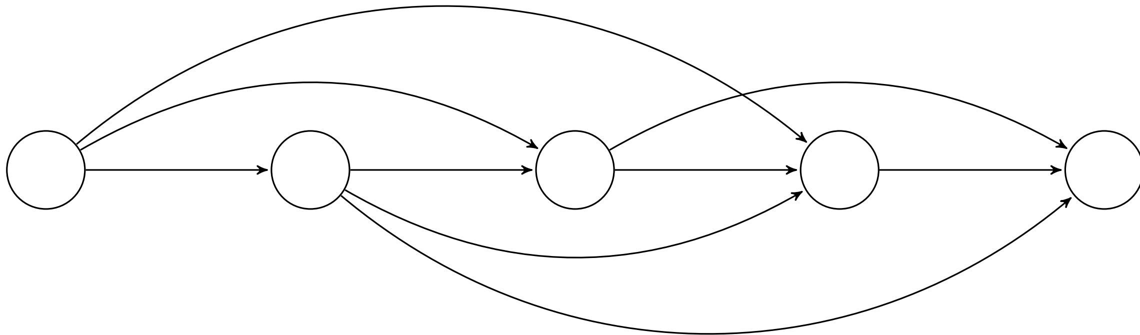


A column generation approach to the flexible job-shop problem with ordering requirements on operations

Master's thesis in Computer Science: Algorithms, Languages and Logic

Lars Niclas Jonsson

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018:NN

A column generation approach to the flexible job-shop problem with ordering requirements on operations

Lars Niclas Jonsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Mathematical Sciences
Mathematical Optimization
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2018

A column generation approach to the flexible job-shop problem with ordering requirements on operations
Niclas Jonsson

© Niclas Jonsson, 2018.

Supervisor: Karin Thörnblad, GKN
Examiner: Ann-Brith Strömberg, Department of Mathematical Science

Master's Thesis 2018:NN
Mathematical Science
Mathematical Optimization
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A visualization of a graph that is used to generate columns of special sequences.

A column generation approach to the flexible job-shop problem (FJSP) with ordering requirements on operations
Niclas Jonsson
Mathematical Sciences
Chalmers University of Technology

Abstract

Manufacturing companies need to have an efficient production and one important effort is to optimize the production schedules. The task of finding good and useful schedules for the products and the production machines is complicated. A further complication is that some production jobs may require a maintenance task of the machine to be performed before the job can be processed.

The company GKN Aerospace Sweden AB has developed a mathematical model that defines feasible sequences of jobs and maintenance tasks to be scheduled in the machines. The model optimizes the schedules in terms of minimizing the finishing times and the tardiness of the jobs. Since the number of feasible sequences is huge for reasonable numbers of jobs to be scheduled, the size of the model grows too large for practical problem settings.

This project presents a column generation approach to generating a useful subset of all existing feasible sequences to be included in GKN's present model. Our results show that the schedules computed by the mathematical model become better when more feasible sequences are included, but the computation times then grow longer.

Keywords: Scheduling, Flexible job-shop, Column generation, Integer linear optimization, CPLEX, Time-indexed model.

Acknowledgements

First and foremost I would like to thank my industrial supervisor Karin Thörnblad at GKN, and my supervisor Ann-Brith Strömberg from the optimization research group. They have provided me with feedback in a highly efficient manner and been very supportive and encouraging during the project. I also wish to thank GKN for giving me the possibility to write my master thesis at their logistical department.

Niclas Jonsson, Gothenburg, May 2018

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem Definition . . . 1
1.2.1 GKN's Problem is NP-hard . . . 1
1.2.2 The Special Jobs . . . 2
1.2.3 The Special Sequences . . . 2
1.2.4 The Families of Special Sequences . . . 3
1.2.5 GKN's Model . . . 3
1.3 Research Questions . . . 3
1.4 Limitations . . . 4
1.5 Thesis Outline . . . 4

2 A Literature Review of Scheduling Problems and Models **5**
2.1 Scheduling Problems . . . 5
2.1.1 The Job-shop Scheduling Problem . . . 5
2.1.2 The Flexible Job-shop Scheduling Problem . . . 5
2.1.3 The Parallel Machine Scheduling Problem . . . 6
2.1.4 The Batch Machine Scheduling Problem . . . 6
2.1.5 Objective Functions . . . 6
2.1.5.1 Minimizing Makespan . . . 6
2.1.5.2 Minimizing Weighted Averages of Completion Times and Tardiness . . . 7
2.2 Scheduling Models and Methods . . . 7
2.2.1 Constraint Programming Applied to Scheduling Problems . . . 7
2.2.2 Model Scheduling Problems as Integer Linear Programs . . . 8
2.2.3 Column Generation for the Parallel Machine Scheduling Problem . . . 9
2.2.4 Column Generation for the Batch Machine Scheduling Problem 10

3 Linear and Integer Optimization **11**
3.1 Linear Programming . . . 11
3.1.1 The Simplex Method . . . 12
3.1.2 Dantzig-Wolfe Decomposition and Column Generation for Linear Programs . . . 13
3.2 Integer Linear Programs . . . 17
3.2.1 Branch and Bound . . . 19

3.2.2 Applying Column Generation to Integer Linear Programs Using Branch-and-Price . . . 19
3.2.2.1 Branch-and-Price . . . 19
3.2.3 Applying Column Generation Without Using Dantzig-Wolfe Decomposition . . . 20

4 An Integer Linear Optimization Model of GKN's Scheduling Problem 23
4.1 The Constraints Defining the Special Jobs . . . 23
4.2 The Scheduling Model . . . 23
4.2.1 Disadvantages with the Scheduling Model . . . 27
4.3 Shortening the Time Span of the Model . . . 28

5 A Column Generation Model for Generating Special Sequences 31
5.1 The Integer Master Problem . . . 31
5.2 The (Integer) Restricted Master Problem . . . 33
5.3 The Linear Programming Dual Problem of the Restricted Master Problem . . . 33
5.4 The Subproblem . . . 34
5.4.1 The Subproblem of Generating SpSs of Solder Jobs . . . 34
5.4.2 The Subproblem of Generating SpSs of Titanium Jobs . . . 35
5.4.3 Generating Several Families of SpSs . . . 35

6 Implementation 37
6.1 Modification of the Instances . . . 37
6.2 Comparisons of the Approaches to be Benchmarked . . . 38
6.3 Computation Time Measurements . . . 38

7 Tests and Results 39

8 Discussion and Conclusion 45

9 Future Research 47
9.1 Parallelize the Scheduling Model . . . 47
9.2 Compute Several Schedules . . . 47
9.3 Develop a New Model . . . 47
9.4 Perform a Study of Domination Criteria . . . 48

Bibliography 49

A Appendix A I

B Appendix B V

1
Introduction

1.1 Background

The company GKN has a heat treating department located in Trollhättan, Sweden. This department handles orders of products from all over the world that require heat treating. These products are engine components for aeroplanes and rockets which GKN develops and produces. The quality of these products are therefore extremely high, and hence, the products take a lot of time to produce. An example of such a product can be seen in Figure 1.1. The machines that the department uses for the production are furnaces. The high quality requirements of the products result in expensive furnaces, which are very large. Few furnaces are therefore used for the production. These furnaces are not identical, since different products require different operations. However, some products can be heat treated in one of several furnaces, but the processing time of the job might depend on the specific furnace. Different cleaning processes, such as the bake-out and the vacuum test, are sometimes required in the furnaces before some specific products can be heat treated. These cleaning jobs are referred as maintenance jobs. Karin Thörnblad has developed an integer optimization model from the model in [1] that schedules the requested jobs for the heat treating department so that the productivity of the furnaces are high, while priority orders are scheduled earlier than other jobs. A job's priority is a function of its deadline. The model in [1] also schedules the maintenance jobs so that the schedule can be used in practise without modifications. The model has increased productivity by 9% (see [2]). While scheduling maintenance jobs in addition to the heat treatment jobs and the titanium jobs is necessary to obtain a schedule that can be used in practise for GKN, it has been found that computing a schedule with many heat treatment jobs and the titanium jobs that require maintenance jobs is very time consuming.

1.2 Problem Definition

1.2.1 GKN's Problem is NP-hard

The scheduling problem at GKN is a generalization of the *Flexible Job-shop Scheduling Problem* (FJSP), which is **NP**-hard (see [3, Chapter 34]). The FJSP is a scheduling problem with a set of jobs and a set of machines. Each job is defined as an ordered set of operations, and the task is to schedule all the given operations to the machines in the given order. Each operation requires a certain amount of time to

Figure 1.1: The compressor rear frame is a product that requires heat treatment.

be processed in one specific machine. The FJSP is a generalization of the *Job-shop Scheduling Problem* (JSP). In the JSP, each operation can only be heat treated in one of the machines (see [4]), while in the FJSP an operation can be heat treated in a subset of the machines (consisting of one or more machines). Hence, the JSP is a special case of the FJSP. It should be noted that most jobs in the heat treatment department at GKN consists of only one operation.

1.2.2 The Special Jobs

Two types of jobs that GKN has are a bit special: the titanium jobs and the solder jobs. These jobs can be heat treated in a furnace only under special conditions. A solder job can only be heat treated in a furnace if that furnace has recently had a bake-out job or if the last job that was heat treated in that furnace was another solder job that requires a higher maximal temperature than the solder job at hand. The titanium jobs can only be heat treated in a furnace if the furnace has recently had a bake-out job and (had a vacuum test job or if the last job that was heat treated in that furnace also was a titanium job). These two types of jobs are harder to schedule than the others, since they require that the furnaces have been prepared for them, which is not the case for the other jobs. The bake-out job and the vacuum test job are refereed to as *maintenance jobs*. The maintenance jobs takes several hours to perform in a furnace and no other job can be heat treated in the furnace during that time.

1.2.3 The Special Sequences

A *special sequence* is defined as an ordered set of maintenance jobs and special jobs if the jobs in the ordered set can be scheduled in a furnace as they are ordered in the set. While any order of the titanium jobs in a schedule is valid, this is not the case for solder jobs. Hence, in this respect the titanium jobs are easier to schedule than the solder jobs.

1.2.4 The Families of Special Sequences

A group of special sequences such that each special job is included in exactly one of the sequences, is called a family of special sequences. All requirements regarding maintenance jobs are satisfied when each special sequence in a family of special sequences is scheduled. Computing and storing the elements (i.e., ordered sets) in the family requires a time and a space that is exponential as a function of the number of special jobs included in a problem instance. Even for instances with just ten special jobs, finding and verifying an optimal solution has been too time consuming for GKN, since such an instance can have more than 150 000 families of special sequences.

1.2.5 GKN's Model

GKN uses a complex integer linear program, that is referred to as the *scheduling-model*, which is composed by several types of constraints, and which model their scheduling problem. The problem is challenging to solve for practical instances as a consequence of the large numbers of variables and constraints. Especially the constraint regarding the special jobs, which require pre-computed special sequences and families of special sequences. The scheduling model requires pre-computed special sequences and families of special sequences, since it can not compute that by it self, and hence, GKN needs to pre-compute special sequences and families of special sequences for the model.

The methodology GKN uses to find a dense schedule when there are many special jobs is to compute some of all the families of special sequences, and then let the scheduling model consider only these families. This speeds up the process of finding a usable schedule. It can be shown that some special sequences will not be part of an optimal schedule, since other special sequences are dominating them, and it is therefore no point for the scheduling model to consider them and the families that include these special sequences (see [5, Chapter 2] for domination). However, it can not be guaranteed that a family that is not computed by GKN is not part of an optimal schedule, since special sequences that exist are not generated even when it is unknown if they are dominated by others or not.

1.3 Research Questions

While it is impossible to know exactly which family of special sequences that will be part of an optimal schedule, it is likely that the special sequences of a family that is part of an optimal, or close to an optimal, schedule have some property that can be measured. We will in this project try to generate special sequences which we believe are part of an optimal or close to optimal schedule using a column generation method (see [6]). The purpose of the project is to answer the following questions:

1. Is it faster to find a dense schedule by using column generation to generate special sequences compared to the current method used by GKN?
2. How "good" are the schedules that are found using the proposed method compared too those found by the current method?

1.4 Limitations

We will generate special sequences for a simpler version of the scheduling problem as compared to the problem faced by GKN. This simpler version will not allow batches of jobs to be scheduled, where a batch consist of a number of jobs that are heat treated in a furnace at the same time. There will be no scheduled breaks such that no jobs can be heat treated during that time in the simpler version of the problem. Furthermore, each schedule are independent of the schedule that was computed for the problem instance of the previous day in the simpler version of the problem. In reality, the department makes a new schedule every day and uses the schedule that was computed the previous day to allow the scheduling model to be able to find more optimized schedules. The department do this by using the fact that some maintenance jobs in some special sequences are not needed if the last job that was processed in a furnace the previous day was a maintenance job or a solder job.

1.5 Thesis Outline

The thesis is organized as follows. Chapter 2 presents an introduction to scheduling problems and models. Chapter 3 gives a basic introduction to the linear and integer optimization theory that is needed to understand the models that are used in the project. A modified version of the scheduling model that GKN uses for their scheduling problem is presented in Chapter 4, and the column generation model that generates special sequences is given in Chapter 5. Chapter 6 discusses implementation details that are relevant for the result, which is presented in Chapter 7. A discussion and conclusions of the results are found in Chapter 8. Future research ideas are presented in Chapter 9.

2

A Literature Review of Scheduling Problems and Models

A literature review of relevant scheduling problems for different strategies, and models to solve them — and how well these strategies and models perform in practise — are presented in this chapter.

2.1 Scheduling Problems

The scheduling problems that are reviewed are the *Job-shop Scheduling Problem* (JSP), the *Flexible Job-shop Scheduling Problem* (FJSP), the *Parallel Machine Scheduling Problem* (PMSP), and the *Batch Machine Scheduling Problem* (BMSP). The latter three of these are generalizations or special cases of the JSP. The task of solving these problems is **NP**-hard (see [7, Chapter 2]).

2.1.1 The Job-shop Scheduling Problem

The JSP is a scheduling problem, in which a finite number of operations shall be scheduled to a finite set of machines. Once an operation starts, a certain time that is dependent on the machine is required until the operation is complete. A machine that starts an operation is occupied until that operation is completed. Each of the operations can only be scheduled to one specific machine, and an operation can only be scheduled in a machine that is not occupied. Furthermore, each operation belongs to a certain job, and for each job, a partial order for the operations that the job consists of is given. An operation can only be scheduled in a machine when all the operations that have a lower order than the operation that is considered has been completed. The JSP has a great practical value for the industry, and has therefore been studied a lot (see [8] for a review).

2.1.2 The Flexible Job-shop Scheduling Problem

The FJSP is an extension of the JSP such that each operation can be scheduled to a subset of all machines in the FJSP (see [4]). While the JSP has a great practical value, the FJSP has an even greater practical value, since it is natural that (at a given point in time) an industry has several machines that can handle the same operation in the case that a machine needs maintenance work, gets broken, etc (see

[9]). FJSP is **NP**-hard to solve, since the JSP is a special case of the FJSP and JSP is **NP**-hard.

While the FJSP is **NP**-hard in general, it is known how to find and verify a solution that minimizes the *makespan* (see Section 2.1.5.1) of an FJSP instance in polynomial time, if the instance has not more than two jobs (see [10, Chapter 7]). However, no polynomial algorithm that solves the FJSP for instances with more than two jobs is known and solving such small instances possesses low practical value.

2.1.3 The Parallel Machine Scheduling Problem

The PMSP is a special case of the FJSP. Each operation in the PMSP can be scheduled to any machine, and each job has one operation, and therefore, none of the machines needs to consider at what time the operations on other machines are completed (see [11, 12]). This is generally not the case in the FJSP. Consider an operation that belongs to a job with several operations in the FJSP. In the case that the operation is the first in the given partial order, then none of the other operations can be scheduled before this operation has been completed.

