Erasmus School of Economics

MSc Econometrics

Specialization: Operations Research & Quantitative Logistics

The Hybrid Flow Shop model with multiprocessor tasks, unrelated parallel machines and sequence dependent setup times

Author

O.N.C. Witsen

471441

Supervisor

Dr. W. van den Heuvel

Second assessor

Dr. K.S Postek

In this thesis we present a study focused on a scheduling problem that is inspired by the production process of mattresses. We can translate our scheduling problem into a multi-objective Hybrid Flow Shop model with non-identical multiprocessor tasks, unrelated parallel machines and sequence dependent setup times. To the best of our knowledge, we are the first to introduce research concerning the Hybrid Flow Shop model with these assumptions. We define our problem mathematically by means of a mathematical formulation, which also allows us to obtain a benchmark for small sized problem instances. Furthermore, we define a constraint programming formulation to solve small problem instances to optimality. Based on our mathematical formulation, we derive a lower and upper bound. Next to exact solution approaches, we adopt heuristic solution approaches from literature, a genetic and memetic algorithm, and adapt these methods to be able to solve our model. The outcomes of the resulting algorithms are compared to the optimal solution. It turns out that on certain small problem instances our heuristic solution approaches are able to produce optimal results. Finally, we study the performance of the heuristic approaches on a large industry based case study. On the large scale problem instances the genetic algorithm in combination with a linear programming optimization problem at the end yields the most successful results.

September 12, 2019

Contents

1 Introduction

In this thesis we will study a scheduling problem that is inspired by the production process of mattresses. Before a mattress is ready for use, several production steps have to be executed. The mattresses are produced by a make to order policy. Due to the fact that a mattress has 4 stages that need to be processed before it is ready for use and that there are around 1300 mattresses that need to be planned on a weekly basis, efficient planning turns out to be an extremely complex task. In the production environment the aim is to schedule the jobs in such a way that the jobs are finished just in time, to prevent unnecessary stock; that the stock between production stages is minimized; the finish time of the time horizon of the production schedule is as low as possible. The minimization of stock plays an important role in the production process of mattresses due to the fact that fully produced mattresses take a lot of space, while its parts before being assembled do not. The aim of our research will be to find an efficient algorithm that creates a production plan for a production environment that can deal with the characteristics just described.

In the rest of this thesis, a literature review concerning our research problem will be done in Section 2. After the literature review, we will propose two exact solution approaches, a mathematical formulation and a constraint programming formulation in Section 3. In Section 4 we present and explain several heuristic solution approaches. We will compare the results of our solution approaches in Section 5. Finally, a conclusion can be found in Section 6 and a discussion in Section 7.

1.1 Problem Description

To further elaborate on the production process of mattresses, this process starts with the production of various types of springs by means of reforming metal bars of which there is ample stock available. In the next stage these different types of springs are assembled together such that different types of bases of mattresses are produced. The next production step is to use these bases to make the main product, namely, different types of mattresses. One can think of different types of mattresses in the sense that there are box springs, single mattresses and double mattresses, for example. A possible way in which the base of a mattress is finished such that it is ready to be sent to the customer, is that it gets coated, then packaged and finally placed on pallets. All the different kinds of mattresses can make use of the same resources, however, the machines do not all have the same production speed and sometimes parts can not be produced on certain specific machines.

In Figure 1 we present an overview of the production environment that is being considered. Figure 1 shows that there are k stages with m_k machines in each stage. We also see that the machines are different from each other. For our specific case the number of machines per stage declines as we continue in the production process. This is, however, not a necessary condition for our model, as can be seen in Figure 1. A formal mathematical model to describe our problem is given in Section 3.

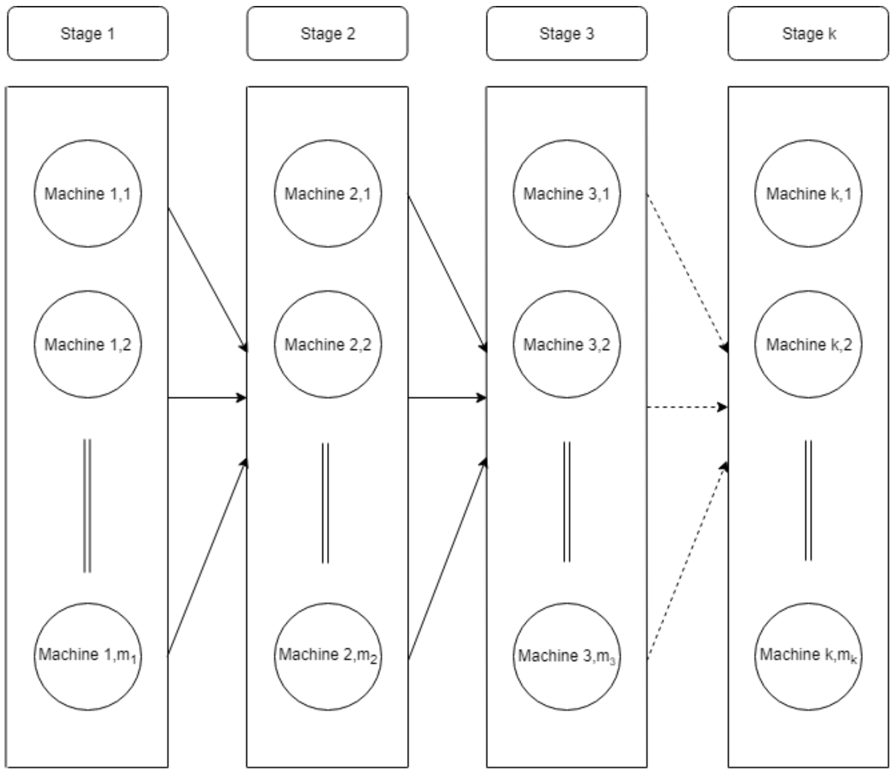

Figure 1: Visual representation of HFS-variant.

In literature, our problem is usually referred to as the Hybrid Flow Shop model (we will refer to this as HFS) and sometimes it is called the Flexible Flow Shop model. The HFS can be seen as a generalization of the Flow Shop model or the Parallel Shop model. A Flow Shop model can be described as having a set of jobs and machines in which each job has a number of operations that need to be executed that is equal to the number of machines. The operations of a job must be executed within a predefined order, which is the same for all jobs. If there is only one machine per stage in a HFS, we can see the HFS as a generalization of the Flow Shop model. A Parallel Shop model can be described as having a set of machines and jobs in which each job needs to be executed by only one of the machines that are available. Thus, the Hybrid Flow Shop model differs from a Parallel Shop model if it has more than one stage.

According to Ruiz and Vázquez Rodríguez (2010) the HFS in standard form assumes that the number of stages is at least 2, that there is at least 1 machine in each stage (in case there is more than 1 machine in a stage, they assume that they are identical and in parallel), that each job passes through all stages in the same predefined order, that all machines and jobs are available at time 0, machines can only process one job at a time and a job can only be processed by one machine at a time, the buffer capacity between stages is infinite and the processing times and job demand and sizes are deterministic and known up-front.

We will extend on the standard assumptions for our specific Hybrid Flow Shop model. In our research we will focus mainly around the HFS with an arbitrary number of stages (≥ 2) with multiprocessor tasks.

The term multiprocessor tasks is used in literature to describe the case where a job can have more than 1 task to be processed in a stage. In literature, multiprocessor tasks are often assumed to be equal in terms of production times and that they need to start at the same time, however, we relax both of these assumptions. A good example that motivates the relaxation of the assumption of equal multiprocessor tasks is the production of springs. For the production of mattresses it can be the case that a mattress consists of different types of springs which differ in size. We also relax the assumption that all machines are identical in a stage such that we can deal with unrelated parallel machines, meaning that each machine can have its own speed for each job. Next to this we add the possibility of sequence dependent setup times to the standard form of the HFS. Sequence dependent setup time refers to setup time that depends on the direct preceding task for the task that is going to be executed next on that machine. The aim is to minimize a multi-objective optimization problem based on total tardiness, earliness, maximum makespan and total flow time.

The main motivation to relax some of the common assumptions used in literature, is that in practice the assumptions that are commonly made usually do not hold¿ This way, our newly introduced HFS-variant matches reality in a much closer way. Also, from an academic perspective there has been a demand for research in this direction, as has been pointed out by Lin et al. (2013), who state that researching the HFS with unrelated parallel machines and multiprocessor tasks would be "challenging, albeit worthwile". Furthermore, the literature reviews by Ribas et al. (2010) and Ruiz and Vázquez Rodríguez (2010) pointed out that there is a lack of realistic settings in the HFS and that new research is needed in this direction.

To visualize a solution to our complex problem, we present an example of a feasible solution to a small problem in Figure 2. This small example contains two stages and three jobs. In this example all jobs have two multiprocessor tasks in both stages. Note that there is no need for all jobs to have the same number of multiprocessor tasks for our problem in general. In both stages there are two machines. One can see in this example that there are multiprocessor tasks in a stage by the fact that a job can be seen multiple times in a stage. The underlying data for the example can be seen in Table 1, which states the processing times of the tasks and Table 2, which states the setup times between the tasks. These tables present a task by the notation $x.y$, in which x denotes the job to which the task belongs and y denotes the ID of the multiprocessor task of that job. The solution presented in Figure 2 shows that it is not always a good idea to start a task as early as possible (see job 1 in stage 1), since the flow time of a job might unnecessarily increase. In line with our problem definition, one can see in Figure 2 that a job can not start production in the next stage unless all tasks have finished production in the previous stage.

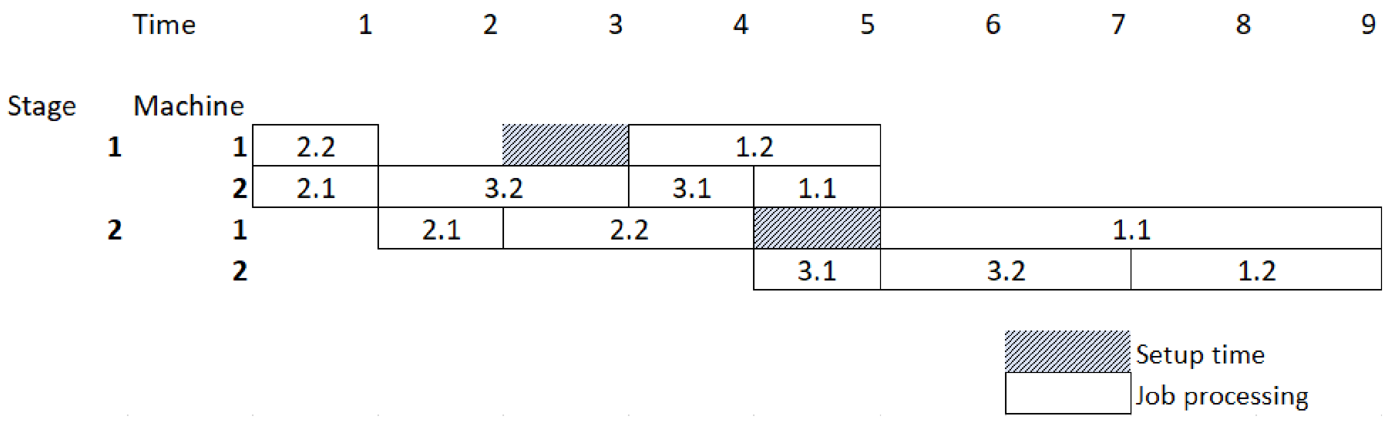

Figure 2: Visual representation of a feasible solution to a small example of the HFS variant.

Table 1: Processing times of tasks

	Stage	1						2					
Machine	Task	1.1	1.2	2.1	2.2	3.1	3.2	1.1	1.2	2.1	2.2	3.1	3.2
1		1	2	4	1	3	3	4	4	1	2	3	3
2		1	4	1	2	1	2	2	2	4	1	2	1

Table 2: Setup times between tasks

	Machine	1						2					
Stage	Task from\to	1.1	1.2	2.1	2.2	3.1	3.2	1.1	1.2	2.1	2.2	3.1	3.2
1	1.1	-	1	1	0	1	0	-	0	1	1	0	1
	1.2	0	-	1	1	1	1	0	-	1	1	0	0
	2.1	0	0	-	0	0	0	0	0	-	0	0	0
	2.2	1	1	1	-	1	1	0	1	1	-	1	1
	3.1	1	1	1	0	-	0	0	1	1	1	-	0
	3.2	1	0	0	1	0	-	1	1	1	1	0	-
2	Task from\to	1.1	1.2	2.1	2.2	3.1	3.2	1.1	1.2	2.1	2.2	3.1	3.2
	1.1	-	1	1	0	1	1	-	1	0	1	1	1
	1.2	1	-	0	1	0	1	0	-	0	1	0	1
	2.1	1	0	-	0	1	1	0	1	-	1	1	0
	2.2	1	1	1	-	0	0	0	1	1	-	0	0
	3.1	0	1	0	0	-	1	1	0	1	0	-	1
	3.2	0	0	0	1	1	-	1	0	1	1	0	-

2 Literature Review

We can translate our problem into the HFS with unrelated parallel machines, multiprocessor tasks and sequence dependent setup times (HFS-UMMTSDS) with a multi-criteria objective value that minimizes earliness, tardiness, flow time and the maximum makespan. As stated earlier in Section 1, to the best of our knowledge the (HFS-UMMTSDS) has not been studied before in literature. However, there has been research on certain aspects of HFS-UMMTSDS, namely: the HFS with unrelated parallel machines and the HFS with multiprocessor tasks. That is why we divide the literature into two categories: one with research done on the HFS with unrelated parallel machines and another with research done on the HFS with multiprocessor tasks.

Examples of papers that consider problems that are the closest in our opinions to our research are presented by Bozorgirad and Logendran (2016) and Kheirandish et al. (2014). Bozorgirad and Logendran (2016) study the HFS with sequence-dependent setup times and unrelated parallel machines. However, they do not take into account multiprocessor tasks and they consider non-constant processing times of machines (the processing times decrease due to so-called learning effects). Kheirandish et al. (2014) study the two-stage HFS with non-identical machines and multiprocessortasks. However, this is done in just a 2-stage manner with only 1 machine in the second stage and without sequence-dependent setup times.

In literature, different types of HFS problems can be described accordingly to the notation that has been introduced by Graham et al. (1979), which was later on further specified by Vignier et al. (1999). Their notation has the following structure: $\alpha|\beta|\gamma$, in which α is used to describe the overall characteristics of the HFS, including the number of stages and the number of machines in each stage (and whether those are all equal, similar or different from one another). α is denoted by the following vector: (α_1,α_2,α_3,α_4). In this notation α_1 denotes the type of flow shop (a hybrid flow shop will be denoted by HF), α_2 denotes the number of stages and (α_3,α_4) together denote the properties of the machines in the several stages to make it possible to have different machine properties in different stages. The properties of the machines are usually denoted by R, P or Q. R relates to unrelated parallel machines (as we have for our HFS), P relates to identical parallel machines, and Q is used for uniform parallel machines (in this case machines can only have different speeds relative to each other, which are the same for all jobs).

In the notation of Graham, β denotes the assumptions deviating from the standard assumptions used within the HFS and γ denotes the objective of the problem. For example, if one would want to model a two-stage HFS with unrelated parallel machines in the second stage and identical machines in the first stage with as objective value the maximum makespan without any deviating assumptions, this could be noted as $HF2,(P^1,Q^2)||C_{max}$. Using this notation, we can easily show that the problem is NP-hard by reducing our problem to the $HF2(P^k)_{k=1}^2||C_{max}$ for which a NP-hardness proof is presented by Gupta (1988). We present a formal proof for this in Section 3.

Fan et al. (2018), Ribas et al. (2010) and Ruiz and Vázquez Rodríguez (2010) present a literature overview of research that has been done on the Hybrid Flow Shop modeling problem in general and give an overview of algorithms used to solve different variants of the problem (both exact and heuristic methods). For exact approaches, Ruiz and Vázquez Rodríguez (2010) state that around 10% of all research on the HFS focuses on branch and bound based techniques and around 15% on mathematical formulations. These techniques have so far only been effective on problem instances with up to 20 jobs and at most five stages (Ribas et al. (2010)).

On the other hand, Ruiz and Vázquez Rodríguez (2010) state that around 50% of all research on HFS considers either simple dispatching rules or simple ad-hoc heuristics. Finally, the remaining 25% of the papers uses metaheuristic approaches like a Tabu Search, Simulated Annealing or Genetic Algorithms. In the metaheuristics that are used in literature, a solution is usually represented by means of a vector that determines the sequence of the execution of a job on a machine in the first stage. The way in which a solution is represented in a vector is usually called an encoding rule. A decoding rule is then used to determine the assignment of jobs to a machine and the starting times of the production order in the rest of the stages. A more detailed description of encoding and decoding rules will be given in Section 4, where we present our heuristic solution approaches. The metaheuristic approaches tend to perform particularly well when the problem sizes become larger. In the papers by Fan et al. (2018), Ribas et al. (2010) and Ruiz and Vázquez Rodríguez (2010) it is also stated that the main focus of the research has been on minimizing the maximum makespan. Ruiz and Vázquez Rodríguez (2010) even state that around 60% of research on HFS has been around minimizing the maximum makespan.

