**Department of Business Administration**


# Flow Shop Scheduling for BRC Ltd

**by**

**Kyriakos Bitsis**

Dissertation Presented for the Degree of
MSc in Business Analytics and Data Science

September 2021

Supervised by
Dr Konstantinos Kaparis

# Abstract

This MSc project considers a multistage flexible flow shop scheduling problem with parallel unrelated machines at each stage. An extensive literature review is given followed by the presentation of a flow shop model. Moreover the model takes into consideration machine dependent setup times and different processing speeds for every machine in each stage. The study is motivated by the necessity for job allocation within the production line for coils and bars on behalf of BRC Ltd, who is among the major steel reinforcement manufacturer in UK. We develop an integer program that minimizes the convex sum of makespan and the lateness of tardy jobs over a finite time horizon. Our model is applicable to a part of the factory and specifically for bay 3. The corresponding optimization problem is quite challenging to be solved optimally even for small instances. We report the mathematical and implementation challenges and how we tackle them. Also we suggest some extensions by adding some extra parameters making the model ready to be compatible with real data. Last but not least, we generate and analyze some preliminary computational results based on a beta version code developed in Python using the Pyomo library.

# Acknowledgments

I would like to express my gratitude to Dr Konstantinos Kaparis for challenging me to the opportunity to collaborate and work with him on this overwhelmingly and very thought-provoking real world problem. His passion for this project inspired me to improve my skills in the combinatorial optimization field. His commitment and willingness to assist and encourage me converted my time working for BRC a rewarding learning experience. Our endless conversations were very crucial for expanding my knowledge.

I extend my gratitude also to Dr Andreas Georgiou, who is an excellent academic tutor. He was very willing to help me overcome different obstacles that I faced during my undergraduate and postgraduate courses by providing valuable comments and suggestions. He also taught me that resignation is not a solution.

I could not omit thanking Dr Ioannis Konstantaras. I appreciate his encouragement and his willing to help me anytime I asked for. His faith led me to continue with my studies and gave me the necessary strength to aim high. Our discussions were very beneficially supportive to my evolution. I wish the best for him.

Last but not least, this MSc would have not been possible without the love and support of my family and friends. They supported me to overcome the difficulties that arose. However, I would like to dedicate this work to my girlfriend Chara who have been a source of encouragement throughout my life and of course for her endless patient, help, support and companionship during all the years of my MSc studies. .

# Own Work Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

# Contents

## List of Tables



## List of Figures

# 1 Introduction

Scheduling is among the most important issues that concern the operation of manufacturing systems. Its aim is the efficient allocation of tasks to machines along with the subsequent time-phasing of this allocation. In general, tasks individually compete for resources which can be of a very different nature, e.g., manpower, money, processors (machines), energy, tools. The same is true for task characteristics, e.g., set up times, due dates, relative urgency weights, and functions describing task processing in relation to allotted resources. Moreover, a structure of a set of tasks, reflecting relations among them, can be defined in different ways. In addition, different criteria which measure the quality of the performance of a set of tasks can be considered ([4]).

In this thesis we discuss *flow shop* scheduling problems, and more precisely we analyse the case of BRC Ltd, which is among the leading UK companies in steel reinforcement. Steel industry is an important industrial sector in UK and one of the biggest worldwide.

In what follows we briefly introduce the case of BRC, we describe its products, the manufacturing line and its major components, the warehouse procedure etc. Then we outline the relevant operational research literature. There will be a more detailed description in the main part and during the modelling.

The purpose of this dissertation is to develop a conceptual model for a part of the production after the "storage" and prior to "loading and dispatch" to the customers. We will construct a multistage flexible flow shop model and we will propose a suitable Mixed Integer Programming model (MIP). Finally, we will present some preliminary results from the application of the MIP model on a set of randomly generated instances.

## 1.1 Steel Industry

In the 19th century, United Kingdom led the world industrial revolution and was the leading manufacturer of steel and iron. When the global recession hit the economy in 2008, the iron industry was at its peak. Since the financial crisis in 2009, there have been many spending cuts, demand for steel has fallen and prices have fallen by 40%, while production in United Kingdom reached 8 million tons in 2016, China produced 808 million tones.

In particular, the production of the iron industry in United Kingdom was around 14 million tons per year from 2005 until the crisis of 2009. Since 2009, production has been decreased to 9 million tons in 2012. By the next 2 years iron production reached 12 million tons, while there was a significant reduction in production in 2015 to 11 million tones, reaching 8 by 2016, as many factories closed, 7,000 people lost their jobs at the end of 2016 and exports were cancelled to a significant extent.