2.1.4 The Batch Machine Scheduling Problem

Batches, and not operations, are scheduled in the BMSP. A batch is a set of operations that can be scheduled in a machine simultaneously (see [13]). The BMSP is of practical value, since machines that can process batches of operations will result in a non-higher total time during which the machines are occupied as compared to machines that can not handle batches. All jobs in the version of BMSP to be discussed in this chapter have only one operation.

2.1.5 Objective Functions

Objective functions are used in optimization problems to measure the quality of solutions. The goal of an optimization problem is to find a feasible solution such that the value of the objective function is minimized (or maximized). A feasible solution may be optimal for some objective function but considered bad for another objective function. A certain objective function can be chosen for several reasons: A solver can compute a solution for a problem faster, or the objective function is more relevant for practical applications of the problem than other objective functions. The goal is often too find an as "cheap" schedule as possible, or to find a schedule that completes all jobs as early as possible. This implies that the objective function for scheduling problems should be minimized. Two often used objective functions for scheduling problems are the *makespan* and the *weighted sum of completion times and tardiness*.

2.1.5.1 Minimizing Makespan

The makespan is defined as the time when the latest job is completed. While this objective function minimizes the time that the machines are needed, it has

several disadvantages. The order in which the operations starts may be such, that several machines could complete the operations they have been assigned to faster if operations were re-ordered, without affecting the minimum value of the makespan. It could also be the case that assigning some operation to another machine would cause some machines to complete their operations faster, without affecting the makespan. Moreover, a certain job may have a priority to be completed faster than others, which is ignored by this objective function.

2.1.5.2 Minimizing Weighted Averages of Completion Times and Tardiness

An objective function that minimizes the weighted averages of completion times and tardiness is often used for scheduling problems. When such an objective function is used, each job is given one or several weights. E.g., one weight applied to the completion time of the job and one weight applied to its tardiness (in case it is completed later than a given deadline). The advantage of this objective function is that jobs can be given priorities. However, this introduces a problem — it is non-trivial to decide what weights the jobs should be given, since it is hard to determine how a certain job is prioritized compared to others.

2.2 Scheduling Models and Methods

Different methods and models that tackle scheduling problems are presented in this section. These methods will either use *constraint programming* or *integer linear programs* to model the scheduling problems that has been discussed above. Two of the methods that will be described below use column generation, which is also the method that will be used in this project.

According to Rossi, Beek and Walsh ([14, Chapter 1]) "The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them." A problem can be modeled as a constraint program in many different ways, and depending of how it is modeled, the time for a solver to find a (optimal) solution may vary.

An integer linear program is a model in which a linear objective function should be minimized (or maximized) so that a set of linear constraints are satisfied. Column generation is a technique that can be used with a linear program to enhance the process of finding an optimal solution by ignoring a large set of the variables that are not part of an optimal solution.

2.2.1 Constraint Programming Applied to Scheduling Problems

The FJSP can be divided into two sub-problems, namely the *routing problem* and the *order assignment problem*. The routing problem is to select a suitable machine for each operation. When the routing problem has been solved, the order assignment problem can be solved, which is to assign an order of the operations on each machine.

The order assignment problem takes the solution of the routing problem as input. The solution of the order assignment problem is also a solution to the FJSP.

The division of the problem results in that only a sub-optimal solution of the original problem can be found when the sub-problems are solved. This is also the case when both of the two sub-problems are solved optimally by first solving the routing problem optimally and then solving the order assignment problem by using the optimal solution of the routing problem as input to the order assignment problem.

Constraint programming will be applied to solve the two sub-problems in this section. While constraint programming can be used to compute and verify optimal solutions for many problems, the approach that will be explain in section is a heuristic method.

The routing problem has a trivial solution in the JSP, since each operation can only be performed by one machine. When a valid routing assignment has been found for an FJSP, the remaining problem is in fact to solve a JSP, i.e, to find an order of the operations on each machine (see [15]).

Depending of how the FJSP is tackled, the problem may implicitly be divided into these two sub-problems. Brandimarte (see [16]) presents a constraint programming model that solves the FJSP by solving the two sub-problems independently with a *hierarchical tabu search* (HTS) method. In a HTS, a feasible *neighbour solution* is being searched for from a feasible solution that is already known. A HTS algorithm terminates when some given stop condition has been satisfied, e.g. a time-limit. A neighbour solution is obtained from another solution by performing a certain process that transforms the current feasible solution to another feasible solution. Brandimarte has defined four methods to obtain neighbour solutions. One of these methods is to swap the order of two operations that belongs to the same machine.

The difference between the HTS method and the *local search* (LS) method is that a neighbour solution that has a higher cost (assuming the problem is a minimization problem) is not ignored by the HLS method, which they are by LS method. The advantages with not ignoring neighbour solutions that are worse than the current solution is that one can escape from a *local minimum*. A local minimum is a solution for which all neighbours have higher costs, but the solution itself does not have be globally optimal. A problem with the HTS method is that a specific solution can be found several times. A tabu list is usually used to keep track of solutions that have been found, otherwise, the algorithm can get stuck in a cycle of already known solutions (see [17] for more about HTS).

2.2.2 Model Scheduling Problems as Integer Linear Programs

There are many approaches to model a scheduling problem as an integer linear program. How fast a scheduling problem can be solved when modeled as an integer linear program depends on how the constraints, the objective function, and the decision variables are defined.

One approach to define the decision variables of an integer linear problem for a scheduling problem is to define a binary variable for each possible order an operation

can be processed by a certain machine. The number of decision variables that are needed for each machine is exponential as a function of the number of operations that can be performed by that machine, since all permutations of the operations might be a valid order. For example, if there are five (5) operations that can be performed in a certain machine, then at most $5! = 120$ binary decision variables are required to model the possible operation orders. The decision variables that represent the order of the operations for a machine in a solution are equal to one, and otherwise equal to zero. Models that employ binary variables of these types belong to the Manne family (see [18, 19]). Another approach is to define the variables such that each binary variable equals one if a certain job starts in a certain machine at a certain time-step. A time-step is a discrete number that approximates the real time.

The number of time-steps that are needed to represent the problem exactly depends on the processing time of the operations. If the processing time of an operation takes a certain number of minutes, which can not be written in integer format in any unit bigger than minutes (as hours or quarters, etc.), then each time-step must represent at most one minute). If the total processing time is several hours, say ten, for example, then $10 \cdot 60 = 600$ time-steps are needed for the model. Furthermore, if a certain machine can process five (5) operations, then $5 \cdot 600$ decision variables are needed to represent all operations' possible starting times for the machine. A model that uses variables that represent job starting times is referred to as a time-indexed model (see [20, 21]).

Thörnblad [1] discovered that even if the time-indexed model results in more variables than the models in which variables represent orders, the time-indexed model can be solved more efficiently in practice. More information about modeling principles for scheduling problem using integer linear programs can be found in [1].

2.2.3 Column Generation for the Parallel Machine Scheduling Problem

Van Den Akker et al. [6] developed a column generation algorithm for the PMSP that generates *machine schedules*. A machine schedule is a list of operations that must be scheduled on the machine. The model is initiated by a few machine schedules to form a feasible solution. New machine schedules are then generated and added to the model until an optimal solution to the model has been found. The model is a linear program, and not an integer linear program, since the method column generation can not be applied on integer linear programs. The columns in their linear program correspond to machine schedules; they can be generated in polynomial time as a function of the input size of the problem instance by solving a shortest path problem in a graph. Van Den Akker's column generation algorithm and implementation was successful, since it could be used to solve instances that other algorithms and software could not solve. Algorithms then had difficulties to solve instances with 20 jobs and five machines, or 30 jobs and four machines, while Van Den Akker's column generation algorithm could solve such instances in 15 seconds (see [22, 6]). It was noted that the smaller the ratio of jobs and machines were, the larger problem instances could be solved. Van Den Akker et al. could solve instances with 50 jobs and three machines, and instances with 100 jobs if the number of machines were

ten or more. Van Den Akker's algorithm will be explained in more detail in Section 3.2.3.

2.2.4 Column Generation for the Batch Machine Scheduling Problem

Tang and Wang (see [23]) has developed a column-row-generation algorithm for the BMSP. This method is a bit more complex than the usual column generation method, since rows are generated as well. The rows in their model correspond to batches and the columns correspond to machine schedules of batches. The reason for generating rows (batches) too is that the number of batches are exponential as a function of the number of jobs, and it likely would take too long time to compute all batches. Therefore, their method is a *column-row-generation algorithm*, since they generate both rows of batches, and columns of schedules that are containing batches.

Before their results is discussed, the *gap* between two solutions is defined as the ratio of their respective objective values. The results of the benchmarks of their algorithm shows that it is very successful, since the average and maximum gaps between the optimal solution for their restricted master program and their integer restricted master program of 300 randomly generated instances were shown to be at most 1% and 2%, respectively (see Section 3.1.2 for the notation (restricted) master program). The restricted master program provides a lower bound for the integer restricted master program, and since the gaps between these two were very low for the instances solved in [23], one can draw the conclusion that their integer solution is near-optimal. Furthermore, as in [22], Tang and Wang found that the running time decreases as the number of machines increases.

3

Linear and Integer Optimization

An introduction to the theory that is needed to understand the models that are used in this project is given in this chapter. Section 3.1 introduces the concept of linear programming, the simplex method, which is an algorithm that finds and verifies optimal solutions for linear programs, and the idea of column generation that can be applied with Dantzig-Wolfe decomposition. Column generation is used with the simplex method to reduce the total search space needed to find an optimal solution.

Section 3.2 presents integer linear programming and methods to solve it, including the algorithm branch-and-bound (BaB), which finds and verifies optimal solutions for integer linear programs, and the branch-and-price (BaP) algorithm, which is a combination of column generation and the algorithm BaB, and Section 3.2.3 presents an example of how column generation can be applied without employing the Dantzig-Wolfe decomposition for an integer linear program.

3.1 Linear Programming

Linear programming is a field in optimization theory that is used for various applications, since many optimization problems can be formulated as linear programs (see [24, Chapter 7] for examples). A linear program (LP) consists of a linear objective function and a set of linear constraints. It can be formulated in a matrix format as to

minimize$_x$

$$z := \boldsymbol{c}^\top \boldsymbol{x}, \tag{3.1a}$$

subject to

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.1b}$$

$$\boldsymbol{x} \geq 0, \tag{3.1c}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ are the decision variables, $\boldsymbol{c} \in \mathbb{R}^n$ are the given objective weights for the decision variables, $\boldsymbol{b} \in \mathbb{R}^m$ is a constant vector and $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a constant matrix such that $\text{rank}(\boldsymbol{A}) = m$. The objective function is $\boldsymbol{c}^\top \boldsymbol{x}$, and each row in (3.1b) and (3.1c) defines constraints.

A feasible solution to the linear program (3.1) assigns a value to each decision variable such that all constraints are satisfied, and an optimal solution is a feasible solution that minimizes the objective function over the set of all feasible solutions. The points in the polyhedron $P = \{\, \boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq 0\}$ define the set of all feasible

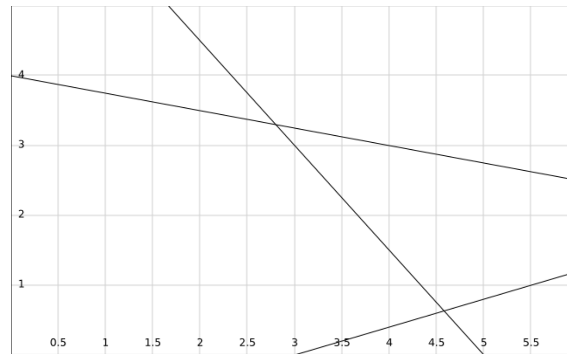


Figure 3.1: The bottom left area illustrates the polyhedron defined by the five linear constraints $x_1 + 4x_2 \leq 16$, $2x_1 - 5x_2 \leq 6$, $6x_1 + 4x_2 \leq 30$, $x_1 \geq 0$ and $x_2 \geq 0$. The points in the polyhedron is the set of all feasible solutions that the linear program (3.1) has. There are five extreme points in this polyhedron and for any objective function, at least one of these points is a minimizer.

solutions to the linear program (3.1). An example of such a polyhedron is illustrated in Figure 3.1. At least one of the extreme points in the polyhedron P is an optimal solution for the linear program (3.1) (see [25, Chapter 1]).

Every linear program, referred to as the primal problem, has a corresponding dual problem. The linear programming dual problem of the primal problem (3.1) is to

maximize$_y$

$$\boldsymbol{b}^\top \boldsymbol{y}, \tag{3.2a}$$

subject to

$$\boldsymbol{A}^\top \boldsymbol{y} \leq \boldsymbol{c}, \tag{3.2b}$$

where $\boldsymbol{y} \in \mathbb{R}^m$ are the dual variables. The dual problem can be used to find an upper bound on the optimal value of the primal problem and is utilized in the column generation method which will be presented later in this chapter.

3.1.1 The Simplex Method

This section presents a short summary of the simplex method (see [25, Chapter 1]). An optimal solution to the linear program (3.1) can be found and verified by iterating over all extreme points in the polyhedron P, by checking which point that results in the lowest value for the objective function. However, an exponential number of such points exist with respect to the number of decision variables; therefore it is not possible in practice to consider all of them for large instances. The simplex method tries to avoid checking all of them by only considering points that have a lower objective value than the value of the best known extreme point.

A point $\hat{\boldsymbol{x}}$ in the polyhedron P is an extreme point if at least $n - m$ variables in the point are equal to zero. These variables are the non-basic variables and the remaining variables are the basic variables. The LP (3.1) can then be rewritten as to

minimize$_{x_B, x_N}$

$$\boldsymbol{c}_{\boldsymbol{B}}^{\top}\boldsymbol{x}_{\boldsymbol{B}} + \boldsymbol{c}_{\boldsymbol{N}}^{\top}\boldsymbol{x}_{\boldsymbol{N}}, \tag{3.3a}$$

subject to

$$\boldsymbol{A}_{\boldsymbol{B}}\boldsymbol{x}_{\boldsymbol{B}} + \boldsymbol{A}_{\boldsymbol{N}}\boldsymbol{x}_{\boldsymbol{N}} = \boldsymbol{b}, \tag{3.3b}$$

$$\boldsymbol{x}_{\boldsymbol{B}}, \boldsymbol{x}_{\boldsymbol{N}} \geq 0, \tag{3.3c}$$

where $\boldsymbol{x}_{\boldsymbol{B}}$ is a vector of the basic variables in the point $\hat{\boldsymbol{x}}$, $\boldsymbol{x}_{\boldsymbol{N}}$ is a vector of the non-basic variables in the point $\hat{\boldsymbol{x}}$, $\boldsymbol{A}_{\boldsymbol{B}}, \boldsymbol{c}_{\boldsymbol{B}}$ and $\boldsymbol{A}_{\boldsymbol{N}}, \boldsymbol{c}_{\boldsymbol{N}}$ are the associated matrices and vectors for the variables $\boldsymbol{x}_{\boldsymbol{B}}$ and $\boldsymbol{x}_{\boldsymbol{N}}$, respectively. Multiplying both left and right hand sides of Equation (3.3b) with $\boldsymbol{A}_{\boldsymbol{B}}^{-1}$ results in

$$\boldsymbol{A}_{\boldsymbol{B}}^{-1}(\boldsymbol{A}_{\boldsymbol{B}}\boldsymbol{x}_{\boldsymbol{B}} + \boldsymbol{A}_{\boldsymbol{N}}\boldsymbol{x}_{\boldsymbol{N}}) = \underbrace{\begin{bmatrix} x_{B_1} & & & \tilde{A}_{N_{1,1}}x_{N_1} & \dots & \tilde{A}_{N_{1,n-m}}x_{N_{n-m}} \\ & \ddots & & \vdots & \dots & \\ & & x_{B_m} & \tilde{A}_{N_{m,1}}x_{N_1} & \dots & \tilde{A}_{N_{m,n-m}}x_{N_{n-m}} \end{bmatrix}}_{n} = \boldsymbol{A}_{\boldsymbol{B}}^{-1}\begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}, \tag{3.4}$$

where x_{B_i} is element i'th in the vector $\boldsymbol{x}_{\boldsymbol{B}}$, x_{A_j} is element j in the vector $\boldsymbol{x}_{\boldsymbol{A}}$ and $\tilde{A}_{N_{i,j}}$ is the element on row i and column j in the matrix $\tilde{\mathbf{A}}_N = \mathbf{A}_B^{-1}\mathbf{A}_N$ such that $1 \leq i \leq m$ and $1 \leq j \leq n - m$. In this format, the variables can be viewed as columns.

