, l_0\} + \max\{0, l_1\} | l_0^{min} \leq l \leq l_0^{max}, l_1^{min} \leq l \leq l_1^{max}\}$$

Let $P = \max\{\sum p_j, \sum p_j^s\}$, the recursive formulation calculates $O(nP^2)$ values. Calculating these values are all done in $O(1)$ and thus the running time of the dynamic programming algorithm for one scenario takes $O(nP^2)$ time.

This dynamic programming algorithm can be extended to solve problems for all sizes of S. In this general case the state variable would be $f_j(l_0, l_1, \ldots, l_{|S|})$ which is equal to the minimum value of the total rejection penalty when the jobs in consideration are $J_j, \ldots, J_n$ and the maximum lateness of the initial schedule is at most l_0 and in scenario schedule s this is at most l_s. Depending on the values of $f_j(l_0, l_1, \ldots, l_{|S|})$, the value of $f_j(l_0, l_1, \ldots, l_{|S|})$ can be calculated. If $P = \max\{\sum p_j, \max_{s \in S} \sum p_j^s\}$, the dynamic programming algorithm will calculate $O(nP^{|S|})$ values. One entry can have $O(2^{|S|})$ minimands which all can be calculated in $O(1)$ time. Thus one value takes $O(2^{|S|})$ time to calculate. Thus the algorithm for the general case $O(n2^{|S|}P^{|S|})$.

Chapter 5

Experiments

To find out which algorithm works best and which settings give the best results, experiments need to be performed. To do so problem instances need to be created, how this is done is explained in Section 5.1. First the branch and bound and branch and price algorithms are tested, here after called the branching algorithms. These experiments are performed in two rounds. In the first round the branching algorithms are tested on instances with $n = 40$ and $s = 2$ to find the best settings, see Section 5.2.1, and the most difficult instances, see Section 5.2.2. In the second round n and s will be increased to see how the performances of the best settings are affected. These results can be found in Section 5.2.3. Next the performance of the last two algorithms, dynamic programming and the direct linear program, is discussed in Sections 5.4 and 5.3.

The problem and the algorithms are implemented using $C\#$ and with ILOG CPLEX 12.6.0. the linear programs are solved. This is necessary to solve the separate recovery algorithm within branch and price and the direct linear program later on. All the experiments in the upcoming sections are run on an ® Intel ^{TM} core i7-3610QM 2.30 GHz processor with 8 GB of RAM.

5.1 Problem Instances

For testing the exact algorithms problem instances need to be created. When creating a problem for the RRML problem, first an initial problem needs to be created. Because minimizing the number of late jobs has a polynomial time algorithm, there is no literature on how to create interesting instances for this problem. The weighted variant, $1||\sum w_j u_j$, however is NP-hard and has literature on creating instances for the problem. To create an initial scenario from these instances for the $RRML$ problem, the weights can be omitted.

For each job J_j an initial processing time p_j is generated from the uniform distribution $[1, 30]$. In [20] now the two parameters R and T are used to create the due dates. The parameter R is the *relative range of due dates* and the parameter T is the *average tardiness factor*. When the processing times are computed, the value of $P = \sum_j p_j$ is known. The due dates are now generated from the uniform distribution $[P(1 - T - \frac{R}{2}), P(1 - T + \frac{R}{2})]$. The jobs will be sorted in non-decreasing due date order. When R is small, the due dates are close together, as R increases the smallest and biggest due date generated will be further apart. Therefore the term relative range of due dates. When T is small, the due dates are close to P and they will be further from P when T increases. When the due dates are close to P most jobs will fit, but when T increases and the due dates get smaller than P then not all jobs will fit and more jobs will become late. Therefore T is called the average tardiness factor.

For each combination of n and s the values of R and T are chosen between 0.2 and 1.0 with steps of 0.2. Only those values are chosen that keep the due dates positive. This gives 14 different combinations per combination of n and s. To complete the initial problem, the value p_0 is randomly created.

To complete the instances the set of scenarios S needs to be created. These problems can be created in different ways. One could create the processing times for each job J_j in the scenario from an uniform distribution as well, then p_j will be independent from each p_j^s. Here is chosen to let the scenario processing times depend on the initial processing times. It may be that p_j^s is bigger or smaller than p_j with a certain factor or that the processing time stays the same. The first two types of problem instances create problems for the RRML problem.

- Random increase and decrease (RID). Here the following procedure is used for each job J_j in each scenario s. With a probability of 33 % it holds that $p_j^s = p_j$ for job J_j, with probability 33% it holds that $p_j^s = 0.75p_j$ and the last 33% makes $p_j^s = 1.5p_j$.
- Opposite increase and decrease (OID). Now all scenarios are created at the same time. Consider job J_j in each scenario, in 33% of the scenarios it will hold that $p_j^s = 0.75p_j$, for another 33% $p_j^s = 1.5p_j$ and for the other scenarios the processing time will stay the same. In which scenario the job increases or decreases is randomly determined.

To also see the effect of the algorithms on instances of the RRML-I problem, the following instances have been created as well.

- Random increase (RI). Here the following procedure is used for each job J_j in each scenario s. With a probability of 50 % it holds that $p_j^s = p_j$ and in the other cases $p_j^s = 1.5p_j$.
- Opposite increase (OI). Now all scenarios are created at the same time. Consider job J_j in each scenario, in half of the scenarios will hold that $p_j^s = 1.5p_j$ and the other half will have $p_j^s = p_j$. In which scenario the job increases of stays equal is randomly determined.

The opposite increase instance types, OID and OI, are not defined properly for all sizes of S. To overcome this problem, both OID and OI will use their random equivalent strategy when there is only one scenario. This solves the problems for OI. OID is also not defined well for two scenarios, in this case a job will increase in one scenario and decrease in the other. Which scenario increases or decreases is chosen randomly.

The probabilities p_s for each scenario $s \in S$ are determined as done in [24]. Probabilities $p_s^{'}$ are set to a value from the uniform distribution $[1, 3]$. These are then scaled such that $p_s = \frac{p_s^{'}}{\sum_{s \in S} p_s^{'} + p_0}$ for each scenario $s \in S$. The value p_0 is scaled in the same way.

5.2 Branching Algorithms Compared

In this section the performances of the two branching algorithms are compared; branch and bound and branch and price. Both algorithms have different parameters for which the optimal value has to be determined, this is done in Section 5.2.1 by testing the different parameters on instances with 40 jobs and two scenarios. In the next section is investigated what are difficult instances and what properties these instances have. Next in Section 5.2.3 the performances of the branching algorithms are tested when the number of jobs and scenarios increase. They are then compared to each other.

The set H was defined in Section 3.4, this set contains all jobs which will be definitely on time in all schedules of the optimal solution. Set H can always be obtained in the root node with the use of algorithm 1.

This reduces the number of jobs for which its configuration needs to determined by the branching algorithms. Using this set to pre set multiple jobs will always give a better performance of the branching algorithms. It simply reduces the number of jobs on which the branching algorithms need to run and the calculation is fairly easy. Therefore in the performance measure of the branching algorithms the set H is always used to reduce the running time of the algorithms. This can be done because the size of this set is always equal for all different settings and algorithms and therefore this will not influence the findings.

For an optimal performance of the branch and price algorithm separate recovery must work well. For each pricing problem with negative reduced cost the corresponding column will be added to the master problem. One could choose to add more than the column with the highest reduced cost and investigate the performance when also the second best is added and so on as is done in [24], but this is not the focus of this thesis.

When an initial solution is added, for each scenario $s \in S$ a solution can be constructed that satisfies the recovery algorithm and thus with the new initial column they would together form a valid solution. This can be done for each scenario by performing the Moore-Hodgson algorithm on the on time jobs of the initial schedule. This gives a valid solution for each scenario. This is exactly how the *Moore-Hodgson initial* upper bound is created. In stead of adding a second best column, the columns corresponding to the solutions for each scenario are added. In Table 5.1 the results are shown of performing separate recovery 100 instances with 40 jobs and two scenarios, with either adding these extra columns or not adding them. The 100 instances are 25 for each instance type with taking the values of R and T randomly while of course keeping the lower bound positive.

	Time (ms)		Added columns		Iterations		
Extra scenarios	avg.	max	avg.	max	avg.	max	integral
Yes	50	629	2158	4837	543	1203	86
No	77	758	2441	5112	938	1945	72

Table 5.1: Results for initial testing of Separate Recovery.

When adding the extra columns, the solution is reached quicker. Even with less columns and less iterations. Most importantly, there are 14 more instances solved to an integral solution directly. This is a much better performance than without adding the extra columns. Therefore, this addition will always be used when performing separate recovery.

5.2.1 Optimal Settings

To perform one of the branching algorithms a decision needs to be made on the branching and bounding heuristic. In Section 3.3 different upper bounds and branching heuristics were discussed. The *everyting late* upper bound made all the non-determined jobs late, where the *Moore-Hodgson initial* upper bound used the Moore-Hodgson algorithm on these jobs. For deciding on which job to branch, the next options were discussed in this section. It can be done in increasing or decreasing due dates order, called *first* or *last* here. Other options were to select in order of increasing or decreasing initial processing times, so taking the *longest* or *shortest* job. The last option was to branch on a job that does not satisfy the recovery algorithm when looking at the Moore-Hodgson solutions. The last option will be called *invalid job* in this chapter. The different ways to traverse the branch tree, which node to choose to do the branching step on, were explained in Section 2.2.3. Recall that it were the following three options; in *breadth first* a new node is added at the end of the queue, *depth first* adds the node in front of the queue and *best first* takes the node with the lowest lower bound first. This all is summarized in Table 5.2.