It is very important to point out the decrease on the selling price observed in recent years, as at the beginning of the crisis of 2009, the price of one ton was 340 pounds and by 2016 it reached half price of 170 pounds. UK's iron ore revenue in 2009 was almost 3.41 billion pounds, while 2012 revenues reached up to 1.6 billion, only 0.1% of the British economy. From 2013 the selling price of a ton of iron started at 300 pounds, reaching 225 pounds in 2015. That year, there were 2.4 billion revenues in United Kingdom, while in 2016 revenues reached up to 1.36 billion pounds.
It is worth noting that production in 2016 decreased 30% from 2015, the second largest decrease since the global crisis of 2008, 46%. Finally, it was observed that 52% of iron exports to United Kingdom are made in the European Union, while 69% of iron imports come from the European Union [1, 2, 3].

## 1.2 Scheduling

Today, resource management is an inevitable part of the performance and efficiency optimization in manufacturing and service industries. *Scheduling* is the allocation of shared resources over time to competing activities. It has been the subject of significant amount of literature in the operations research field. Emphasis has been on investigating machine scheduling problems where jobs represent activities and machines represent resources; each machine can process at most one job at a time. The resources include the use of equipment, the utilization of raw material or intermediates, the employments of operators, etc. The purpose of scheduling is to optimally allocate the limited resources to

processing tasks over time and the decisions to be determined include the optimal sequence of tasks taking place in each machine, the amount of material being processed at each time in each machine and sometimes the processing time of each job in each machine.

In addition scheduling problems could be classified into *offline* and *online*. In an offline problem, the number of jobs, release dates, delivery dates, processing times, due dates and other input data are known in advance. When data are not known in advance but they are realised only when a job is released then the problem is classified under the label of online scheduling. Such problems have been extensively used for resource planning in distributed systems [6, 5].

Scheduling problems are among the most widely studied topics in operational research literature. The significance of these problems rests in the fact that scheduling decisions greatly affect the success or failure of the manufacturer to efficiently allocate limited resources. The two most common types of problems, which are native to manufacturing jobs, are *Job-Shop Scheduling Problem (JSSP)* and *Flow-Shop Scheduling Problem (FSSP)*.

An important classification is based on the nature of the production facility to manufacture the required number of products utilizing a limited set of units. If production orders have different routes (require different sequences of tasks) and some orders may even visit a given unit several times it is known as multipurpose plant and the related optimization problems are also called *job-shop* problems. If every job consist of the same set of tasks to be performed in the same order and the units are accordingly arranged in production line, it is classified as a multiproduct plant called *flow-shop* problem [7]. The latter class of problems is the ones that are mostly met in practise.

### 1.2.1 Job-Shop Scheduling

The Job-Shop scheduling (JSP) problem is a case where jobs are assigned to a particular machine for processing at a particular time. These types of jobs are usually non-continuous or non-repetitive in nature.
The job-shop problem may be formulated as follows. Consider $n$ jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ and $m$ different machines $\mathcal{M} = \{M_1, \ldots, M_m\}$. Each job $J_i$ consists of a number $n_i$ of operations $\mathcal{O} = \{O_{i1}, \ldots, O_{in}\}$ which have to be processed in this order. Furthermore, assume that operation $O_{ik}$ can be processed only by one machine $\mu_{ik}$ $(i = 1, ..., n;\ k = 1, ..., n_i)$. Denote by $p_{ik}$ the corresponding processing time. There is only one machine of each type which can only process one operation at a time. Such an operation must be processed without preemption. Moreover, a job cannot be processed by two machines at the same time. According to these restrictions we call for an order of all operations $O_{ik}$ with $\mu_{ik} = M_j$ for each machine $M_j$ such that for the corresponding schedule the performance criterion is the optimum [9].

It has been known that the job-shop problem is strongly $\mathcal{NP}$-hard [10] and also it is one of the worst in the class. That means when the size of problem grows up, the time for achieving optimality grows exponentially, unless $\mathcal{P} = \mathcal{NP}$ [12]. Job Shop problems are not only theoretically hard though. Solving such combinatorial optimization problems is hard from the practical point of view and this is highlighted by the fact that a 10-job, 10-machine problem formulated in 1963 [11] was unsolved for more than 25 years [8].

Typically, there are $M$ machines and $N$ jobs for scheduling. Jobs have to be processed on these machines with different routes or sequences. So the complexity of scheduling depends on number of machines, number of jobs and finally the sequences themselves. There are *N!* ways (solutions/ possible schedules) to sequence jobs on each machine. For some type of scheduling problem, all machines use the similar sequence. So the number of feasible solutions to find the optimal one are *N!* solutions. For the job-shop problem each machine has different sequences and there are $M$ different machines; therefore the number of feasible solution expand to $(N!)^M$ solutions [13].

To find exact solutions of job-shop problem several Branch & Bound algorithms have been developed. For many years an algorithm by McMahon and Florian [14] has been the most efficient one. Others were less successful or led to improvements only in some special cases. The first algorithm which solved the $10 \times 10$ benchmark problem of Muth and Thompson and proved optimality of the solution was developed by Carlier and Pinson in 1987 [8].

Besides branch and bound methods for finding exact solutions of the job shop scheduling problem heuristics have been developed. The most popular heuristics in practise rely on priority rules. There are some more sophisticated, among those a method called "Shifting Bottleneck" by Adams et al. [15]. For most heuristics there exist instances for which these heuristics perform badly as well [9].