The point $\hat{\boldsymbol{x}}$ might be a non-optimal solution if there exists a non-basic variable x_{N_k} such that $c_{N_k}^{\top} < 0$, since the value of the objective function would decrease by $c_{N_k}^{\top}x_{N_k} < 0$ when $x_{N_k} > 0$, where $1 \leq k \leq m$. When this variable is assigned a positive value, the decision variables need to be assigned as follow:

$$\begin{bmatrix} x_{B_1} \\ \vdots \\ x_{B_m} \end{bmatrix} = \boldsymbol{A}_{\boldsymbol{B}}^{\boldsymbol{-1}} \begin{bmatrix} b_1 - A_{N_{k,1}}x_{N_k} \\ \vdots \\ b_m - A_{N_{k,m}}x_{N_k} \end{bmatrix}. \tag{3.5}$$

Otherwise, (3.3b) will not hold. The larger value the variable x_{N_k} is assigned, the lower is the value of $c_{N_k}^{\top}x_{N_k} < 0$. Therefore, x_{N_k} should be assigned the largest value such that no x_{B_j}, $1 \leq j \leq m$ is assigned a negative value. When x_{N_k} and x_{B_j} have been assigned their new values, an improved solution has been found. At least one x_{B_j} will then be equal to zero. In the improved solution, the variable x_{B_j} is a non-basic variable and the variable x_{N_k} is a basic variable. The improved solution is optimal if there does not exists a non-basic variable x_{N_k} such that $c_{N_k}^{\top} < 0$.

Spielman and Teng [26] have shown that the simplex method has an exponential running time in the worst case as a function of the number of decision variables, but its running time is polynomial in most cases.

3.1.2 Dantzig-Wolfe Decomposition and Column Generation for Linear Programs

A disadvantage with instances of linear programs with gigantic numbers of (non-basic) variables is that iterating over all of these variables, which is necessary to find

the variable (or column) with the highest reduced cost, takes very long time. This makes the simplex method slow, even when only a polynomial number of iterations with respect to the input size is needed to find and verify an optimal solution. The column generation method tackles this problem by considering a subset of the variables and generates variables with a negative reduced cost to the linear program in an iterative process. The optimal solution is found when no variable, or column, with a negative reduced cost can be generated. Most variables that exist are never included in the program in practise when column generation is used. Depending of how the linear program is formulated, it may have to be transformed to an equivalent program before column generation can be applied. The Dantzig-Wolfe decomposition is often used for this task.

The Dantzig-Wolfe decomposition transforms a linear program into an equivalent linear program that is referred to as the master problem (MP). A restricted master problem (RMP) and one or several sub problems (SP) are then defined. The RMP has the same set of constraints and objective function as the MP, but only a subset of all its variables. The SP is used to generate columns to the RMP until no column with a negative reduced cost can be generated. The SP requires the values of an optimal solution of the linear programming dual problem of the RMP to calculate new columns. When no column with a negative reduced cost can be found, the value of the objective function for an optimal solution of the current RMP equals that of the MP. The Dantzig-Wolfe decomposition technique can be applied to the linear program (3.1) by first dividing the constraints into two sets. The linear program (3.1) is equivalent to the program to

minimize$_x$

$$\boldsymbol{c}^\top \boldsymbol{x}, \tag{3.6a}$$

subject to

$$\boldsymbol{A_1 x} = \boldsymbol{b_1}, \tag{3.6b}$$

$$\boldsymbol{A_2 x} = \boldsymbol{b_2}, \tag{3.6c}$$

$$\boldsymbol{x} \geq 0, \tag{3.6d}$$

where $\boldsymbol{A_1} \in \mathbb{R}^{m_1 \times n}$, $\boldsymbol{A_2} \in \mathbb{R}^{m_2 \times n}$ $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \end{bmatrix}$, and $\boldsymbol{b} = \begin{bmatrix} \boldsymbol{b_1} \\ \boldsymbol{b_2} \end{bmatrix}$, where $\boldsymbol{b_1} \in \mathbb{R}^{m_1}$ and $\boldsymbol{b_2} \in \mathbb{R}^{m_2}$, such that $m_1 + m_2 = m$. Let us assume that the constraints (3.6b) are the non-complicated constraints in (3.6). The polyhedron $S = \{\, \boldsymbol{x} \mid \boldsymbol{A_1 x} \leq \boldsymbol{b_1}, \boldsymbol{x} \geq 0\}$ includes all the values the constraints (3.6b) allow the decision variables to be equal to. Any point $\hat{\boldsymbol{x}} \in S$ can be written as

$$\hat{\boldsymbol{x}} = \sum_{i=1}^{|\Omega|} \lambda_i \hat{x}_i \quad , \tag{3.7a}$$

where

$$\sum_{i=1}^{|\Omega|} \lambda_i = 1, \tag{3.7b}$$

$$\lambda_i \geq 0, \qquad i = 1, \ldots, |\Omega|, \tag{3.7c}$$

and each $\hat{x}_i \in \Omega$, where Ω is the set of the extreme points of the polyhedron S. The linear program (3.6) can therefore be written as to

$$\text{minimize}_\lambda \quad \sum_{i=1}^{|\Omega|} c^\top(\lambda_i \hat{x}_i), \tag{3.8a}$$

subject to

$$\boldsymbol{A_2} \sum_{i=1}^{|\Omega|} (\lambda_i \hat{x}_i) = \boldsymbol{b_2}, \tag{3.8b}$$

$$\sum_{i=1}^{|\Omega|} \lambda_i = 1, \tag{3.8c}$$

$$\lambda_i \geq 0, \qquad i = 1, \dots, |\Omega|, \tag{3.8d}$$

This problem is equivalent to the linear program (3.6) but has $m_2 + 1$ instead of $m_1 + m_2$ rows. The number of variables on the other hand is likely more than in the original formulation, since the number of columns is equal to the number of extreme points in the polyhedron S.

It might be possible to decompose the problem further if the decision variables in the linear program (3.6) can be divided into p disjoint parts such that none of the constraints in (3.6) depending on variables in several of these disjoint sets. An illustration of this can be seen in Figure 3.2. The notations can then be written such that $\boldsymbol{c}^\top = [\overrightarrow{\boldsymbol{c_1}}, \dots, \overrightarrow{\boldsymbol{c_p}}]$, $\boldsymbol{x}^\top = [\overrightarrow{\boldsymbol{x_1}}, \dots, \overrightarrow{\boldsymbol{x_p}}]$, $\boldsymbol{b}^\top = [\overrightarrow{\boldsymbol{d}}, \overrightarrow{\boldsymbol{b_1}}, \dots, \overrightarrow{\boldsymbol{b_p}}]$ and

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{D_1} & \boldsymbol{D_2} & \dots & \boldsymbol{D_p} \\ \boldsymbol{F_1} & & & \\ & \boldsymbol{F_2} & & \\ & & \ddots & \\ & & & \boldsymbol{F_p} \end{bmatrix}.$$

The linear program program (3.1) can then be written as to

$$\text{minimize}_x \quad \sum_{k=1}^{p} \overrightarrow{\boldsymbol{c_k}}^\top \overrightarrow{\boldsymbol{x_k}}, \tag{3.9a}$$

subject to

$$\sum_{k=1}^{p} \boldsymbol{D_k} \overrightarrow{\boldsymbol{x_k}} = \overrightarrow{\boldsymbol{d}} \quad , \tag{3.9b}$$

$$\boldsymbol{F_k} \overrightarrow{\boldsymbol{x_k}} = \overrightarrow{\boldsymbol{b_k}}, \qquad k = 1, \dots, p, \tag{3.9c}$$

$$x_{ki} \geq 0, \qquad i = 1, \dots, |\Omega_k|, k = 1, \dots, p, \tag{3.9d}$$

where the set Ω_k is defined as the extreme points in the polyhedron $S_k = \{\, \boldsymbol{x} \mid \boldsymbol{F_k} \leq \overrightarrow{\boldsymbol{b_k}}, \boldsymbol{x} \geq 0\}$ and x_{ki} is the i'th element in Ω_k for $1 \leq k \leq p$ and $1 \leq i \leq |\Omega_k|$. The linear program (3.1) can then be decomposed into the linear program that

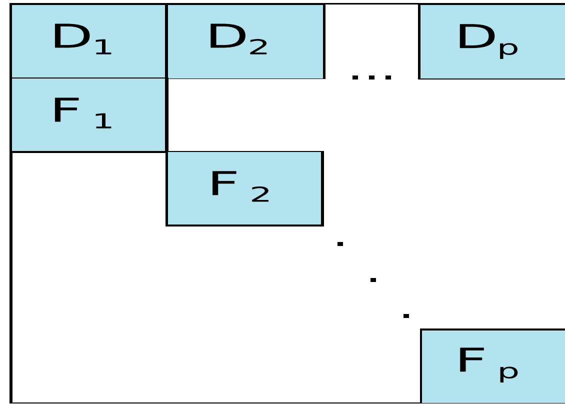


Figure 3.2: An illustration of how a linear program's variables can be divided into p disjoint sets.

minimize$_\lambda$

$$\sum_{k=1}^{p}\sum_{i=1}^{|\Omega_k|} c_{ki}^{\top}\left(\lambda_{ki}x_{ki}\right), \tag{3.10a}$$

subject to

$$\sum_{k=1}^{p} \boldsymbol{D_k}\left(\sum_{i=1}^{|\Omega_k|}\lambda_{ki}x_{ki}\right) = \overrightarrow{\boldsymbol{d}}, \tag{3.10b}$$

$$\sum_{i=1}^{|\Omega_k|}\lambda_{ki} = 1, \qquad k = 1\dots p, \tag{3.10c}$$

$$\lambda_{ki} \geq 0, \qquad i = 1\dots|\Omega_k|, k = 1\dots p, \tag{3.10d}$$

where c_{ki} is the i'th element in $\overrightarrow{\boldsymbol{c_k}}$ for $1 \leq k \leq p$ and $1 \leq i \leq |\Omega_k|$. This problem is referred to as MP (see the beginning of this section). The number of rows has been reduced from m, to the rank of the matrix $\boldsymbol{D} = [\boldsymbol{D_1}, \dots, \boldsymbol{D_p}]$, but the number of variables are many more than in (3.1). Most of these variables, or columns, will not be part of an optimal solution and is in that aspect not needed. Is it often computationally intractable to generate all of these variables. The RMP is therefore useful since it does not need all the variables. The RMP is to

minimize$_\lambda$

$$\sum_{k=1}^{p}\sum_{i=1}^{|\hat{\Omega}_k|} c_{ki}^{\top}\left(\lambda_{ki}x_{ki}\right), \tag{3.11a}$$

subject to

$$\sum_{k=1}^{p} \boldsymbol{D_k} \left(\sum_{i=1}^{|\hat{\Omega}_k|} \lambda_{ki} x_{ki} \right) = \overrightarrow{\boldsymbol{d}}, \tag{3.11b}$$

$$\sum_{i=1}^{|\hat{\Omega}_k|} \lambda_{ki} = 1, \qquad k = 1 \ldots p, \tag{3.11c}$$

$$\lambda_{ki} \geq 0, \qquad i = 1 \ldots |\hat{\Omega}_k|, k = 1 \ldots p, \tag{3.11d}$$

where $\hat{\Omega}_k \subset \Omega_k$, $1 \leq k \leq p$.

To find columns to add to the set $\hat{\Omega}_k$, the subproblem that corresponds to the k'th constraints (3.11b) must be solved. Let $\bar{\pi}$ and $\bar{\gamma}_k$, $k = 1, \ldots, p$ be the optimal values of the linear programming dual variables corresponding the constraints (3.11b) and (3.11c), respectively. The sub-problems are to generate new columns that will be added to the sets $\hat{\Omega}_k$. A column with a negative reduced cost can be found (if it exists) by solving, for each $k \in \{1, \ldots, p\}$, the problem

$$\hat{c}_k := \text{minimum}_{x_k} (c_k^\top - \bar{\pi}_k^\top D) x_k - \bar{\gamma}_k, \tag{3.12a}$$

subject to

$$F_k x_k = b_k, \tag{3.12b}$$

$$x_k \geq 0. \tag{3.12c}$$

The resulting column is denoted $(\bar{x}_1^\top, \ldots, \bar{x}_k^\top)^\top$. The column is included in the RMP if $c_k < 0$, since it then improves the current solution. The complete process of column generation is illustrated in the seven upper levels in Figure 3.3.

3.2 Integer Linear Programs

Integer linear programs (ILP; see [27]), are linear programs with integer decision variables, i.e, the decision variables can only be assigned integer values. Integer linear programming is an important field in optimization, since many real-world problems can be represented by ILPs. Unfortunately, the Simplex method can not be used to find solutions to ILPs. In fact, no algorithm that solves an ILP optimally with a polynomial running as a function of the number of decision variables is known, since ILPs are **NP**-hard.

An optimal solution to a linear program is an (or several) extreme-point(s) in the polyhedron P (see Section 3.1), but this (these) point(s) might be non-integer. A very naive idea to find a good integer solution is to first find the optimal solution in P for the integer relaxed ILP (i.e., the ILP without the integer requirements on the variables), and then round this point to an integer point. This idea has several problems; Let x be the optimal point for an integer relaxed ILP and $\hat{x}$ be the integer point that has been obtained by rounding the point x. The ratio $\frac{c^\top \hat{x}}{c^\top x}$ can be arbitrarily large. Even worse, the point $\hat{x}$ might not even be in the polyhedron P which means that the point is not a feasible solution, i.e, $\hat{x} \notin P$. The technique Branch and Bound can be used to find and verify optimal solutions for ILPs.