Ruiz and Vázquez Rodríguez (2010) also address the need for research on more robust approaches in case of varying input and the need for more research on metaheuristic solution approaches on complex versions of the hybrid flow shop scheduling problem. Next to this, Ruiz and Vázquez Rodríguez (2010) state that there is a lack of industry based objective values and that it might be interesting to develop flexible heuristics which work well in multiple settings of varying constraints and objectives.

Furthermore, Ruiz and Vázquez Rodríguez (2010) state that a lot of the metaheuristic approaches use the fact that the optimal solution is usually at least reasonably close to the permutation schemes of jobs within a stage (a vector containing the sequence of jobs in which they are being assigned to a machine) and that the full solution can be derived from this permutation scheme by applying simple dispatching rules. These simple dispatching rules are designed for the HFS based on a setting without multiprocessor tasks, but can be extended quite easily so they would work for a HFS with multiprocessor tasks as well (we will describe how this can be done in Section 4). Although this could still yield an intractable large search space for enumeration, this does limit the search space considerably, which might be very useful for our problem.

2.1 Hybrid Flow Shop with unrelated parallel machines

Most research has been on the Hybrid Flow Shop model with identical parallel machines. However, recent research about the HFS with non-identical machines has become more popular. Jungwattanakit et al. (2008) was one of the first papers that addressed the HFS with unrelated parallel machines and the most recent state-of-the-art has been resulting from applying (hybrid) metaheuristics.

For example, Chen et al. (2015) propose an ant colony optimization-based hyper-heuristic with genetic programming approach for a k-stage hybrid flow shop problem with one stage consisting of non-identical batch processing machines and the others of non-identical single processing machines. Rashidi et al. (2010) consider a hybrid multi-objective parallel genetic algorithm for a HFS with unrelated parallel machines. Sun and Gu (2017) consider a hybrid estimation of distribution algorithm for a HFS with unrelated parallel machines.

In literature, usually relatively small problems are investigated with a maximum of 20 machines in total and around 100 jobs (see Chen et al. (2015), Siqueira et al. (2018) and Jungwattanakit et al. (2007)). Exact approaches are almost always based on branch and bound and often have cases with less than 20 machines and 100 jobs. This is due to the fact that when there are more jobs and machines, the computation times get out of hand quickly (see for example Oğuz and Ercan (1997), which state that exact approaches get intractable already when there are more than 10 jobs in a two-stage flow-shop). This implies the use of heuristic approaches when the number of jobs, stages or machines increases (especially when we look at our large industry based case study, which contains 1.300 jobs and more than 100 machines).

In the HFS with unrelated parallel machines a solution is usually represented by means of a random keys representation. The random keys representation uses a vector with a size equal to the number of tasks in the first stage with an integer part and a fractional part to model the first stage of the HFS. The integer part denotes the machine that a task is assigned to and the sequence of tasks on the machines is determined according to the fractional part. The random keys representation will be explained in further detail in Section 4, where the methodology is explained in detail.

A decoding rule is used to determine the starting and ending time of the tasks corresponding to a job in the remaining stages after the first stages based on the random keys representation. For the HFS with unrelated parallel machines, list scheduling (LS) is commonly used as a decoding rule. In LS, which was introduced by Oğuz and Ercan (2005), the jobs are extrapolated through the rest of the stages using a "First Come First Serve" (FCFS) principle. A further explanation of LS will be given in Section 4.

2.2 Hybrid Flow Shop with multiprocessor tasks

Recently, research for the HFS with multiprocessor tasks has gotten more attention (Kurdi (2018), Xu et al. (2013), Chou (2013)). For this HFS-variant it holds, just as in the HFS-variant with unrelated parallel machines, that it is strongly NP-hard, since it can be easily reduced to a regular HFS model. Just as with the

HFS with unrelated parallel machines, mainly metaheuristics are being used to derive sub-optimal solutions (see Jouglet et al. (2009), Engin et al. (2011), Wang, Chou, and Wu (2011)).

In HFS with multiprocessor tasks a solution is usually represented by means of a job permutation encoding rule. The job permutation encoding rule uses a vector containing a sequence of jobs which denote the sequence in which they will be assigned to a machine according to a certain assignment rule to model the first stage in a HFS. Job permutation will be explained in further detail in Section 4, where we will also explain how job permutation can be adjusted such that it is able to deal with multiprocessor tasks. Examples of decoding rules that are commonly used in literature for the HFS with multiprocessor tasks include LS, for which a further explanation will be given in Section 4.

3 Exact Solution Approaches

In this section we will describe a mathematical formulation corresponding to the HFS-UMMTSDS to formally define the HFS-UMMTSDS and to be able to compare the results of heuristic approaches to optimal solutions for relatively small problem instances. First, we will look at the complexity class to which the HFS-UMMTSDS belongs. Then we will introduce the notation used for the sets, parameters and decision variables that we will use in the mathematical formulation, which will then be presented and explained in detail. As a base for our mathematical formulation we used the MIPs presented by Dai et al. (2013) and Chen et al. (2013). Furthermore, we will introduce a constraint programming formulation to solve the problem. Constraint programming formulations have been successfully implemented in scheduling environments (see for example Laborie et al. (2018)), which motivates the use of it for the HFS-UMMTSDS. Finally, we will introduce bounds that can speed up the process of solving the constraint programming formulation to optimality and that will give us an idea of the performance of heuristic solution approaches on large problem instances.

3.1 Problem Complexity

To show the complexity of our problem, we can take a look at special cases of the HFS considered by Gupta (1988) and Hoogeveen (1996). Gupta (1988) showed that the HFS with two stages and with only two or more machines in parallel in one stage and only one machine in the other stage is already NP-hard in the strong sense (see Lawler et al. (1993) for this proof) and adding stages would only add to the complexity. Hoogeveen (1996) states that the HFS with at least 2 machines in a stage is NP-hard, since finding the minimal maximum makespan on a parallel machine scheduling problem in a single stage is already NP-hard. Besides this, they show that the HFS with two stages and preemption is NP-hard in the strong sense. Thus, the problem that is shown to be NP-hard by Gupta (1988) can be denoted by $HF2(P^k)_{k=1}^{2}||C_{max}$, when described by the notation of Graham (Graham et al. (1979) and Vignier et al. (1999)).

Our problem can be easily reduced to the $HF2(P^k)_{k=1}^{2}||C_{max}$ by setting the number of stages to two, having equal production times on each machine for each job per stage, having only one multiprocessor task per job per stage and only considering the C_{max} in the multi-objective optimization objective by setting the weight of the other objective criteria to 0. This shows that our problem is also strongly NP-hard and that we therefore do not expect to obtain an algorithm that solves the problem to optimality in polynomial time. Furthermore, Du and Leung (1990) show that minimizing the total tardiness in the single machine is strongly NP-hard. The single machine problem can be relaxed to the parallel machine scheduling problem, which again can be relaxed to a HFS and our objective value can also easily be reduced to the total tardiness when setting the weight of all other objective value components to 0.

3.2 Mathematical Programming

To formally define our problem, we present a mathematical formulation. Besides this, we will use our mathematical formulation to show how quickly exact solution approaches can get out of hand from a computational perspective on small instances and to be able to get an idea about the performance of our heuristic solution approaches.

3.2.1 Sets and Parameters

Sets with corresponding indices

For our mathematical formulation the following sets with corresponding indices are used:

- Jobs J, indexed by j, j_1 and j_2
- Stages K, indexed by k
- Machines M_{ijk} in stage k that task i of job j can be executed on indexed by m
- Machines $\bar{M}_k$ in stage k, indexed by m
- Tasks I_{jk} for job j in stage k, indexed by i, i_1 and i_2

Parameters

For our mathematical formulation the following parameters are used:

- Setup time when task i_1 of job j_1 precedes task i_2 of job j_2 directly on machine m in stage, denoted by k $s_{i_1 i_2 j_1 j_2 km}$
- Production time of task i of product j on machine m in stage k, denoted by p_{ijkm}
- Due date of job j, denoted by d_j
- Penalty associated with tardiness ω_t
- Penalty associated with earliness ω_e
- Penalty associated with flow time ω_f
- Penalty associated with maximum completion time ω_c
- A large number L

3.2.2 Decision variables

The following decision variables will be used within our proposed mathematical formulation:

- Starting time of task i belonging to job j in stage k on machine m, denoted by S_{ijkm}
- Starting time of production of job j, denoted by $\tilde{S}_j$
- Completion time of task i belonging to job j in stage k on machine m, denoted by C_{ijkm}

- Completion time of job j in stage k, denoted by $\tilde{C}_{jk}$
- Earliness of job j, denoted by E_j
- Tardiness of job j, denoted by T_j
- Flow time of job j, denoted by F_j
- Maximum completion time denoted by C_{max}
- Binary decision variable, denoting whether task i of job j is assigned to machine m in stage k X_{ijkm}
- Binary decision variable, denoting whether task i_1 of job j_1 precedes task i_2 of job j_2 on machine m in stage k $Y_{i_1 i_2 j_1 j_2 km}$

3.2.3 Mixed Integer Programming Formulation (MIP)

With the defined parameters and decision variables described above, we can setup the following mathematical model for the HFS-UMMTSDS:

$$\min \quad \omega_t \sum_{j\in J} T_j + \omega_e \sum_{j\in J} E_j + \omega_f \sum_{j\in J} F_j + \omega_c C_{max} \tag{1}$$

$$\sum_{m\in M_{ijk}} X_{ijmk} = 1 \qquad \forall k \in K, j \in J, i \in I_{jk}, \tag{2}$$

$$X_{ijmk} = 0 \qquad \forall k \in K, j \in J, i \in I_{jk}, m \in \bar{M}_k \backslash M_{ijk}, \tag{3}$$

$$S_{ijkm} \geq \tilde{C}_{jk-1} \qquad \forall i \in I_{jk}, j \in J, k \in K\backslash\{0\}, m \in M_{ijk}, \tag{4}$$

$$C_{ijkm} = S_{ijkm} + p_{ijkm} X_{ijkm} \qquad \forall i \in I_{jk}, j \in J, k \in K, m \in M_{ijk}, \tag{5}$$

$$C_{i_1 j_1 km} + s_{i_1 i_2 j_1 j_2 km} \leq S_{i_2 j_2 km} + L(1 - Y_{i_1 i_2 j_1 j_2 km}) \qquad \forall (j_1, j_2) \in J^2, i_1 \in I_{j_1 k}, i_2 \in I_{j_2 k}, \forall k \in K, m \in \bar{M}_k, \tag{6}$$

$$Y_{i_1 i_2 j_1 j_2 km} \leq X_{i_1 j_1 km} \qquad \forall (j_1, j_2) \in J^2, i_1 \in I_{j_1 k}, i_2 \in I_{j_2 k}, \forall k \in K, m \in \bar{M}_k, \tag{7}$$

$$Y_{i_1 i_2 j_1 j_2 km} \leq X_{i_2 j_2 km} \qquad \forall (j_1, j_2) \in J^2, i_1 \in I_{j_1 k}, i_2 \in I_{j_2 k}, \forall k \in K, m \in \bar{M}_k, \tag{8}$$

$$\sum_{j\in J, i\in I_{jk}} X_{ijkm} - 1 \leq \sum_{(j_1,j_2)\in J^2, i_1\in I_{j_1 k}, i_2 \in I_{j_2 k}} Y_{i_1 i_2 j_1 j_2 km} \qquad \forall k \in K, m \in \bar{M}_k, \tag{9}$$

$$\sum_{j_1\in J, i_1\in I_{j_1 k}} Y_{i_1 i_2 j_1 j_2 km} \leq 1 \qquad \forall i_2 \in I_{j_2 k}, j_2 \in J, k \in K, m \in M_{i_2 j_2 k}, \tag{10}$$

$$\sum_{j_2\in J, i_2\in I_{j_2 k}} Y_{i_1 i_2 j_1 j_2 km} \leq 1 \qquad \forall i_1 \in I_{j_1 k}, j_1 \in J, k \in K, m \in M_{i_1 j_1 k}, \tag{11}$$

Equation (1) states the objective value of our mathematical model. Here we see a multi-objective optimization criterion with respectively the total tardiness, total earliness, total flow time and maximum completion time. In constraint set (2) and (3) we ensure that each task is assigned to exactly 1 machine out of the available machines that a task can be processed on and we prevent that a task is assigned to a machine that it can't be processed on. With constraint set (4) we model that we cannot start with processing a task of a job in a stage before all the tasks of the job in the previous stage have finished their operation. With constraint set (5) the completion time of a task is set equal to the starting time of that task plus its production time on the assigned machine. In constraint set (6) it is stated that the starting time of a task needs to be larger or equal to the completion time of the preceding task plus the sequence dependent setup times (note that the completion time and the starting time may also be not real, if the task is not executed on that machine). Constraint set (7) together with constraint set (8) make sure that the Y-variables can only be used to determine the sequence between tasks that are actually assigned to the machine concerned in that stage. To make sure that there are enough Y-variables that take on a value of one on a machine in a stage such that a sequence of tasks is created on this machine, constraint set (9) is used. The constraint set (10) makes sure that each task can have at most one direct predecessor while constraint set (11) makes sure that each task can have at most one direct successor.

$$\tilde{C}_{jk} \geq C_{ijkm} \qquad \forall i \in I_{jk}, j \in J, k \in K, m \in M_{ijk}, \tag{12}$$
$$C_{max} \geq \tilde{C}_{j|K|} \qquad \forall j \in J, \tag{13}$$
$$\tilde{S}_j \leq S_{ij0m} \qquad \forall i \in I_{jk}, j \in J, m \in M_{ij0}, \tag{14}$$
$$E_j \geq \tilde{C}_{j|K|} - d_j \qquad \forall j \in J, \tag{15}$$
$$T_j \geq d_j - \tilde{C}_{j|K|} \qquad \forall j \in J, \tag{16}$$
$$F_j \geq \tilde{C}_j - \tilde{S}_j \qquad \forall j \in J, \tag{17}$$

Constraint sets (12) – (17) are used to determine the objective values of the HFS-UMMTSDS. In constraint set (12) the completion time of a job within a stage is set. Constraint set (13) sets the maximum completion time. The starting time of a job is ensured by constraint set (14). Constraint set (15) tracks the earliness of a job, (16) the tardiness and (17) the flow time. Furthermore, we have the following domain restrictions on our decision variables ensured by constraint sets (18)-(24).

$$X_{ijkm} \in \mathbb{B} \qquad \forall i \in I_{jk}, j \in J, k \in K, m \in \bar{M}_k, \tag{18}$$
$$Y_{i_1 i_2 j_1 j_2 km} \in \mathbb{B} \qquad \forall i_1 \in I_{j_1 k}, i_2 \in I_{j_2 k}, (j_1, j_2) \in J^2, k \in K, m \in \bar{M}_k, \tag{19}$$
$$S_{ijkm} \geq 0 \qquad \forall i \in I_{jk}, j \in J, k \in K, m \in \bar{M}_k, \tag{20}$$
$$S_j \geq 0 \qquad \forall j \in J, \tag{21}$$
$$F_j \geq 0 \qquad \forall j \in J, \tag{22}$$
$$T_j \geq 0 \qquad \forall j \in J, \tag{23}$$
$$E_j \geq 0 \qquad \forall j \in J \tag{24}$$

3.3 Constraint Programming

Constraint programming has been proven to be useful in applications with highly constrained problems such as timetabling, sequencing and scheduling problems (see, for example, Kroon et al. (2009) and Laborie et al. (2018)). Since we are dealing with a highly complicated scheduling problem, a successful implementation of constraint programming might yield insightful results.

In constraint programming a problem is usually modelled as a Constrain Satisfaction Problem (CSP), which is a problem that tries to find a feasible solution given a set of constraints on decision variables. Here, the problem is formulated by using 3 different data types: variables, domains which limit the range of the variables and constraints on the variables. So far, this might not look too different from mathematical programming. However, within constraint programming the aim is usually to reduce the search space by resetting the domains due to all constraints that are set. One of the great benefits of constraint programming is that it makes use of so-called global constraints to be able to effectively reduce the search space. For these global constraints, dedicated methods have been developed to restrict the feasible region of the variables. An example of an often used global constraint is the *AllDifferent(x,y,z)* constraint, in which the values of all variables that are specified (in this case x, y, z) need to be different from one another. To cope with this global constraint, the observation has been made that a bipartite matching problem can be formulated just as the *AllDifferent* constraint (see Régin (1994)) and thus, also its solution approaches can be used.

One can say that the general idea behind constraint programming is that it finds feasible solutions rather than an optimal solution and that it works best for highly constrained problems. The constraint program could be modelled in such a way that all constraints as we defined in Section 1 would be satisfied and an optimization procedure could be executed by iteratively adding a constraint to the model that requires to come up with a better solution than in the previous iteration.

3.3.1 Formulation

We base the constraint programming formulation on the formulations setup by Zeballos et al. (2011), which gives a constraint programming formulation for the planning of products in batches in a multistage setting and Castro et al. (2006), which addresses how to deal with sequence and machine dependent setup times in constraint programming.