Not all of these settings will work equally well. In this section will be tested what choices give the best result for these two algorithms. The branching heuristic *invalid job* is not valid in branch and bound, as mentioned earlier. To test the branching algorithms the following problem instances are created for 40 jobs and two scenarios. For each type of instance, RID, OID, RI and OI, 25 instances are created with random values for R and T from the set $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. A combination is of course only feasible if the lower bound of the uniform distribution the due dates are taken from stays positive, such that each due date is positive. This gives 100 instances to test the branching algorithms in the first round. These were the same instances on which the separate recovery algorithm was tested. For solving each instance in this first round a running time limit of three minutes is set, if this run exceeds the three minutes it has failed.

Node selection	**Branch Heuristic**	**Upper bound Heuristic**
Best first	First	Everyting late
Depth first	Last	Moore-Hodgson initial
Breadth first	Longest	
	Shortest	
	Invalid job	

Table 5.2: Different heuristics for the branch and bound and branch and price algorithm.

		Time (ms)		Visited nodes			Weighted	
Node selection	Branch	avg.	max	avg.	max	failed	avg. time	avg. nodes
Best first	First	692	15280	4761	116673	16	29381	66220
	Last	1984	110547	10601	521147	16	30466	46137
	Longest	4770	174710	11559	270411	9	20540	43189
	Shortest	403	24790	3669	190918	15	27342	57797
	Invalid job	3976	163091	14181	811663	12	25098	38865
Depth first	First	7872	166314	185953	3537245	9	23363	552774
	Last	10549	177950	254712	4585096	14	34272	808128
	Longest	2787	117819	70827	3076575	5	11647	288064
	Shortest	9595	148687	196004	3224191	24	50492	1041304
	Invalid job	4736	157881	111769	3713715	6	15251	363279
Breadth first	First	17677	171154	110389	856934	75	139419	696808
	Last	9475	93524	76464	528986	74	135663	692960
	Longest	6969	107067	46052	590140	77	140202	679940
	Shortest	15135	121777	96662	683319	65	122297	596245
	Invalid job	17636	114787	150216	1044344	69	129667	686307

Table 5.3: Results when looking at the different branching heuristics, node selection options and the upper bound is everything late for branch and bound.

In Table 5.3 the results for the branch and bound algorithm can be found when the upper bound heuristic is *everything late*. In this table the columns *time* shows the average time and maximum time that was needed given all the successful runs, thus the runs that completed within three minutes. The *visited nodes* column show the average and maximum number of visited nodes of all the non failed runs. The columns below

weighted show the average time and nodes when the failed runs are taken into account. For each failed run three minutes run time is counted and their by then number of visited nodes.

What can be seen is that branching on the jobs with the longest initial processing time or on an *invalid job* works better than all other branching options, independent of the node selection decision. They solve more problems within the given time, are quicker overall and with less average nodes. Here it is important to look at the weighted columns, where the failed runs are also taken into account. The best choice for the node selection is depth first. Best first is not much worse, but breadth first does not work very well.

		Time (ms)		Visited nodes			Weighted	
Node selection	Branch	avg.	max	avg.	max	failed	avg. time	avg. nodes
Best first	First	1850	64843	11900	317883	11	21446	40773
	Last	572	30249	3517	149275	14	25691	43015
	Longest	3037	175883	11281	266837	7	15424	46688
	Shortest	876	53710	4464	173681	14	25953	36621
	Invalid job	575	20723	5678	190163	11	20311	42030
Depth first	First	7451	156452	108174	2185875	8	21254	309610
	Last	7327	168329	104358	2378465	13	29774	432716
	Longest	2496	104584	36622	1491601	6	13146	196393
	Shortest	5438	171673	75942	2460039	18	36859	505244
	Invalid job	5228	149511	68695	1926698	7	17462	233920
Breadth first	First	5858	144653	50401	1411457	11	25013	162183
	Last	3138	140019	28104	1620457	15	29667	184261
	Longest	4958	174131	47230	1491601	7	17210	131670
	Shortest	2369	146005	12942	580892	20	37895	203568
	Invalid job	4648	171282	39820	1471149	9	20429	146787

Table 5.4: Results when looking at the different branching heuristics, node selection options and the upper bound is Moore-Hodgson initial for branch and bound.

The results with the Moore-Hodgson upper bound for branch and bound can be found in Table 5.4. Branching on the longest initial job or an *invalid job* still works best and depth first is the best way to traverse the branch tree. What is interesting is that selecting the nodes breadth first is not much worse than the other two strategies as it was with the *everything late* upper bound. This will be caused by the tighter upper bound which causes that less nodes are necessary. This is also seen when comparing the two different upper bounds with each other, the used number of nodes decreased drastically. For the *Moore-Hodgson initial* upper bound less nodes are used and therefore in the most cases this upper bound finds the optimal solution the quickest. In the two best configurations, depth first with longest job or *invalid job*, a bit less problems get solved than with the *everything late* upper bound. However on 100 instances a difference of one problem solved is not that much. Therefore the Moore-Hodgson upper bound is still considered the best setting.

Thus for the branch and bound algorithm there are two settings that work pretty good. Because one best setting is wanted, this will be traversing the tree in depth first order, branching on the job with the longest initial processing time and using the Moore-Hodgson initial upper bound.

The question is whether the same settings work the best for the branch and price algorithm. Branching on the *invalid job* is of course not defined for this algorithm, so this option is not valid here. From all the 100

instances there are 86 instances that have an integral solution in the root node of the branch tree. Thus that the average number of visited nodes is very low makes a lot of sense because for all these directly integral cases the number of visited nodes is one.

		Time (ms)		Visited nodes		Added columns		Iterations		
Node selection	Branch	avg.	max	avg.	max	avg.	max	avg.	max	failed
Best first	First	918	27317	21	1288	721	20513	206	5423	0
	Last	1294	64897	18	611	1143	53569	299	13895	0
	Longest	411	10715	7	326	363	9626	95	2743	0
	Shortest	1637	63455	26	996	1432	62098	407	18038	0
Depth first	First	1773	57000	38	1359	1580	61252	437	17006	0
	Last	1953	118950	30	1326	1520	90430	414	24281	1
	Longest	677	19082	12	491	582	14816	155	4059	0
	Shortest	2648	159608	45	1916	2288	134432	656	38675	2
Breadth first	First	1238	45398	24	1288	1014	34336	284	9177	0
	Last	2184	117461	25	921	1816	90584	476	23452	0
	Longest	932	35745	11	341	805	28190	199	6577	0
	Shortest	2087	68453	37	996	1833	68061	520	19761	1

Table 5.5: Results when looking at the different branching heuristics, node selection options and the upper bound is everything late for branch and price.

In Table 5.5 the results for the upper bound heuristic *everything late* are given. Branching on the job with the longest initial processing time works the best here, as it was for the branch and bound algorithm. This is clear from every measurement in this table of results. The best node selection strategy is however not the same as for the branch and bound algorithm. Here it is best to choose best first and depth first, which was best for branch and bound, is even the worst option.

For the Moore-Hodgson initial upper bound the results are found in Table 5.6. It is clear that for the branch and price algorithm this upper bound works better. Less time, nodes, columns and iterations are needed. Here branching on the longest job in the initial problem works best again, as is traversing the tree in a best first order.

Thus for the branch and price algorithm, use the *Moore-Hodgson initial* upper bound heuristic and branch on the longest initial job. This can be done while traversing the tree in best first order. This gives the quickest results.

After this first round of testing the best settings for the branching algorithms are found. For the branch and bound algorithm the tree will be traversed depth first, while branching on the longest initial job and having the *Moore-Hodgson initial* upper bound. This setting will be used for this algorithm in the rest of this thesis. For the branch and price algorithm this is almost the same, only the tree is traversed in best first order. Next will be investigated with the use of these settings what instances are hardest.

5.2.2 Difficulty of Instances

The following set of instances is created, again for 40 jobs and two scenarios. For each type of scenario instances, RID, OID, RI and OI, 140 instances are created. These 140 instances are 10 instances for each possible combination of R and T. These instances are then solved with the branch and bound algorithm

		Time (ms)		Visited nodes		Added columns		Iterations		
Node selection	Branch	avg.	max	avg.	max	avg.	max	avg.	max	failed
Best first	First	792	23613	21	1288	698	20513	200	5423	0
	Last	1162	60422	17	611	1102	53569	290	13895	0
	Longest	351	10147	7	326	320	9626	83	2743	0
	Shortest	1580	60029	26	996	1432	62098	407	18038	0
Depth first	First	1120	52586	24	1323	992	45899	276	11826	0
	Last	885	30580	21	811	875	30197	240	8430	1
	Longest	522	18065	11	491	446	13419	122	4059	0
	Shortest	3095	112818	71	3693	2241	72174	666	22814	1
Breadth first	First	1035	41997	23	1288	892	34336	253	9177	0
	Last	1732	106255	22	921	1572	90584	413	23452	0
	Longest	555	19751	9	326	497	16580	126	3887	0
	Shortest	1876	59183	37	996	1755	62098	499	18038	1

Table 5.6: Results when looking at the different branching heuristics, node selection options and the upper bound is Moore-Hodgson late for branch and price.

and the branch and price algorithm. The best settings as found previously are used for these algorithms.