### 1.2.2 Flow-Shop Scheduling

Consider a job shop problem such that each job goes through the different machines following the same order, is called Flow Shop Scheduling problem (FSP). This is a special case of a job shop problem and was introduced by Johnson for two machines with the objective of minimizing makespan [20]. In scheduling theory, the makespan ($C_{max}$) is defined as the completion time of the final job. In the Flow Shop problem the jobs follow a strict sequence of tasks to be performed at the various machines, and are continuous in nature (in contract with the JSP). Though "easier" than the general case (i.e. the JSP), the FSP remains $\mathcal{NP}$-hard [16] and it is far more applicable in real cases.

In the basic FSP we assume a set of $n$ independent jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ that must be sequentially processed on a set of $m$ different machines $\mathcal{M} = \{M_1, \ldots, M_m\}$. All jobs have the same processing order through the machines and a job is composed of a well ordered list of $m$ tasks with known requirements where the $i^{th}$ task of each job is determined by the same machine. Moreover each job $J_j \in \mathcal{J}$ has a processing time $p_{ji} > 0$ on machine $M_i \in \mathcal{M}$ and also has a release date $r_j \geq 0$. It is not permitted to process any job before its release date. At any given time each machine can handle at most one task and each task can be processed by at most one machine. Each task of a job requires a different machine, no job visits a work station more than once. Preemption is forbidden, that is, any commenced operation has to be completed without interruptions. Each machine process the jobs in a first come first served manner. Unless specified otherwise, all jobs are available simultaneously at the beginning of the time horizon, and remain available without interruption until all work on them is finished. A schedule S is defined as a vector of starting times of the jobs on each machine: $S = \{s_{11}, \ldots, s_{1m}, s_{21}, \ldots, s_{2m}, s_{n1}, \ldots, s_{nm}\}$, where $s_{ji}$ is the starting time of job $J_j$ on machine $M_i$ in schedule $S$. Given a schedule $S$, we denote by $C_{ji}$ the completion time of job $J_j$ on machine $M_i : C_{ji} = s_{ji} + p_{ji}$ [19].

The main distinction between the classical flow-shop and a job-shop is that, in the former case each job passes the machines in the same order whereas in the latter case the machine order may vary per job. So the arrival of a job at a particular machine is not stochastic and most of the jobs that flow through that machine are similar in nature. Since workflow in a job shop in not unidirectional, scheduling becomes quite harder and tedious. Jobs in a FSP are produced either continuously or in batches [17]. We consider the batch process in the sense that once processing of a batch is started, it cannot be interrupted and other jobs cannot be introduced into the batch. On the other hand, the jobs in a JSP usually differ from one another and their arrival to the machines is usually stochastic.

As the size of a problem increases, due to uncertainty of arrival time and difference in jobs, the complexity and the time required to determine an optimal solution increases exponentially. This makes scheduling of jobs in JSP much more difficult than an FSP and Taillard (1993) has published a set of benchmark FSPs, JSPs and Open Shop Problems of different sizes and this is open for all to solve and send their results to compare the solutions achieved [18]. However though a JSP is more difficult from complexity perspective, the flow-shop problems are more applicable in real life. The majority of the real cases are actually a flow shop problem because usually there are precedence constraints and different procedures that must be followed.

Due to the wide use of the flow shop scheduling problems there exist some extensions in the literature like the *hybrid*, the *permutation*, the *multi-stage flexible*, etc flow shop problems. Briefly we will introduce the above cases. The Hybrid flow shop problem could be every variation of the initial one, however it is very common to be referred in this situation where the properties of a flow shop and parallel machine (processor) is combined. So in hybrid FSP at each stage there are one or more identical machines to process the tasks in contrast with the basic model where there is one machine in every stage. In the permutation flow shop, each machine processes the jobs in the same order. Thus, in this case the job sequence on the first machine is fixed it will stay the same for all the remaining machines. Finally an interesting generalization of the FSP is the multi-stage flexible flow-

shop environment. There are $k$ stages, each job consists of $k$ tasks and each task can be scheduled on a set of parallel processors (identical, unrelated, etc). The processors at each stage might be indistinguishable and the tasks of each job have to be scheduled in the order indicated by the stages from 1 to $k$.