Parameters

In the constraint programming formulation we will make use of the same sets and indices as defined in the mathematical formulation in Section 3. We add the following parameters with corresponding indices:

- $Transition_{km}$ sequence dependent setup time between two tasks on machine m in stage k where the matrix contains all the tasks on both dimensions with the sequence dependent setup time on the interfering cell

Variables

In constraint programming one can make use of different types of variables for modelling purposes. The first type is a regular variable which is very much similar to the variables used in a mathematical programming formulation, although setting the bounds of these variables as tight as possible can be of significant importance. Another variable type that can be used is the so-called interval variable, which is mainly used in scheduling problems and has several attributes associated such as the start, end, size, length and availability of the interval. In these attributes, the start and the end denote the start and the end of that interval. The size denotes the difference between the start and the end of the interval. The length denotes the period of time in which the interval variable is actually being worked on, which does not have to be equal to the size of an interval variable (e.g. in case of preemption, a product does not necessarily need to be produced in one go, but it could be produced in multiple shifts). Furthermore, the availability can be used to define optional interval variables which do not necessarily need to be used in the final solution (e.g. if one needs to assign a task to a machine, one could create an optional variable for each machine and then require that one of those optional variables needs to be present in the final solution).

Finally, there are sequence variables which contain interval variable(s). This type of variable is used to track the sequence of those interval variables. With the sequence variable there is also a special type of parameter that can be used for these sequence variables, namely the transition matrix. This transition matrix can, for example, be used to determine that there are sequence dependent setup times on a machine.

In our formulation we will use the following variables:

- $Task_{ijk}$, interval variable that refers to task i of job j in stage k. These interval variables are not optional and must therefore all be present in the final solution
- M_Task_{ijkm}, task i of job j in stage k on machine m. These are all optional interval variables with a size equal to the processing time of task i of job j in stage k on machine m
- $Sequence_{km}$, sequence variable which contains all the M_Task_{ijkm} interval variables per machine m in stage k
- E_j, regular variable denoting the earliness of job j (an obvious bound on this variable would be the due date of the job)
- T_j, regular variable denoting the tardiness of job j (an obvious bound on this variable would be the sum of all production times and setup times)
- F_j, interval variable denoting the flow time of job j
- C_{max}, interval variable denoting the maximum completion time

Constraint Programming Formulation

$$\min \quad \sum_{j\in J} (\omega_e E_j + \omega_f F_j.size + \omega_t T_j) + \omega_c C_{max} \tag{25}$$

$$alternative(Task_{ijk}, M\_Task_{ijkm} \quad \forall m \in M_{ijk}) \qquad k \in K, j \in J, i \in I_{jk}, \tag{26}$$

$$No\_overlap(Sequence_{km}, Transition_{km}) \qquad \forall k \in K, m \in \bar{M}_k, \tag{27}$$

$$end\_before\_start(M\_Task_{ijk-1m}, M\_Task_{ijkm'}) \qquad \forall j \in J, k \in K\backslash\{0\}, i \in I_{jk}, (m,m') \in \bar{M}_k^2, \tag{28}$$

$$F_j = span(Task_{jkm} \quad \forall k \in K, m \in \bar{M}_k) \qquad \forall j \in J, \tag{29}$$

$$E_j = \max(d_j - \max_{i\in I_{j|K|},} (Task_{ij|K|}.end), 0) \qquad \forall j \in J, \tag{30}$$

$$T_j = \max(\max_{i\in I_{j|K|}} (Task_{ij|K|}.end) - d_j, 0) \qquad \forall j \in J, \tag{31}$$

$$C_{max} = \max_{j\in J, i\in I_{j|K|}} (Task_{ij|K|}.end) \tag{32}$$

Explanation Formulation

Equation (25) states the objective value of the formulation in which the flow time is derived by the duration of the processing job. In the objective value F_j is an interval variable, from which respectively the size attribute is accessed by $F_j.size$. The other attributes corresponding to an interval variable can be accessed in a similar way. In constraint set (26) it is stated that a task needs to be executed on at least one of the machines that it is eligible to be executed on. Furthermore, in constraint set (27) we ensure that there is no overlap between tasks on machine, taking into account also the sequence dependent setup times that are defined in the transition matrix. In the *no_overlap*() global constraint, interval variables are forced to be non-overlapping. In case optional variables are specified within the constraint, it only enforces interval variables to be non-overlapping when they are present in the final solution. The constraint set (28) is there to force the tasks to be able to start only if all the tasks corresponding to the same job in the previous stage have all been processed.

Finally constraint sets (29)-(32) are used to track the objective values of the resulting planning. Here, constraint set (29) is used for the flow time of each job making use of the global constraint *span*() which enforces an interval variable to have the minimum start and maximum end of all the interval variables that are defined in the constraint. The earliness of a job is set by constraint set (30), constraint set (31) is used for the tardiness of a job and constraint (32) is used for the maximum completion time. An idea to improve the performance of solving the formulation stated above can be to derive bounds on the objective value of the problem to be able to further reduce the search space.

3.4 Bounds on objective value

We will present both an upper and lower bound on the objective value of our problem. The upper bound can be used to speed up the performance of solving the constraint programming formulation to optimality, since this upper bound can quickly reduce the range of the domains of the decision variables. We can use the lower bound to evaluate the performance of heuristic solution approaches on large problem instances that cannot be solved to optimality.

3.4.1 Upper Bound on objective value

To create an upper bound on the objective value, a strategy may be to create a feasible schedule by means of a greedy algorithm such that this feasible schedule is generated in a fast way. An advantage of creating an upper bound by means of a greedy heuristic, is that we also can use the upper bound to evaluate the performance of our heuristic solution approaches. For the HFS-UMMTSDS a simple way to create a feasible schedule is by means of assigning the tasks to the machine on which it has the least processing time, then ordering the processing of the tasks on the machine by means of the due date that it has.

To assign tasks to machine with lowest processing time, we need to take the minimum of the processing time over all machines in that stage that the task can be processed on for each task. The assignment of all tasks to machines can be done in $\mathrm{O}(\sum_{k\in K}\sum_{j\in J}|I_{jk}||m_k|)$, since in this case for every task its processing time on each machine is determine and compared. The determination of the sequence on each machine can be done in $\mathrm{O}(\sum_{k\in K}\sum_{j\in J}|I_{jk}|\log(|I_{jk}|)|m_k|)$ by applying the sorting of tasks on each machine. Here, the worst case scenario would be that all tasks are assigned to one machine in a stage. A sorting algorithm takes $\mathrm{O}(nlog(n))$ to compute, for example by means of a heapsort, which concludes the computation time needed.

When all tasks are assigned to a machine, we only still need to evaluate the objective value associated with this feasible schedule. This takes $\mathrm{O}(\sum_{k\in K}\sum_{j\in J}|I_{jk}|)$ time, which can be done my means of a recursive function that determines that completion time of each job. The recursive function can be formally described as follows: $C_{ijk} = \max\{\max_{i\in I_{jk-1}}\{C_{ijk-1}+p_{ijkm}\}, C_{i'jk}+p_{ijkm}+s_{ii'jj_{i'}km}\}$ in which m refers to the machine to which the task was assigned in the previous phase of our algorithm and i' refers to the predecessor of task i that we determined in the previous phase of our algorithm. After we determine the completion times by means of the dynamic program, the earliness of a job can be computed from the following equation: $E_j = \max\{d_j - \max_{i\in I_{j|K|}}\{C_{ij|K|}\}, 0\}$, the tardiness from $T_j = \max\{\max_{i\in Ij|K|}\{C_{ij|K|}\} - d_j, 0\}$, and the flow time from $F_j = \max_{i\in I_{j|K|}}\{C_{ij|K|}\} - \min_{i\in I_{j0}}\{C_{ij0} - p_{ij|K|0}\}$, in which m again denotes the machine to which task i was assigned in the previous phase of our algorithm, the C_{max} can be computed by $\max_{j\in J, i\in I_{jk}}\{C_{ijk}\}$. The C_{ijk} are initialized for $k = 0$ with value 0. This leads to a dynamic program with $|K|$ recursions with $|J|$ * $|I_{jk}|$ variables which all require $|I_{jk}|$ iterations to compute, yielding a total computation time of $\mathrm{O}(\sum_{k\in K}\sum_{j\in J}|I_{jk}|)$.

From the dynamic program the objective values can in its turn be calculated in $\mathrm{O}(\sum_{j\in J}|I_{j|K|}|)$ time, since the earliness takes $\mathrm{O}(\sum_{j\in J}|I_{j|K|}|)$ time, the tardiness $\mathrm{O}(\sum_{j\in J}|I_{j|K|}|)$ time, the flow time takes $\mathrm{O}(2\sum_{j\in J}|I_{j|K|}|)$ time and the maximum completion time takes $\mathrm{O}(\sum_{j\in J}|I_{j|K|}|)$ time and $\mathrm{O}(4\sum_{j\in J}|I_{j|K|}|)$ has the same time complexity as $\mathrm{O}(\sum_{j\in J}|I_{j|K|}|)$. This way, our proposed algorithm yields an upper bound on our problem while having a time complexity of $\mathrm{O}(\sum_{k\in K}\sum_{j\in J}|I_{jk}||m_k| + \sum_{k\in K}\sum_{j\in J}|I_{jk}|log(|I_{jk}|)|m_k| +$

$\sum_{k\in K}\sum_{j\in J}|I_{jk}|+\sum_{j\in J}|I_{j|K|}|)$ which can also be stated as O($\sum_{k\in K}\sum_{j\in J}|I_{jk}|log(|I_{jk}|)|m_k|$) and is therefore within polynomial time.

3.4.2 Lower Bound on objective value

The lower bound on the objective value will be used for large problem instance that cannot be solved to optimality, to have an idea about the performance of heuristic solution approaches. We will obtain our lower bound by bounding each of the components of the objective value.

$$\omega_f \sum_{j\in J} F_j + \omega_e \sum_{j\in J} E_j + \omega_t \sum_{j\in J} T_j + \omega_c C_{max} \tag{33}$$

$$= \omega_f \sum_{j\in J}(C_j - S_j) + \omega_e \sum_{j\in J}\max(d_j - C_j, 0) + \omega_t \sum_{j\in J}\max(C_j - d_j, 0) + \omega_c \max_{j\in J}(C_j) \tag{34}$$

$$\geq \omega_f \sum_{j\in J}(C_j - S_j) + \omega_e \max(\sum_{j\in J} d_j - \sum_{j\in J} C_j, 0) + \omega_t \max(\sum_{j\in J} C_j - \sum_{j\in J} d_j, 0) + \omega_c \max_{j\in J}(C_j) \tag{35}$$

In our lower bound we can make use of the derivation stated in Equations (33)-(35). Equation (34) follows from Equation 33 by filling in the definitions of the flow time, earliness, tardiness and maximum completion time respectively. Then, Equation (35) can be derived from moving the maximum operator in the earliness and tardiness operator out of the summation.

$$F_j \geq \sum_{k\in K} \max_{i\in I_{jk}} \min_{m\in M_{ijk}} p_{ijkm} \tag{36}$$

$$SPT_j = \frac{\sum_{i\in I_{jk_b}} \min_{m\in M_{ijk_b}} p_{ijk_bm}}{|m_{k_b}|} \tag{37}$$

$$C_j \geq F_j = C_j - S_j \tag{38}$$

Each objective value component can be bounded by looking at the expression in (35). The flow time can be bounded from below by the expression stated in (36). In Equation (36) we have a summation over the number of stages where in each stage a job needs at least the production time of the multiprocessor task that has the highest production time when the machine is selected for all multiprocessor tasks that can process the tasks in the least amount of time. When looking at the tardiness, we can observe that this can be bounded in two ways. We can bound the tardiness by bounding the completion time of an individual job or by bounding the sum of all completion times. The final bound on the tardiness can be obtained by taking the maximum of both resulting tardiness bounds. Note that we can derive a bound for the maximum completion time from both the bounds on the completion times as well and that we can take the maximum out of those to use in our final bound. Finally, the earliness of a stage can be bounded by maximizing the completion time. Strictly speaking there is not a maximum for the completion time. However, the value of earliness can easily be bounded from below by 0, due to the definition of earliness.

The completion time of a job can be bounded in two ways. Both bounds on the completion time will be evaluated on their resulting tardiness and we will use the maximum resulting tardiness within the lower bound. The first bound is given in Equation (38) which denotes a bound for the individual completion time of a job. This simply states that a job will always have a higher completion time than its flow time. For the other bound, we bound the total completion time. To bound the total completion time, we make use of the property that the total completion time ,in case of a parallel machine scheduling, problem is minimized by applying a shortest processing time first (SPT) rule (see for example Pinedo (2012) for a formal proof on this). This bound will be useful when there are relatively many jobs compared to the number of machines in a stage. We can apply the SPT rule in the bottleneck stage and then assume that there is ample capacity in the stages before and after the bottleneck for production.

To determine the processing time of a job for the SPT rule, we use the processing time of a job in the bottleneck stage as stated in Equation (37). In Equation (37) we spread all processing time needed over all machines in a stage, where k_b denotes the bottleneck stage. In the HFS-UMMTSDS we will assume integer processing times, which allows us to ceil the resulting processing time, as can be seen in our final bound in (39). Furthermore, we add the minimum time that a job needs to be processed in all remaining stages to its completion time. For each job this is equal to at least $\sum\limits_{k \in K \setminus k_B} \max\limits_{i \in I_{jk}} \min\limits_{m \in M_{ijk}} p_{ijkm}$. If no bottleneck can be pointed out up front, all stages can be tried as a bottleneck to test what yields the highest total completion time.

In Equation (39) our lower bound resulting from Equations (33)–(38) is represented. In this equation, J_o denotes the ordered set of jobs, sorted according to the SPT rule presented in (37) in ascending order. Furthermore, ord(j) refers to the location of job j in the ordered set J_o.

$$
\begin{aligned}
LB = {} & \omega_f \sum_{j \in J} \max_{i \in I_{jk}} \min_{m \in M_{ijk}} p_{ijkm} + \omega_t \max \Bigg(\Big(\sum_{j \in J_o} \left\lceil (|J| - \text{ord}(j) + 1) \frac{\sum\limits_{i \in I_{jk_b}} \min\limits_{m \in M_{ijk_b}} p_{ijk_bm}}{|m_{k_b}|} \right\rceil + \\
& \sum_{j \in J} \sum_{k \in K \setminus k_B} \max_{i \in I_{jk}} \min_{m \in M_{ijk}} p_{ijkm} - \sum_{j \in J} d_j \Big), \Big(\sum_{j \in J} \max(\max_{i \in I_{jk}} \min_{m \in M_{ijk}} p_{ijkm} - d_j, 0) \Big) \Bigg) + \\
& \omega_c \max \Big(\sum_{j \in J} \left\lceil \frac{\sum\limits_{i \in I_{jk_b}} \min\limits_{m \in M_{ijk_b}} p_{ijk_bm}}{|m_{k_b}|} \right\rceil, \max_{j \in J} (\max_{i \in I_{jk}} \min_{m \in M_{ijk}} p_{ijkm}) \Big)
\end{aligned} \tag{39}
$$

As can be seen, setup times are not taken into account in our lower bound and it will in our opinion be hard to implement setup times. One challenge that needs to be overcome, is that it is unclear which tasks suffer from setup times, since a task does not encounter setup times if it is the only task on a machine. Furthermore, the amount of setup times present, will depend on the sequence of tasks on a machine. To determine the minimum amount of setup time required based on a sequence of tasks one needs to solve a single machine scheduling problem with sequence dependent setup times.

4 Heuristic Solution Approaches

As has been stated in our literature review in Section 2, researchers obtained the most promising results when applying metaheuristics on the HFS. This motivates the use of a genetic algorithm for solving the multi objective HFS-UMMTSDS. Furthermore, we stated in Section 3 that in a much more general setting of scheduling problems, constraint programming has been proven to yield useful results. Besides the separated use of the genetic algorithm and constraint programming, we also investigate the combination of those two in the form of a memetic algorithm. Researchers obtained promising results on a case of a HFS with multiprocessor tasks by using a combination of a genetic algorithm and constraint programming (Jouglet et al. (2009)). Besides the combination of a genetic algorithm and constraint programming, we also study a heuristic that combines a genetic algorithm with a linear programming formulation.

In this section we will first introduce genetic algorithms in general together with the encoding and decoding possibilities that we will consider in our research. Afterwards, we will propose the combination of a genetic algorithm with a constraint programming formulation in the form of a memetic algorithm. Finally, we will introduce the possibility of combining a genetic algorithm with a linear programming formulation in the form of a memetic algorithm.

4.1 Genetic Algorithm

Genetic algorithms are popular metaheuristics that are inspired by the evolution theorem and were first introduced by Holland et al. (1992). In Algorithm 1 we present the general outline of a genetic algorithm, in which p_m denotes the probability that a mutation procedure is being executed, $PopSize$ denotes the size of the population and $f(x)$ denotes the objective value which is associated with a certain chromosome x. We start by initializing each chromosome until a population of $PopSize$ is created. Subsequently, the best and the worst chromosome are recorded. Note that the hyper parameters $PopSize$ and p_m will need to be determined up front, which we will do by testing different values on a test data case. We will refer to these test runs as hyper parameter runs.

After the initialization a loop is started, which lasts until a certain stopping criterion is met (e.g. maximum computation time reached, or a number of iterations without an improvement in the objective value). In each iteration of this loop, 2 chromosomes are selected which are used to produce a new chromosome. If the objective value of the newly created chromosome is better than the best value, we replace one chromosome in the current population with the newly created chromosome. If the objective value of the newly created chromosome is not better than the best value, we mutate the newly created chromosome with a certain possibility. Afterwards, we check whether the objective value of the chromosome is better than the worst value out of the current population. If the objective value is not better, the newly created chromosome is discarded and otherwise it replaces a chromosome from the current population and we repeat the process. We will further elaborate on the methods used for selection, crossover generation, mutation, replacing and the local search method in this section.