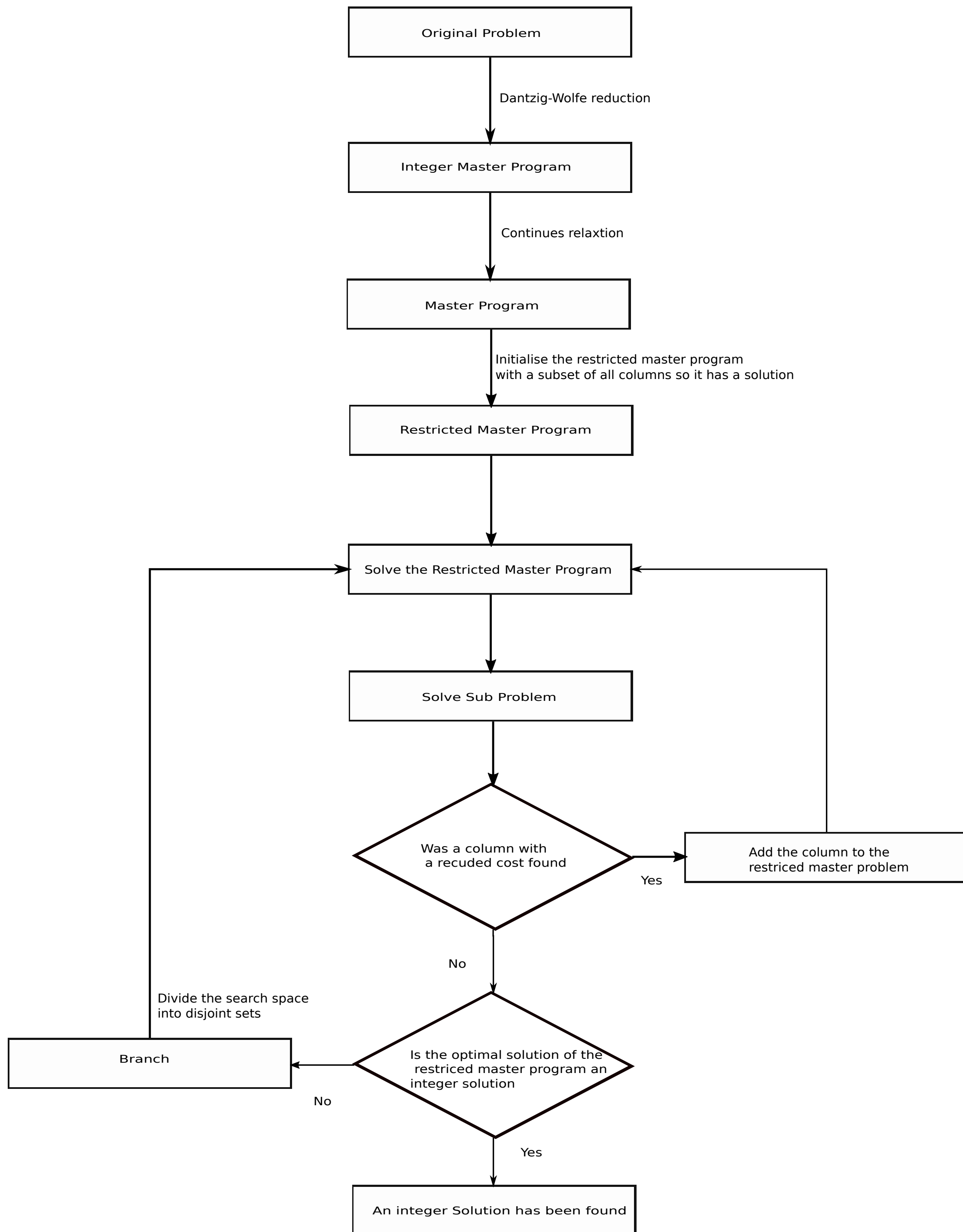


Figure 3.3: A Branch and Price algorithm that terminates when an integer solution has been found.

3.2.1 Branch and Bound

The algorithm Branch and Bound finds and verifies one, or several optimal solutions to ILPs. The algorithm uses branching, and the Simplex method (or some other algorithm that can find an optimal solution for LPs) to find an optimal integer solution.

The Branch and Bound algorithm searches for an optimal solution for the *LP relaxation* of the ILP. The LP relaxation of an ILP is the LP that is obtained by replacing the integer constraints in the LP by continuous constraints. For example, the (binary) integer constraint $x \in \{0, 1\}$ would be replaced by the continuous constraints $0 \leq x \leq 1$.

The Branch and Bound algorithm terminates if the optimal solution for the LP relaxation is an integer point (for the complete search space), since an optimal integer solution has then been found. Otherwise, the search space of the LP relaxation is cut into two disjoint branches such that no integer point is in the cut. The same process is repeated in each of the resulting branches, i.e, a solution is being searched for by the simplex method. The optimal integer solution must be in either of these two sets. This process is done recursively until the branches can be pruned, which can be done if the best LP solution in the branch is worse than the best known integer solution or if the best ILP solution that is found is worse than the best known ILP solution. This method is in practise better than the brute force, but visits all extreme points in the worst case (see [28]). A deeper introduction to Branch and Bound is found in [29].

3.2.2 Applying Column Generation to Integer Linear Programs Using Branch-and-Price

Column generation can be applied to ILPs by formulating an integer master problem (IMP) from the original problem. The MP is then defined by dropping the integer constraints from the IMP. The RMP and the SP can then be formulated from the MP. The integer constraints are added to the RMP once no column with a negative reduced cost can be found by the SP. The RMP with integer constraints is the integer restricted master problem (IRMP). The optimal solution to the IRMP is an integer solution, since the variables of the IRMP has integer restrictions. Unfortunately, the ratio of the values of the objective functions of the optimal solutions for the IRMP and the IMP might be larger than one. This ratio is referred to as the integrality gap.

The algorithm BaP, which is a combination of BaB and column generation, is presented in the next subsection, which can be used to find an optimal integer solution when column generation is applied to ILPs. An example of how column generation can be applied, without the use of the Dantzig-Wolfe decomposition, is presented as well.

3.2.2.1 Branch-and-Price

The algorithm BaP is a combination of BaB and column generation. As stated above, an integer gap may exists between the IMP and the IRMP. While the integer

gap might be very small, the optimal solution to the problem might be preferred if it can be found in a reasonable time. The integer gap can often be small when the values of the decision variables are large enough, which they often are when column generation is needed to be applied.

The algorithm Branch-and-Price applies column generation as described in Section 3.1.2, and divides the search space into branches as Branch and Bound does. The steps in Branch and Price can be seen in the two lower levels of Figure 3.3. The first step is to reformulate the original problem to an integer master problem (IMP), with a corresponding MP, RMP, and SP. The RMP and the SP are solved iteratively until no column with a negative reduced cost is found by the SP. The Integer Restricted Master Problem (IRMP) and the Integer Master Problem (IMP) are then solved. If the integrality gap is equal to one, the algorithm terminates. If not, the search space is divided into two or several branches. A new IMP, MP, RMP, and SP are then defined for each branch and the process is repeated until the integer solution has been found. Branch and Price can be studied in more detail in [30].

3.2.3 Applying Column Generation Without Using Dantzig-Wolfe Decomposition

Column generation can be applied directly to models without transforming them with the Dantzig-Wolfe decomposition if they are formulated in such fashion that each decision variable can be viewed as a column.

The Parallel Machine Scheduling Problem (PMSP) is one of the problems that can be formulated like that, and Van Den Akker [6] has, as written in Section 2.2.3, developed a column generation algorithm for the PMSP. In the PMSP, a set of jobs $\mathcal{J}$, a set of identical machines $\mathcal{K}$, and a process time p_j for each job $j \in \mathcal{J}$ are given. Each job can be processed by any machine, but no machine can process several jobs simultaneously. The goal is to schedule each job in a machine such that the makespan is minimized.

The model uses the concept of *machine schedules*. A machine schedule s is a vector of length $|\mathcal{J}|$ in which each element s_j states if job $j \in \mathcal{J}$ is included in the machine schedule or not, i.e., $s = [s_1, \dots, s_{\mathcal{J}}]$ where s_j is equal to one if job $j \in \mathcal{J}$ is included in the machine schedule, and otherwise zero. The set $\mathcal{S}$ is defined as the set of all possible machine schedules.

In the ILP, a decision variable x_s is defined for each machine schedule $s \in \mathcal{S}$ and the variable T measured the makespan value. The PMSP can now be modeled as the binary LP to

$$\text{minimize}_{x,T} \quad T, \tag{3.13a}$$

subject to

$$\sum_{s\in\mathcal{S}} x_s \leq |\mathcal{K}|, \tag{3.13b}$$

$$\sum_{s\in\mathcal{S}} s_j x_s = 1, \qquad j \in \mathcal{J}, \tag{3.13c}$$

$$\sum_{j\in\mathcal{J}} s_j x_s p_j \leq T, \qquad s \in \mathcal{S}, \tag{3.13d}$$

$$x_s \in \{0,1\}, \qquad s \in \mathcal{S}, \tag{3.13e}$$

where the objective function (3.13a) minimizes the makespan, the constraints (3.13b) make sure each machine uses at most one machine schedule, the constraints (3.13c) control that each job is processed in exactly one machine schedule that is used, and the constraints (3.13d) control the value of the makespan variable.

This program is defined as the Integer Master Problem (IMP). The Master Problem (MP) is the integer relaxed IMP, the integer Restricted Master Problem IRMP is defined as the IMP but for a subset $\hat{\mathcal{S}} \subset \mathcal{S}$ instead of the set $\mathcal{S}$, and the Restricted Master Problem (RMP) is the integer relaxed IRMP. A sub-problem that can generate any machine schedule $s \in S$ can then be defined as a pricing problem using the optimal values of the dual variables of the RMP (see [6, Chapter 2]).

4

An Integer Linear Optimization Model of GKN's Scheduling Problem

This chapter presents the constraints of the special jobs in Section 4.1, and the ILP (hereafter referred to as the *scheduling model*) that models the scheduling problem that will be described in detail in Section 4.2. A model that can be used to decrease the time span that the scheduling model uses is presented in Section 4.3.

4.1 The Constraints Defining the Special Jobs

The constraints for the special jobs (described in Section 1.2.2) are presented in this section.

A solder job j^{sold} can be heat treated in a furnace at a certain time step if at least one of the following two conditions hold:

1. The previous job scheduled in the furnace was a bake-out job, j^{bo}.
2. The previous job scheduled in the furnace was a solder job with a higher maximum temperature than the solder job, j^{sold}.

A titanium job, j_1^{tit} can only be scheduled in a furnace at a certain time step if at least one of the following two conditions hold:

3. The previous job scheduled in the furnace was a vacuum test job, j_1^{vac}, and the job scheduled before j_1^{vac} was a bake-out job, j^{bo}.
4. The previous scheduled job in the furnace was a vacuum test job j_1^{vac}, the job scheduled before j_1^{vac} was a titanium job j_2^{tit}, and if the special sequence of maintenance jobs and titanium job is never longer than five, hence, the longest allowed special sequence of titanium jobs is defined by: j^{bo}, j_2^{vac}, j_2^{tit}, j_1^{vac}, j_1^{tit}.

4.2 The Scheduling Model

The scheduling model is a modification of the ILP Thörnblad [1] formulated for solving GKN's scheduling problem. The task is to schedule a set of given jobs $\mathcal{J} = \{1, \dots, n\}$ in a given set of furnaces $\mathcal{K} = \{1, \dots m\}$ in a given time-range, which is divided into a set of time steps $\mathcal{T} = \{1, \dots, T\}$ so that some given constraint are satisfied. Each job $j \in \mathcal{J}$ can be scheduled in a subset of the furnaces, $\mathcal{M}_j \subseteq \mathcal{K}$. Once a job $j \in \mathcal{J}$ has been scheduled in furnace $k \in \mathcal{M}_j$, furnace k is occupied

until the job is complete, which takes p_{jk} time steps. No job can be scheduled in a furnace that is occupied.

A job $j \in \mathcal{J}$ can neither be scheduled before time step $r_j \in \mathcal{T}$ nor after time step $(T - \delta_j) \in \mathcal{T}$, and job $j \in \mathcal{J}$ has a due date (time) $d_j \in \mathcal{T}$, at which it is supposed to be completed. Furthermore, furnace $k \in \mathcal{K}$ can not be used before time step $a_k \in \mathcal{T}$.

For each job $j \in \mathcal{J}$, two weights are given; v_j and w_j. The weight v_j is assigned to the completion time of the job and the weight w_j is assigned to the tardines of the job and which is applied if the job is completed later than time step $d_j \in \mathcal{T}$. The notation $(u + p_{jk} - d_j)_+ := \max\{u + p_{jk} - d_j, 0\}$ denotes the number of time steps after its deadline that the job is completed.

A subset of the given jobs are either solder jobs or titanium jobs. These jobs are referred to as special jobs. A special job $j \in \mathcal{J}$ can only be scheduled in a furnace if the requirements given in the list in Section 4.1 are satisfied. To deal with this problem, the set $\mathcal{I}$ is defined as the set of all special sequences. A special sequence $i \in \mathcal{I}$ contains the special jobs $\mathcal{J}_i^{\text{corr}} \in \mathcal{J}$ and the corresponding maintenance jobs that are required for the special sequence (see Section 4.1). When a special sequence is scheduled, the special jobs and the maintenance jobs are scheduled in the order they are given in the sequence. Once a special sequence is scheduled in a furnace $k \in \mathcal{M}_i^{\text{spec}}$, all special jobs and maintenance jobs must be scheduled and completed in furnace k before anything else is scheduled in that furnace, where $\mathcal{M}_i^{\text{spec}} \subseteq \mathcal{K}$ is the set of all furnaces in which special sequence $i \in \mathcal{I}$ can be scheduled.

The set $\mathcal{P}$ corresponds to the families of all special sequences that together include each special job once. The special sequences that are included in the family that corresponds to $p \in \mathcal{P}$ are denoted $\mathcal{I}_p$. For each $\mathcal{I}_{p \in \mathcal{P}}$, $|\bigcup_{i \in \mathcal{I}_p} \mathcal{J}_i^{\text{corr}}|$ is equal to the number of special jobs that exist in the instance, and $|\mathcal{J}_{i_1}^{\text{corr}} \cap \mathcal{J}_{i_2}^{\text{corr}}| = 0$ for each $i_1, i_2 \in \mathcal{I}_{p \in \mathcal{P}}$. The parameter $s_{ik}^{\text{spec}} \in \mathcal{T}$ denotes the first time step special sequence i can start in furnace $k \in \mathcal{M}_i^{\text{spec}}$ and the parameter $\delta_{ik}^{\text{spec}} \in \mathcal{T}$ denotes the last time step special sequence i can start in furnace k.

For each job $j \in \mathcal{J}$, furnace $k \in \mathcal{M}_j$, and time step $u \in \mathcal{T}$, a binary variable x_{jku} is defined to be equal to one if and only if job j starts in furnace k at time step u, and zero otherwise. Analogously, the variable x_{iku}^{spec} is equal to one if and only if sequence $i \in \mathcal{I}$ starts in furnace $k \in \mathcal{M}_i^{\text{spec}}$ at time step $u \in \mathcal{T}$, and zero otherwise. A binary variable y_p is defined for each $p \in \mathcal{P}$, which is equal to one if and only if the family of special sequences p is scheduled, and zero otherwise. A summary of all parameters, sets, and variables used in the model is shown in Table 4.1. The objective of the scheduling model is to

Sets and families	Description
$\mathcal{J}$	the set of all jobs
$\mathcal{J}^{\text{maint}} \subseteq \mathcal{J}$	the set of all maintenance jobs
$\mathcal{I}$	the set of all special sequences
$\mathcal{P}$	the set of all families of special sequences that includes all job exactly once
$\mathcal{I}_p \subseteq \mathcal{I}$	the special sequences that are included in the family $p \in \mathcal{P}$
$\mathcal{J}_i^{\text{corr}} \subseteq \mathcal{J}$	the set of all jobs in special sequence $i \in \mathcal{I}$
$\mathcal{K}$	the set of all furnaces
$\mathcal{M}_j \subseteq \mathcal{K}$	the set of all furnaces in which job $j \in \mathcal{J}$ can be processed
$\mathcal{M}_i^{\text{spec}} \subseteq \mathcal{K}$	the set of all furnaces in which special sequence $i \in \mathcal{I}$ can be processed
$\mathcal{T}$	the ordered set of all discrete time steps, such that $\mathcal{T} = \{1, 2, \ldots, T\}$
Variables	
x_{jku}	$= 1$ if job $j \in \mathcal{J}$ starts at time step $u \in \mathcal{T}$ in furnace $k \in \mathcal{K}$, $= 0$ otherwise
x_{iku}^{spec}	$= 1$ if special sequence $i \in \mathcal{I}$ starts at time step $u \in \mathcal{T}$ in furnace $k \in \mathcal{K}$, $= 0$ otherwise
y_p	$= 1$ if the family $p \in \mathcal{P}$ of special sequences is scheduled, $= 0$ otherwise
Parameters	
v_j	the completion weigh of job $j \in \mathcal{J}$
w_j	the penalty weigh for job $j \in \mathcal{J}$
d_j	the time step when job $j \in \mathcal{J}$ should be completed
δ_j	the last time step when job $j \in \mathcal{J}$ can be scheduled
p_{jk}	the processing time for job $j \in \mathcal{J}$ in furnace $k \in \mathcal{K}$
a_k	the first time step when furnace $k \in \mathcal{K}$ is available
s_{ik}^{spec}	the first time step when special sequence $i \in \mathcal{I}$ can start in furnace $k \in \mathcal{M}_i^{\text{spec}}$
r_j	the first possible time step to start job $j \in \mathcal{J}$
$\delta_{ik}^{\text{spec}}$	the last time step when special sequence $i \in \mathcal{I}$ can start in furnace $k \in \mathcal{M}_i^{\text{spec}}$