### 1.2.3 Techniques to improve the solution efficiency

Scheduling problems are inherently hard to solve both in theory and in practise. There are cases where it is extremely hard even to come up with a feasible solution. To reduce the difficulty by solving a large Mixed Integer Linear Programming (MILP) problem, a number of techniques have been proposed to improve the solution efficiency by exploiting the characteristics of the problem. According to C. Floudas & Xiaoxia Lin [21] the exact methods that have been proposed for tackling such problems can be grouped in the following main categories:

- reformulation techniques that increase the size of the formulation and reduce the gap between the optimal solution and its linear programming (LP) relaxation counterpart, for example, Sahinidis and Grossmann (1991b) [22], Shah et al. (1993) [23] and Yee and Shah (1998) [24] reformulated the allocation and/or batch-sizing constraints based on variable aggregation/disaggregation
- cutting plane inequalities, which remain valid for the integer polytope, but cut off fractional solutions produced when solving a relaxation of the integer program such as those proposed by Dedopoulos and Shah (1995) [25] and Yee and Shah (1998) [24]
- intervening in the branch and bound solution procedure for instance, Shah et al. (1993) [23] developed ways to reduce the size of the relaxed LP and perform post analysis of the LP solution at each node of the branch and bound tree. Also Dedopoulos and Shah (1995) [25] proposed variable reduction techniques which are invoked during the branch and bound procedure
- decomposition that divides a large and complex problem to smaller subproblems,for example, Bassett, Pekny, et al. (1996) [26] proposed a number of time-based decomposition approaches and Elkamel et al. (1997) [27] developed another algorithm consisting of both spatial decomposition and temporal decomposition.

## 1.3 Case Description for BRC Ltd

BRC ltd is the UK's largest supplier of steel reinforcement and associated products for concrete. They fabricate cut & bent rebar to the specs of BS8666:2005 and governed by the independent steel reinforcement governing body C.A.R.E.S. In 2009 BRC was acquired by the Celsa Steel Services UK group and currently has 4 depots in the UK with the largest being in Newport South Wales which can produce up to 2000 tonnes of fabricated reinforcement for the construction industry per week. The rest are located in Romsey near Southampton, Mansfield in the midlands and Newhouse up in Scotland. BRC manufactures bespoke products for the construction industry with a lead-time of 5-7 days where each batch is unique and can be up to 2 tonnes of steel in one product batch. These can be in the form of simple straight bar, "U" shaped bars to complicated 99 shape codes where it could be 3D shapes. The process is to cut and shape from stock lengths of straight or coiled rebar and go through the flow process which will be explained with more details below.

Production transforms the stock into products which are placed by cranes in the finished product area. The orders (batches) are fulfilled by placing the various finished products, that these products are composed of, on to some trailers. At this phase there is a scanning procedure where each product gets a time stamp. When the order is complete the batch is ready for shipment to customer. All the material movements inside the production line are made by cranes which are a limited shared resource. The company reported that considering an additional crane is not an option due to space limitations.

After receiving the raw materials the company stocks them into its warehouse in bars or in coils. After receiving an order, they check whether there exist enough raw material to proceed into production. The coil material in order to be converted into final product can either bending in different shapes (it depends with the product code) or straightening and then cut to length. After the completion of

coil production, the final products wait to the finished product area to be loaded and transferred to the customers.

On the other hand the bars go through a more elaborate procedure to be a final product having more stages to be processed. A bar can be cut to length on shearlines or dispatch as mill lengths regarding the order. The next job after the cutting to lengths is either to dispatch the bar for shipping or threading and coupling. The next step is to ship the threaded or coupled bars or proceed to the final stage that is of bending. After being processed the final product is being dispatched.

Although the company has a large production capacity, the lack of a business intelligent system prevents it from finding an efficient production plan. The production mainly rely on some rules of thumb, like the loading procedure on the lay down area, and also based on people with many years of experience. In the current processing system the operator of each crane has a list of products that need to be moved but not an "optimised" order to do that. Thus the operator also applies some rules of thumb to do that. The principle in general suggests placing at the bottom of lay down area the straight bars, bent items are going next and small links at the very top. Since the is no picking system, the positioning and tracking of products/orders is rather problematic. While crane operators are looking for some products they move other finished products around. As a result a product might be under a lot of items when the operator is trying to locate it and that causes major delays.

Idle time is occurred due to delays of cranes. The processing of a product might have finished in some station but there might be a delay caused by a crane unavailability to move the item hence the machine remains idle at this point. That obstacle arises due to absence of real time data about the cranes' movements. The company within the Factlog program will install some sensors so as to give time stamps when the crane pick an item. Finally we will focus on job scheduling part and we will incorporate the time that a crane need to move an item to the machine *setup* time before start the production.

# 2 Literature Review

Over the last fifty years a considerable amount of research effort has been focused on deterministic and stochastic scheduling. In our case we will focus on deterministic Flow Shop problems. The number and variety of models considered is astounding. The FSP is one of the most complex scheduling problems, and finding an optimal solution for real size instances in a reasonable amount of time is practically infeasible. Sometimes is infeasible or at least extremely hard even to find a feasible solution, respecting all the constraint, and relatively 'fast'. During this time a notation has evolved that succinctly captures the structure of many (but not for sure all) deterministic models that have been considered in the literature.

In scheduling terminology a distinction is often made between a sequence, a schedule and a scheduling policy. A sequence usually corresponds to a permutation of the n jobs or the order in which jobs are to be processed on a given machine. A schedule usually refers to an allocation of jobs within a more complicated setting of machines, allowing possibly for preemptions of jobs by other jobs that are released at later points in time. The concept of a scheduling policy is often used in stochastic settings: a policy prescribes an appropriate action for any one of the states the system may be in. In deterministic models usually only sequences or schedules are of importance.