	Time (ms)		Visited nodes			
Instance type	avg.	max	avg.	max	failed	weighted avg. time
OID	4868	164469	26571	587364	14	22381
RID	5180	160177	33002	914477	23	33901
OI	1953	170758	7070	417751	1	3225
RI	1127	155711	5346	721370	0	1127

Table 5.7: Results when looking at the different instance types for Branch and Bound

	Time (ms)		Visited nodes		Added columns		Iterations			
Instance type	avg.	max	avg.	max	avg.	max	avg.	max	integral	failed
OID	859	18151	19	326	737	15001	200	3915	92	0
RID	3024	46603	49	936	2066	38547	580	9531	82	0
OI	2679	93818	103	4552	1327	54203	488	19453	119	3
RI	272	17720	8	591	161	7771	52	3012	131	1

Table 5.8: Results when looking at the different instance types for Branch and Price

In tables 5.7 and 5.8 the results for the different instance types for the branching algorithms are shown. From the branch and bound results it is clear that the instances when only increasing the processing times in the scenarios and not also decreasing some processing time compared to the initial processing time are easiest, except for one, all problems get solved in time. The solutions are found using a lot less nodes

than when also decreasing processing times. Random increase and decrease is the most difficult of the four different types.

When looking at the results for the branch and price algorithm, the scenario types with only increasing processing times have the most directly integral solutions. This might be an indicator that these instances are easy. However, these are only the only instances for which some runs failed. This seems really contradictory. A direct cause is not clear. Looking at the running time and the number of visited nodes, OID and RI are the quickest, with the random only increasing processing times clearly the number one. It also needs least number of columns and iterations. The RID here is the most difficult when looking at all the numbers accept the number of fails, as it was for the branch and bound algorithm.

		Time (ms)		Visited nodes			
R	T	avg.	max	avg.	max	failed	weighted avg. time
0.2	0.2	6224	109920	23710	257742	2	14913
	0.4	11819	160177	83191	914477	7	41251
	0.6	9165	170758	36439	721370	10	51874
	0.8	6403	164469	27932	544861	1	10743
0.4	0.2	2	50	30	534	0	2
	0.4	6077	106203	36319	587364	3	19121
	0.6	1667	22496	12947	122166	7	32875
0.6	0.2	0	0	1	1	0	0
	0.4	2704	85437	19687	580186	1	7136
	0.6	1701	49388	9450	186943	7	32903
0.8	0.2	0	0	1	1	0	0
	0.4	105	3585	1343	46217	0	105
1.0	0.2	0	0	1	1	0	0
	0.4	83	3289	914	35912	0	83

Table 5.9: Results when looking at the instances with different R and T for Branch and Bound

The results for the branching algorithms for the different values of the parameters R and T are given in tables 5.9 and 5.10. Recall that the parameter R is the relative range of due dates and the parameter T is the average tardiness factor. The results for the branch and bound algorithm show that when increasing T, it takes longer to solve the problem and more instances fail. Except for the combination of $R = 0.2$ and $T = 0.8$, these are even the easiest instances for $R = 0.2$. This seems to show that the more jobs are late, the harder the problems are.

When increasing R, the problems get solved quicker and less instances fail. When R increases the range of due dates is larger and thus more jobs can be on time. Thus this again shows that the problems are harder when more jobs will be late. These same patterns are seen when looking at the results for the branch and price algorithm. This is also shown in the number of directly integral solutions.

The explanation for these results is quite easy when looking at the following values for the different instances. The first measurement is the size of set H, the set of jobs that are all on time in all schedules of the optimal solution. As explained earlier, a larger set H is expected to result in quicker results. The other measurements are the distances from the initial lower and upper bound to the optimal solution. These values are calculated as follows. Let the lower and upper bound calculated in the root node be LB_{root} and

		Time (ms)		Visited nodes		Added columns		Iterations			
R	T	avg.	max	avg.	max	avg.	max	avg.	max	integral	failed
0.2	0.2	578	10706	17	396	200	2719	89	1766	30	0
	0.4	2343	34673	39	476	1595	23361	443	6365	21	0
	0.6	4353	77951	178	4552	3300	54203	1041	19453	15	1
	0.8	1129	6464	21	201	1248,25	6801	325	1677	23	0
0.4	0.2	38	877	1	31	12	201	6	142	39	0
	0.4	2889	31946	62	591	1869	23671	550	6402	25	1
	0.6	4613	46603	74	726	3148	38547	840	9531	17	1
0.6	0.2	16	168	1	1	5	7	2	3	40	0
	0.4	5058	93818	191	3866	1966	29361	828	14696	32	0
	0.6	2491	40007	33	381	1569	21785	446	6654	25	1
0.8	0.2	17	276	1	1	5	7	2	3	40	0
	0.4	363	12860	12	456	96	3160	47	1608	37	0
1.0	0.2	12	74	1	1	4	6	2	3	40	0
	0.4	22	446	1	1	7	50	2	14	40	0

Table 5.10: Results when looking at the instances with different R and T for Branch and Price

UB_{root} and the optimal value is *opt*. The distance to the lower bound is then $opt - LB_{root}$ and the distance to the upper bound is $UB_{root} - opt$. Because the lower bound calculation is different for branch and bound then for branch and price, two values are shown. If these are values are small, the results are also expected sooner.

The results for the random only increase instances are shown in Table 5.11. The size of set H decreases when T increases and thus when more jobs can be on time. The size of the set increases when R increases, this again nicely reflects the property of the parameter R. The distances to the upper and lower bound have exactly the opposite pattern and this again reflects the expected results; larger distances result in more difficult instances.

In Table 5.12 the results are shown for the opposite increase instances. The patterns for the size of set H and the distances of the lower and upper bound are the same as for the random increase instance. However, the sizes of set H are a bit smaller and the distances are a bit larger. This implies that the opposite increase instances are a bit harder than the random increase instances. This was clear in the running times of the branch and bound algorithm and in the number of integral solution for the branch and price algorithm.

Next the values are given for the random increase and decrease instances are given in Table 5.13. Compared to the random only increase instances the differences are not that clear. When $R = 0.2$ the size of set H is larger, but for all other values of R the size is smaller. The distances to the initial lower and upper bound are a lot larger than the random only increase instances. These then explain the difficulty the algorithms had with the random increase and decrease instances.

At last the values for the opposite increase and decrease are given in Table 5.14. First these will be compared to the values of the opposite only increased values. The size of set H is larger in all cases, this is caused by the fact that the decreased jobs make more room for jobs to be on time. The distance to the initial upper and lower bound however are larger in most cases. This explains that the branch and bound algorithm needs more time to solve the opposite increase and decrease instances and less instances are directly integral in the branch and price algorithm.

Compared to the random increase and decrease instances, the distances to the initial lower and upper

		size set H		Distance to LB - BB		Distance to LB - BP		Distance to UB	
R	T	avg.	max	avg.	max	avg.	max	avg.	max
0.2	0.2	22	26	0.00	0.00	0.00	0.00	0.09	0.62
	0.4	19	24	0.04	0.36	0.01	0.1	0.07	0.67
	0.6	16	26	0.03	0.35	0.12	0.41	0,14	0.65
	0.8	13	22	0.07	0.70	0.00	0.00	0.16	0.61
0.4	0.2	25	31	0.00	0.00	0.00	0.00	0.01	0.12
	0.4	20	23	0.08	0.80	0.01	0.06	0.18	0.65
	0.6	18	22	0.01	0.13	0.14	0.87	0.33	0.87
0.6	0.2	29	36	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	23	27	0.00	0.00	0.00	0.00	0.28	1.00
	0.6	20	25	0.00	0.00	0.40	1.00	0.23	1.00
0.8	0.2	32	39	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	23	28	0.00	0.00	0.00	0.00	0.23	1.00
1.0	0.2	37	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	25	31	0.00	0.00	0.00	0.00	0.03	0.28

Table 5.11: Difficulty of instances RI

		size set H		Distance to LB - BB		Distance to LB - BP		Distance to UB	
R	T	avg.	max	avg.	max	avg.	max	avg.	max
0.2	0.2	19	24	0.04	0.41	0.00	0.00	0.24	1.00
	0.4	17	23	0.04	0.31	0.02	0.10	0.12	0.76
	0.6	15	18	0.25	1.00	0.12	0.41	0.21	0.90
	0.8	11	15	0.04	0.43	0.00	0.00	0.11	0.53
0.4	0.2	21	27	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	19	22	0.01	0.12	0.01	0.06	0.17	0.80
	0.6	17	22	0.09	0.46	0.05	0.34	0.33	0.83
0.6	0.2	22	29	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	19	25	0.10	0.44	0.05	0.29	0.20	0.65
	0.6	18	25	0.05	0.32	0.01	0.09	0.15	0.65
0.8	0.2	25	34	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	19	24	0.04	0.42	0.03	0.28	0.19	0.97
1.0	0.2	33	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	19	30	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.12: Difficulty of instances OI

bound however are sometimes smaller and sometimes larger. This is not a clear indicator whether these problems are harder than the random instances. The size of the set H however is larger than for the random increase and decrease instances. This shows why the RID instances where the hardest to solve.