Table 4.1: A summary of the sets, variables, parameters, and constants that are used in the models in Chapters 4 and 5.

$$
\begin{aligned}
&\text{minimize}_{x,x^{\text{spec}},y} \\
&\qquad \sum_{j\in\mathcal{J}\setminus\mathcal{J}^{\text{maint}}} \sum_{k\in\mathcal{M}_j} \sum_{u\in\mathcal{T}} \Big[v_j(u+p_{jk}) + w_j(u+p_{jk}-d_j)_+\Big] x_{jku}, && && (4.1a)\\
&\text{subject to} \\
&\sum_{k\in\mathcal{M}_j} \sum_{u\in\mathcal{T}} x_{jku} = 1, && j\in\mathcal{J}\setminus\mathcal{J}^{\text{maint}}, && (4.1b)\\
&\sum_{k\in\mathcal{K}/\mathcal{M}_j} \sum_{u\in\mathcal{T}} x_{jku} = 0, && j\in\mathcal{J}, && (4.1c)\\
&\sum_{j\in\mathcal{J}} \sum_{\mu=(u-p_{jk}+1)_+}^{u} x_{jk\mu} \le 1, && k\in\mathcal{K}, u\in\mathcal{T}, && (4.1d)\\
&\sum_{p\in\mathcal{P}} y_p = 1, && && (4.1e)\\
&\sum_{k\in\mathcal{M}_i^{\text{spec}}} \sum_{u\in\mathcal{T}} x_{iku}^{\text{spec}} \ge y_p, && i\in\mathcal{I}_p, p\in\mathcal{P}, && (4.1f)\\
&x_{jk\alpha_\beta} \ge x_{iku}^{\text{spec}}, && j\in\mathcal{J}_i^{\text{corr}}, u\in\mathcal{T}, && (4.1g)\\
& && k\in\mathcal{M}_i^{\text{spec}}, i\in\mathcal{I}, \\
&x_{jku} = 0, && u\in\{0,\ldots,\max\{r_j,a_k\}-1\}, && (4.1h)\\
& && k\in\mathcal{M}_j, j\in\mathcal{J}, \\
&x_{iku}^{\text{spec}} = 0, && u\in\{0,\ldots,s_{ik}^{\text{spec}}-1\}, && (4.1i)\\
& && k\in\mathcal{M}_i^{\text{spec}}, i\in\mathcal{I} \\
&x_{jku} = 0, && u\in\{T-\delta_j+1,\ldots,T-1\}, && (4.1j)\\
& && k\in\mathcal{M}_j, j\in\mathcal{J}, \\
&x_{iku}^{\text{spec}} = 0, && u\in\{T-\delta_{ik}^{\text{spec}}+1,\ldots,T-1\}, && (4.1k)\\
& && k\in\mathcal{M}_i^{\text{spec}}, i\in\mathcal{I}, \\
&x_{jku} \in\{0,1\}, && k\in\mathcal{K}, u\in\mathcal{T}, j\in\mathcal{J}, && (4.1l)\\
&y_p \in\{0,1\}, && p\in\mathcal{P}, && (4.1m)\\
&x_{iku}^{\text{spec}} \in\{0,1\}, && u\in\mathcal{T}, k\in\mathcal{M}_i, i\in\mathcal{I}. && (4.1n)
\end{aligned}
$$

The objective function (4.1a) is to minimize the weighted average finishing times of the jobs and the tardiness weight.

The constraints (4.1b) make sure that each job that is not a maintenance job starts exactly once in a furnace in which the job can be heat treated, the constraints (4.1c) control that no job starts in a furnace in which the job can not be heat treated, and the constraints (4.1d) control that no job interrupts any other job that is being processed.

The constraints (4.1e) regulate that only one family of special sequences is chosen, and the constraints (4.1f) control that each special sequence in the family of special sequences that is chosen starts at some time-step, and the constraints (4.1g) make sure the jobs in each special sequence in the chosen family starts. The parameter α_β is not explained for confidentiality reasons.

The constraints (4.1h) make sure that no job starts before it has arrived or in a furnace that is not ready, the constraints (4.1i) control that no special sequence is scheduled before it is allowed to be scheduled. Constraints (4.1j) and (4.1k) make sure that all jobs and special sequences that start are also completed before the last time-step, and constraints (4.1l)–(4.1n) are the binary constraints.

4.2.1 Disadvantages with the Scheduling Model

GKN has set a time limit, before which a schedule must be computed by the scheduling model. Usually, an optimal schedule can not be found and verified with the scheduling model, before this time limit has passed. This is mainly due to the large number of time steps that are needed to represent the real time accurately enough, and the number of families of special sequences that exists. Currently, the best schedule that is found within the time limit is used for production. The scheduling model does not consider all time steps to represent the real time accurate, and not all families of special sequences, since this would slow down the solver that solves the scheduling model too much. However, most families are considered, although not all of them.

The optimal schedule that the scheduling model finds when only a subset of the variables are used, is therefore a heuristic solution to the problem, and the lower bounds on the objective value of the optimal solution to the problem is unknown. A model that is used to decrease the number of time steps that is needed to represent an accurate real time is discussed in the next section. The remaining part of this section discusses the complications with the special sequences.

Consider the ordered set $\mathcal{S} := \langle r^{\mathrm{bo}}, j_1, j_2, j_3 \rangle$ of jobs, where r^{bo} denotes the mandatory bake-out job before scheduling any solder job $j_i \in \mathcal{J}$ (see condition 1 in Section 4.1). We define $t_i^{\max}$ to be the maximal temperature for solder job j_i. Then, for the set S to be a valid special sequence of solder jobs, the inequalities $t_3^{\max} \leq t_2^{\max} \leq t_1^{\max}$ must hold (see condition 2 in Section 4.1). If the set $S \subset \mathcal{I}$, then all ordered subsets of S are in the family $\mathcal{I}$, i.e., $\{\langle r^{\mathrm{bo}}, j_1 \rangle, \langle r^{\mathrm{bo}}, j_2 \rangle, \langle r^{\mathrm{bo}}, j_3 \rangle, \langle r^{\mathrm{bo}}, j_1, j_2 \rangle, \langle r^{\mathrm{bo}}, j_1, j_3 \rangle\}$, $\langle r^{\mathrm{bo}}, j_2, j_3 \rangle\} \subset \mathcal{I}$. If it holds that $t_3^{\max} = t_2^{\max} = t_1^{\max}$, then even more special sequences are valid. The number of valid special sequences of solder jobs in an instance with m solder jobs is equal to or less than 2^m.

A special sequence of titanium jobs can never be longer than five (see condition 4 in Section 4.1), but there is no temperature restriction on the order of these jobs, so the number of valid special sequences of titanium jobs in an instance is $\frac{n!}{(n-1)!}$ + $\frac{n!}{(n-2)!} = n^2$, where n is the number of titanium jobs in the instance, since there are at most two titanium jobs in a special sequence of titanium jobs, the other jobs in such special sequence are maintenance jobs. As a consequence of the fact that only a subset of the set $\mathcal{I}$ of all special sequences are used by the scheduling model, subfamilies of the family $\mathcal{P}$ are used by the scheduling model.

4.3 Shortening the Time Span of the Model

The number of time steps that are used by the scheduling model is calculated such that all jobs in the instance can be processed. The time needed to processes all jobs in an instance can often take 24 hours or more. The processing time of a job is measured in minutes. To represent all minutes in one day with discrete time steps, $24 \cdot 60 = 1440$ time steps are needed. This results in very many variables in the scheduling model, since several decision variables exist for each time step.

Thörnblad [1] has created a model (referred to as TIM — Time Iterative Model in this thesis) that calculates how long the *time span* needs to be at most such that a feasible solution exists when a time step represent fifteen minutes. The time span for an instance is defined as the number of time steps in the instance, i.e., $\mathcal{T}$. No solution can exists if the time span is smaller than the makespan. The job that requires most time to process takes 14 hours, and the shortest takes around one hour and 40 minutes. A Unified Modeling Language diagram of TIM is shown in Figure 4.1. The department considers schedules that are found when a time step represents fifteen minutes to be usable, but solving the scheduling model when the time span is 24 hours results in $7 \cdot 24 \cdot 4 = 762$ time steps, which is still too many for a solution to be found in a reasonable time.

The idea that TIM is based on is that the jobs in the scheduling model can often be scheduled when the time span is less than first calculated. The TIM starts by solving the scheduling model when each time step represents 12 hours. This can be done fast since the number of time steps then are quite few compared to how many they are when each time step represents 15 minutes. The schedule that is obtained from a solution to that scheduling model is very "airy" — most jobs take more than 12 hours to process, but less than 20 hours, but since one time step represent 12 hours, 14 hours, for example, is represented by 2 time steps. Let us refer to the schedule computed using 12 time steps as the *original schedule*.

The original schedule is then modified as if each time step represents 4 hours, which results in the so-called *modified schedule*. If a job started at time step 2 in the original schedule, it starts not later than at time step 6 in the modified schedule, since 6 time steps represent 24 hours in this schedule. However, a job is scheduled earlier if that is possible, with respect to the real processing times and a 4-hour discretization. That might be possible if there is another job that starts before this job in the same furnace. For example, if that job takes 4 hours to process and starts at time step 0, then it is completed at time step 1, which represents hour 4. Then the job considered can be scheduled at time step 1, which is much better (with respect to the objective function) than to schedule it at time step 6. Thörnblad has named this technique to *squeeze* the schedule (see [1]).

The real time corresponding to the time step that is equal to the makespan in the modified schedule is lower than the real time that the corresponding time step in the original schedule represents.

Let us give an example of this. Say that the latest job that was completed in the original schedule was completed at time step 2, which represents hour $2 \cdot 12 = 24$. In the modified schedule, assume that the time step when the latest job was completed was 4, which represents hour $4 \cdot 4 = 16$. We then know that the time span required

is not greater than 16 hours, which is less than 24 hours.

A new scheduling model is then created and solved, with the time step representing 4 hours, with the new calculated time span (24 in our example). Then $24/4 = 6$ time steps are needed. If a time step represents 4 hours and the time span is 1 week, then $7 \cdot 24/4 = 42$ time steps is needed. The process described above is repeated five times, and for each iteration, a time step represents fewer hours, down to 0.25 hours, i.e., 15 minutes, in the last iteration.

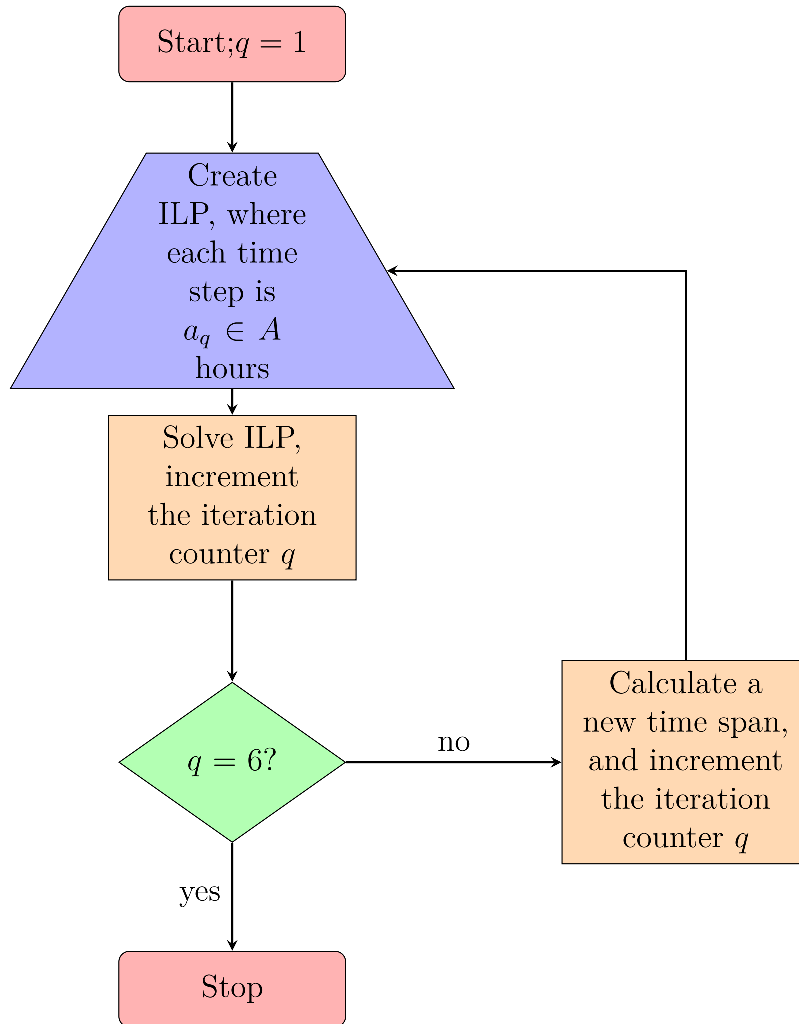


Figure 4.1: The time-iteration model (TIM). The scheduling problem is solved six times. In each iteration, a time step represents fewer hours than in the previous one, and the time span is reduced. The first time the ILP is solved, a time step represent 12 hours. The ordered set $A = \{12, 4, 2, 1, 0.5, 0.25\}$ contains the number of hours a time step can represent.

5

A Column Generation Model for Generating Special Sequences

A column generation method that generates special sequences (SpSs) that the scheduling model uses is introduced in this chapter. An optimal schedule can not be guaranteed to be found using this approach, since only a subset of all variables are used by the scheduling model, as a consequence of the fact that only a subset of the SpSs families are considered by the scheduling model. It is practically impossible to know which family that is part of an optimal schedule without defining all the families in the scheduling model. The goal of this approach is to find one or several families of SpSs that cover each special job exactly once, and that are part of an optimal, or near optimal schedule. This approach will hopefully find as good as or better schedules than the current approach, and be faster, since the fewer families of SpSs that are considered by the scheduling model, the faster a good-quality schedule can be computed. GKN currently generates almost all families that exist for the scheduling model, which can be more than 150 000, even when the number of special jobs is just ten, which makes the scheduling model require a lot of computing time to find a good schedule.

The purpose of the Integer Master Problem (IMP) to be defined in Section 5.1 is to choose a family of SpSs, so that the SpSs in that family together include each special job exactly once and such that the family is likely to be part of an (near) optimal schedule. Since we want to avoid generating the complete set of SpSs, we define the (Integer) Restriced Master Problem ((I)RMP) in Section 5.2, that contains a subset of the family of SpSs, the dual linear program of the RMP in Section 5.3, which will be used by the subproblem (SP), to be defined in Section 5.4 that will generate SpSs as columns to the RMP until the objective value of optimal solutions to the RMP and the MP are equal. Then, the IRMP will be solved and the family that is selected will be used by the scheduling model. See Section 3.2.3 for how the MP and RMP are obtained from the IMP and the IRMP, respectively.

5.1 The Integer Master Problem

The purpose of the IMP (see Chapter 2 for IMP) is to select a family of the SpSs, which corresponds to columns, such that each special job is included in exactly one of the SpSs in the family, and so that the SpSs in the family are likely to be part of an optimal or near–optimal schedule.