## 2.1 Classification of Scheduling Problems and Notation

In all the scheduling problems considered the number of jobs and the number of machines are thought to be finite. The number of jobs is denoted by $n$ and the number of machines by $m$. More often than not, the subscript $j$ refers to a job while the subscript $i$ refers to a machine. In case where a job demands several processing steps or operations then the pair (i, j) refers to the processing step/operation of job j on machine i. The forthcoming data are associated with job j.

**Processing time** ($p_{ij}$) The $p_{ij}$ represents the processing time of job j on machine i. The subscript i is omitted if the processing time of job j does not depend on the machine or if job j is only to be processed on one given machine.

**Release date** ($r_j$) The release date $r_j$ of job j may also be referred to as the ready date. It is the time the job arrives at the system, i.e., the earliest time at which job j can start its processing.

**Due date** ($d_j$) The due date $d_j$ of job j represents the committed shipping or completion date (i.e. the date the job is promised to the customer). Completion of a job after its due date is allowed, but then a penalty is incurred. When a due date must be met it is referred to as a deadline and denoted by $\bar{d}_j$.

**Weight** ($w_j$) The weight $w_j$ of job j is basically a priority factor, denoting the importance of job j relative to the other jobs in the system. For example, this weight may represent the actual cost of keeping the job in the system. This cost could be a holding or inventory cost; it also could represent the amount of value already added to the job [30].

The great variety of scheduling problems that exist in the literature motivates the introduction of a systematic notation that could serve as a basis for a classification scheme. Such a notation of problem types would greatly facilitate the presentation and discussion of scheduling problems. According to G. Vairaktarakis and H. Emmons [19], J. Blazewicz et al. [4] and P. Brucker [28] we will follow a notation proposed by Graham et al. (1979) [29].

A scheduling problem is described by a triplet $\alpha \mid \beta \mid \gamma$. The $\alpha$ field describes the machine environment or more precisely the type and size of the shop , the $\beta$ field provides details of processing characteristics and constraints and may contain a single entry, multiple entries, or not entries at all. Finally the $\gamma$ field describes the objective function and often contains a single entry.

### 2.1.1 Type and size

In the first field $\alpha = \alpha_1, \alpha_2$ is denoted the type and size of the shop. Parameter $\alpha_1 \in \{\varnothing, P, Q, R, O, F, J\}$ characterizes the type of processors used:
$\alpha_1 = \varnothing$ : single processor
$\alpha_1 = P$ : identical processors
$\alpha_1 = Q$ : uniform processors
$\alpha_1 = R$ : unrelated processors
$\alpha_1 = O$ : dedicated processors: open shop system
$\alpha_1 = F$ : dedicated processors: flow shop system
$\alpha_1 = J$ : dedicated processors: job shop system

Parameter $\alpha_2 \in \{\varnothing, k\}$ denotes the number of processors in the problem:
$\alpha_2 = \varnothing$ : the number of processors is assumed to be variable
$\alpha_2 = k$ : the number of processors is equal to k, where k is a positive integer

### 2.1.2 Processing Characteristics and Special Features

In the second field are listed the special features of the shop and describes task and resource characteristics. Parameter $\beta_1 \in \{\varnothing, pmtn\}$ indicates the possibility of task premption:
$\beta_1 = \varnothing$ : no preemption is allowed
$\beta_1 = pmtn$ : preemption (i.e interruption) of a task is allowed. Preemptions imply that is not necessary to keep a job on a machine, once started, until its completion. The scheduler is allowed to interrupt the processing of a job (preempt) at any point in time and put a different job on the machine instead. The amount of processing a preempted job already has received is not lost. When a preempted job is afterwards put back on the machine (or on another machine in the case of parallel machines), it only needs the machine for its remaining processing time. The preempted task may be resumed without penalty.

Parameter $\beta_2 \in \{\varnothing, res\}$ characterizes additional resources:
$\beta_2 = \varnothing$ : no additional resources exist
$\beta_2 = res$ : there are specified resource constraint.

Parameter $\beta_3 \in \{\varnothing, prec, uan, tree, chains\}$ reflects the precedence constraints:

$\beta_3 = \varnothing, prec, uan, tree, chains$ : denotes independent tasks, general precedence constraint, uniconnected activity networks, precedence constraints forming a tree or a set of chains respectively. **Precedence constraints** may appear in a single or in a parallel machine environment, requiring that one or more jobs may have to be completed before another job is allowed to start its processing. There are several special forms of precedence constraints: for example if each job has at most one predecessor and at most one successor, the constraints are referred as to *chains*. If each job has at most one successor, the constraints are referred as *tree*.

Parameter $\beta_4 \in \{\varnothing, r_j\}$ describes ready times:
$\beta_4 = \varnothing$: all ready times are zero
$\beta_4 = r_j$ : ready times differ per task.