So when the value R increases the size of set H increases and the distances to the lower and upper

		size set H		Distance to LB - BB		Distance to LB - BP		Distance to UB	
R	T	avg.	max	avg.	max	avg.	max	avg.	max
0.2	0.2	22	28	0.25	0.53	0.03	0.13	0.27	1.00
	0.4	17	24	0.46	0.62	0.11	0.42	0.58	1.00
	0.6	14	19	0.18	0.49	0.02	0.20	0.57	0.97
	0.8	12	17	0.48	0.99	0.05	0.22	0.46	1.00
0.4	0.2	28	39	0.10	0.53	0.00	0.00	0.17	0.83
	0.4	17	24	0.44	1.00	0.14	0.34	0.57	1.00
	0.6	13	18	0.36	1.00	0.08	0.38	0.40	1.85
0.6	0.2	34	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	21	25	0.14	1.00	0.04	0.16	0.54	1.00
	0.6	18	23	0.33	0.74	0.09	0.43	0.70	1.00
0.8	0.2	39	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	24	36	0.21	0.90	0.02	0.22	0.12	0.72
1.0	0.2	40	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	28	40	0.13	0.70	0.00	0.00	0.17	0.65

Table 5.13: Difficulty of instances RID

		size set H		Distance to LB - BB		Distance to LB - BP		Distance to UB	
R	T	avg.	max	avg.	max	avg.	max	avg.	max
0.2	0.2	25	28	0.34	1.56	0.03	0.17	0.29	0.76
	0.4	18	21	0.26	0.59	0.07	0.44	0.52	0.87
	0.6	16	21	0.39	0.58	0.10	0.40	0.30	0.72
	0.8	13	15	0.49	1.04	0.04	0.29	0.14	1.04
0.4	0.2	29	32	0.15	1.00	0.01	0.13	0.21	0.74
	0.4	21	27	0.47	1.46	0.08	0.38	0.43	0.99
	0.6	18	22	0.45	1.00	0,10	0.24	0.34	0.69
0.6	0.2	35	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	23	27	0.26	0.88	0.05	0.24	0.51	1.00
	0.6	20	25	0.13	0.42	0.06	0.32	0.30	1.00
0.8	0.2	40	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	26	35	0.07	0.39	0.00	0.00	0.20	0.83
1.0	0.2	40	40	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	29	35	0.06	0.48	0.00	0.00	0.13	0.76

Table 5.14: Difficulty of instances OID

bounds decrease. The increase of T causes the opposite effect. Small sets H and small distances are easy to solve instances, therefore the instances are hardest to solve when $R = 0.2$ and $T = 0.8$, because increasing T to 0.8 is the only exception. This pattern cannot be checked for other R, as with these values the lower bound of the uniform distribution the initial due dates are taken from will then be negative. As the negative due dates are unwanted, and leaving them out is not an option, creating instances with these R values is not

feasible.

The random increase and decrease instances are the hardest to solve. These have the smallest sets H in most cases, which leaves the most jobs to be set. The distances to the optimal solution are the largest, which also indicate hard solutions. This matches the results found when performing the branching algorithms on these instances.

In the next section will be investigated what influence the increase of the number of jobs n and the size of the set S has on the performance of the branching algorithms.

5.2.3 Increased Number of Jobs and Scenarios

In the previous section results showed that the best setting for the branch and bound algorithm is traversing the tree best first, while branching on the job with the longest initial processing time and using the *Moore-Hodgson initial* upper bound. For the branch and price algorithm the *Moore-Hodgson initial* upper bound is used as well and the tree is traversed in depth first order. In a node will also be branched on the job with the longest initial processing time. These two algorithms will now be tested further for these settings when the number of jobs and scenarios increase.

Besides testing on the branching algorithms, the performance of the separate recovery algorithm will be tested on its own as well. This is interesting because when separate recovery gets impossible to solve, branch and price will have no purpose. The influence of the increase of jobs and scenarios on the running time and the number of integral solutions will be tested.

To investigate the influence of the number of jobs instances sets will be created for $n = 20, 40, 60, 80, 100, 120$ and all these will have only two scenarios. To investigate the influence of the number of scenarios, sets will be created for $|S| = 1, 2, 3, 5, 8, 10$ with 40 jobs. Each test set will contain 100 instances; for each of the four types of instances 25 instances are created and these instances are made with random values for R and T.

First the influence of the increasing number of jobs and scenarios on the size of set H is investigated. These seemed to be of great influence on the performance of the branching algorithms. By looking at their values and later comparing them to the results of the branching algorithms, this hopefully gives more insight into what makes an instance difficult.

Influences on the Size of Set H

In the previous section the results of the initial experiments showed that when the size of set H is close to the number of jobs of the problem, then they are solved quicker than when this set is small. The effect of the parameters R and T on the size of H were discussed in the previous section. Here will be shown what effect the increase of jobs or the number of scenarios has on the size of the set. The results are shown in Figure 5.1. The black line in the left graph indicates a size equal to the number of jobs. When the number of jobs increases, the size of H increases, but the size gets further away from the maximum size while increasing n. The number of jobs in set H stay about half the number of jobs in total. This is a first indication that problems with a higher number of jobs are harder to solve.

In the graph on the right, the influence of the increase in scenarios is displayed. For all these instances the maximum size of set H is 40, because this is the number of jobs for the instances used in these experiments. The decrease is large when increasing the size of S from 1 to 2 to 3. After that the average size seems to stabilize around 17 jobs. This indicates that for a small set of scenarios the problems will be a lot easier to solve than when increasing the size of the set to 10. Here it however seems that when $|S|$ is 8 or 10, the difficulty is the same.

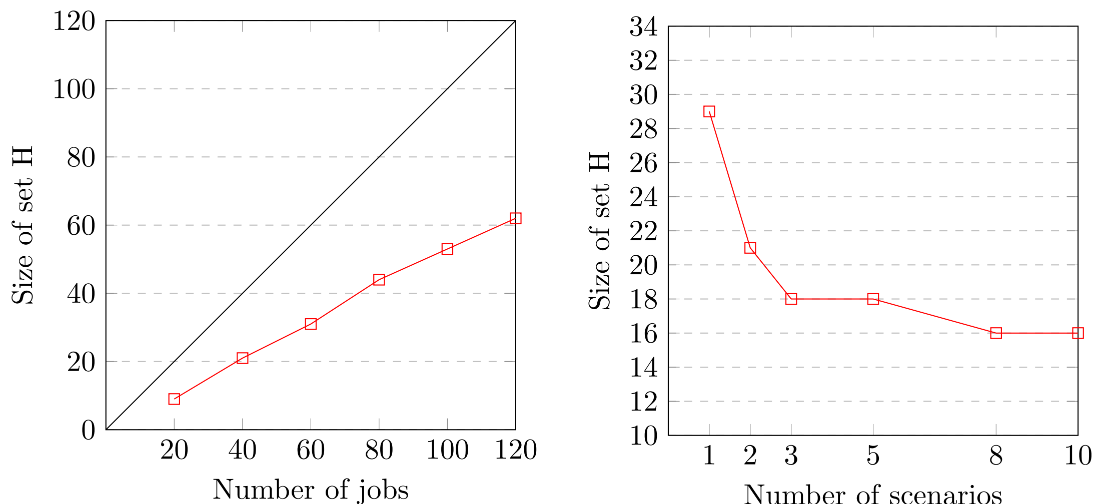

Figure 5.1: Influence of the number of jobs and scenarios on the size of set H

Separate Recovery

The performance of the separate recovery algorithm is of huge influence on the performance of the branch and price algorithm. When the separate recovery algorithm takes a long while to solve the problem in the root node, the chance will be small that the problem will be solved at all. Unless this solution is directly integral of course. In Figure 5.2 the results can be found when increasing the number of jobs from 20 to 120, for all problems with two scenarios. These values are all averages over all 100 instances.

An exponential increase in time can be found here. Overall the running time increases to an average of 17.000 ms. The number of runs that result in an integral solution, also decreases a lot. For 20 jobs almost all instances have an integral solution, for 120 jobs it is less than half. The combination of these two results will probably result in more fails for the branch and price algorithm when the number of jobs increases.

The number of columns and iterations needed reflect the running time. More columns and iterations are needed to reach a solution, thus this results in the increase of running time. The influence of the number of jobs is quite big, but 120 jobs is quite a lot thus this is not a direct problem for this algorithm.

In Table 5.3 the results can be found of the experiments with 40 jobs and an increasing number of scenarios. As expected here the number of columns and iterations also increase. What is interesting to see is that the number of columns needed increases to about the same number as when increasing the number of jobs. The number of iterations however does not increase as much. This can be caused by the fact that when the number of scenarios is larger, more columns are added during one iteration because of an increased number of pricing problem and extra added columns. In total this again has an increasing result in the running time, but for ten scenarios this did not get dramatical. Within two seconds most problems are solved. What may cause longer running time for the branch and price algorithm is the decrease in integral solutions. For the 100 instances, there are only 35 integral when reaching the ten scenarios. Thus a solution gets found quite quickly, but they are mostly non-integral. Next, the effect on the branching algorithms is investigated.