Some notations that will be used in this chapter and that was not used in Chapter

Sets and families	Description
$\mathcal{J}^{\text{spec}}$	the set of all special jobs
$\tilde{\mathcal{I}} \subseteq \mathcal{I}$	the set of the SPSs that is considered by the (I)(R)MP
Variables	
γ_i	$= 1$ if SpS $i \in \mathcal{I}$ is selected by the (I)(R)MP, 0 otherwise
π_j	the corresponding dual variable for $j \in \mathcal{J}^{\text{spec}}$ for constraint (5.4b)
Parameters	
g_i	the best possible furnace for SpS $i \in \mathcal{I}$
c_i	the cost of including the SpS i in solution for the (I)(R)MP
$\mathbf{A}$	is a matrix, which element $A_{ij} = 1$ if $j \in \mathcal{J}_i^{\text{spec}}$ for SpS $i \in \mathcal{I}$, 0 otherwise
Constants	
a	constant used as a penalty for each SpS that is selected by the (I)(R)MP

Table 5.1: A summary of the sets, variables, parameters, and the constant used in Chapter 5 but that is not included in Table 4.1.

4 will now be presented. The new notations can be overviewed in Table 5.1. There is one decision variable, γ_i, for each SpS $i \in \mathcal{I}$ in the IMP. The variable γ_i is equal to one if SpS i is included in the family chosen by the IMP, and zero otherwise, where $i \in \mathcal{I}$. The set $\mathcal{J}^{\text{spec}}$ is the set of all special jobs and the element A_{ij} in the binary matrix $\mathbf{A}$ is equal to one if the SpS i contains the special job $j \in \mathcal{J}^{\text{spec}}$, and zero otherwise. For each decision variable γ_i, a cost element c_i is defined as

$$c_i := a + \sum_{j \in \mathcal{J}_i^{\text{corr}}} p_{j,g_i}, i \in \mathcal{I}, \tag{5.1}$$

where the constant $a > 0$ is a penalty cost for each SpS that is chosen by the IMP and g_i is the index of the furnace that minimize the total processing time of all the jobs in the SpS, among of all furnaces $\mathcal{M}_i^{\text{spec}}$, i.e., g_i is defined by

$$g_i := \operatorname{argmin}_{k \in \mathcal{M}_i^{\text{spec}}} \left\{ \sum_{j \in \mathcal{J}_i^{\text{corr}}} p_{jk} \right\}, \quad i \in \mathcal{I}. \tag{5.2}$$

The IMP is then stated as to

minimize$_y$

$$\sum_{i \in \mathcal{I}} \gamma_i c_i, \tag{5.3a}$$

subject to

$$\sum_{i \in \mathcal{I}} \gamma_i A_{ij} = 1, \qquad j \in \mathcal{J}^{\text{spec}}, \tag{5.3b}$$

$$\gamma_i \in \{0, 1\}, \qquad i \in \mathcal{I}. \tag{5.3c}$$

The cost element c_i is dependent of the parameter p_{jk}, which is the processing time for job $j \in \mathcal{J}_i^{\text{corr}}$ and $k \in \mathcal{M}_i^{\text{spec}}$ (see Table 4.1), where $i \in \mathcal{I}$. The value of the parameter p_{jk} depends on how many minutes a time-step represents, which might result in that the IMP finds different solutions for different time-discretizations. It is more likely that the optimal schedule is using the SpSs choosen by the IMP if the parameters p_{jk} are precise. Therefore, each time-step represents 0.25 hours, i.e., 15 minutes during the column generation process (which is performed before the scheduling model is used). The cost element c_i is also dependent on the parameter a, which is used to adjust how many SpS that is prefered to select by the IMP, where $i \in \mathcal{I}$. The larger the value of the parameter a, the fewer SpSs will be selected, i.e., fewer but longer SpSs will be selected.

5.2 The (Integer) Restricted Master Problem

We now define the IRMP, which contains only a subset $\tilde{\mathcal{I}} \subseteq \mathcal{I}$ of all SpSs. The set $\tilde{\mathcal{I}}$ is initialized with all SpSs with one special job from start. The task of the IRMP is to

minimize$_y$

$$\sum_{i \in \tilde{\mathcal{I}}} \gamma_i c_i, \tag{5.4a}$$

subject to

$$\sum_{i \in \tilde{\mathcal{I}}} \gamma_i A_{ij} = 1, \qquad j \in \mathcal{J}^{\text{spec}}, \tag{5.4b}$$

$$\gamma_i \in \{0, 1\}, \qquad i \in \tilde{\mathcal{I}}. \tag{5.4c}$$

The RMP is defined as the IRMP with the exception that the variables of the RMP are restricted to non-negative values instead of binary values.

5.3 The Linear Programming Dual Problem of the Restricted Master Problem

The linear programming dual problem of the RMP is defined in this section. The optimal solution of this program is needed by the SP. The linear programming dual of the RMP is to

maximize$_\pi$

$$\sum_{j \in \mathcal{J}^{\text{spec}}} \pi_j, \tag{5.5a}$$

subject to

$$\sum_{j \in \mathcal{J}} \pi_j A_{ij} \leq c_i, \qquad i \in \tilde{\mathcal{I}}, \tag{5.5b}$$

where each variable π_j corresponds to the constraint (5.4b) for special job $j \in \mathcal{J}^{\text{spec}}$ in the RMP.

5.4 The Subproblem

The purpose of the SP is to generate SpSs that will be included to the set $\tilde{\mathcal{I}}$, which is used by the (I)RMP. The structure of the SpSs of solder jobs and titanium jobs are different, but the task of finding SpSs for either kind of special jobs can be reduced to a *shortest path problem* in a graph defined by the special jobs, the values of the dual variables for an optimal solution for the linear programming dual problem of the RMP and the process times of the special jobs. A path will then correspond to a column, and the total cost of the path will correspond to the reduced cost of the column (see Section 3.1.2 for reduced cost) .

The shortest path problem for a weighted graph $G = \langle V, E\rangle$ can be solved in $\mathcal{O}(|V| \cdot |E|)$ time with Bellman–Ford's algorithm ((see [3, Chapter 4]), where V is the set of vertices and E is the set of edges. Dijkstra's algorithm (see [3, Chapter 4]) has a lower time-complexity than Bellman-Ford's algorithm, but can only be used for graphs with none-negative edge lengths. Dijkstra's algorithm can therefore not be used to solve the SP, since the lengths of the edges of the graphs that will be constructed for the sub-problem are functions of an optimal solution of the RMP (5.5), which can contain negative variable values. The fact that it is possible to find new columns for the RMP in polynomial time is important, since the SPs will be solved many times, and hence, the SPs need to be solved in an efficient manner.

The columns generated by the SP that will be included in the set $\tilde{\mathcal{I}}$ are the ones that have a negative reduced cost. The reduced cost of a generated SpS i for the furnace $g_i \in \mathcal{M}_i^{\text{spec}}$ is equal to $c_i - \pi_j = a + \sum_{j \in \mathcal{J}_i^{\text{corr}}} p_{j,g_i} - \pi_j$, where the value of π_j is such that it is part of an optimal solution for (5.5).

We will first present a shortest path model that can be used to find SpSs of solder jobs and then illustrate how the problem of finding SpSs of titanium jobs can be modeled as a shortest path problem as well. Since an SpS can not contain both titanium and solder jobs, the SPs can be partitioned in this manner, without losing the possibility of generating any SpSs that exist.

5.4.1 The Subproblem of Generating SpSs of Solder Jobs

The task of finding SpSs of solder jobs can be reduced to a path finding problem, for several Directed Acyclic Graphs (DAGs). Each DAG contains one node for each solder job, one start node r^{bo}, and one destination node d. Since the objective function of the RMP uses the parameters p_{jk}, the objective value of a variable in the RMP that corresponds to an SpS is dependent on in which furnace the SpS is processed. Therefore, one DAG per furnace $k \in \mathcal{K}$ is constructed to find the SpS of solder jobs that corresponds to the column of the RMP with the lowest possible reduced cost. For a specific furnace, only special jobs that are suitable for that furnace are inserted in the graph that is created.

Each of these DAGs contains one edge from the start node r^{bo} to each node in the graph that corresponds to a solder job. Furthermore, from each node that corresponds to a solder job $j_\wp$, there is an edge to the destination node d and an edge to each node that corresponds to a solder node j_ℓ for all $\ell \geq \wp$. The weight on the outgoing edges for the start node is set to a, and the weight that is outgoing from

each job node j_ℓ is set to $p_{j_\ell k} - \pi_{j_\ell}$. Since $\ell > \wp$, the solder job j_ℓ must have a lower maximal temperature than the solder job $j_\wp$. If two solder jobs $j_\wp$ and j_ℓ require the same maximal temperature, then a decision of which job node in the graph that should have an outgoing edge to the other needs to be made. This decision is made in algorithm 1 (see Table 4.1 for the notations), in which $m \in \mathcal{K}$:

Algorithm 1 The algorithm returns true if the special job j_ℓ shall be before the special job $j_\wp$ in a SpS with the assumption that the SpS will be scheduled in furnace k. The job with the highest penalty weight should be first in the SpS. If the penalty weights are the same for the two jobs, then the job with the shortest process time should be first. If the process times for both the jobs are the same, then the job with the earliest deadline should be first. If these three parameters are the same for both jobs, then the decision can be made arbitrarily, since the jobs are equivalent in every fashion. (virtually all jobs have the same completion weight, so that weight does not need to be considered.)

1: **procedure** MyProcedure($j_\ell, j_\wp, k$)
2:     **if** $w_{j_\ell} \neq w_{j_\wp}$ **then return** $w_{j_\ell} > w_{j_\wp}$
3:     **if** $p_{j_\ell k} \neq p_{j_\wp k}$ **then return** $p_{j_\ell k} < p_{j_\wp k}$
4:     **if** $d_{j_\ell} \neq d_{j_\wp}$ **then return** $d_{j_\ell} < d_{j_\wp}$
       return true

An SpS of solder jobs can then be computed by finding a path $P = \langle r^{\text{bo}}, \ldots, d\rangle$ in such a DAG, illustrated in Figure 5.1. An advantage with DAGs compared to graphs in general is that an optimal path can be found faster in a DAG than in graphs with cycles. Furthermore, an optimal path in a graph with negative weights might not exists if the graph has cycles (see [3, Chapter 4] for properties about graphs with negatives weights on their edges).

5.4.2 The Subproblem of Generating SpSs of Titanium Jobs

The SpSs of titanium jobs have no restrictions on the order of the titanium jobs due to the jobs' temperature, but they have a length restriction – they can only contain one or two special jobs. Therefore, the maximum number of SPSs of titanium jobs is $|\mathcal{K}| \left(\binom{n}{1} + \binom{n}{2}\right)$, where n is the number of titanium jobs. Hence, $\binom{n}{2}$ DAGs used to generate SpSs of titanium jobs per furnace needs to be created. In reality it is, however, unlikely that all SpSs of titanium jobs will be suitable for all furnaces in the instance. DAGs that can be used to generate SpSs of titanium jobs can be created in a similar fashion as for the solder jobs, but each such DAG can only contain two nodes that correspond to titanium jobs. The same order priorities of the nodes as for the solder nodes can be applied for these graphs (see Algorithm 1).

5.4.3 Generating Several Families of SpSs

A solution to the IRMP corresponds to one family of SpSs. This family will be used by the scheduling model to compute a schedule. Even if the scheduling model is solved faster when only one family of SpSs is used by the scheduling model as

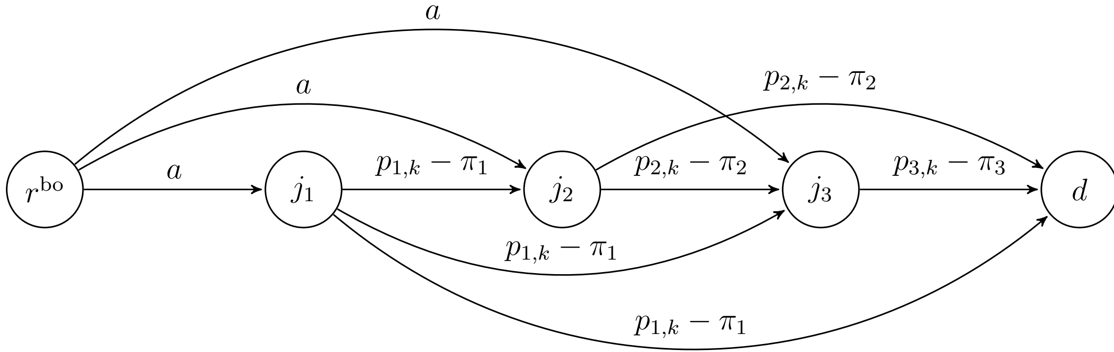


Figure 5.1: A DAG used to generate columns of SpSs of solder jobs. The node r^{ba} is the start node, the nodes j_1, j_2 and j_3 correspond to solder jobs, and the node d is the destination node. A path $S = \langle r^{\text{ba}}, \ldots, d\rangle$ contains the special jobs that are in the SpS the path represent. This graph is constructed for the furnace $k \in \mathcal{K}$ and the set $\mathcal{J}_i^{\text{corr}}$ contains exactly the special jobs that are in this SpS. Then the sum $a + \sum_{j \in \mathcal{J}_i^{\text{corr}}} (p_{jk-\pi_j})$, which is equal to the cost of all edges that were traversed for this path, is the reduced cost of the column that the SpS corresponds to.

compared with many families, a few families might result in a better schedule, and it may not take a lot more CPU time to solve the scheduling model when a few families of SpSs are used by the scheduling model, compared to only one family.

Several families might exist in the set $\tilde{\mathcal{I}}$ when no column with reduced cost can be found. If several families exist, then they can be selected by different approaches. In this project, we have tested two approaches. One approach is to compute several families by random, i.e., selecting the first n families found, where n is the maximal number of families considered. The second approach is made by computing all families that exist in the set $\tilde{\mathcal{I}}$, and then sorting them with respect to the objective coefficients c_i of the IRMP (5.4). The n cheapest families are then chosen.

6
Implementation

This chapter gives the reader information about implementation details, including the modifications of the instance files that were used for the benchmark and that was provided by GKN, and how the time-measurement was made. The software IBM ILOG CPLEX Optimization Studio [31] was used to solve all the (integer) linear programs in the project. C++ was used for the implementation. The programming language *A Mathematical Programming Language* (AMPL) (see [32]) is often used to model linear programs since the language is constructed for this task, but since implementing the TIM (see Figure 4.1) and modifying the given instances so that they can be used by the model (3.13) was easier in C++ than in AMPL, the implementation was performed in C++.

6.1 Modification of the Instances

The original scheduling problem that GKN has allows batches of jobs to be scheduled in a furnace, i.e., several jobs can be scheduled simultaneously in one furnace if certain requirements are satisfied. Some SpSs that GKN has generated contain batches of jobs, but these SpSs can not be scheduled in the model (4.1), since it does not allow batches to be scheduled. The solution implemented in this project removes batched jobs from the SpSs and treat them as normal jobs.

When a SpS containing batched jobs is included in the given input instance, all but one of the batched jobs are removed from all SpSs that contain these jobs. This procedure is carried out for all SpSs containing batched jobs. As a consequence, these jobs are no longer treated as special jobs, and can be scheduled without maintenance jobs (see Section 4.2). This modification of the given problem instances means that we are not solving real-world problem instances.

The reason why these jobs are removed from all SpSs and not just in the ones in which they are batched, is that some jobs would be treated as special jobs depending of which family that is selected.

When a job that is removed from an SpS S_1 is included in another SpS S_2, and therefore removed from the SpS S_2 too, it might be the case that the SpS S_2 needs to be modified by removing a maintenance job from it. For example, if the job that was removed from an SpS is the only job in that SpS, then the resulting SpS consists of only one or two maintenance jobs. Such an SpS can not contribute to an optimal schedule and can therefore be removed. When a job is removed from a titanium SpS, and that job is not batched in the SpS, then a vaccum-test maintenance job can be removed from that SpS.