Algorithm 1 Genetic algorithm

```
1: procedure Genetic algorithm($PopSize, p_m$)
2:     Initialize population of size $PopSize$ of chromosome
3:     $b \leftarrow$ best chromosome and $w \leftarrow$ worst chromosome
4:     while Stopping criteria not yet met do
5:         Select 2 chromosomes from the population
6:         Create offspring called $n$, with the 2 selected chromosomes by means of a crossover procedure
7:         if $f(n) \leq f(b)$ then
8:             Mutate $n$ with probability $p_m$
9:         if $f(w) \leq f(n)$ then
10:            Discard $n$
11:        else
12:            Replace $r$ with $n$, where $r$ is selected by by means of reverse binary tournament selection
13:            Update best and worst chromosome
14:    return $b$
```

4.1.1 Encoding

Since the feasible region of the HFS-UMMTSDS is extremely large, regular solution representations which contain all the information about a solution might not work very well. Therefore, we use a more tailored representation structure to encapture the specific characteristics (see for an example of a tailored neighborhood structure in HFS the paper of Bozorgirad and Logendran (2016)). A common way of reducing the search space by using a tailored representation scheme in HFS settings for metaheuristic purposes is by means of an encoding rule. Two examples of encoding rules used in practice are the job permutation encoding rule and the random keys representation. These encoding rule describes the sequence of the tasks within the first stage. A decoding rule, which we will elaborate further on, is then used is then used to schedule the tasks in the remaining stages. To increase our chances of success, we want to compare two different types of neighborhood structures in the genetic algorithm.

Random keys representation

The random keys representation (RK) has been introduced by Bean (1994) and has since been successfully applied within HFS scheduling problems with unrelated parallel machines (see, for example, Rashidi et al. (2010) and Gholami et al. (2009)). A random keys representation can be used for our problem with a vector with the size of the number of tasks to be executed in that stage, containing a real number in which the integer part describes the machine to which a task is assigned and the fractional part is used to determine in which order the tasks are being processed (a task with the lowest fractional part will be processed the earliest).

For example, if one would have a chromosome which is encoded by the random keys representation with the following vector: (1.56, 2.63, 2.42, 2.53, 1.12, 1.06), this would translate into a schedule that is presented in Figure 3. This example is based on the data that we used earlier for the example of a feasible solution to the HFS-UMMTSDS in Section 1 and uses Table 1 and Table 2. A potential benefit of using a random keys representation over a job permutation encoding rule might be that one only needs one vector to be able to encode the HFS by means of a random keys representation, while a job permutation encoding rule needs an additional procedure to assign jobs to machines in order to present a solution in full. To reduce the search space and to prevent for any infeasibilities in a solution with this representation scheme, one could assign the set of machines that a task is eligible to be executed on to choose from.

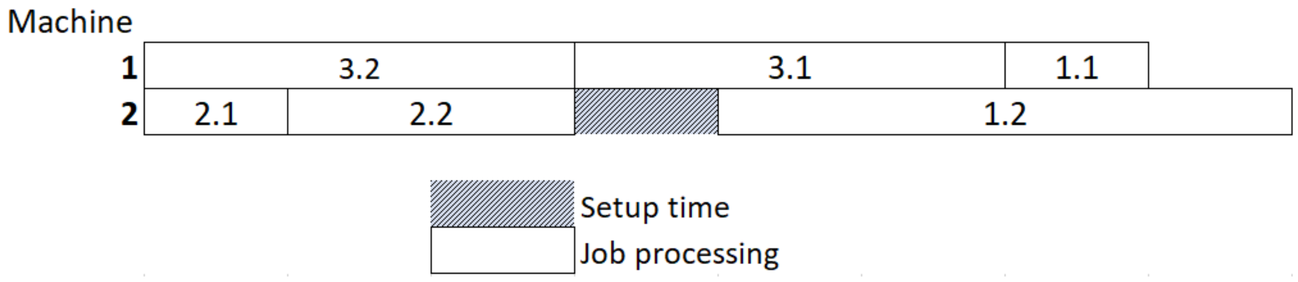

Figure 3: Example of the random keys representation encoding rule.

Job permutation

A job permutation encoding rule has been the mostly used encoding rule in HFS settings with multiprocessor tasks. An analysis of different job permutation (JP) representation schemes used within a genetic algorithm (GA) to solve a HFS with unrelated parallel machines is given by Urlings et al. (2010). In a job permutation schedule there is a vector with a size that is equal to the number of tasks to be processed in the first stage in which the order of the task IDs presents the order to which tasks are assigned to a machine (so in addition to this encoding rule one needs a machine assignment rule).

A commonly used machine assignment rule is the Earliest Completion Time (ECT) rule, which we will use as well. In case we have 6 tasks to be processed, in which task 2 cannot be executed on machine 2, a chromosome which is encoded by the job permutation scheme with the following vector: (2.2,1.2,1.1,3.2,2.1,3.1) with as machine assignment rule ECT would lead to the schedule presented in Figure 4. Again, we use the data presented in Table 1 and Table 2.

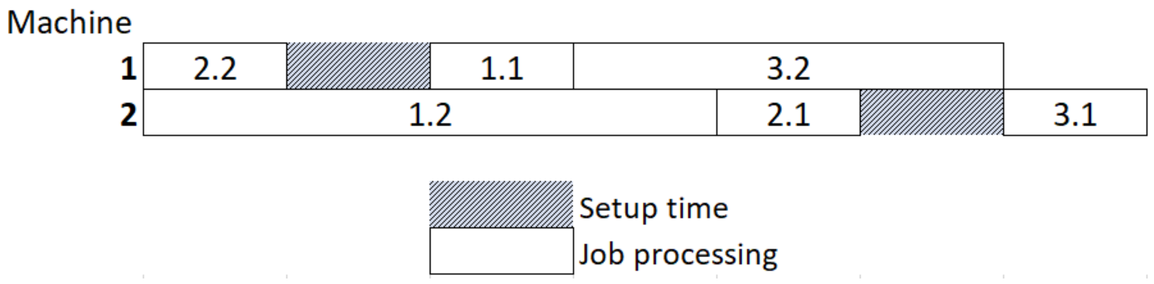

Figure 4: Example of the job permutation representation encoding rule.

4.1.2 Initialization and Selection Procedure

A common method to initialize all chromosomes within the population would be by doing so randomly. The implementation of a random initialization is slightly different for the random keys representation compared to the job permutation encoding rule. In the random keys representation the random initialization can be done by selecting a random machine for each task on which it should be executed and add a random value which is generated between 0 and 1 from a uniform distribution for the fractional part of the random keys representation. A job permutation encoding rule can be randomly initialized by a random shuffle of all tasks.

To know which chromosomes are used as parent chromosomes to produce offspring, a selection procedure is used. The strategy that is often used as a selection procedure within a GA is random selection. With the random selection, two chromosomes are being chosen randomly in order to produce offspring.

4.1.3 Crossover Procedure

The crossover procedure is used to create new chromosomes such that the population gets further diversified and good structures from the current population are exploited to eventually move to an optimal solution. Procedures which are often found in literature are: one point crossover, two point crossover, uniform crossover. These procedures can be easily incorporated for the random keys representation encoding rule.

As stated above one point crossover, two point crossover and uniform crossover can be used for the random keys representation encoding rule. The one point crossover procedure randomly selects a gene in the chromosome and then adds all the genes to the left of the selected gene from the first parent chromosome to the offspring and the rest is added from the other parent. The two point crossover is quite similar to the one point crossover, but now two genes are randomly selected instead of one. Then, from one parent the genes on the right from the first gene and the genes on the right from the second gene are added to the offspring, while the genes in between the selected genes are added from the other parent. In uniform crossover for each gene of the offspring one of the genes of the parent chromosomes is selected on the same location with equal probability. If it is desired that multiple offspring are created, the parents can be interchanged such that two different types of offspring are created. We will test the three described crossover procedures when determining the hyper parameters needed for the GA.

The crossover procedure for the job permutation encoding rule is a little more difficult since infeasibilities can easily arise when the crossover methods mentioned in the paragraph above are applied. In literature, the Partially Matched Crossover (PMX, introduced by Goldberg et al. (1985)) operator is mainly applied when a job permutation encoding rule is used. PMX can be viewed as an extension to the two-point crossover operator such that the offspring is a feasible solution to the problem. With PMX also two random genes are selected, the genes between the selected genes are selected from the first parent chromosome and copied to the offspring. Then we look at the genes that are in between the selected genes of the second parent that are not in the offspring yet. These genes are added to the offspring in the same location as the offspring as they are in the second parent. If one would want to make two offspring from two parents, the parents can simply be interchanged.

For example, when PMX crossover is applied with as parent 1: (5,4,3,2,1) and as parent 2: (1,2,3,4,5) where the selected indices for creating an initial offspring are 2 and 3, this would result in the first iteration the following offspring: (-,4,3,-,-). In the second iteration we observe that in the offspring that we copied 2 is not in the offspring, but is within the indices of parent 2. Now, we check which number is at the index where 2 is located in parent 1, which in this case is 4. We locate 4 in parent 2 and we notice that this index is not covered by the offspring yet, so we add 2 at this position, such that the offspring becomes: (-,4,3,2,-). Now, all the numbers between the selected indices in parent 2 are also in the offspring, which means that we can simply copy the rest of parent 2 to the remaining blank spots in the offspring. The final offspring then becomes: (1,4,3,2,5).

4.1.4 Mutation Procedure

The mutation procedure is used to prevent premature convergence among the genetic algorithm. For the mutation procedure there is, just as with the crossover operation, a distinction between the procedures that can be used for the job permutation encoding and the random keys representation encoding rule. A uniform mutation can be used for the random keys representation. The uniform mutation selects a certain number of genes in a chromosome randomly and then adjusts the information in that gene. In our case, this would entail the mutation of a gene such that the task that is represented by this gene is executed on a different machine. The fractional part in a gene is also adjusted by regenerating a new random number between 0 and 1 from a uniform distribution.

For the job permutation encoding scheme, the following mutation procedures can be used: insertion, swap, scramble and inversion. The insertion mutation rule selects two genes randomly and inserts the last gene directly next to the first gene and moves the rest of the genes one place to the right. Scramble mutation selects two genes randomly and then shuffles the location of all the genes between the selected genes randomly. With inversion mutation, a similar procedure is done, in which two genes are selected randomly again, but now the location of the genes between the two selected genes are inverted instead of shuffled. We will test all the mutation procedures when determining the hyper parameters needed for the GA and choose the mutation procedure that performs best for the job permutation encoding rule.

4.1.5 Replacing procedure

If the offspring is accepted into the population, a method needs to determine which chromosome is replaced by the offspring, such that the size of the population remains manageable. Our method will only depend on the objective value associated with a chromosome and therefore no distinction needs to be made between the job permutation and random keys representation encoding rules. A popular procedure used for determining which chromosome to replace within a GA is by means of reverse binary tournament selection. This method selects two chromosomes randomly and then chooses the chromosome with the worst objective value out of these two to be replaced. Always replacing the worst chromosome out of the population usually results in a GA that converges too quickly, which motivates the use of reverse binary tournament selection.

4.1.6 Decoding rules

To schedule all tasks of all jobs at the stages after the stage that is modelled by the encoding rule, a decoding rule is needed. A decoding rule which is commonly used in literature is List Scheduling (LS). However, LS is designed for a standard HFS setting without multiprocessor tasks. We will explain here how the methods can be adapted to be able to cope with an HFS with multiprocessor tasks.

List scheduling

In list scheduling, jobs are scheduled to the machine that can process a job as soon as possible when it has finished processing in the previous stage (this can be referred to as a first come first serve principle as well). Ties can be broken by choosing the machine with the shortest processing time plus least amount of setup time required. This decoding rule can be easily extended to be able to deal with multiprocessor tasks. A task can start in the next phase as soon as possible after all tasks in the previous stage of that job have been completed. The order in which tasks are assigned (since all tasks of the same job are available at the same time in the next stage) by assigning the task with the shortest processing time and setup time on that machine such that all tasks will be finished in that stage as soon as possible.

4.2 Memetic algorithm

The memetic algorithm (MA) has been introduced by Moscato (2000) and is closely related to the genetic algorithm, but differs mainly in the sense that it tries to prevent premature convergence by adding a local search method to its procedure. Neri and Cotta (2012) provide a literature overview on memetic algorithms. Memetic algorithms have been successfully applied to various kinds of scheduling problems where they are being considered as state of the art (Deng and Wang (2017), Cheng et al. (2008)). In Algorithm 2 we give an outline of the memetic algorithm (based on Jouglet et al. (2009)) to be used for our HFS-variant, which is very much similar to the genetic algorithm which was presented in Algorithm 1, with the exception of the local search step being added to the procedure in line 9. In the memetic algorithm we use the same encoding, selection, crossover, mutation and decoding procedures as was earlier presented in this section.

Algorithm 2 Memetic algorithm

```
1: procedure MEMETIC ALGORITHM($PopSize, p_m, p_{ls}$)
2:     Initialize population of $PopSize$ of memes.
3:     $b \leftarrow$ best meme and $w \leftarrow$ worst meme
4:     while Stopping criteria not yet met do
5:         Select 2 memes from the population
6:         Create offspring called $n$, with the 2 selected memes by means of a crossover procedure
7:         if $f(n) \leq f(b)$ then
8:             Mutate $n$ with probability $p_m$
9:         Execute a local search with probability $p_{ls}$
10:        if $f(w) \leq f(n)$ then
11:            Discard $n$
12:        else
13:            Replace $r$ with $n$, where $r$ is selected by means of reverse binary tournament selection
14:            Update best and worst chromosome
15:    return $b$
```

4.2.1 Local search method

The local search method is what differentiates a MA from a GA. In our case, we will use the local search method to try to solve the decoding sub-problem in a near-optimal way. The main motivation behind the local search procedure is to prevent premature convergence and to be able to discover good structures within the memes. However, this does come at the cost of more computation time per iteration. For our research we will consider two different local search methods. One will uses constraint programming and the other will use a linear programming formulation.

Constraint Programming

A method that can be used for the local search procedure in our proposed memetic algorithm is the use of constraint programming. Jouglet et al. (2009) present a memetic algorithm with constraint programming as local search method for a HFS with multiprocessor tasks with promising results.

We can specify the constraint programming to our case by formulating it in such a way that it solves the problem of the HFS, except for the stage that is encoded by the meme. Also, we add the constraint that it needs to have a better objective value than the meme obtained by means of the decoding rule. This is done, such that we find out what good structures within the memes are, since the decoding rule creates a feasible schedule from the encoded meme heuristically. The addition of the constraint that the objective value associated with a meme needs to better than the objective value resulting from the decoding rule, also should reduce the amount of computation time needed, since this would give the opportunity to the CP solver to reduce the bounds of the variables quickly.

When using a constraint programming formulation in the local search phase, we can make use of the constraint programming formulation that was stated earlier in Section 3, with some slight modifications and additions. First, we fix the sequence of the *M_Task* decision variables and their assignment to a machine in the first stage as it is represented by the encoded meme in the constraint programming formulation. Finally, an additional constraint is added to the formulation that states that the objective value should be lower than the already known objective value that resulted from decoding the meme. The additional constraint is formally stated in constraint (40). In this constraint, *Bound* refers to the bound that is set by applying the decoding rule to the encoded meme.

$$\sum_{j \in J} (\omega_e E_j + \omega_f F_j + \omega_t T_j) + \omega_c C_{max} \leq Bound \tag{40}$$

As stated earlier, a potential threat of the local search algorithm in the memetic algorithm is that it can consume a lot of computation time. That is why we intend to cut the constraint program (CP) short if no improvements are made within a certain period of time (to be found out in practice how much exactly). Jouglet et al. (2009) show that their proposed CP formulation either made improvements very fast, or it did not for a very long time.

Linear Programming

When looking at our constraint programming formulation it can be noted that there are still many decisions to be made. To reduce the amount of decisions to be made, one could use the sequence and machine assignments as they are already determined by the encoding and decoding rule in all stages in the GA in the local search step. With these sequence of tasks and machine assignments already fixed, the HFS-UMMTSDS can be modelled by means of a linear programming (LP) formulation. It is good to note that this LP formulation can also be used as a final improvement step with the final result of the GA.