Parameter $\beta_5 \in \{\varnothing, p_j = p, \underline{p} \leq p_j \leq \bar{p}\}$ describes task processing times:
$\beta_5 = \varnothing$ : tasks have arbitrary processing times
$\beta_5 = (p_j = p)$: all tasks have processing times equal to $p$ units
$\beta_5 = \underline{p} \leq p_j \leq \bar{p}$: no $p_j$ is less than $\underline{p}$ or greater than $\bar{p}$

Parameter $\beta_6 \in \{\varnothing, \tilde{d}\}$ describes deadlines:
$\beta_6 = \varnothing$ : no deadlines are assumed in the system (however, due dates may be defined if a due date that involves a criterion is used to evaluate schedules)
$\beta_6 = \tilde{d}$ : deadlines are imposed on the performance of a task set.

Parameter $\beta_7 \in \{\varnothing, n_j \leq k\}$ describes the maximal number of tasks constituting a job in case of job shop systems:
$\beta_7 = \varnothing$ : the above number is arbitrary or the scheduling problem is not a job shop problem
$\beta_7 = (n_j \leq k)$ : the number of tasks for each job is not greater than k

Parameter $\beta_8 \in \{\varnothing, \text{no-wait}\}$ describes a no-wait property in the case of scheduling on dedicated processors:
$\beta_8 = \varnothing$: buffers of unlimited capacity are assumed
$\beta_8 =$ no-wait: buffers among processors are of zero capacity and a job after finishing its processing on one processor must immediately start on the consecutive processor.
The no-wait requirement is another variation of flow shop problems. Buffers are not allowed between two successive machines. This implies that the starting time of a job at the first machine has to be delayed to ensure that the job can go through the flow shop without having to wait for any machine. An example of such an operation is a steel rolling mill in which a slab of steel is not allowed to wait as it would cool off during a wait. It is clear that under no-wait the machines also operate according to the FIFO (First In First Out) discipline.

### 2.1.3 Objective Function

In the third or $\gamma$ field we enter the criterion to be minimized (maximization criteria rarely arise naturally in scheduling problems, and can easily be converted to minimizations by sign reversal). We will sometimes refer to the objective as a cost function to emphasize that we are minimizing. Scheduling objectives are functions of the completion times of the tasks. Indeed, throughout this dissertation all objectives are functions of job completion times; that is, they depend only on the times that the last tasks of each job are completed. Properly, the criterion is a function of the schedule chosen, but we understand that it depends on the schedule only through the completion times that result. Some common objectives are encoded as follows [19, 30]:

- **Makespan ($C_{max}$)** : the maximal or latest completion time of any job. The makespan is defined as $\max(C_1, ..., C_n)$ where $C_i$ is the completion time on the last machine for job j. Also this criterion has been by far the most exhaustively studied. A minimum makespan usually implies a good utilization of the machine(s).

- ***Total weigted completion time*** $(\sum C_j)$: total weighted completion time of all jobs, $\sum_{j=1}^{n} w_j C_j$ which is equivalent to the mean completion time (they differ only by a constant factor n). The sum of the completion times is in the literature often referred to as the flow time. The total weighted completion time is then referred to as the weighted flow time.

- ***Maximum Lateness*** *($L_{max}$)* The maximum lateness of a job $j$ in a schedule is the difference between its completion time $C_j$ minus its due date $d_j$ ($L_j = C_j - d_j$) and is defined as $L_{max} = \max(L_1, ..., L_n)$. In fact, if a job completes before its due date, its lateness can be negative. So this criterion measures the worst violation of the due dates.

- ***Throughout***: is the number of jobs that complete their execution before their deadline.

- ***Total weighted tardiness*** $(\sum w_j T_j)$: the tardiness of job j is defined as

$$T_j = \max(C_j - d_j, 0) = \max(L_j, 0)$$

- ***Weighted number of tardy jobs*** $(\sum w_j U_j)$ The weighted number of tardy jobs is not only a measure of academic interest but it is often an objective in practise as it is a measure that can be recorded very easily. The difference between the tardiness and the lateness lies in the fact that the tardiness never is negative. The *unit penalty* of a job j is defined as

$$U_j = \begin{cases} 1 & \text{if } C_j > d_j \\ 0 & \text{otherwise} \end{cases}$$

  The lateness, the tardiness and the unit penalty are the three basic due date related penalty functions.

Sometimes we are concerned with two objectives, both of which we would like to make as small as possible. While we cannot generally minimize both simultaneously, we want our objective to somehow incorporate both goals. Such objectives are called multicriteria objectives. There are three ways to formulate them:

- **hierarchical objective,** $(A|B)$: minimize A subject to a constraint on B. For example, we might minimize total flow time while keeping the makespan under some prespecified upper bound, written $(\sum C_j | C_{max} \leq D)$.

- **composite objective,** $(\alpha A + \beta B)$ : minimize a linear combination of the two objectives, for prespecified relative weights $\alpha$ and $\beta$.