6.2 Comparisons of the Approaches to be Benchmarked

The procedure of removing jobs that are batched in SpSs must be applied, nonetheless if the column generation method that generates SpSs is used or not. The reason for this is to make a fair comparison between the benchmarks. We wish to compare the results from the case when the SpSs are generated by column generation with that obtained for the case when the SpSs given by GKN are used. The same set of jobs in both tests must be special jobs, and since the procedure described in Section 6.1 might turn special jobs into non-special jobs, this procedure has to be applied to all benchmarks.

6.3 Computation Time Measurements

Several time measurements can be made for each instance that is benchmarked. We wish to compare the results from the case when the SpSs are generated by column generation with that obtained for the case when the SpSs given by GKN are used.

For a scheduling problem to be solved, the model and the data file needs to be read, the column generation method to generate SpSs might be used, the model is created and solved, and a result file is created containing the solution. Furthermore, when TIM (see Figure 4.1) is used, the obtained solution is modified for the next step and saved, a new model is created in which a time-step represents fewer minutes than in the previous step, and the time-span is shortened. This model is then solved, and the cycle is iterated six times. For each iteration, the time used to calculate the discrete time-steps is measured.

The time used to read the file is not included in any time measurements, but the time used to build the model in C++ is included in the total solution time. Once an optimal solution for an instance with a certain time-step representation has been found, the time measurement is halted and the result is written to a file (this time is not included in any time measurement). If TIM is used, the time to calculate new time-spans, and to initialize the new model with the previous solution is included in the time measurements for that TIM-iteration. Furthermore, the time used to solve the model in the previous TIM-iteration is included in the following TIM-iteration. When the column-generation method is used to generate SpSs, the time used for that is included in the total solution time.

7 Tests and Results

To answer the questions that were posed in the introduction of this thesis, a number of benchmarks have been performed. The questions to answer is how the column generation approach to generate SpSs and families and then let the scheduling model use these, performs, as compared to the performance when the schedules that are obtained with the families of SpSs generated by GKN are used. The notation of performance is in this context in terms of the cost of the schedules and of the execution time. The results from these benchmarks are presented in this chapter.

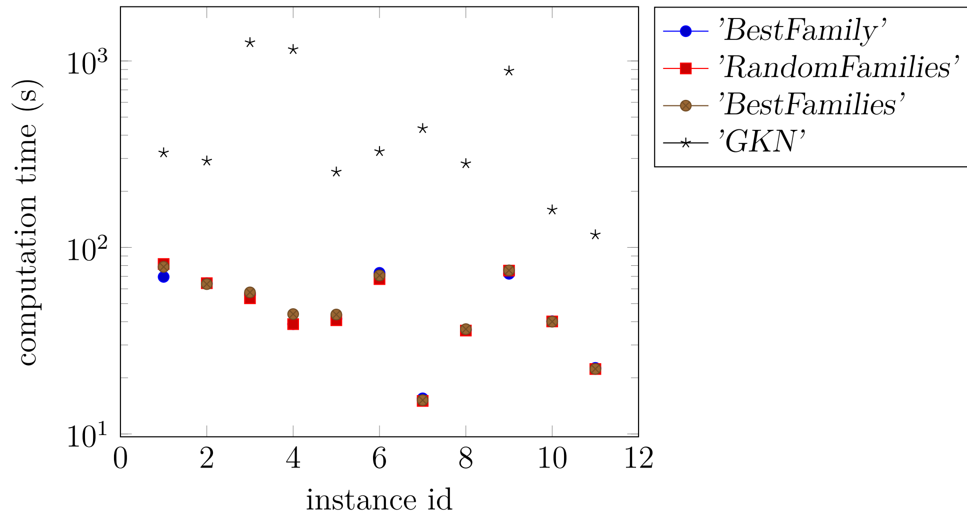
The instances that have been used for the benchmarks are modifications of real-world instances that GKN has used to create schedules for their production. Each of these instances has at least five special jobs after the batching modification has been performed (see Section 6.1). A mapping between the instance ids in this thesis and their names at GKN's department is seen in Table A.1. Three procedures have been used to test the proposed approach. The difference between these procedures is how the families of SpSs are selected. The different procedures of selecting families include computing the family of SpSs that is considered the cheapest by the IRMP, computing three random families of SpSs of all the SpSs that has been generated by the SP until no new column with a negative reduced cost can be found, and computing the three families of SpSs that are considering being the best by the IRMP. In the case when more families than one should be computed but there exists less than three families, then all existing families are computed, i.e., one or two families. The three procedures are refereed to as *'BestFamily'*, *'RandomFamilies'* and *'BestFamilies'*, respectively. Each of these three procedures have been tested for two values of the constant a (see Table 5.1): 0 and 500. The approach GKN currently uses, refereed to as *'GKN'*, has been benchmarked as well. The results of the benchmark test are now ready to be presented.

The results in terms of running time, and the *normalized cost* with respect to the cost of the schedules that were computed by the approach *'GKN'* for $a = 0$ can be seen in Figure 7.1a and Figure 7.1b, respectively, and in Table A.2. The formula for normalized cost is defined as

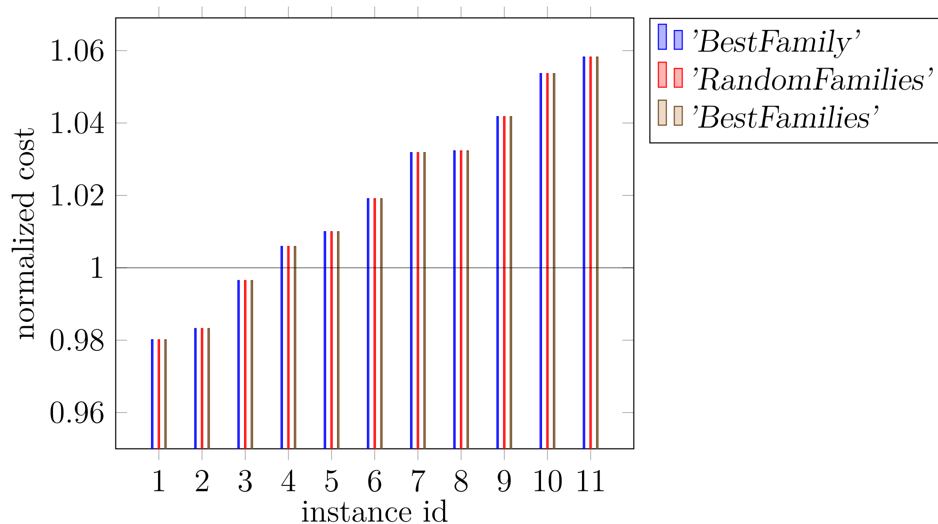
$$z = \frac{A_c}{\text{'GKN'}_c}, \tag{7.1}$$

where A_c is the cost of a schedule computed when approach or procedure A was used and 'GKN'_c is the cost of a schedule computed when the approach GKN' was used. As shown in Figure 7.1b, for each instance, the normalized cost is identical over the three procedures of the proposed approach. That is due to when $a = 0$, all families of SpSs have the same cost (see (5.4)), and therefore, no SpS are generated by the SP. Therefore, the same family of SpS is computed by all three procedures.

The time used for each procedure is therefore approximately the same (see Figure 7.1a). While most solutions found by the different procedures are worse than the approach *'GKN'*, the running time is many times shorter.



(a) The CPU time in seconds used to solve each instance by each approach.

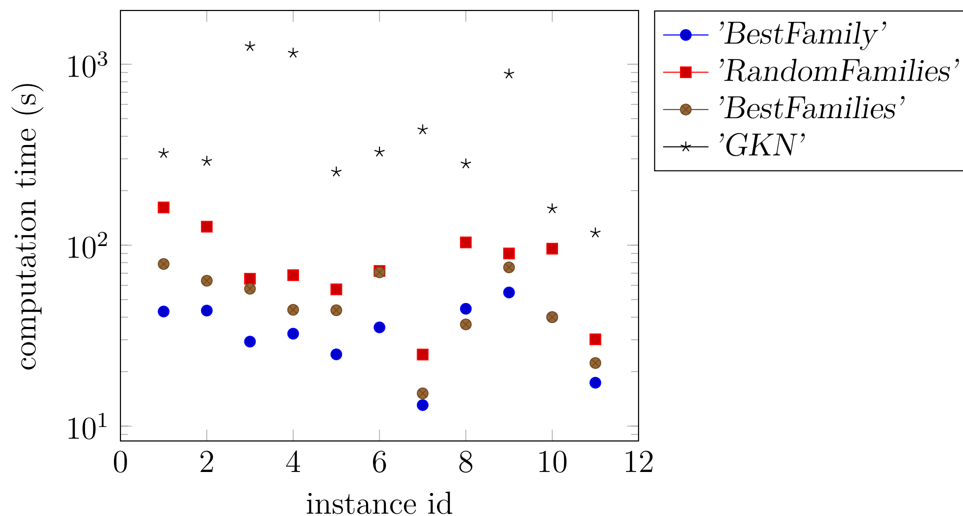


(b) The normalized cost for each approach and instance with respect to the approach *'GKN'*. The horizontal line illustrates the normalized cost of the approach *'GKN'*.

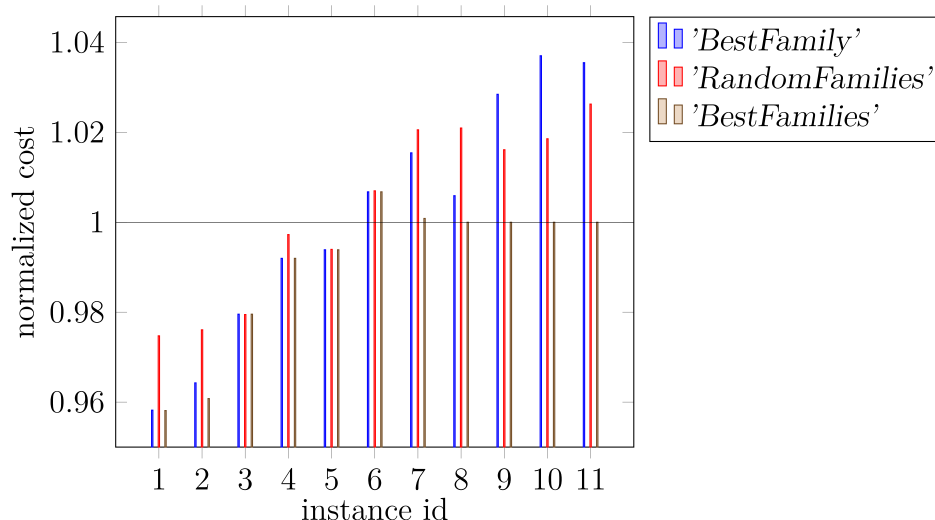
Figure 7.1: The computation time and the normalized cost visualized for each approach and instance from Table A.2. The instances were sorted after the results were computed.

The corresponding results of the benchmark when $a = 500$ can be seen in Figure 7.2a and Figure 7.2b, respectively, and in Table A.3. All procedures of the proposed approach that was allowed to generate up to three families computed three families in this benchmark test. It is clear that the number of families affects the computation time a lot — when one family is used as input to the scheduling model it is a around two minutes for most instances, but up to 15 minutes when several families are used.

The procedure *'RandomFamilies'* has for some instances the same normalized cost as the procedure *'BestFamily'*, but is for most instances worse, and never better. The procedure *'BestFamily'* is better than the approach *'GKN'* for five out of eleven instances and worse for the rest. The procedure *'BestFamilies'* is equally good for one instance, better for five and worse for five instance as compared to the approach *'GKN'*. While the computation time is much shorter for the proposed procedures, the normalized cost is close or below one for several instances, especially for the procedures *'BestFamilies'*, which is non-worse for all instances but one compared to *'GKN'*.



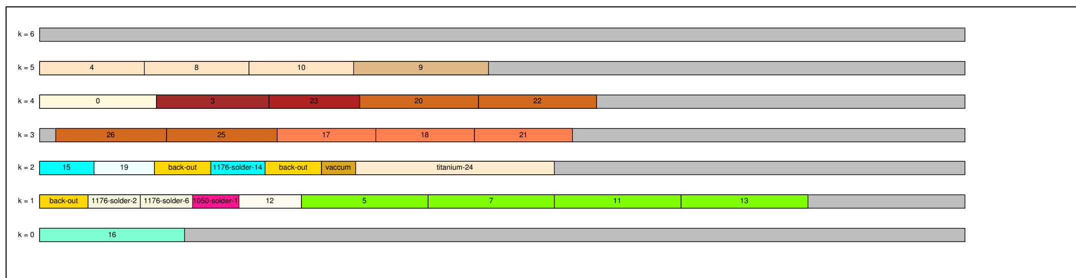
(a) The CPU time in seconds used to solve each instance by each approach.



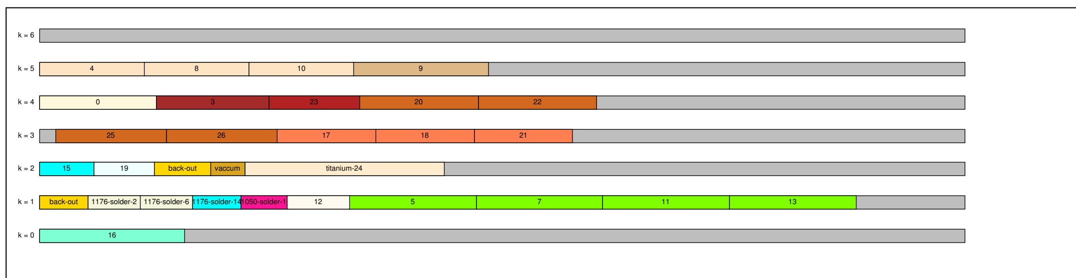
(b) The normalized cost for each approach and instance with respect to the approach *'GKN'* The horizontal line illustrates the normalized cost of the approach *'GKN'*.

Figure 7.2: The computation time and the normalized cost visualized for each approach and instance from Table A.3. The time for the approach *'GKN'*. is taken from Table A.2.

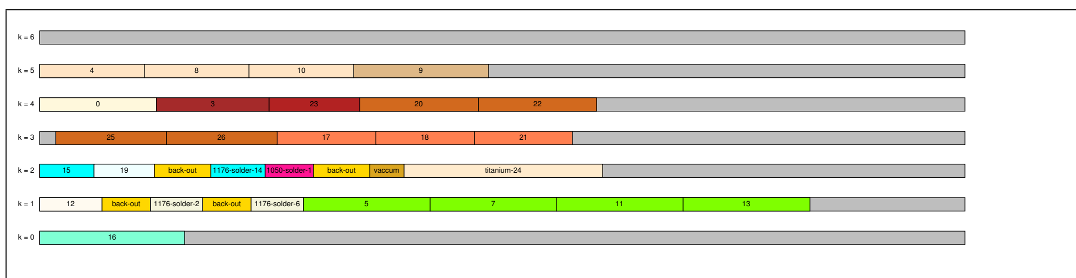
The computed schedules for the approaches *'GKN'*, *'BestFamily'*. *'RandomFamilies'*, and *'BestFamilies'* for instance 7 when $a = 500$ are illustrated in the Figures 7.3a, 7.3b, 7.3c and 7.3d, respectively. In the schedules, each of the seven rows corresponds to a furnace. The gray area in the rectangle for each row is the time when no jobs were being processed for the furnace that correspond to that row and the non-gray color is the time when jobs are being processed. Each job that is being processed is visualized as a rectangle. For a certain job, the most left side in the rectangle is the time when the job starts and the most right side is the time when it is completed. The text in the middle of the rectangle is information about the job. For the non-special jobs, the information is a number, which is the index of the job. The maintenance jobs are written as "back-out" and "vaccum", respectively. A titanium job is written as "titanium-id" and a solder job is written as "maximal temperature-solder-id". For example, furnace 1 in the schedule seen in Figure 7.3d starts with a back-out job, followed by three solder jobs with the maximum temperature 1176, the first processed solder job has index 2, the second has index 6 and the third has index 15. The non-special jobs 5, 7, 11 and 13 are then processed.