In our linear programming formulation we use the same sets with corresponding indices and the same parameters and decision variables as in our mathematical formulation that we introduced in Section 3. There is only a small difference in the decision variable of the starting and completion time of a task in a stage, since we do not need the index for the machine anymore, since we already fixed the assignment of a task to a machine. Therefore, the decision variables S_{ijkm} and C_{ijkm} become S_{ijk} and C_{ijk} in our linear programming formulation. Furthermore, we use m_{ijk} to denote the machine that task i of job j in stage k is assigned to and we use t_{ijk} to denote the directly preceding task of task i of job j in stage k on its assigned machine. With b_{ijk} we refer to the job of the directly preceding task of task i of job j in stage k on its assigned machine. If there is no predecessor for a task, we create a dummy task that has a corresponding setup time and completion time of 0. The LP-model will then look as follows:

$$\min \quad \omega_t \sum_{j\in J} T_j + \omega_e \sum_{j\in J} E_j + \omega_f \sum_{j\in J} F_j + \omega_c C_{max} \tag{41}$$

$$S_{ijk} \geq \tilde{C}_{jk-1} \qquad \forall i \in I_{jk}, j \in J, k \in K\backslash\{0\}, \tag{42}$$

$$C_{ijk} = S_{ijk} + p_{ijkm_{ijk}} \qquad \forall i \in I_{jk}, j \in J, k \in K, \tag{43}$$

$$C_{t_{ijk}b_{ijk}k} + s_{t_{ijk}ib_{ijk}jkm_{ijk}} \leq S_{ijk} \qquad \forall j \in J, i \in I_{jk}, k \in K, \tag{44}$$

$$\tilde{C}_{jk} \geq C_{ijk} \qquad \forall i \in I_{jk}, j \in J, k \in K, \tag{45}$$

$$C_{max} \geq \tilde{C}_{j|K|} \qquad \forall j \in J, \tag{46}$$

$$\tilde{S}_j \leq S_{ij0} \qquad \forall i \in I_{jk}, j \in J, \tag{47}$$

$$E_j \geq \tilde{C}_{j|K|} - d_j \qquad \forall j \in J, \tag{48}$$

$$T_j \geq d_j - \tilde{C}_{j|K|} \qquad \forall j \in J, \tag{49}$$

$$F_j \geq \tilde{C}_j - \tilde{S}_j \qquad \forall j \in J, \tag{50}$$

$$S_{ijk} \geq 0 \qquad \forall i \in I_{jk}, j \in J, k \in K, \tag{51}$$

$$S_j \geq 0, F_j \geq 0, T_j \geq 0, E_j \geq 0 \geq 0 \qquad \forall j \in J \tag{52}$$

The objective value in (41) is the same as in our mathematical formulation. Constraint set (42) states that a task can only start production in a stage as long as it has finished production in the stage before. Constraint set (43) sets the completion time of a task equal to its starting time plus the production time that it needs on its assigned machine. In constraint set (44) it ensures that there is no overlap between tasks and that the sequence dependent setup times are taken into account. Constraint set (45) states that the completion time of a job should be at least as large as the completion time of all its tasks in a stage. Furthermore, constraint set (46) is used to set the maximum completion time, constraint set (47) to set the starting time of a job, constraint set (48) to track the earliness, constraint set (49) to track the tardiness of a job and constraint set (50) to track the flow time of a job. Finally Constraint sets (51) and (52) ensure the non-negativity of the decision variables.

In the remainder of this thesis we will refer to the memetic algorithm that uses constraint programming in the local search step as MACP and to the memetic algorithm that uses linear programming in the local search step as MALP.

5 Results

In this section we will study the performance of our solution approaches that have been introduced in Section 3 and Section 4. We will do this on two different types of data sets. The fist type of data set considers relatively small, randomly generated instances of the HFS-UMMTSDS. On these data sets we will compare the results of our heuristic solution approaches to the exact solution approaches and we study the tightness of our proposed lower and upper bound. The second type of data set is based on a large use case with around 1300 jobs and 4 stages and will be used to compare the proposed heuristic approaches to each other and to the lower and upper bound.

As stated earlier, our problem has, to the best of our knowledge, not been studied before. However, in literature research has been done on the HFS with multiprocessor tasks and on the HFS with unrelated parallel machines. Hence, besides evaluating our algorithms on the data sets stated in the previous paragraph, we could also use special case benchmark data from literature. For the HFS with multiprocessor tasks, benchmark data from literature is available (see for example Oğuz et al. (2003)), but in literature multiprocessor tasks are assumed to be processed simultaneously, which means that our algorithms will need an additional procedure to prevent cases in which multiprocessor tasks are not starting at the same time. Therefore, we do not expect our algorithms to be competitive on these benchmark instances due to the overhead in our code. Unfortunately, for the HFS with unrelated parallel machines we did not find benchmark data that fit the HFS-UMMTSDS.

All our proposed algorithms have been implemented in Python 3.7 by using Spyder 3.3.2. For the MIP and LP formulation, we used CPLEX 12.9 and implemented this in Python by using the python API of CPLEX by means of the PuLP 1.20 library. We implemented the constraint programming formulation in Python and solved it with IBM ILOG CPLEX CP Optimizer 12.9 with the help of the python API. We executed all runs on a laptop with an Intel i7 processor with 2.30 GHZ and 16 GB of RAM memory.

5.1 Setup of experiments for hyper parameters

For our genetic algorithm and memetic algorithms, the following parameters have to be tuned:

- p_m probability that a mutation happens in an iteration in the GA/MA
- $PopSize$ number of chromosomes/memes in the GA/MA
- p_{ls} probability that a local search is executed in the MA
- $MACP_{time}$ amount of computation time that is allowed to find an improved solution is found in the CP-based local search phase of the MACP

We evaluate the performance of the hyper parameters on a test data set that was generated with 5 jobs, 3 stages, a due date of 12, 2 multiprocessor tasks per stage, processing time from a discrete uniform distribution ranging from 1–5, number of machines in a stage from a discrete uniform distribution ranging from 2–4 where all tasks that can be executed on all machines in a stage and the sequence dependent setup time from a discrete uniform distribution ranging from 0–2. Besides a choice for the hyper parameters, also a choice is made for both the encoding rules, concerning the crossover procedure and mutation procedure. For the population size $PopSize$ we used values in the range of 100–1000, with a step size of 100. The mutation probability p_m is tested with values in the range of 0.1–1.0, with a step size of 0.1. The local search probability p_{ls} needs to be tested specifically for the MAs, where we used values ranging between 0.005–0.05 with a step size of 0.005 and between 0.05–0.2 with a step size of 0.05. The computation time allowed until an improvement is made $MACP_{time}$ is tested on values in the range of 0.5–2.0 seconds with a step size of 0.5.

Table 3: Values of hyper parameters

Metaheuristic	Crossover Procedure	Mutation Procedure	$PopSize$	p_m	p_{ls}	CP_{time}
GA JP	PMX	Insertion	400	0.7	-	-
GA RK	Two Point	Uniform	400	0.8	-	-
MACP JP	PMX	Insertion	400	0.7	0.005	0.5
MACP RK	Two Point	Uniform	400	0.8	0.005	0.5
MALP JP	PMX	Insertion	400	0.7	0.02	-
MALP RK	Two Point	Uniform	400	0.8	0.02	-

In Table 3 the values of the hyper parameters are presented that we will use in our runs on the different data sets, based on their performance on our test case. Table 3 shows, when using the job permutation encoding rule, the insertion procedure is the chosen mutation procedure and PMX is the used crossover procedure. Our results showed that the insertion procedure outperformed the scramble, swap and inversion procedures as mutation procedure. If the random keys encoding rule is used, two point is the preferred crossover procedure since it outperformed the uniform crossover procedure significantly. Furthermore, the uniform mutation procedure is used. Note, that for the uniform mutation procedure the fraction of genes that should be mutated within the mutation procedure should be specified. For this, we used a value of 0.15, because it performed best on our test instances. Regarding the population size, we noticed that a population size of 400 yielded the best results on our test instances. The mutation probability is best set relatively high for the same reasons as for the population size.

5.2 Benchmark with exact solution methods

In this section we study the performance of our exact solution approaches and heuristic solution approaches on small data sets by only considering 3–5 jobs and 2 or 3 stages. These instances are a factor 300 times as small, in terms of jobs, as will be the case in our large scale industry based problem. This shows us the performance of the exact approaches relative to the heuristic solution approaches and we can see how fast the computation time increases when solving the problem to optimality with our exact solution approaches as the problem size increases. Furthermore, we present the values of our lower and upper bounds on the small data sets and we compare them to the optimal solution.

5.2.1 Data outline small randomly generated data set

In the small data sets, we consider the following characteristics: 3 or 5 jobs, 2 or 3 stages, 2–4 machines per stages, production time on a machine of a task ranging between 1–5, 3–8 or 5–10, sequence dependent setup time for each task on each machine in the range of 0–2 or 0–9. When the production time ranges between 1–5, the due date of a job ranges from 4–10; if the production time ranges between 3–8, the due date of a job ranges from 6–12; if the production time ranges from 5–10, the due date of a job ranges from 8–14. Here, all values are generated from a discrete uniform distribution. Furthermore, each job has 2 multiprocessor tasks in each stage. In the objective value we will use a weight of 1 for all objective value components.

5.2.2 Results exact and heuristic solution approaches on small data sets

Tables 4–7 and Table 9 present the results of solution approaches on the small data sets. In these tables, the column named 'instance' denotes the characteristics of the data set on which the run is done. For example, J3S2P1-5S0-2 denotes that this instance has 3 jobs (J3), 2 stages (S2), the production time of a task on a machine is randomly generated between 1 and 5 (P1-5) and the setup time between two tasks on machine is randomly generated between 0 and 2 (S0-2). In Table 4 we present the results of the exact solution approaches (CP and MIP) and the upper and lower bound on the small data set. Table 5 presents the results of the GAs with the 2 different encoding rules. The linear programming optimization problem is applied on the best solution resulting from our GAs, the objective value resulting from this is presented in Table 6. Table 7 presents the results of the MACP. Finally, Table 9 shows the result of our MALP.

In Table 4 the computation time (in seconds) needed to reach the optimal is presented for each instance for both the MIP and the CP solver and the objective value is presented, with the gap to the optimal solution in brackets. Both the MIP and the CP solver were cut short when a time limit of 3600 seconds is reached. The lower and upper bound are also presented in Table 4 in which we also added the gap to the optimal solution in brackets. In Table 4 the fastest computation time to reach the optimal solution per instance is highlighted in bold.

Table 4 shows that for the instances with only 3 jobs and 2 stages, both the MIP and the CP solver perform quite well and they find the optimal solution within a few seconds. On these instances, solving our MIP formulation generally goes faster than solving our CP formulation to optimality. However, we can observe a relatively large increase in computation time for the MIP solver when the stages are increased from 2 to 3, while the computation time needed to solve the CP formulation to optimality remains similar. When the number of jobs is increased to 5 we can see that the MIP is not able to produce the optimal solution for 10 out of the 12 cases with 5 jobs within the time limit of 3600 seconds, while the CP solver is able to find the optimal solutions for all cases. However, also for the CP solver the computation times increases considerably.

The fact that the CP solver is able to outperform the MIP solver, confirms our earlier expectation that the HFS-UMMTSDS is a highly constrained problem.

Table 4: Computational results MIP and CP on small randomly generated data sets

Instance	CP time (seconds)	CP result (gap)	MIP time (seconds)	MIP result (gap)	UB (gap)	LB (gap)
J3S2P1-5S0-2	**0.65**	27 (0.0%)	1.52	27 (0.0%)	80 (196.3%)	19 (29.6%)
J3S2P1-5S0-9	7.15	36 (0.0%)	**2.99**	36 (0.0%)	140 (288.89%)	19 (47.2%)
J3S2P3-8S0-2	3.38	63 (0.0%)	**0.66**	63 (0.0%)	162 (157.1%)	47 (25.4%)
J3S2P3-8S0-9	7.05	74 (0.0%)	**1.21**	74 (0.0%)	286 (286.5%)	47 (36.5%)
J3S2P5-10S0-2	2.00	95 (0.0%)	**1.25**	95 (0.0%)	232 (144.2%)	77 (18.9%)
J3S2P5-10S0-9	1.80	106 (0.0%)	**0.72**	106 (0.0%)	356 (235.8%)	77 (27.4%)
J3S3P1-5S0-2	**3.92**	41 (0.0%)	31.83	41 (0.0%)	124 (202.4%)	27 (34.1%)
J3S3P1-5S0-9	**1.26**	48 (0.0%)	14.98	48 (0.0%)	334 (595.9%)	27 (43.8%)
J3S3P3-8S0-2	**2.26**	91 (0.0%)	30.59	91 (0.0%)	226 (148.4%)	74 (18.7%)
J3S3P3-8S0-9	**2.30**	115 (0.0%)	57.51	115 (0.0%)	391 (240.0%)	74 (35.7%)
J3S3P5-10S0-2	**9.71**	137 (0.0%)	17.23	137 (0.0%)	324 (136.5%)	116 (15.3%)
J3S3P5-10S0-9	**5.68**	161 (0.0%)	46.91	161 (0.0%)	489 (203.7%)	116 (28.0%)
J5S2P1-5S0-2	**17.99**	38 (0.0%)	320.55	38 (0.0%)	158 (315.8%)	29 (23.7%)
J5S2P1-5S0-9	**83.50**	69 (0.0%)	2613.97	69 (0.0%)	368 (433.3%)	29 (58.0%)
J5S2P3-8S0-2	**262.34**	130 (0.0%)	3600 *	136 (4.6%)	363 (179.2%)	106 (18.5%)
J5S2P3-8S0-9	**143.96**	150 (0.0%)	3600 *	169 (12.7%)	636 (324.0%)	106 (29.3%)
J5S2P5-10S0-2	**500.35**	190 (0.0%)	3600 *	194 (2.1%)	497 (161.6%)	166 (12.6%)
J5S2P5-10S0-9	**283.81**	208 (0.0%)	3600 *	213 (2.4%)	770 (270.2%)	166 (20.2%)
J5S3P1-5S0-2	**58.13**	79 (0.0%)	3600 *	85 (7.6%)	248 (213.9%)	62 (21.5%)
J5S3P1-5S0-9	**964.24**	124 (0.0%)	3600 *	136 (9.7%)	560 (351.6%)	62 (50.0%)
J5S3P3-8S0-2	**420.68**	186 (0.0%)	3600 *	197 (5.9%)	521 (180.1%)	155 (16.7%)
J5S3P3-8S0-9	**1841.72**	230 (0.0%)	3600 *	246 (7.0%)	801 (248.3%)	155 (32.6%)
J5S3P5-10S0-2	**700.37**	270 (0.0%)	3600 *	272 (0.7%)	699 (158.9%)	235 (13.0%)
J5S3P5-10S0-9	**2841.17**	314 (0.0%)	3600 *	318 (1.3%)	979 (211.8%)	235 (25.2%)

* optimal solution was not found within the time limit of 3600 seconds.

Table 4 shows that the upper bound is relatively far from the optimal solution with gaps for certain problem instances that are over 400%. Since we did not take setup times into account for determining the upper bound, we observe larger gaps for the problem instances with a setup time ranging between 0–9 compared to the problem instances with a setup time ranging between 0–2. However, it did take considerably less computation time to generate the upper bounds compared to the time needed to obtain the best solutions in our heuristic solution approaches. The upper bounds take approximately 0.008 seconds to compute, which remained approximately equal when increasing the number of jobs and stages. The lower bounds have a gap ranging between 12%–58% compared to the optimal solution. This gap might be explained by the fact that we do not evaluate setup times within the lower bound, which also explains the lower bounds that are equal for the problem instances that only differ in setup time, and that we set earliness at a value of 0. The lower bounds took approximately 0.001 seconds to generate for the problem instances with 3 jobs and approximately 0.002 seconds for the problem instances with 5 jobs.

Table 5: Computational results GA JP and GA RK on small randomly generated data sets

Instance	GA JP best result (gap)	GA JP avg result (gap)	GA JP avg time (seconds)	GA RK best result (gap)	GA RK avg result (gap)	GA RK avg time (seconds)
J3S2P1-5S0-2	36 (33.3%)	36 (33.3%)	8.51	**32 (18.5%)**	**32 (18.5%)**	8.00
J3S2P1-5S0-9	48 (33.3%)	48 (33.3%)	8.62	**41 (13.9%)**	**41 (13.9%)**	8.67
J3S2P3-8S0-2	68 (7.9%)	68 (7.9%)	8.86	**66 (4.8%)**	**66 (4.8%)**	8.60
J3S2P3-8S0-9	**83 (12.2%)**	**83 (12.2%)**	8.51	**83 (12.2%)**	**83 (12.2%)**	7.19
J3S2P5-10S0-2	104 (9.5%)	104 (9.5%)	8.84	**102 (7.4%)**	**102 (7.4%)**	7.09
J3S2P5-10S0-9	117 (10.4%)	117(10.4%)	8.53	**115 (8.5%)**	115.8 (9.2%)	8.40
J3S3P1-5S0-2	48 (17.1%)	48 (17.1%)	12.82	**46 (12.2%)**	46.1 (12.4%)	12.69
J3S3P1-5S0-9	70 (45.8%)	70 (45.8%)	12.82	**65 (35.4%)**	65.6 (36.7%)	13.66
J3S3P3-8S0-2	98 (7.7%)	98 (7.7%)	12.43	**95 (4.4%)**	**95 (4.4%)**	13.20
J3S3P3-8S0-9	136 (18.3%)	136 (18.3%)	12.95	**131 (13.9%)**	**131 (13.9%)**	12.97
J3S3P5-10S0-2	148 (8.0%)	148 (8.0%)	12.39	**141 (2.9%)**	141.4 (3.2%)	14.29
J3S3P5-10S0-9	192 (19.3%)	192 (19.3%)	12.91	**177 (9.9%)**	**177 (9.9%)**	14.15
J5S2P1-5S0-2	48 (26.3%)	48 (26.3%)	18.20	**43 (13.2%)**	**43 (13.2%)**	20.54
J5S2P1-5S0-9	86 (24.6%)	86.5 (25.3%)	26.35	**83 (20.3%)**	85.2 (23.5%)	25.89
J5S2P3-8S0-2	158 (21.5%)	158.3 (21.8%)	33.02	**145 (11.5%)**	149.2 (14.8%)	29.26
J5S2P3-8S0-9	183 (22.0%)	183 (22.0%)	26.85	**182 (21.3%)**	184.9 (23.3%)	27.85
J5S2P5-10S0-2	236 (24.2%)	236.6 (24.5%)	21.31	**210 (10.5%)**	218.9 (15.2%)	27.07
J5S2P5-10S0-9	260 (25%)	260.3 (25.1%)	20.98	**248 (19.2%)**	253.3 (21.8%)	23.66
J5S3P1-5S0-2	90 (13.9%)	90 (13.9%)	29.34	**89 (12.7%)**	89.9 (13.8%)	30.10
J5S3P1-5S0-9	152 (22.6%)	152.1 (22.7%)	33.82	**141 (13.7%)**	143.7 (15.9%)	40.58
J5S3P3-8S0-2	207 (11.3%)	207 (11.3%)	33.80	**191 (2.7%)**	193.6 (4.1%)	41.50
J5S3P3-8S0-9	255 (10.9%)	255 (10.9%)	40.68	**247 (7.4%)**	**248.6 (7.4%)**	50.96
J5S3P5-10S0-2	298 (10.4%)	298 (10.4%)	34.21	**275 (1.9%)**	277.1 (2.6%)	43.87
J5S3P5-10S0-9	350 (11.5%)	350.1 (11.5%)	34.69	**334 (6.4%)**	337.8 (7.5%)	44.83

In Table 5 the results of the GA with the 2 different encoding rules are presented. We cut the GAs short if no improvement in the objective value was found for 10000 iterations or if a total computation time of

3600 seconds per run was reached. The GAs were run 10 times on each problem instance. For both the GA JP and the GA RK the average result of the converged solution and the best result are presented, together with the gap to the optimal solution. For each problem instance, the best found solution is highlighted in bold.