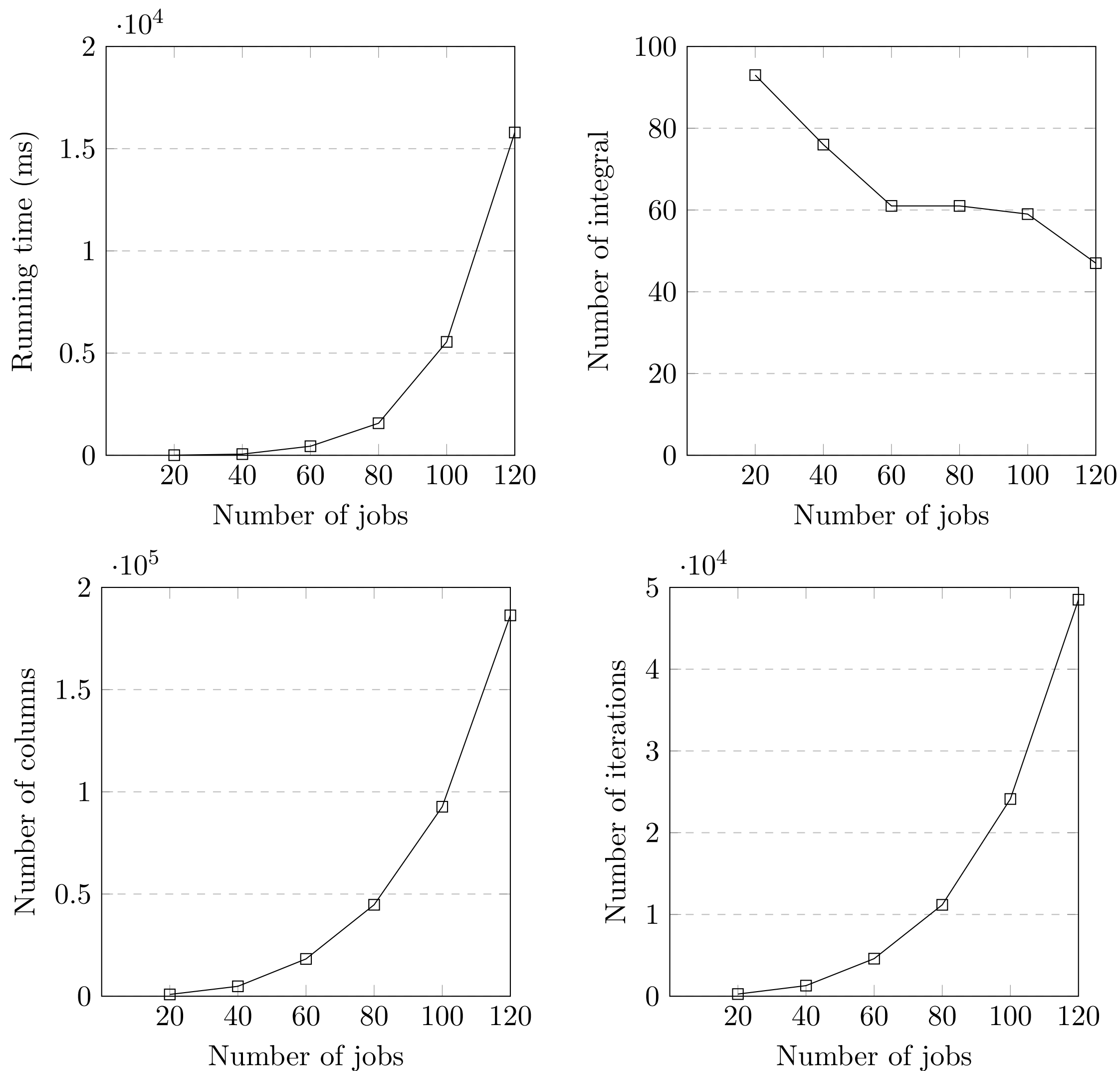

Figure 5.2: Influence of the number of jobs on different performance measures of Separate Recovery.

The Branching Algorithms

When the number of jobs increases the depth of the branching tree increases, because there are more jobs to be set. Even when leaving the jobs that are in set H out, the number of jobs to determine their settings for increases. This suggests that problems will get harder when the number of jobs increases from 20 to 120. The results for the separate recovery algorithm already suggested this, because the running time to solve the algorithm increased drastically. The figures in Table 5.4 show the results for both branching algorithms. The tables show results for both algorithms for the average over all instances that reached a solution and the weighted results also take the failed instances in account. These are stopped when reaching the three minutes.

As expected the problem increases in difficulty. The number of fails increases for both algorithm to 50.

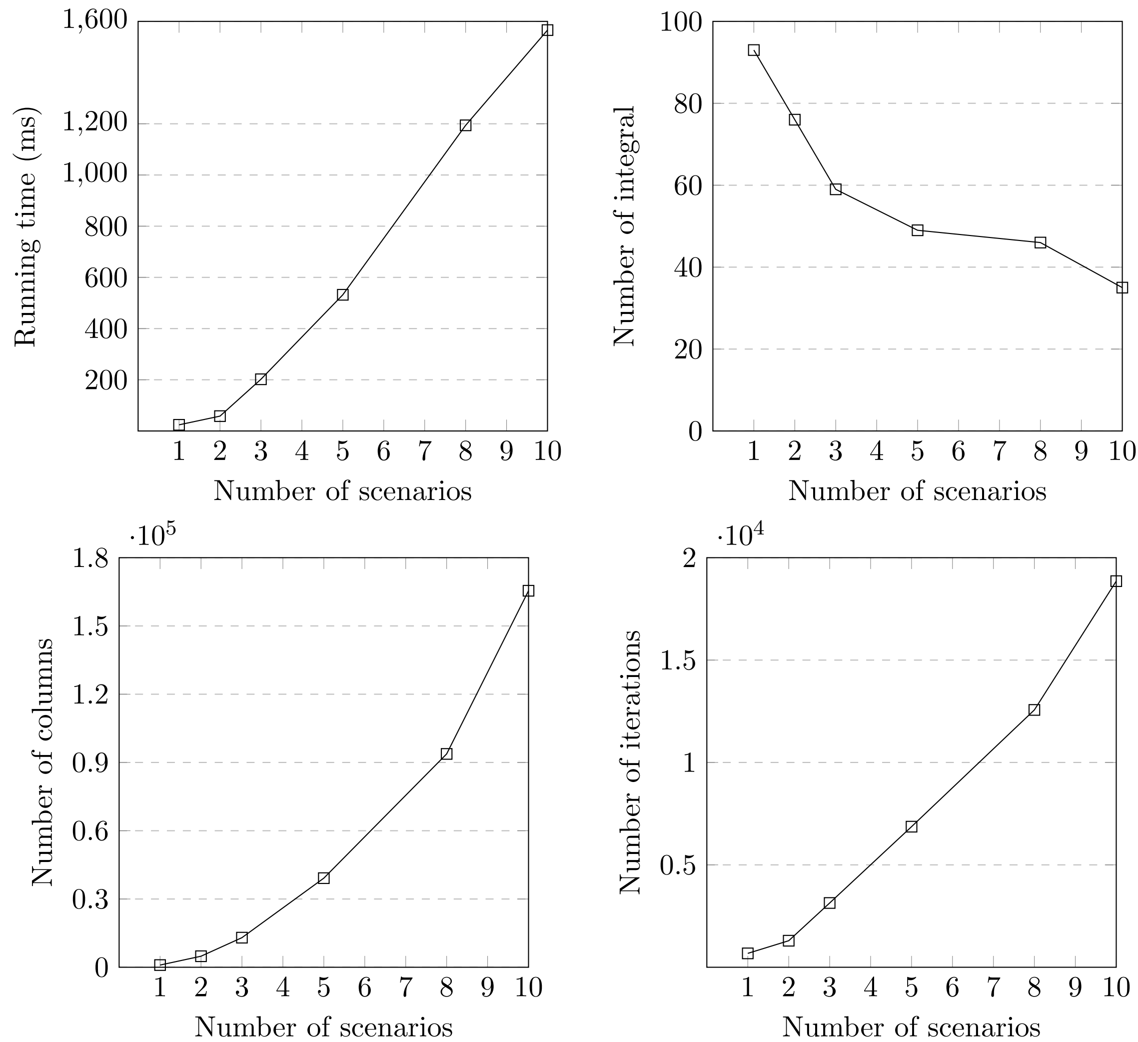

Figure 5.3: Influence of the number of scenarios on different performance measures of Separate Recovery.

Branch and price however solves more instances in general. Weighted, the branch and price algorithm does this quicker, the branch and bound algorithm solves the problems that it solves quicker than the branch and price algorithm. This is probably caused by the drastic increase in running time for the separate recovery algorithm in the root node. For 120 jobs, the branch and price algorithm solves the problems either in the root node or not at all.