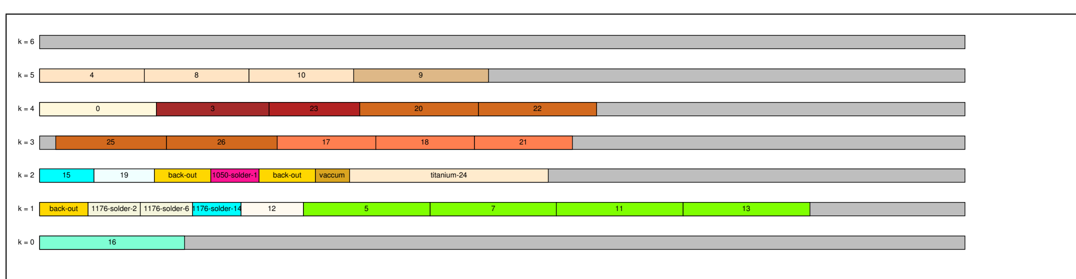
(a) The schedule computed by the approach *'GKN'*



(b) The schedule computed by the procedure *'OneFamily'*



(c) The schedule computed by the procedure *'RandomFamilies'*



(d) The schedule computed by the procedure *'BestFamilies'*

Figure 7.3: The schedules that were computed for instance 7 for each approach and procedure when $a = 500$ (see Table 5.1). If two jobs' colors are the same, then the jobs could form a batch of jobs that could be schedules simultaneously in the original scheduling model develop by GKN. For the scheduling model in this thesis that does not use the concept of batching, the colors do not have any meaning.

8 Discussion and Conclusion

A proposal to generate families of special sequences has been given and tested in this project. Our results indicate that cheaper schedules than GKN computes with their current method can be computed by the proposed approach much faster. However, it can not be guaranteed that the schedules that are computed by the proposed approach are cheaper. Our results also indicate that the objective function used by the model we developed to find families was not well suited, since the procedure *'OneFamily'* gave worst result than *'BestFamilies'* in seven out of eleven cases.

There is a strong reason to criticize the statement that some families of special sequences that are generated by the proposed approach are better than the one that GKN has computed. That is due to that the given special sequences from GKN were modified so that they did not contain batches of jobs. It is likely that some special sequences that was generated by the proposed algorithm were not computed by GKN because they had special sequences with batches that dominated such a special sequence.

The original idea of the project was to transform the scheduling model to another model to which column generation would be applied, but due to the complex constraints of the scheduling model, especially the constraints regarding scheduled pauses, this was deemed to be a very complicated task, and the focus switched to generating families of special sequences. These families of special sequence were generated for a simplification of GKN's model in which batching is not allowed. This made it easier to generate families of special sequences, since batching was not considered. Unfortunately, by removing the possibility of batches, some special jobs and special sequences needed to be removed from each instance. The need to generate special sequences was therefore smaller by our model than by GKN's model. The column generation never took more than one (1) CPU second for any instance – the most time-consuming part for all instances was to solve the scheduling model.

In conclusion, this project shows that generating a few families of special sequences with column generation results in a much better running time for the scheduling model than generating all of them. Furthermore, the schedules that are calculated by the scheduling model when these families are used by the scheduling model can give a better result than the method GKN uses today.

9 Future Research

Future research ideas and suggestions that has emerged during this project are presented in this chapter.

9.1 Parallelize the Scheduling Model

The first suggestion is to parallelize the scheduling model. All families of SpSs is needed to be used by the scheduling model for it to calculate the optimal schedule, but only one family is used per schedule. Therefore, one can solve the problem in parallel in the following fashion. First, compute all families of SpSs and then divide the families into groups of the same size as far as possible. Then create an instance for the scheduling model for each group and let each instance only contain the families of SpSs that are in the group it was created for. These instances can now be solved in parallel, and the instance with the cheapest solution is the optimal solution for the scheduling model, since each family of SpSs has been considered.

9.2 Compute Several Schedules

Another suggestion is to solve the scheduling model twice, one time with only a few families of SpSs, and a second time with all SpSs. The motivation for this is to quickly find a schedule that is acceptable so that some jobs can start to be processed quickly. Most jobs require more than four hours in the furnaces, and during the time these jobs are being processed, an optimal schedule can be computed for the remaining jobs in the instance, i.e., the second time a schedule is computed, the jobs that started first in the schedule that was first computed are not considered, since these jobs have already been processed.

9.3 Develop a New Model

A third suggestion is to create a new model which does not use the concept of SpSs. Such a model needs to keep track of the temperatures of the jobs, the number of titanium jobs that have been scheduled before a new clean-up is performed, and will contain many more variables (with time indices) than the scheduling model. Creating such a model will remove the difficulties related to SpSs, but introduces a new difficulty. Such a model can only be used for a workshop that do not have other special jobs than the solder and titanium jobs.

9.4 Perform a Study of Domination Criteria

The last suggestion is to perform a study of domination criteria for the SpSs. Domination criteria in this context mean that an SpS will always result in a better schedule compared to another SpS. There exist trivial *domination criteria* for the SpSs, for example, assume there is a set of solder jobs that are identical in every way, except their completion weight. An SpS of these jobs such that the solder jobs starts decreasingly with respect to their completion weight dominates all other SpSs of the same length and jobs. The number of SpSs and therefore families of SpSs that is needed to be generated is drastically decreased when a domination criterion is known. Therefore, a study with the goal of finding more domination criteria would be interesting.

Bibliography

[1] Karin Thörnblad. *On the Optimization of Schedules of a Multitask Production Cell, Licentiate thesis.* Chalmers University of Technology and University of Gothenburg, 2011.

[2] GKN. Framgångsrikt projekt finalist i EURO Excellence in Practice Award. www.gkngroup.com/GKNSweden/news/Pages/A_160414_EURO_award.aspx, 2016.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1997.

[4] F. Pezzella, G. Morganti, and G. Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers and Operations Research*, 35(10):3202–3212, 2008.

[5] Guy Doumeingts and Jimmie Browne. *Modelling Techniques for Business Process Re-engineering and Benchmarking.* Springer US, Bordeaux, 1997.

[6] Marjan Van Den Akker, Han Hoogeveen, and Steef Van De Velde. Applying column generation to machine scheduling. In *Column Generation*, pages 303–330. Springer US, Boston, MA, 2005.

[7] J. van Leeuwen. *Handbook of Theoretical Computer Science.* Elsevier, 1990.

[8] Anant Jain and Sheik Meeran. A state-of-the-art review of job-shop scheduling techniques. *European Journal of Operations Research*, 113:390–434, 1999.

[9] E. Moradi, S. M T. Fatemi Ghomi, and M. Zandieh. An efficient architecture for scheduling flexible job-shop with machine availability constraints. *International Journal of Advanced Manufacturing Technology*, 51(1–4):325–339, 2010.

[10] P. Brucker and R. Schlie. Job-shop scheduling with multi-purpose machines. *Computing*, 45(4):369–375, 1990.

[11] E. Mokotoff. Parallel machine scheduling problems — a survey. *Asia-Pacific Journal of Operational Research*, 18:193, 2001.

[12] Rabadi Ghaith, Moraga J. Reinaldo, and Al-Salem Ameer. Heuristics for the unrelated parallel machine scheduling problem with setup times. *Journal of Intelligent Manufacturing*, 17:85–97, 2006.

[13] Gregory Dobson and Ramakrishnan S. Nambimadom. The batch loading and scheduling problem. *Operations Research*, 49(1):52–65, 2001.

[14] Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. In *Handbook of Knowledge Representation*, volume 6526, pages 181–212. Elsevier, Amsterdam, 2008.

[15] S. David Wu, Eui-Seok Byeon, and Robert H. Storer. A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness. *Operations Research*, 47(1):113–124, 1999.

[16] Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993.

[17] D. de Werra and A. Hertz. Tabu search techniques. *OR Spektrum*, 11(3):131–141, 1989.

[18] Cemal Özgüven, Lale Özbakır, and Yasemin Yavuz. Mathematical models for job-shop scheduling problems with routing and process plan flexibility. *Applied Mathematical Modelling*, 34:1539–1548, 2010.

[19] Alan S. Manne. On the job-shop scheduling problem. *Operations Research*, 8:219–223, 1960.

[20] Jorge P. Sousa and Laurence A. Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, 54(1–3):353–367, 1992.

[21] J. M. Van Den Akker, C. A. J. Hurkens, and M. W. P. Savelsbergh. Time-indexed formulations for machine scheduling problems: column generation. *INFORMS Journal on Computing*, 12(2):111–124, 2000.

[22] J. M. Den, Van Akker, J. A. Hoogeveen, and S. L. Van De Velde. Parallel machine scheduling by column generation. *Operations Research*, 47(6), 1999.

[23] Gongshu Wang and Lixin Tang. A row-and-column generation method to a batch machine scheduling problem. *Proceedings of the Ninth International Symposium on Operations Research and its Applications (ISORA-10)*, page 301–308, 2010.

[24] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education, New York, 2008.

[25] Leon S. Lasdon. *Optimization Theory for Large Systems*. Dover Publications, 2002.

[26] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 2001.

[27] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Amsterdam, 1987.

[28] Der-San Chen, Robert G. Batson, and Yu. Dang. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons, New York City, 2010.

[29] Pamela H. Vance, Cynthia Barnhart, Ellis L. Johnson, and George L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3(2):111–130, 1994.

[30] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[31] IBM. IBM ILOG CPLEX Optimization Studio. www.ibm.com/products/ilog-cplex-optimization-studio, 2014.

[32] Robert Fourer, David M. Gay, and Brian W. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36:519–554, 1990.

A

Appendix A

Instance id	File name
1	20150925_113338
2	20150925_111745
3	20151008_141908
4	20151026_110356
5	20151001_110346
6	20150925_125446
7	20150921_100250
8	20150923_154604
9	20150923_193105
10	20150923_155904
11	20150924_141916

Table A.1: A mapping of each instance id to its corresponding file name.

Instance id	#jobs			'GKN'			'BestFamily'				'RandomFamilies'				'BestFamilies'			
	$\lvert\mathcal{J}\rvert$	$\lvert\mathcal{J}^{\text{titan}}\rvert$	$\lvert\mathcal{J}^{\text{sold}}\rvert$	#seg	#fam	t[CPU s]	#seg	z	t[CPU s]	t_{cg}[CPU s]	#seg	z	t[CPU s]	t_{cg}[CPU s]	#seg	z	t[CPU s]	t_{cg}[CPU s]
1	32	4	4	12	8	321.97	8	0.98	69.54	0.02	8	0.98	81.54	0.01	8	0.98	78.70	0.02
2	32	4	4	12	8	291.11	8	0.98	64.27	0.02	8	0.98	64.33	0.01	8	0.98	63.60	0.02
3	30	3	4	25	43	1255.18	7	1.00	55.97	0.05	7	1.00	53.40	0.04	7	1.00	57.43	0.04
4	30	3	4	26	54	1152.92	7	1.01	38.83	0.02	7	1.01	38.81	0.01	7	1.01	43.93	0.01
5	31	3	4	16	18	254.08	7	1.01	41.44	0.02	7	1.01	40.72	0.01	7	1.01	43.69	0.01
6	33	3	4	16	18	327.20	7	1.02	73.08	0.02	7	1.02	67.66	0.02	7	1.02	70.77	0.02
7	27	1	4	22	25	434.62	5	1.03	15.52	0.03	5	1.03	15.03	0.02	5	1.03	15.17	0.02
8	28	3	3	10	8	281.62	6	1.03	36.05	0.02	6	1.03	35.85	0.02	6	1.03	36.48	0.02
9	30	4	3	11	8	884.55	7	1.04	72.26	0.03	7	1.04	75.16	0.03	7	1.04	75.38	0.04
10	28	3	3	10	8	159.09	6	1.05	40.22	0.01	6	1.05	40.11	0.01	6	1.05	40.00	0.01
11	23	3	3	10	8	117.01	6	1.06	22.61	0.01	6	1.06	22.29	0.01	6	1.06	22.32	0.01

Table A.2: The result of a benchmark test when each instance was solved using the approach *GKN'* and the three different procedures *BestFamily'*, *RandomFamilies'* and *BestFamilies'* of the proposed approach. Each of these procedures and the approach *'GKN'* are presented in Chapter 7. Instance id denotes the id of the instance (the name mapping to each id can be seen in Table A.1). The notations $\lvert\mathcal{J}\rvert$, $\lvert\mathcal{J}^{\text{titan}}\rvert$ and $\lvert\mathcal{J}^{\text{sold}}\rvert$ denote the number of jobs, the number of titanium jobs, and the number of solder jobs in each instance. The notations #seg, #fam, and t[CPU s] represent the number of SpSs and the number of families for each approach/procedure, and the computing time (in seconds) used to find an optimal schedule. The column z denotes the normalized cost of a schedule (see the formula (7.1)). t_{cg}[CPU s] denotes the time (in seconds) used to generate the SpSs by the column generation procedures. The value of the variable a was set to 0 during this benchmark test (see Table 5.1).

Instance id	*'BestFamily'*				*'RandomFamilies'*				*'BestFamilies'*			
	#seg	z	t[CPU s]	t_{cg}[CPU s]	#seg	z	t[CPU s]	t_{cg}[CPU s]	#seg	z	t[CPU s]	t_{cg}[CPU s]
1	4	0.96	42.99	0.03	14	0.97	161.62	0.13	14	0.96	106.38	0.12
2	4	0.96	43.50	0.02	14	0.98	126.52	0.11	14	0.96	78.05	0.13
3	3	0.98	29.31	0.03	11	0.98	65.22	0.04	11	0.98	75.44	0.04
4	3	0.99	32.42	0.03	11	1.00	68.28	0.04	11	0.99	70.42	0.04
5	3	0.99	24.92	0.01	11	0.99	57.00	0.04	11	0.99	59.35	0.04
6	3	1.01	35.13	0.03	11	1.01	72.02	0.04	11	1.01	83.32	0.04
7	2	1.02	13.07	0.01	8	1.02	24.86	0.01	8	1.00	17.38	0.01
8	4	1.01	44.51	0.02	10	1.02	103.56	0.05	10	1.00	75.87	0.02
9	5	1.03	54.77	0.03	11	1.02	89.99	0.02	11	1.00	84.63	0.04
10	4	1.04	40.02	0.02	10	1.02	95.62	0.02	10	1.00	59.80	0.03
11	4	1.04	17.37	0.03	10	1.03	30.18	0.02	10	1.00	33.49	0.04

Table A.3:
The result of a benchmark test when each instance was solved using the three different procedures *'BestFamily'*, *'RandomFamilies'* and *'BestFamilies'* of the proposed approach when $a = 500$ (see Table 5.1). An explanation of the notations used in this table is given in Table A.2. The normalized cost (see the formula (7.1)) was calculated from the result of the approach *'GKN'* that was obtained from the benchmark that can be viewed in Table A.2.

B

Appendix B

A table of all abbreviations that are used in this thesis and the meaning of them are presented in this Appendix.

Abbrevation	Meaning
BaB	Branch-and-Bound
BaP	Branch-and-Price
FJSP	Flexible Job-Shop Problem
JSP	Job-Shop Problem
PMSP	Parallel Machine Scheduling Problem
BMSP	Batch Machine Scheduling Problem
HTS	Hierarchical Tabu Search
MP	Master Problem
RMP	Restricted Master Problem
SP	Sub-Problem
LP	Linear Program
ILP	Integer Linear Program
IMP	Integer Master Problem
IRMP	Integer Restricted Master Problem
TIM	Time-Iteration Model
SpS	Special Sequence
DAG	Directed Acyclic Graph

Table B.1: The meaning of the abbreviations used in this thesis.