- **bicriteria objective, (A,B)**: instead of somehow combining the two objectives into one, we find the schedules that produce the "best" pairs of values, and present these options to the decision maker for final selection. To define "best", suppose arbitrary schedule $S_i$ has values $(a_i, b_i)$ for the two objectives (A,B) that we wish to make small. If $a_i \leq a_j$ and $b_i \leq b_j$, with at least one inequality strong, we say that $S_i$ dominates $S_j$ . Clearly, only non-dominated or efficient solutions are of interest, and it is the set of all these solutions that we wish to determine.

As a simple example of this notation, the classic problem of scheduling n jobs in a simple $m$-machine flow shop so as to complete all work as soon as possible is encoded $F_m||C_{max}$. Note that the simple shop contains all default assumptions, and thus there are no entries in the middle or $\beta$ field. Finally the $F_2||C_{max}$ problem can be solved in *Polynomial* time since it requires $O(n \log n)$ computational steps by using Johnson's algorithm (1954) [20]. When it comes to solution approaches, a variety of methods have been developed for different flow shop scheduling problems [31]. Johnson's rule has served as a basis for developing heuristic methods for general flow shop scheduling problems [32].

## 2.2 Solution methodologies

A great deal of research has been focused on solving flow-shop problems. As mentioned before, the FSP is among the hardest combinatorial optimization problems and due to its intractability several approaches has been developed trying to find an optimal or a feasible (for large instances) solutions. We will try to address some of the most popular methods for solving a flow shop problem using as main reference the *Hand Book on Scheduling* by J. Blazewicz et al. [4].

### 2.2.1 Exact Methods

In this section we will describe very briefly two general methods of solving many combinatorial problems, namely the method of **Dynamic Programming (DP)** and the method of **Branch and Bound (B&B)**. Few remarks should be made at the beginning. First, we will not go into details, since both methods are well documented, including the basic scheduling books [33, 34, 35]. Before passing to the description of the methods let us mention that they are of implicit enumeration variety, because they consider certain solutions only indirectly, without actually evaluating them explicitly.

#### 2.2.1.1 Dynamic Programming

Fundamentals of dynamic programming were elaborated by Bellman in the 1950's and presented in [36]. The name "Dynamic Programming" is slightly misleading, but generally accepted. A better description would be "recursive" or "multistage" optimization, since it interprets optimization problems as multistage decision processes. It means that the problem is divided into a number of stages, and at each stage a decision is required which impacts on the decisions to be made in later stages. Now, Bellman's principle of optimality is applied to draw up a recursive equation which describes the optimal criterion value at a given stage in terms of the previously obtained one. This principle can be formulated as follows: Starting from any current stage, an optimal policy for the rest of the process, i.e. for subsequent stages, is independent of the policy adopted in the previous stages. Of course, not all optimization problems can be presented as multistage decision processes for which the above principle is true. However, the class of problems for which it works is quite large. For example, it contains problems with an additive optimality criterion. If dynamic programming is applied to a combinatorial problem, like the flow shop one, then in order to calculate the optimal criterion value for any subset of size k, we first have to know the optimal value for each subset of size k-l. Thus, if our problem is characterized by a set of n elements, the number of subsets considered is $2^n$. It means that dynamic programming algorithms are exponential. However, for problems which are NP-hard (but not in the strong sense) it is often possible to construct pseudopolynomial dynamic programming algorithms which are of practical value for reasonable instance sizes. There are some very good DP models for flow shop problems and the interested reader is referred to [37, 38]

#### 2.2.1.2 Branch & Bound

Branch-and-Bound is another method for solving combinatorial optimization problems. It is based on the idea of intelligently enumerating all feasible solutions. To explain the details, we assume again that the discrete optimization problem P to be solved is a minimization problem (e.g $\min C_{max}$). We also consider subproblems of P which are defined by a subsets S′of the set S of feasible solutions of P. It is convenient to identify P and its subproblems with the corresponding subset $S' \subseteq S$. As its name implies, the branch and bound method consists of two fundamentals procedures: branching and bounding [28].

Branching is the procedure of partitioning a large problem into two or more sub-problems usually mutually exclusive. Furthermore, the sub-problems can be partitioned in a similar way, etc. Bounding calculates a lower bound on the optimal solution value for each subproblem generated in the branching process.

The choice of a node from the set of generated nodes which have so far neither been eliminated nor led to branching is due to the chosen search strategy. Two search strategies are used most frequently: *jumptracking* and *backtracking*. Jumptracking implements a frontier search where a node with a

minimal lower bound is selected for examination, while backtracking implements a depth first search where the descendant nodes of a parent node are examined either in an arbitrary order or in order of non-decreasing lower bounds. Thus, in the jumptracking strategy the branching process jumps from one branch of the tree to another, whereas in the backtracking strategy it first proceeds directly to the bottom along some path to find a incumbent solution and then retraces that path upward up to the first level with active nodes, and so on. It is easy to notice that jumptracking tends to construct a fairly large list of active nodes, while backtracking maintains relatively few nodes on the list at any time. However, an advantage of jumptracking is the quality of its incumbent solutions which are usually much closer to optimum than the incumbent solutions generated by backtracking, especially at early stages. So the things are needed for a B&B algorithm:

1. **Branching** : S is replaced by smaller problems $S_i (i = 1, ..., r)$ such that $\bigcup_{i=1}^{r} S_i = S$. This process is called **branching**. Branching is a recursive process, i.e. each $S_i$ is the basis of another branching. The whole branching process is represented by a branching tree. S is the root of the branching tree, $S_i (i = 1, ..., r)$ are the children of S, etc. The discrete optimization problems created by the branching process are called **subproblems**.