The number of nodes that are needed to solve the problems stay a lot smaller for the branch and price algorithm as for branch and bound. This was also the case in Section 5.2.1. The number of nodes needed increases until about 60 jobs are reached. This can be caused by a longer time for calculations in one node, because of the increased number of jobs. This increases the running time of for example the Moore-Hodgson solution for calculation the lower bound in the branch and bound algorithm. For the branch and price algorithm this was already established. An other explanation is the fact that only the easy problems get

solved and they do not need many nodes. For the weighted problem this can be caused by the fact that the algorithm is stopped and is not continued until finished.

The last graphs show the number of added columns and needed iterations, these are the total columns and iterations used for all the separate recovery algorithms performed in the different nodes. The same arguments as for the nodes can be used to explain these results.

This means when increasing the number of jobs the branch and price algorithm performs the best. More instances get solved and it gets done quicker. The question of course stays how both algorithms perform when there is no limit un the running time, this would however cost a lot of time, because this could take forever.

Next to the figures in Table 5.5 the results are shown when increasing the number of scenarios. When the number of scenarios increases, the depth of the tree stays the same. The number of children for one child however increases. The number of children was equal to $2^{|S|}+1$ and thus for eight scenarios this already gives 257 children. Where for two scenarios this was only five. The separate recovery algorithm showed not a large increase in running time, but the number of integral solutions decreased.

Here the running time for the separate recovery algorithm did not increase as much as for the increase scenarios. For both branching algorithms this seems true, thus the increased number of children of a node has a large influence. This is again caused by the large increase in number of failing instances. Even more than 60 are failed for both algorithm. For the branch and price algorithm for eight or ten instances, problems get either solved in the root node or not at all, as was seen for the increased number of jobs.

The total number of nodes that needs to be traversed before finding the optimal solution is larger than when increasing the number of jobs. What happens here again, as we saw when increasing the number of jobs, is that the number of nodes decrease again after some time, which again can be explained by the fact that less nodes can be explored because of the longer calculation time needed in a node. The same pattern can again be seen for the number of columns and iterations.

So again the branch and price algorithm performs a bit better than the branch and bound algorithm. This makes that the branch and price algorithm is the algorithm someone should use to solve the recoverable robustness problem for minimizing the number of late jobs. However, the number of jobs should not increase too much and the same holds for the number of scenarios. Otherwise even the branch and price algorithm does not solve the problems quickly.

5.3 Direct Linear Program

The next algorithm to test is the direct linear program as explained in Section 3.3.5. As mentioned earlier, the set H can be predetermined to make to problem easier to solve. In the case of the linear program however, this means for each job that was in set H, constraints need to be added to force the information of set H. Because u_i denotes whether the job on spot i is late or not, this cannot help to force job J_j to be on time or late. Therefore variables $\tilde{u}_k$ and $\tilde{u}_k^s$ are defined such that they are one when job J_k is late in the schedules. To get the right values for these variables we need them to satisfy these constraints:

$$\begin{aligned} \tilde{u}_j &= u_i \quad \text{if } x_{ij}=1 \quad \forall i,j \\ \tilde{u}_j^s &= u_i^s \quad \text{if } x_{ij}^s=1 \quad \forall i,j,s \in S \end{aligned}$$

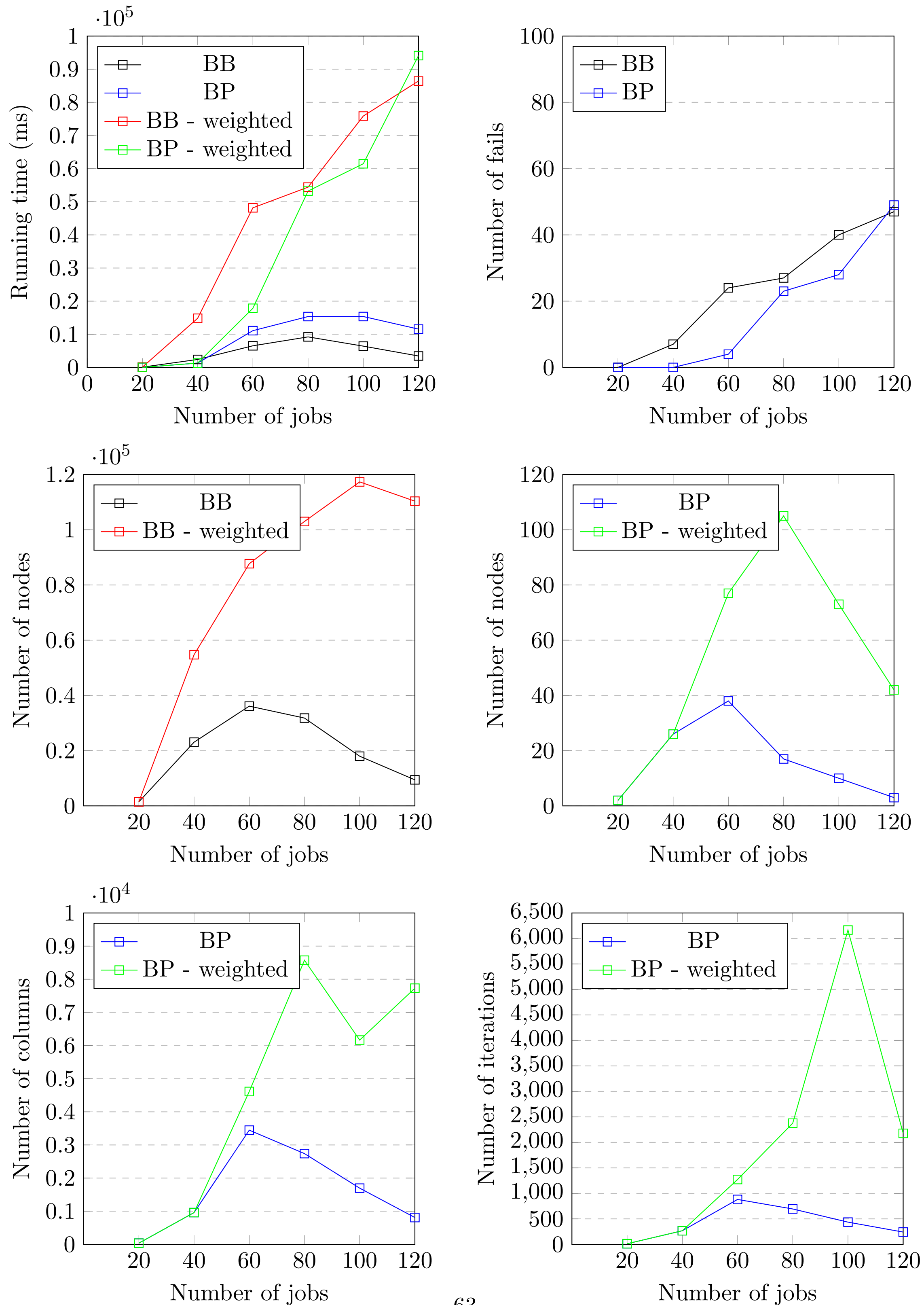

Figure 5.4: Influence of the number of jobs on different performance measures of both Branching Algorithms.