From Table 5 it can be observed that the computation time of the GA until convergence slightly increases as the number of jobs increases. However, it increases at a slower rate than it does for the CP and MIP solvers. Furthermore, when we look at the results when the GA is converged, we can see that the gaps are between 1.9%–45.8% from the optimal result with an average gap of 18.7% for the GA JP and 12.9% for the GA RK. We can explain the gaps for a large part by the list scheduling heuristic that decodes the solution. With the list scheduling decoding rule it may be the case that it is not even possible for a problem instance to be solved to optimality by the GA. To illustrate this, we checked the assignment of tasks to machines and their sequence on the machines in the first stage in the optimal solution resulting from the CP formulation from problem instance J3S2P1-5S0-2. We translated this into the job permutation and random keys representation encoding rules such that this would result in the same assignment to the machines and sequence of tasks on the machines in the first stage. The list scheduling decoding rule then respectively yields an objective value of 49 and 40.

The results of the GA JP and the GA RK show that the random keys representation on almost all problem instances is able to produce a converged solution with a better objective value than the job permutation encoding rule. We expect that this can be explained by the additional assignment rule that the job permutation encoding rule needs, which reduces the number of possible solutions that this GA can produce. In terms of computation time needed, we can see that the GA RK generally converges faster than the GA JP does on the instances with 3 jobs and 2 stages, but that as number of stages and jobs increases, it generally converges slower. Finally, we observe that the GA RK is not as stable as the GA JP (the objective value of the converged solution varies more).

When looking at the gaps in further detail, there does not seem to be a relationship between the number of stages in the problem and gap of the converged solution to the optimal solution. However, we do see a difference in the gaps when the number of jobs increases. In this case, as the number of jobs increases, the gap decreases. Also, the gap generally speaking decreases as the production time increases and the gap increases as the setup time increases. The difference in the gaps might be caused by the fact that the impact of a deviation from the optimal solution will have a relatively high impact on the gap for problem instances with a relatively low optimal objective value.

Table 6: Computational results GA JP and GA RK with the linear programming optimization at the end on small randomly generated data sets

Instance	GA JP* best result (gap with optimum)	GA JP* avg result (gap with optimum)	GA RK* best result (gap with optimum)	GA RK* avg result (gap with optimum)
J3S2P1-5S0-2	32 (18.5%)	34.6 (28.1%)	**31 (14.8%)**	31.7 (17.4%)
J3S2P1-5S0-9	48 (33.3%)	48 (33.3%)	**41 (13.9%)**	41.2 (14.4%)
J3S2P3-8S0-2	**64 (1.6%)**	**64 (1.6%)**	**64 (1.6%)**	**64 (1.6%)**
J3S2P3-8S0-9	81 (9.5%)	81 (9.5%)	**77 (4.1%)**	78.2 (5.7%)
J3S2P5-10S0-2	**96 (1.1%)**	97.6 (2.7%)	**96 (1.1%)**	96.4 (1.5%)
J3S2P5-10S0-9	115 (8.5%)	115 (8.5%)	**107 (0.9%)**	113.5 (7.1%)
J3S3P1-5S0-2	**42 (2.4%)**	**42 (2.4%)**	44 (7.3%)	44.4 (8.3%)
J3S3P1-5S0-9	**60 (25.0%)**	**60 (25.0%)**	65 (35.4%)	65.7 (36.9%)
J3S3P3-8S0-2	**92 (1.1%)**	**92 (1.1%)**	94 (3.3%)	94.1 (3.4%)
J3S3P3-8S0-9	129 (12.2%)	129 (12.2%)	**123 (7.0%)**	126.9 (10.3%)
J3S3P5-10S0-2	**138 (0.7%)**	**138 (0.7%)**	140 (2.2%)	140 (2.2%)
J3S3P5-10S0-9	181 (12.4%)	183.1 (13.7%)	**169 (5.0%)**	169.8 (5.5%)
J5S2P1-5S0-2	44 (15.8%)	44.5 (17.1%)	**41 (7.9%)**	41.1 (8.2%)
J5S2P1-5S0-9	81 (17.4%)	81 (17.4%)	**76 (10.1%)**	80.2 (16.2%)
J5S2P3-8S0-2	156 (20.0%)	156 (20.0%)	**145 (11.5%)**	146.7 (12.8%)
J5S2P3-8S0-9	179 (19,3%)	179 (19,3%)	**164 (9.3%)**	176.7 (17.8%)
J5S2P5-10S0-2	207 (8.9%)	212 (11.6%)	**204 (7.4%)**	211.7 (11.4%)
J5S2P5-10S0-9	247 (18.8%)	247 (18.8%)	**226 (8.7%)**	241.8 (16.3%)
J5S3P1-5S0-2	90 (13.9%)	90 (13.9%)	**85 (7.6%)**	87.9 (11.3%)
J5S3P1-5S0-9	140 (12.9%)	140 (12.9%)	**134 (8.1%)**	138.9 (12.0%)
J5S3P3-8S0-2	196 (5.4%)	196.1 (5.4%)	**190 (2.2%)**	191.9 (3.2%)
J5S3P3-8S0-9	242 (5.2%)	242 (5.2%)	**234 (1.7%)**	239.2 (4.0%)
J5S3P5-10S0-2	283 (4.8%)	285.6 (5.8%)	**274 (1.5%)**	275.1 (1.9%)
J5S3P5-10S0-9	324 (3.2%)	324 (3.2%)	**323 (2.9%)**	327.7 (4.4%)

* LP optimization added to the final solution resulting from the GA.

In Table 6 we present the best and average result of the GA JP and GA RK with the linear programming optimization problem added to the final solution found by the GAs. The gap with the optimal solution is added between brackets and the best found solution is highlighted in bold for each problem instance.

Table 6 shows significant improvements in the objective value as a result of the LP optimization problem on the final result from the GA. The gap with the optimum solution value ranges from 0.7%–36.9% with an average gap of 12.1% for the GA JP (compared to an average of 18.7% for the GA JP without the LP optimization step at the end) and 9.7% for the GA RK (compared to an average of 12.9% for the GA RK without the LP optimization step at the end). The computation time of the LP optimization is around 0.05 seconds on these small instances. Furthermore, Table 6 shows that on 20 out of 24 problem instances the random keys representation performs at least as good as the job permutation encoding rule. In 18 out of 24 problem instances the random keys representation outperforms the job permutation encoding rule.

Table 7: Computational results MA JP and MA RK on small randomly generated data sets

Instance	MACP JP best result (gap with optimum)	MACP JP avg result (gap with optimum)	MACP JP avg time (seconds)	MACP RK best result (gap with optimum)	MACP RK avg result (gap with optimum)	MACP RK avg time (seconds)
J3S2P1-5S0-2	**27 (0.0%)**	27.1 (0.4%)	129.98	**27 (0.0%)**	27.2 (0.7%)	126.35
J3S2P1-5S0-9	**37 (2.7%)**	37.7 (4.7%)	144.69	**37 (2.7%)**	38.5 (6.9%)	99.91
J3S2P3-8S0-2	**65 (3.2%)**	**65 (3.2%)**	120.12	**65 (3.2%)**	**65 (3.2%)**	93.79
J3S2P3-8S0-9	**74 (0.0%)**	75.5 (2.0%)	141.27	**74 (0.0%)**	76.3 (3.1%)	127.13
J3S2P5-10S0-2	**97 (2.1%)**	97.8 (2.9%)	150.57	**97 (2.1%)**	**97 (2.1%)**	153.75
J3S2P5-10S0-9	**106 (0.0%)**	108.3 (2.2%)	137.26	**106 (0.0%)**	108.9 (0.9%)	130.63
J3S3P1-5S0-2	**43 (4.9%)**	**43 (4.9%)**	118.71	**43 (4.9%)**	**43 (4.9%)**	90.27
J3S3P1-5S0-9	55 (14.6%)	55 (14.6%)	134.75	**49 (2.1%)**	53 (10.4%)	170.54
J3S3P3-8S0-2	93 (2.2%)	94.8 (4.2%)	108.0	**92 (1.1%)**	93 (2.2%)	118.25
J3S3P3-8S0-9	122 (6.1%)	123.9 (7.7%)	225.40	**119 (3.5%)**	122.4 (6.5%)	165.44
J3S3P5-10S0-2	141 (2.9%)	142.8 (4.2%)	106.6	**138 (0.7%)**	138.2 (0.9%)	130.25
J3S3P5-10S0-9	171 (5.6%)	172.6 (7.2%)	210.93	**169 (5.0%)**	170 (5.6%)	228.40
J5S2P1-5S0-2	**42 (10.5%)**	44.9 (18.2%)	219.15	43 (13.2%)	43 (13.2%)	237.76
J5S2P1-5S0-9	**81 (17.3%)**	84.7 (22.8%)	210.62	83 (20.3%)	84.3 (22.2%)	315.76
J5S2P3-8S0-2	148 (13.8%)	149.2 (14.8%)	288.28	**145 (11.5%)**	147.7 (13.6%)	257.96
J5S2P3-8S0-9	172 (14.7%)	179.5 (19.7%)	188.37	**168 (12.0%)**	179.5 (19.7%)	328.79
J5S2P5-10S0-2	221 (16.3%)	223.6 (17.7%)	214.54	**204 (7.4%)**	218.2 (14.8%)	250.73
J5S2P5-10S0-9	237 (13.9%)	250.5 (20.4%)	226.88	**239 (14.9%)**	249.5 (20.0%)	324.35
J5S3P1-5S0-2	90 (13.9%)	90 (13.9%)	144.95	**88 (11.4%)**	89.6 (13.4%)	199.27
J5S3P1-5S0-9	146 (17.7%)	150 (21.0%)	202.28	**141 (13.7%)**	143.7 (15.9%)	252.49
J5S3P3-8S0-2	205 (10.2%)	206.6 (11.1%)	198.02	**193 (3.8%)**	194.3 (4.5%)	257.90
J5S3P3-8S0-9	255 (10.9%)	255 (10.9%)	158.57	**247 (7.4%)**	249.5 (8.5%)	267.60
J5S3P5-10S0-2	296 (9.6%)	297 (10.0%)	200.44	**275 (1.9%)**	279.5 (3.5%)	246.06
J5S3P5-10S0-9	350 (11.5%)	350 (11.5%)	146.17	**334 (6.4%)**	335.7 (6.9%)	264.22

In Table 7 the results of the 2 different MACP versions are presented. We stopped the MACP if no improvement in the objective value was found for 10000 iterations or if a computation time of 3600 seconds was reached per run. The MACPs were run 10 times on each problem instance. For both the MACP JP and the MACP RK the average result of the converged solution and the best result are presented with the best solution value per problem instance highlighted in bold. Also the gap with the optimal value is presented in percentages and the average computation time in seconds is presented.

The results presented in Table 7 show that with the addition of the local search step by means of constraint programming to the GA, we are able to produce better solutions at the cost of additional computation time. We can see a gap to the optimal solution that ranges from 0%-23% with an average of 10.4% for the MACP JP (compared to an average of 18.7% for the GA JP) and an average of 8.5% for the MACP RK (compared to an average of 12.9% for the GA RK). These results show that with the addition of the constraint programming formulation to the GA we are able to outperform the GA. However, we also observe an increase in the amount of computation time needed, especially when the number of jobs increases from 3 to 5. Furthermore, just as with the GA we observe that the average time needed to convergence for the MA RK is larger than for the MA JP and that the result of the converged solution is generally better for the MA RK when compared to the MA JP (except for the problem instances with 3 jobs and 2 stages).

Strangely, the time needed to converge decreases for the case with 5 jobs when the number of stages increases from 2 to 3. Table 8 shows the percentage of the cases where the local search step yielded an improvement to the objective value associated with a meme. Besides that, Table 8 shows the average amount of times that the local search step is given an additional 0.5 second of computation time, because the objective value of that meme improved. From Table 8 we observe that the time to convergence decreases when the amount of stages goes from 2 to 3, because the CP search is not able to find as many improvements in the local search phase, such that no additional seconds are allowed within this search step and the iterative search phase is cut short. We can conclude that as the number of stages increases the constraint programming formulation turns out to be less successful in the local search step.

Table 8: MA RK analysis of improvements made by local search step

	Local search resulting in improvements	Improvements per local search
3 jobs 2 stages	95%	5.57
3 jobs 3 stages	92%	4.58
3 jobs 2 stages	96%	6.8
5 jobs 3 stages	89%	4.68

In Figure 5 we show the performance of the GA and the MA on problem instance J3S3P1-5S0-2 and 5S3P1-5S0-2 respectively. In this visualization it can be seen that on a problem instance with 3 jobs, the

MA is able to outperform the GAs in a computational efficient manner. However, in the problem instance with 5 jobs the MA does not yield any improvements in the objective value and it takes more time to converge. For the other small benchmark data problems, similar results can be found. This also suggests that the MA might not be an effective method when the number of jobs increases.

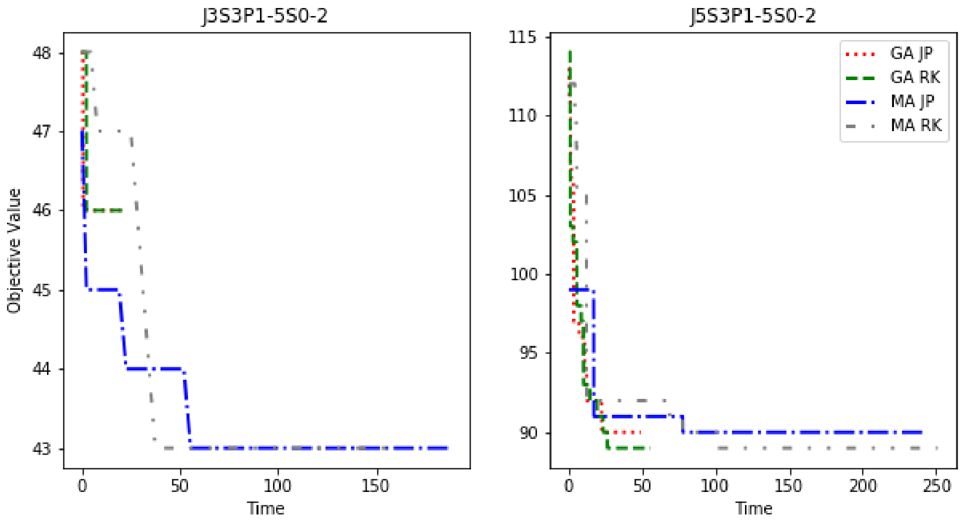

Figure 5: GA vs MACP on small randomly generated data sets.

In Table 9 we present the average and best results over 10 iterations of the MALP with the JP and RK encoding rules, together with the average computation time in seconds. Again, we cut the metaheuristic short if no improvement is made for 10000 iterations or a time limit of 3600 seconds is reached. For each problem instance the best found objective value is highlighted in bold.

We observe that for certain instances the MALP is able to find the optimal solution. Furthermore, the gap with the optimal solution ranges from 0.0%–27.9% with an average gap of 8.6% for the MALP JP (compared to an average of 18.7% for the GA JP) and an average gap of 7.1% for the MALP RK (compared to an average of 12.9% for the GA RK). The computation times approximately increase by a factor 2 when the amount of jobs are increased from 3 to 5. On these small problem instances, this is not necessarily a problem. However, for large problem instances this could be something to keep in mind. Similar to the results of the GA, we see that the gap generally speaking decreases as the production time and setup time increases.