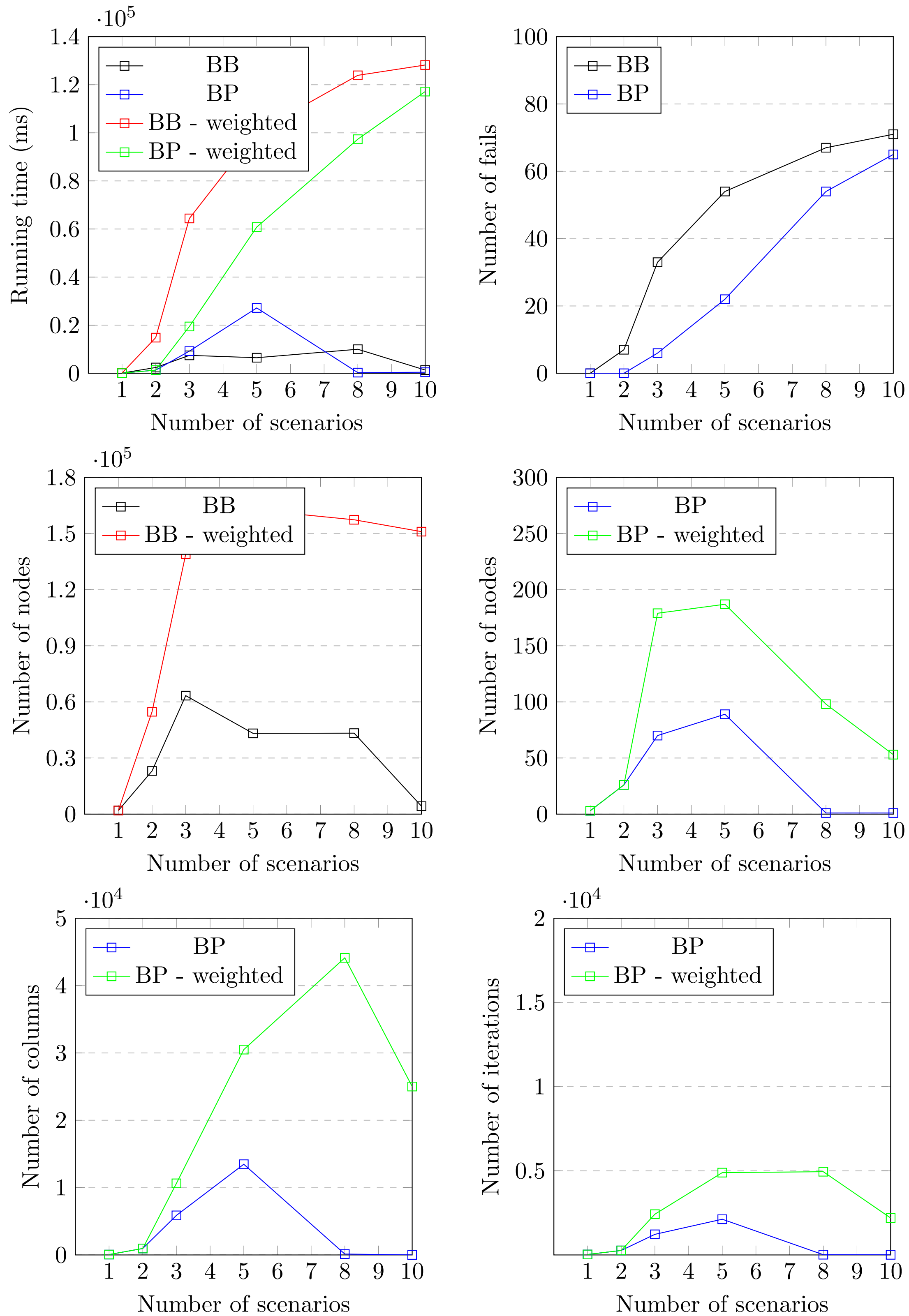

Figure 5.5: Influence of the number of scenarios on different performance measures of both Branching Algorithms.

This means that when job J_j is on spot i and the job on spot i is on time, job J_j needs to be on time as well. The same holds when the job J_j is on spot i and the job on spot i is late, then job J_j must be late too. These two constraints can be written in a linear function as follows:

$$\begin{aligned} x_{ij} - 1 \leq \quad & \tilde{u}_j - u_i \quad \leq 1 - x_{ij} \quad \forall i, j \\ x_{ij}^s - 1 \leq \quad & \tilde{u}_j^s - u_i^s \quad \leq 1 - x_{ij}^s \quad \forall i, j, s \in S \end{aligned}$$

Now for each job J_j in H, the constraint $\tilde{u}_j^s = 0$ needs to be added for each scenario $s \in S$ and $\tilde{u}_j = 0$ is also added to maintain it in the initial problem.

An instance set for 20 jobs with one scenario is created, that consists of 100 instances with 25 of each instance type, all with random values of R and T. Solving these problems gave the following results. Without the extra constraints 31 instances failed and the average time was 27303 ms. Adding the extra constraints decreased the number of fails to three and the average time to solve the problems was 10273 ms. Therefore, it is clear that although extra constraints are necessary, it results in much better results.

Additional testing with the extra constraints was done for multiple instances sets, the results are shown in Table 5.17. For each combination of n and $|S|$ again the same instance set of 100 instances is created with 25 instances of each instance type with random values for R and T. The results show that the performance is much worse than for either branching algorithm. Here for 40 jobs and only one scenario, which is the same set as used previously, already 59 instances fail, although branch and price solves all instances and branch and bound fails only for seven. The used time was also only one tenth of the time. This shows that it is better to use one of the branching algorithms, as they are a lot quicker. This algorithm was not tested more, because no promising results where expected.

$\|S\|$	1		2	
n	avg. time (ms)	failed	avg. time (ms)	failed
10	138	0	515	0
15	1495	0	10273	3
20	4892	2	38080	25
30	19538	25		
40	24280	59		

Table 5.15: Direct Linear Programming results for different n and the sizes of S

Instead of adding the extra constraints, one could also leave the jobs from set H out of the calculation. To do so, the due dates of the jobs that are left of the initial problem and each scenario need to be adapted, so that the left out jobs can always fit. This can be done. However, this would reduce the number of jobs to half, as was seen for the results for set set H. This would still not give great results that perform better than the branching algorithms.

5.4 Dynamic Programming

In Section 3.3.4 was explained how the RRML problem can be solved with dynamic programming. For implementing this algorithm efficiently for only one schedule, thus with no scenarios, a two-dimensional

array with dimensions n and $P = \sum p_j$ is used. When the recoverable robustness problem is being solved, this two-dimensional array becomes a $|S|+2$-dimensional array. When the size of the scenario set S is known, implementing this is not a problem, when $|S|$ is unknown however, initializing this multi-dimensional array becomes hard and this gives the first problems with this algorithm. To solve this problem, all the values of $f_j(r, t_0, \ldots, t_{|S|})$ for all jobs j are stored in one long array.

Let $P = \max\{\max_{s\in S} \sum p_j^s, \sum p_j\}$, the length of this array will then be $nP^{|S|+1}$. Storing all values in one array seems nice solution to the initialization problem, an array however has a length limit of 2147483648. Which is approximately $2.1 \cdot 10^9$. This seems a lot, but Table 5.16 shows how little this is. Assuming that a job has an average processing time of 15, because the processing times are taken from the uniform distribution $[1, 30]$, and thus P is approximately $15n$, the estimated length of the array is shown in this table.

n $\backslash \|S\|$	1	2	3	4
5	$2.8 \cdot 10^4$	$2.1 \cdot 10^6$	$1.6 \cdot 10^8$	$1.2 \cdot 10^{10}$
10	$2.3 \cdot 10^4$	$3.4 \cdot 10^6$	$5.1 \cdot 10^8$	$7.6 \cdot 10^{10}$
15	$7.5 \cdot 10^5$	$1.7 \cdot 10^7$	$3.8 \cdot 10^9$	$8.6 \cdot 10^{11}$
20	$1.8 \cdot 10^6$	$5.4 \cdot 10^8$	$1.6 \cdot 10^{11}$	$4.9 \cdot 10^{13}$
25	$3.5 \cdot 10^6$	$1.3 \cdot 10^9$	$4.9 \cdot 10^{11}$	$1.9 \cdot 10^{14}$
30	$6.0 \cdot 10^6$	$2.7 \cdot 10^9$	$1.2 \cdot 10^{12}$	$5.5 \cdot 10^{14}$

Table 5.16: The length of the array given n and the size of S

This shows that when there are only two scenarios, a problem cannot have more than 20 jobs. There might even be instances where this would fail, because this is only an estimation. For three scenarios this decreases to only 10 jobs. For one scenario this is still solvable when the number of jobs increases to 20. This shows the problem of the solution of the initializing problem. Unfortunately, this cannot be improved. So testing the performance for this algorithm can only be done for a very little number of scenarios and jobs.

In Section 3.4 the set H was defined as the set of jobs that are on time in every schedule of the optimal solution. This set H can be determined before running the algorithm with the Algorithm 1 and while running the dynamic programming algorithm this information can be used to reduce the running time.

This algorithm will now be tested for the following instances. For one scenario it will be tested for 5, 10, 15, 20, 30 and 40 jobs, so that it can be compared to the results of the branching algorithms. For two scenarios it will be tested for 5, 10, 15 and 20 jobs and for three scenarios only for 5 jobs. The following instance set is created for each combination. For each type of instance, RID, OID, RI and OI, 25 instances are created with random values for R and T from the set $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. A combination is of course only feasible if the lower bound stays positive.

The results for the experiments are shown in Table 5.17. It is clear that the average array size was a little under estimated, because the number of instances that go outside the size of the array, given in the columns with 'out', is 100 quite soon when the number of scenarios is two. For three scenarios five jobs was even too much. This can be explained by the fact that many instances only have increased jobs in the scenarios and thus P^s will be higher than $15n$ in these scenarios.

The results that are here, show that increasing the number of scenarios from one to two results in a drastic increase in running time. Increasing the number of jobs for one scenario also shows a nice time increase, but it is not as drastic as for increasing the number of scenarios. For 40 jobs this is solved in an average time of 3 seconds. The branching algorithms however solved this within one second and thus it is better to use these algorithms. The dynamic programming algorithm does outperform the direct linear program, because all problems get solved here and on average they are solved in a tenth of the time.

$\|S\|$	1			2			3		
n	avg. time (ms)	failed	out	avg. time (ms)	failed	out	avg. time (ms)	failed	out
5	7	0	0	1265	0	0	-	-	100
10	54	0	0	10673	0	44	-	-	-
15	203	0	1	-	-	100	-	-	-
20	203	0	1	-	-	100	-	-	-
30	1395	0	1	-	-	-	-	-	-
40	3513	0	1	-	-	-	-	-	-

Table 5.17: Dynamic programming results for different n and the sizes of S

5.5 Conclusion

In this chapter the performances of the four algorithms that were developed in Section 3.3 were tested; Branch and bound, branch and price, the direct linear program and dynamic programming. Firstly different instances needed to be created. Four different types were made, differing in increasing and decreasing the processing times in the scenarios and forcing the scenarios to oppose each other or letting it happen in random. The parameters R, the relative range of due dates, and T, the average tardiness factor, where used to create different due dates.