Table 9: Computational results MALP JP and MALP RK on small randomly generated data sets

Instance	MALP JP best result (gap with optimum)	MALP JP avg result (gap with optimum)	MALP JP avg time (seconds)	MALP RK best result (gap with optimum)	MALP RK avg result (gap with optimum)	MALP RK avg time (seconds)
J3S2P1-5S0-2	31 (14.8%)	31 (14.8%)	32.2	**29 (7.4%)**	29.4 (8.9%)	33.2
J3S2P1-5S0-9	45 (25%)	45.6 (26.7%)	36.2	**39 (8.3%)**	39.1 (8.6%)	35.5
J3S2P3-8S0-2	64 (1.6%)	64 (1.6%)	30.5	**63 (0.0%)**	63.6 (1.0%)	30.9
J3S2P3-8S0-9	78 (5.4%)	78 (5.4%)	26.7	**77 (4.1%)**	77.1 (4.2%)	35.3
J3S2P5-10S0-2	96 (1.1%)	96 (1.1%)	29.1	**95 (0.0%)**	95.3 (0.3%)	34.1
J3S2P5-10S0-9	110 (3.8%)	110 (3.8%)	26.4	**107 (0.9%)**	108.5 (2.4%)	29.9
J3S3P1-5S0-2	**42 (2.4%)**	**42 (2.4%)**	32.0	**42 (2.4%)**	42.1 (2.7%)	34.1
J3S3P1-5S0-9	**60 (25.0%)**	**60 (25.0%)**	32.8	**60 (25.0%)**	61.4 (27.9%)	38.0
J3S3P3-8S0-2	**91 (0.0%)**	**91 (0.0%)**	36.9	**91 (0.0%)**	**91 (0.0%)**	37.6
J3S3P3-8S0-9	126 (9.6%)	126 (9.6%)	33.3	**123 (7.0%)**	123.5 (7.4%)	38.3
J3S3P5-10S0-2	138 (0.7%)	138 (0.7%)	35.7	**137 (0.0%)**	137.1 (0.1%)	38.2
J3S3P5-10S0-9	171 (6.2%)	171 (6.2%)	35.7	**169 (5.0%)**	169.3 (5.2%)	40.4
J5S2P1-5S0-2	43 (13.2%)	43.9 (15.5%)	56.8	**39 (2.6)**	39.5 (3.9%)	84.5
J5S2P1-5S0-9	81 (17.4%)	81.5 (18.1%)	70.5	**76 (14.5%)**	80.7 (17.0%)	74.3
J5S2P3-8S0-2	144 (10.8%)	144.6 (11.2%)	71.6	**143 (10.0%)**	144.5 (11.1%)	78.9
J5S2P3-8S0-9	167 (11.3%)	169.4 (12.9%)	70.6	**164 (9.3%)**	171.2 (14.1%)	72.7
J5S2P5-10S0-2	206 (8.4%)	208.5 (9.7%)	66.3	**205 (7.9%)**	209.3 (10.1%)	72.1
J5S2P5-10S0-9	**228 (9.6%)**	233.8 (12.4%)	62.4	**228 (9.6%)**	238.6 (14.7%)	67.3
J5S3P1-5S0-2	**85 (7.6%)**	85.3 (8.0%)	85.5	**85 (7.6%)**	85.9 (8.7%)	73.9
J5S3P1-5S0-9	139 (12.1%)	139.4 (12.4%)	95.4	**134 (8.1%)**	136.3 (9.9%)	105.8
J5S3P3-8S0-2	191 (2.7%)	191.5 (3.0%)	91.2	**188 (1.1%)**	191.6 (3.0%)	106.2
J5S3P3-8S0-9	234 (1.7%)	234.3 (1.9%)	87.5	**231 (0.4%)**	237.8 (3.4%)	119.6
J5S3P5-10S0-2	273 (1.1%)	275.1 (1.9%)	87.2	**272 (0.7%)**	276.6 (2.4%)	89.6
J5S3P5-10S0-9	318 (1.3%)	319.3 (1.7%)	85.1	**314 (0.0%)**	324.7 (3.4%)	113.8

In Table 10 we present the average gap over the best solutions found, the percentage of best found solutions that are optimal and computation time needed for both our exact and heuristic solution approaches concerning the instances with 3 jobs and the instances with 5 jobs. For both the problem instances with 3 and 5 jobs there were 12 different data sets. From the results on the small data sets, we observe that the exact solution approaches get out of hand in terms of computation time when the number of jobs increases from 3 to 5, but that the constraint programming formulation outperforms the mathematical formulation in terms of computation time.

Table 10: Summary of best results solution approaches on small data sets

	Average gap 3 jobs	Optimal 3 jobs	Average time 3 jobs (s)	Average gap 5 jobs	Optimal 5 jobs (%)	Average time ' 5 jobs (s)
MIP	0.0%	100%	17.3	4.5%	16.7%	3244.5
CP	0.0%	100%	3.9	0.0%	100%	676.5
GA JP	18.6%	0%	10.7	16.6%	0%	29.4
GA RK	12.2%	0%	10.2	13.6%	0%	33.8
GA JP*	10.5%	0%	10.8	12.1%	0%	29.5
GA RK*	8.1%	0%	10.3	6.6%	0%	33.9
MACP JP	4.9%	16.7%	144.0	16.0%	0%	199.9
MACP RK	4.0%	16.7%	136.2	13.0%	0%	266.9
MALP JP	8.0%	8.3%	32.3	8.1%	0%	77.5
MALP RK	5.0%	33.3%	35.5	6.0%	8.3%	88.2

* LP optimization added to the final solution resulting from the GA.

Table 10 also shows that our heuristic solution approaches are able to converge relatively quickly (especially the GAs and the MALPs), even if the number of stages and/or jobs increases. Furthermore, we observe the added value of the local search step in the memetic algorithm compared to our genetic algorithm with the side note that it does come at the cost of additional computation time. It seems that although the MALP does not necessarily produce better results compared to the MACP, it is a more effective solution approach in terms of computation time. We also observed that the MA in some cases produces results that are optimal, but that its result is at worst within a few percentages of the optimal results on small problem instances. However, as the number of jobs increases we notice that the effectiveness of the MACP compared to the GA and MALP decreases and that the GA and/or MALP generally speaking are better at finding good solutions in a time efficient manner.

5.3 Large scale industry inspired data

In this section we will consider a large scale data set and compare the performance of our GA with our MALP to test the performance when applying our methodology to such a large scale scheduling problem. Since we observed in the previous section that exact solution approaches get out of hand for problem sizes with 5 jobs, we will not consider those on such a large scheduling problem. Furthermore, we will not consider the MACP, since in the results of the heuristic solution approaches on the small data sets, we observed that the MALPs and the GAs outperformed the MACPs as the number of jobs increases.

5.3.1 Outline of data

The following data set is based on a production environment encountered in practice. This refers to the production of mattresses that was described in the problem description in Section 1. The industry case contains around 1300 production orders that need to be planned on a weekly basis and all those orders need to be processed in 4 production stages. All jobs have a due date with bounds of 4320 and 10080 (3–7 days expressed in minutes from the start of the planning). We will obtain the due date of a job from a discrete uniform distribution. In Table 11 a detailed outline of the data set is given for the characteristics of the parameters for each stage. Four data sets were generated based on the characteristics of the case sketched in this paragraph and Table 11. Besides these four data sets, on which we will evaluate the performance of our heuristic algorithms, we will test the effects of changing the values of the weights in the objective value.

Table 11: Large industry based case data outline

	Stage 1	Stage 2	Stage 3	Stage 4
Production time (minutes)	10-240	10-600	90-180	1-40
Setup time (minutes)	60-760	0-10	0-10	0-10
Eligible machines	9-34	9-18	13-26	5
Multiprocessor Tasks	2-3	1-2	1	1
Total number of machines	70	25	40	5

Table 12 presents the different sets of values used for the weights of the objective value. We are mainly interested in investigating the strength of the effect of changing the weights on the resulting flow times, earliness and tardiness of a job and the maximum completion time. We will increase the value of the weight of the maximum completion time 400 times instead of 4 times, since some experiments showed that the maximum completion time has a relatively low value on these large problem instances compared to the other objective value components. Note that the effect of increasing the weight associated with earliness will have a negative impact on the tardiness. Contrarily, the effect of increasing the weight of the maximum completion time might reduce the total flow time, since jobs will have to be produced more efficiently in this case.

Table 12: Outline of different weights of the objective criteria

Case	ω_f	ω_e	ω_l	ω_c
1	1	1	1	1
2	4	1	1	1
3	1	4	1	1
4	1	1	4	1
5	1	1	1	400

5.3.2 Results large case

In Table 13 and 14 we present respectively the best and average results from five runs of the GA and MALP on each data set on the four different data sets with each objective value component having a weight of 1. Furthermore, we present the objective value resulting from the LP optimization problem on the final result of the GA. As a stopping criterion we used a maximum computation time of 3600 seconds per run, or if no improvements were made for 10.000 iterations. In this case the time limit always stopped the GA and MALP. Some tests showed that waiting until convergence would take approximately 8 hours per run with an improvement of 15% compared to the objective value of the GA when being stopped at 3600 seconds. Finally, it is good to note that the LP optimization step for such large scale problem instances takes anywhere between 5–10 seconds to be solved optimally.

Table 13: Best results of heuristic solution approaches over 5 runs

Case	GA JP	GA RK	GA JP*	GA RK*	MALP JP	MALP RK
1	21,117	20,105	17,390	16,959	17,740	17,419
2	21,524	19,530	16,963	16,376	17,511	17,550
3	21,387	20,084	17,027	17,157	17,869	17,289
4	22,064	20,606	17,535	17,194	17,826	17,740

* LP optimization added to the final solution resulting from the GA.

Interestingly, Table 13 and Table 14 show that the GA with the LP at the end outperforms the MALP. This can be explained by the fact that in the MALP some random memes will have the LP optimization as a local improvement step. Furthermore, the improvements in the objective value are usually that good that ‘regular’ memes that do not have the local search step, do not improve the best objective value, such that the LP optimization problem at the end is done with a meme for which this problem had already been solved in the local search phase. However, if the LP optimization is only done at the best solution representation found by the GA, this means that we do the LP optimization step at a meme that is probably a meme with a high potential. Due to fact that the GA with the LP optimization at the final result outperforms the MALP, we will evaluate the effect of adjusting the weights of the objective value components only for the GAs with the LP optimization problem at the end.

Table 14: Average results of heuristic solution approaches over 5 runs

Case	GA JP	GA RK	GA JP*	GA RK*	MALP JP	MALP RK
1	21,528	20,305	17,530	17,343	17,854	17,600
2	21,639	20,124	17,505	17,105	17,870	17,648
3	21,668	20,399	17,435	17,277	17,941	17,619
4	22,173	20,776	17,716	17,456	17,926	17,897

* LP optimization added to the final solution resulting from the GA.

Table 15: GA JP vs GA RK on large industry based data sets

Case	ω_e	ω_t	ω_f	ω_c	GA JP* best (x1000)	GA JP* avg (x1000)	GA RK* best (x1000)	GA RK* avg (x1000)	UB (x1000)	LB (x1000)
1	1	1	1	1	17,390	17,530	**16,959**	17,343	500,398	254
	1	1	1	400	26,684	26,981	**26,527**	26,725	751,037	1,928
	1	1	4	1	42,896	43,282	**42,486**	42,795	794,016	1,002
	1	4	1	1	38,685	39,449	**38,091**	38,567	1,706,073	254
	4	1	1	1	17,289	17,385	**16,953**	17,177	500,416	254
2	1	1	1	1	16,963	17,505	**16,376**	17,105	528,840	252
	1	1	1	400	26,782	27,008	**26,548**	26,678	788,737	1,880
	1	1	4	1	42,621	43,383	**41,432**	42,445	873,211	996
	1	4	1	1	39,385	39,489	**37,592**	37,890	1,768,990	252
	4	1	1	1	17,385	17,482	**16,880**	17,212	528,883	252
3	1	1	1	1	**17,027**	17,435	17,157	17,277	520,909	252
	1	1	1	400	26,829	27,224	**26,567**	26,784	778,036	1,874
	1	1	4	1	42,917	43,097	**41,776**	42,317	845,462	996
	1	4	1	1	39,304	39,695	**37,523**	38,697	1,757,109	252
	4	1	1	1	17,138	17,426	**16,840**	17,108	520,953	252
4	1	1	1	1	17,535	17,716	**17,194**	17,456	507,441	253
	1	1	1	400	26,969	27,376	**26,703**	26,830	759,155	1,895
	1	1	4	1	**42,214**	42,854	42,315	42,933	812,764	999
	1	4	1	1	40,122	40,410	**38,983**	39,345	1,722,512	253
	4	1	1	1	17,556	17,642	**17,295**	17,466	507,478	253

* LP optimization added to the final solution resulting from the GA.

Table 15 shows the results of the GA with job permutation and random keys representation with the LP optimization problem done with the final result of the GA. The different sets of values for the weights of the objective value components together with our upper and lower bound on the optimal objective value are also presented. For each problem instance and settings for the weights for the objective value components, the best objective value is highlighted in bold.

Similar to the small benchmark data, we can see that the random keys representation outperforms the job permutation encoding rule on almost all instances with an average improvement of 2% on the large scale data sets. This suggests that when combining multiprocessor tasks with unrelated parallel machines in a HFS, the random keys representation, which is commonly used in HFS with unrelated parallel machines, might be a better choice to use than a job permutation encoding rule, which is commonly used in HFS settings with multiprocessor tasks.

We can see that our GAs with the LP optimization at the end outperform the upper bound significantly. However, determining the upper bound can still be done fast compared to the GAs with a computation time of approximately 0.15 seconds. The lower bound is about 66 times lower than the best result in the case of equal objective component weights. This large gap shows that there is still a possibility for improvement in our lower bound, which we will leave for further research. Furthermore, we observe that changing the weight of earliness of jobs has a relatively low impact on the objective value compared to the other objective components. This can be explained by the fact that earliness can be prevented reasonably easily, by just postponing jobs. Since the GA RK with the LP optimization at the end outperformed the GA JP with the LP optimization at the end on both the small and the large data sets, we will only consider the GA RK with the LP optimization at the end for further analysis.

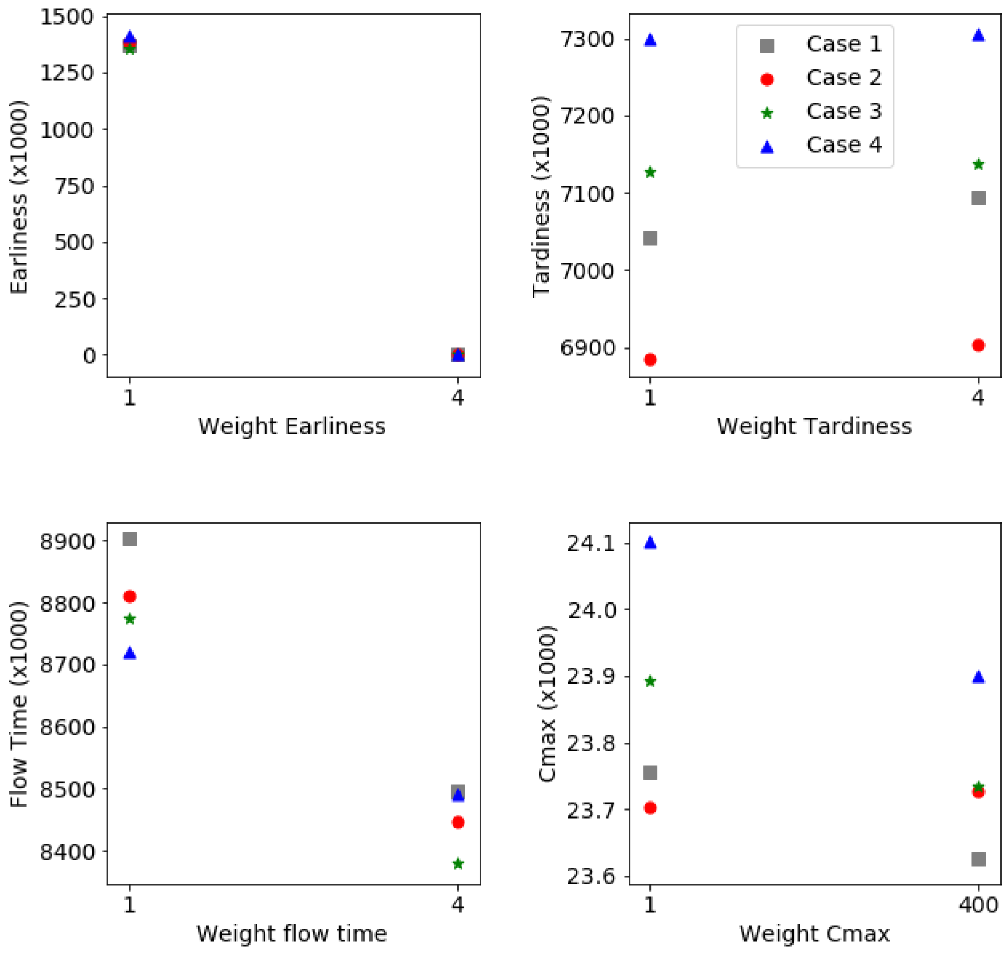

Figure 6: Effect adjusting weight of objective criteria on individual objective criterion

In Figure 6 we present the effect of changing the weight of an objective component while keeping the other weights equal in the GA RK with the LP optimization at the end to the result of that individual objective component for each data case. The average result of 5 runs is denoted by a different symbol and color for each data case. For all objective components, except the tardiness, we observe that the value of that component decreases as the weight increases. Most interesting in Figure 6 is the decrease in earliness when the weight of the earliness is increased from 1 to 4. In this case there is no earliness at all. Contrary to the earliness, the tardiness does not seem to be affected by the weight in the objective value. This could be caused by the fact that it will be harder to reduce the tardiness, since there is only limited capacity available compared to the earliness where jobs can simply be postponed.