The direct linear program performed the worst. Instances with only one scenario already fail when having only 20 jobs and when increasing this number to 40, already more than half of the instances fail. Increasing the number of scenarios to two gives already failing instances for 15 jobs. The branch and price algorithm solves these problems all, within a second. The implementation of this algorithm is fairly easy, because only the constraints have to be added to the CPLEX solver, but the running times are too long.

The dynamic programming algorithm performs better. All instances that can be solved, are solved within time and quicker than the direct linear program. However, because of the implementation limitations, no instances with more than two scenario can be solved and for two scenarios the number of jobs needs to stay small. This limitation is bad, however the instances that can get solved are solved much slower than by either of the branching algorithms.

For the branching algorithms many different settings were available, the best settings were determined for the branching heuristic, the node selection options and the two possible upper bound. The *Moore-Hodgson initial* upper bound turned out to be the best and branching on the job with the longest initial processing time gives the quickest results. Best first is the best way to select a node for the branch and bound algorithm and depth first for the branch and price algorithm.

The difficulty of the problems strongly depended on the size of the set H. The set is large when all the due dates of the instances are wide apart and are close to the sum of the processing times of the initial problem. The set is smaller when the jobs only increase in the scenarios. When set H is smaller, more jobs are left to determine and this resulted in longer processing time. The instances where the processing times increase and decrease however had less tight initial upper and lower bounds, which resulted in the fact that these were the most difficult to solve problems.

Growing the number of jobs and the size of the scenario set had a big influence on the performance of the Separate Recovery algorithm. The number of integral results decreases to only half of the instances and the running time, number of added columns and needed iterations increase exponentially when the number of jobs increase. When the scenario set gets larger, the time increases as well, however less drastically. These

results were not very promising for the results for the branch and price algorithm.

These expectations became reality. When increasing the number of jobs, for both algorithms the number of fails increase until half of the instance set. When the number of scenarios is ten, it is even more than 60. The average weighted time came closer to the maximum of 180.000 ms with each increase. The problems that do get solved however, do that quicker on average, because only the easy instances are left. The number of used nodes increases, but decreases again later on. This is caused by longer times needed in each node. Overall the performance of the branch and price algorithm is a bit better, mostly because it solves more instances.

Chapter 6

Conclusion and Further Research

To handle uncertainties in the data of various machine scheduling problems, the recoverable robustness method was applied to these problem in this thesis. For minimizing the number of late jobs, the recoverable robustness problem was proven to be weakly NP-hard when there is only one scenario, even for the case with common due dates. For this problem a dynamic programming algorithm was developed that runs in pseudo-polynomial time. It might be interesting to research whether this problem turns strongly NP-hard when the number of scenarios is unknown.

The Moore-Hodgson algorithm [19], that solved the problem of minimizing the number of late jobs for one machine when all the data is certain, turned out to give some viable information for the problem. Applied to each scenario and the initial problem this gives the lower bound to the optimal solution. If these solutions do not violate the recovery algorithm, this of course is then the optimal solution. When an instance has only one scenario with one increased job, a simple polynomial time algorithm exists with the use of the Moore-Hodgson algorithm. When the scenarios may only contain increased processing times, instances for two or three jobs have easy solutions. When this increases to four, the solution is not directly clear. It might be equal to the lower bound, or one higher.

For jobs that are on time in the Moore-Hodgson solutions of each scenario and the initial problem, it was proven that these are on time in the optimal solution of the recoverable robustness problem when these jobs have shorter processing times than the jobs for which this not holds. The property of these jobs can be used to decrease the problem size as these jobs can be left out of calculations. Finding more jobs that can be left out of calculation might make the problem even easier.

Next the recoverable robustness approach was applied to problems of scheduling with rejection. For minimizing the makespan with rejection a polynomial time algorithm was developed for the recoverable robustness method. When release dates are available, minimizing the makespan with rejection is already NP-hard when all data is certain. A dynamic programming algorithm was extended to handle uncertainty covered in scenarios. The same was done for minimizing the lateness and tardiness with rejection. These problems can not be solved in polynomial time, because the recoverable robustness problem is NP-hard when the initial problem is already NP-hard.

For the recoverable robustness problem of minimizing the number of late jobs, three algorithms where developed next to the dynamic programming algorithm. An integer linear program was made and solved with CPLEX, results however showed no promising results. The dynamic programming algorithm preformed better, however it was limited by its implementation.

The two well performing algorithms are the branch and bound algorithm and the branch and price

algorithm. Both algorithms performed best while using the *Moore-Hodgson initial* upper bound, which uses the Moore-Hodgson solution of the initial problem as schedule for the initial problem and then performs the Moore-Hodgson algorithm on the jobs that were on time in this schedule for each scenario $s \in S$ to obtain the jobs that can be on time in these scenarios. This gives a valid solution and thus an upper bound to the problem. Researching whether a tighter upper bound exists might be interesting to do.

Branching on the jobs with the longest initial processing time gave the best results for both algorithms. The lower bound in each node for the branch and bound algorithm was calculated by calculating the Moore-Hodgson solutions for each scenario. For the lower bound in the branch and price algorithm the Separate Recovery approach developed by [5] was used. One could also use the combined recovery approach from [5] to research if this gives better results.

Increasing the number of jobs increases the average running time a lot. The number of failed instances increase to about half the number of instances. The same holds for increasing the number of scenarios. The average time needed in a node just becomes too large. Branch and price however solves more problems than the branch and bound algorithm and thus performs better.

For further research it might be interesting to find good approximation algorithms, instead of the exact algorithms. These might give quite good results, although not optimal, but have a lot better running time. The *Moore-Hodgson initial* upper bound gave quite good solutions most of the time. Research could be done on how tight this upper bound is. One might also consider to not solve the direct linear program to optimality, but until the duality gap is small. This gives not the optimal solution, but a quite good solution.

The experiments also showed that when scenarios contained increasing and decreasing jobs compared to the initial processing times and thus where not only instances to the RRML-I problem, the instances were harder to solve. Interesting research might be done on exactly why this is the case. The question what makes an instances difficult is an important one to solve, to fully grasp the difficulty of the recoverable robustness problem.

Bibliography

[1] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust optimization.* Princeton University Press, 2009.

[2] John R Birge and Francois Louveaux. *Introduction to stochastic programming.* Springer Science & Business Media, 2011.

[3] J. Blazewicz, K.H. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer Publishing Company, Incorporated, 2nd edition, 2012.

[4] P.C. Bouman, J.M. van den Akker, and J.A. Hoogeveen. Recoverable robustness by column generation. In Camil Demetrescu and MagnúsM. Halldórsson, editors, *Algorithms – ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 215–226. Springer Berlin Heidelberg, 2011.

[5] P. Bouwman. A column generation framework for recoverable robustness. Master's thesis, Universiteit Utrecht, 2011.

[6] C. Büsing, A. MCA Koster, and M Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optimization Letters*, 5(3):379–392, 2011.

[7] Z. Cao and Y. Zhang. Scheduling with rejection and non-identical job arrivals. *Journal of Systems Science and Complexity*, 20(4):529–535, 2007.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[9] R.L. Graham. Bounds on performance of scheduling algorithms. *Computer and Job Scheduling Theory.*

[10] H. Hoogeveen and V. T'kindt. Minimizing the number of late jobs when the start time of the machine is variable. *Operations Research Letters*, 40(5):353 – 355, 2012.

[11] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical report, DTIC Document, 1955.

[12] J.A. Hoogeveen J.M. van den Akker. Minimizing the number of tardy jobs. In J.Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms Models and Performance Analysis*, pages 227–243. CRC Press Inc., 2004.

[13] J.A. Hoogeveen J.M. van den Akker and S.L. Van de Velde. Applying column generation to machine scheduling. In J. Desrosiers G. Desaulniers and M.M. Solomon, editors, *Column Generation*, pages 305–330. Springer, 2005.

[14] A. B. Keha, K. Khowala, and J.W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Computers and Industrial Engineering*, 56:357–367, 2009.

[15] L. Lu L. Zhang and J. Yuan. Single machine scheduling with release dates and rejection. *European Journal of Operational Research*, 198(3):975 – 978, 2009.

[16] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity.

[17] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Bucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[18] C. Liebchen, M. Lübbecke, R.H. Möhring, and S. Stiller. Recoverable robustness. Technical report, ARRIVAL-Project, August 2007.

[19] J.M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):102–109, 1968.

[20] C.N. Potts and L.N. van Wassenhove. A branch and bound algorithm for the total weighted tardiness problem. *Operations Research*, 33(2):363–377, 1985.

[21] S. Sengupta. Algorithms and approximation schemes for minimum lateness/tardiness scheduling with rejection. In *Algorithms and Data Structures*, pages 79–90. Springer, 2003.

[22] D. Shabtay, N. Gaspar, and M. Kaspi. A survey on offline scheduling with rejection. *Journal of Scheduling*, 16(1):3–28, 2013.

[23] V. T'kindt, H. Scott, and J. Billaut. *Multicriteria scheduling: theory, models and algorithms*. Springer, 2006.

[24] D.D. Tönissen. The size robust multiple knapsack problem, 2013.

[25] L. Zhang, L. Lu, and J. Yuan. Single-machine scheduling under the job rejection constraint. *Theoretical Computer Science*, 411(16–18):1877 – 1882, 2010.