In Figure 7 we present the strength of the effect of changing the weight of the earliness and tardiness on both the earliness and tardiness. Interestingly, we observe that the earliness does not increase if more weight is given to the tardiness. Vice versa, increasing the weight of the earliness does yield in more tardiness.

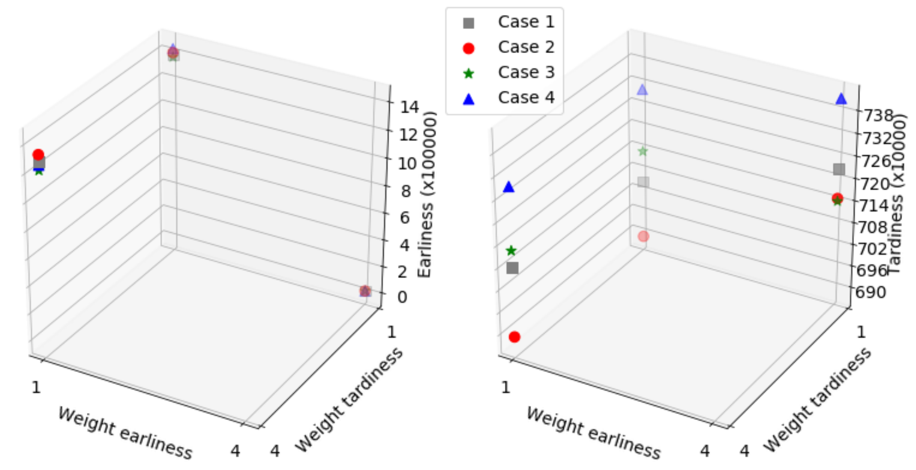

Figure 7: Effect weight of earliness and tardiness to resulting earliness and tardiness

From the results on the large industry case inspired data set, we can conclude that the GAs with the LP optimization at the end have significant performance advantages over the GAs and the MALPs. Also, we observe that the random keys representation consistently outperforms the job permutation encoding rule. This suggests that the random keys representation is a better fit for the HFS-UMMTSDS than the job permutation encoding rule. Furthermore, we investigated the effect of changing the weights within the objective value to the objective components.

6 Conclusion

In this thesis we studied a scheduling problem that is based on the production process of mattresses. The industry case that we base our research on has around 1300 jobs that need to be processed on a weekly basis in such a way that the maximum completion time, total flow time, total earliness and total tardiness are minimized. Each mattress has 4 production steps that have to be completed before it is ready for use. Within each production step it can be the case that several tasks need to be processed and that there are different machines that, dependent on their age and design, have different processing times for a task. It can also be the case that a task can only be processed by a certain set of machines. With the described characteristics, the problem can be translated into the Hybrid Flow Shop model with multiprocessor tasks and unrelated parallel machines in which we consider sequence dependent setup times (HFS-UMMTSDS).

To the best of our knowledge, we are the first to study the HFS-UMMTSDS. However, in literature there has been related research that studies the HFS with multiprocessor tasks and there has been related research that studies the HFS with unrelated parallel machines. We showed that the HFS-UMMTSDS is strongly NP-hard which motivated the use of heuristic solution approaches. To formally define our problem and to be able to benchmark heuristic solution approaches, we introduced a mathematical programming formulation. Since constraint programming techniques have been successfully applied within scheduling problems, we set up a constraint programming formulation as well. Finally, we derived a lower and upper bound to be able to asses the performance of heuristic solution approaches on large data sets. Furthermore, we can use our upper bound to shorten the amount of computation time needed for solving our constraint programming formulation to optimality.

Next to exact solution approaches, we adopted promising metaheuristics, a genetic and memetic algorithm, from literature on the HFS with multiprocessor tasks and on the HFS with unrelated parallel machines and adjusted these existing solution approaches such that we are able to solve the HFS-UMMTSDS. The memetic algorithm uses a local search step for which we studied two different methods. In one method a constraint programming formulation, based on the constraint programming formulation used to solve the HFS-UMMTSDS to optimality, is used for the local search step. The other method uses a mathematical formulation for the local search step, based on the the mathematical formulation used to solve the HFS-UMMTSDS to optimality.

On small randomly generated data sets we compared our exact solution approaches with our heuristic solution approaches. We observed that our constraint programming formulation is a better fit for the HFS-UMMTSDS than our mathematical formulation, which confirmed our earlier expectation that the HFS-UMMTSDS is a highly constrained problem. When considering problem instances with 3 jobs, solving the CP or MIP formulation to optimality takes just a few seconds. However, as the number of jobs increases from 3 to 5, computation times start to become intractable. This suggests that heuristic solution approaches will be a better fit for the HFS-UMMTSDS when there are more jobs that need to be scheduled. We also observe that the upper bounds are relatively far from the optimal solution compared to the heuristic solution approaches and observe that the lower bounds tend to perform relatively poor when the setup time is relatively large compared to the production time on a problem instance.

We could see that our heuristic solution approaches were able to get within a few percentages of the optimal solution on a large part of the problem instances of the small randomly generated data sets. When looking at the encoding rules used, we could see that the random keys representation was consistently producing better results than the job permutation encoding rule, but at the costs of additional computation time. On the smaller instances, the MACP and MALP were able to produce results that are significantly better than the results of the GA. However, as the number of jobs increases from 3 to 5, we could see that the MACP did not yield any significant improvements over the GA anymore and that it needed more computation time to converge. This suggests that our MALP and GA are the best fit from our proposed solution approaches for the HFS-UMMTSDS as the number of jobs increases.

In the data set that is based on the industry case with 1300 jobs and 4 stages, we evaluated the performance of both our GA and MALP. Those results showed a significant performance advantage of the GA with the LP optimization at the end over the GA and MALP. Our results showed that the random keys representation, which is commonly used within the HFS with unrelated parallel machines, might be a better fit when combining multiprocessor tasks and unrelated parallel machines within a HFS than the job permutation encoding rule, which is commonly used within the HFS with multiprocessor tasks. Furthermore, we studied the effect of changing the weights of the different objective components and we could see the effects of changing a single weight on the result of other objective components. The upper bound is significantly outperformed by all our heuristic solution approaches and from the lower bound we observed that further research will be needed to further improve the performance of the lower bound on such large scale problem instances.

Finally, we can conclude that we developed a GA which is able to solve large scale industry cases on a problem that, to the best of our knowledge, has not been studied in literature before. In our heuristic solution approaches we observed that the random keys representation, for our tested problem instances, outperformed the job permutation encoding rule. Furthermore we observed that the GA and the MALP outperform the MACP as the number of jobs increases in a problem instance. For large problem instances the GA with the addition of a LP optimization at the end produces the best results over our tested problem instances. Also, on relatively small problem instances we could see the advantage of constraint programming over solving the HFS-UMMTSDS to optimality with CPLEX.

7 Discussion

With our research we are the first to consider the HFS-UMMTSDS. This thesis shows the complexity of the HFS-UMMTSDS and in this thesis we develop two exact solution approaches to be able to solve the problem to optimality on small problem instances. We also introduced heuristic solution approaches that were based on previous research on the HFS with multiprocessor tasks and on the HFS with unrelated parallel machines. These heuristic solution approaches were able to produce optimal results on certain small problem instances. Furthermore, our heuristic solution approaches produce results on certain small scale problem instances that outperform CPLEX. On the large problem instances it is still hard to assess the result of our solution approaches, since our lower bound appears to perform poorly. Overall, we can say that this thesis has slightly reduced the gap between scheduling problems encountered in theory and practice.

As has been pointed out in literature reviews on the HFS, there is still a lot of room left for further research to be able to cope with challenges encountered in HFS scheduling that occur in practice. For example, it can be interesting to implement the possibility of maintenance on machines for a certain time period in the HFS-UMMTSDS. Instead of planned maintenance, it is also likely to have machine breakdowns in practice such that unplanned maintenance is required, which motivates the research on rescheduling for the HFS-UMMTSDS. Another interesting addition might be to consider (unequal) release dates for jobs. Furthermore, a production environment usually has limited intermediate storage between its production stages that constrain the amount of allowed intermediate storage, which we did not consider in our solution approaches. Finally, our lower bound seems to perform poorly on large scale data and it might be improved by making use of mathematical programming techniques as branch and bound or with column generation techniques.

References

J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6(2):154–160, 1994.

M. A. Bozorgirad and R. Logendran. A comparison of local search algorithms with population-based algorithms in hybrid flow shop scheduling problems with realistic characteristics. *The International Journal of Advanced Manufacturing Technology*, 83(5-8):1135–1151, 2016.

P. M. Castro, I. E. Grossmann, and A. Q. Novais. Two new continuous-time models for the scheduling of multistage batch plants with sequence dependent changeovers. *Industrial & engineering chemistry research*, 45(18):6210–6226, 2006.

L. Chen, H. Zheng, D. Zheng, and D. Li. An ant colony optimization-based hyper-heuristic with genetic programming approach for a hybrid flow shop scheduling problem. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 814–821. IEEE, 2015.

Y.-Y. Chen, C.-Y. Cheng, L.-C. Wang, and T.-L. Chen. A hybrid approach based on the variable neighborhood search and particle swarm optimization for parallel machine scheduling problems—a case study for solar cell industry. *International Journal of Production Economics*, 141(1):66–78, 2013.

H.-C. Cheng, T.-C. Chiang, and L.-C. Fu. A memetic algorithm for parallel batch machine scheduling with incompatible job families and dynamic job arrivals. In *2008 IEEE International Conference on Systems, Man and Cybernetics*, pages 541–546. IEEE, 2008.

F.-D. Chou. Particle swarm optimization with cocktail decoding method for hybrid flow shop scheduling problems with multiprocessor tasks. *International Journal of Production Economics*, 141(1):137–145, 2013.

M. Dai, D. Tang, K. Zheng, and Q. Cai. An improved genetic-simulated annealing algorithm based on a hormone modulation mechanism for a flexible flow-shop scheduling problem. *Advances in Mechanical Engineering*, 5:124903, 2013.

J. Deng and L. Wang. A competitive memetic algorithm for multi-objective distributed permutation flow shop scheduling problem. *Swarm and evolutionary computation*, 32:121–131, 2017.

J. Du and J. Y.-T. Leung. Minimizing total tardiness on one machine is np-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990. doi: 10.1287/moor.15.3.483. URL `https://doi.org/10.1287/moor.15.3.483`.

O. Engin, G. Ceran, and M. K. Yilmaz. An efficient genetic algorithm for hybrid flow shop scheduling with multiprocessor task problems. *Applied Soft Computing*, 11(3):3056–3065, 2011.

K. Fan, Y. Zhai, X. Li, and M. Wang. Review and classification of hybrid shop scheduling. *Production Engineering*, 12(5):597–609, 2018.

M. Gholami, M. Zandieh, and A. Alem-Tabriz. Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns. *The International Journal of Advanced Manufacturing Technology*, 42(1-2):189–201, 2009.

D. E. Goldberg, R. Lingle, et al. Alleles, loci, and the traveling salesman problem. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 154, pages 154–159. Lawrence Erlbaum, Hillsdale, NJ, 1985.

R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979. ISSN 0167-5060. doi: 10.1016/S0167-5060(08)70356-X.

J. N. Gupta. Two-stage, hybrid flowshop scheduling problem. *Journal of the Operational Research Society*, 39(4):359–364, 1988.

J. H. Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

V. Hoogeveen, Lenstra. Preemptive scheduling in a two-stage multiprocessor flow shop is np-hard. *European Journal of Operational Research*, 89(1):172—175, 1996.

A. Jouglet, C. Oğuz, and M. Sevaux. Hybrid flow-shop: a memetic algorithm using constraint-based scheduling for efficient search. *Journal of Mathematical Modelling and Algorithms*, 8(3):271–292, Aug 2009. ISSN 1572-9214. doi: 10.1007/s10852-008-9101-1. URL `https://doi.org/10.1007/s10852-008-9101-1`.

J. Jungwattanakit, M. Reodecha, P. Chaovalitwongse, and F. Werner. Constructive and simulated annealing algorithms for hybrid flow shop problems with unrelated parallel machines. *Thammasat international journal of science and technology*, 12, 01 2007.

J. Jungwattanakit, M. Reodecha, P. Chaovalitwongse, and F. Werner. Algorithms for flexible flow shop problems with unrelated parallel machines, setup times and dual criteria. *International Journal of Advanced Manufacturing Technology*, 37:354 – 370, 05 2008. doi: 10.1007/s00170-007-0977-0.

O. Kheirandish, R. Tavakkoli-Moghaddam, and M. Karimi-Nasab. An artificial bee colony algorithm for a two-stage hybrid flowshop scheduling problem with multilevel product structures and requirement operations. *International Journal of Computer Integrated Manufacturing*, 28:1–14, 02 2014. doi: 10.1080/0951192X.2014.880805.

L. Kroon, D. Huisman, E. Abbink, P.-J. Fioole, M. Fischetti, G. Maróti, A. Schrijver, A. Steenbeek, and R. Ybema. The new dutch timetable: The or revolution. *Interfaces*, 39(1):6–17, 2009.

M. Kurdi. Ant colony system with a novel non-daemonactions procedure for multiprocessor task scheduling in multistage hybrid flow shop. *Swarm and Evolutionary Computation*, 44, 11 2018. doi: 10.1016/j.swevo.2018.10.012.

P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2): 210–250, 2018.

E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *Sequencing and scheduling : algorithms and complexity*, pages 445–522. Handbooks in Operations Research and Management Science. North-Holland Publishing Company, Netherlands, 1993. ISBN 0-444-87472-0.

S.-W. Lin, K.-C. Ying, and C.-Y. Huang. Multiprocessor task scheduling in multistage hybrid flowshops: A hybrid artificial bee colony algorithm with bi-directional planning. *Computers & Operations Research*, 40: 1186–1195, 05 2013. doi: 10.1016/j.cor.2012.12.014.

P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms. *Caltech Concurrent Computation Program*, 10 2000.

F. Neri and C. Cotta. Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14, 2012.

C. Oğuz, M. F. Ercan, T. E. Cheng, and Y.-F. Fung. Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. *European Journal of Operational Research*, 149(2):390–403, 2003.

C. Oğuz and M. Ercan. Scheduling multiprocessor tasks in a two-stage flow-shop environment. *Computers & Industrial Engineering*, 33:269–272, 10 1997. doi: 10.1016/S0360-8352(97)00090-9.

C. Oğuz and M. Ercan. A genetic algorithm for hybrid flow-shop scheduling with multiprocessor tasks. *Journal of Scheduling*, 8:323–351, 07 2005. doi: 10.1007/s10951-005-1640-y.

M. Pinedo. *Scheduling*, volume 29. Springer, 2012.

E. Rashidi, M. Jahandar, and M. Zandieh. An improved hybrid multi-objective parallel genetic algorithm for hybrid flow shop scheduling with unrelated parallel machines. *The International Journal of Advanced Manufacturing Technology*, 49:1129–1139, 08 2010. doi: 10.1007/s00170-009-2475-z.

J.-C. Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

I. Ribas, R. Leisten, and J. Framinan. Review and classification of hybrid flow shop scheduling problems for a production system and a solutions procedure perspective. *Computers & Operations Research*, 37: 1439–1454, 08 2010. doi: 10.1016/j.cor.2009.11.001.

R. Ruiz and J. A. Vázquez Rodríguez. The hybrid flow shop scheduling problem. *European Journal of Operational Research*, 205:1–18, 08 2010. doi: 10.1016/j.ejor.2009.09.024.

E. Siqueira, M. Souza, and S. Souza. A multi-objective variable neighborhood search algorithm for solving the hybrid flow shop problem. *Electronic Notes in Discrete Mathematics*, 66:87–94, 04 2018. doi: 10.1016/j.endm.2018.03.012.

Z.-w. Sun and X.-s. Gu. A novel hybrid estimation of distribution algorithm for solving hybrid flowshop scheduling problem with unrelated parallel machine. *Journal of Central South University*, 24:1779–1788, 08 2017. doi: 10.1007/s11771-017-3586-6.

T. Urlings, R. Ruiz, and F. Sivrikaya Şerifoğlu. Genetic algorithms with different representation schemes for complex hybrid flexible flow line problems. *Int. J. Metaheuristics*, 1, 01 2010. doi: 10.1504/IJMHEUR. 2010.033122.

A. Vignier, J. Billaut, and C. Proust. Hybrid flowshop scheduling problems: State of the art. *Rairo-Recherche Operationnelle-Operations Research*, 33(2):117–183, 1999.

H.-M. Wang, F.-D. Chou, and F.-C. Wu. A simulated annealing for hybrid flow shop scheduling with multi-processor tasks to minimize makespan. *The International Journal of Advanced Manufacturing Technology*, 53:761–776, 03 2011. doi: 10.1007/s00170-010-2868-z.

Y. Xu, L. Wang, S.-y. Wang, and M. Liu. An effective immune algorithm based on novel dispatching rules for the flexible flow-shop scheduling problem with multiprocessor tasks. *The International Journal of Advanced Manufacturing Technology*, 67, 07 2013. doi: 10.1007/s00170-013-4759-6.

L. J. Zeballos, J. M. Novas, and G. P. Henning. A cp formulation for scheduling multiproduct multistage batch plants. *Computers & Chemical Engineering*, 35(12):2973–2989, 2011.