2. **Lower Bounding**: An algorithm is available for calculating a lower bound for the objective values of all feasible solutions of a subproblem. Sometimes is used a heuristic to find a solution for the subproblem.

3. **Upper Bounding** : We calculate an upper bound U of the objective value of P. The objective value of any feasible solution will provide such an upper bound. If the lower bound of a subproblem is greater than or equal to U (for minimization problem), then this subproblem cannot yield a better solution for P. Thus, we don't need to continue to branch from the corresponding node in the branching tree and we *prune* this tree by optimality. To stop the branching process in many nodes of the branching tree, the bound U should be as small as possible. Therefore, at the beginning of the branch-and-bound algorithm we apply some heuristic to find a good feasible solution with small value U. After branching many times we may reach a situation in which the subproblem has only one feasible solution. Then the lower bound LB of the subproblem is set equal to the objective value of this solution and we replace U by LB if LB $<$ U. The best objective value until iteration l is the incumbent value and the solution is named incumbent solution.

One of the most effective of the branch and bound algorithms is that of Potts [39]. Although this algorithm usually solves three-machine problems using only moderate computer resources, large search trees are commonly generated when there are four or more machines and over 20 jobs. For practical purposes, therefore, it is often more appropriate to apply a heuristic method which generates an approximate solution at relatively minor computational expense.

Summing up the above considerations we can say that in order to implement the scheme of the branch and bound method, i.e. in order to construct a branch and bound algorithm for a given problem, one must decide about

- the branching procedure and the search strategy
- the bounding procedure or elimination criteria

Making the above decisions one should explore the problem specificity and observe the compromise between the length of the branching process and time overhead concerned with computing lower bounds or trial solutions. However, the actual computational behavior of branch and bound algorithms remains unpredictable and large computational experiments are necessary to recognize their quality. It is obvious that the computational complexity function of a branch and bound algorithm is exponential in problem size when we search for an optimal solution. However, the approach is often used for finding suboptimal solutions, and then we can obtain polynomial time complexity by stopping the branching process at a certain stage or after a certain time period elapsed.

### 2.2.2 Heuristics and Approximation Algorithms

As already mentioned, scheduling problems belong to a broad class of combinatorial optimization problems. To solve these problems one tends to use optimization algorithms which for sure always find optimal solutions, namely the exact approaches like branch and bound and dynamic programming. However, not for all optimization problems, polynomial time optimization algorithms can be constructed. This is because some of the problems are $\mathcal{NP}$-hard. In such cases one often uses heuristic algorithms which tend toward but do not guarantee optimality for any instance of an optimization problem. Of course, the necessary condition for these algorithms to be applicable in practice is that their worst-case complexity function is bounded from above by a low-order polynomial in the input length. A sufficient condition follows from an evaluation of the distance between the solution value they produce and the value of an optimal solution. This evaluation may concern the worst case or a mean behavior.

As referred by J. Blazewicz et al. [4] there are a lot of approximation algorithms and heuristics to tackle the difficulty of a scheduling and more precisely in our case a flow shop problem. The approximation algorithms usually use a ratio which evaluates the quality of the solution achieved in compare with the optimal that could be obtained. Almost always we cannot have the optimal solution, since this is the required goal, so as to compare with our approximation solution. Instead we can use a lower bound (LB) to evaluate our solution and learn how far we are from the global optimum usually in percentage terms.

#### 2.2.2.1 Approximation Algorithms and Evaluation Method

The approximation algorithms actually are the heuristic algorithms with analytical evaluated accuracy. To be more precise, we give some definitions, starting with the worst case analysis [40]. If P is a minimization (maximization) problem, and I is any instance of it, we can define the ratio $R_A(I)$ for an approximation algorithm A as:

$$R_A(I) = \frac{A(I)}{OPT(I)} \qquad (R_A(I) = \frac{OPT(I)}{A(I)})$$

where A(I) is the value of the solution constructed by an algorithm A for instance $I$, and OPT(I) is the value of an optimal solution for $I$. The *absolute performance* ratio $R_A$ for an approximation algorithm A for problem P is then given as:

$$R_A = \inf\{r \geq 1 \,|\, R_A(I) \leq r \text{ for all instances of P}\}$$

The asymptotic performance ratio $A_A^\infty$ for A is given as:

$$R_A^\infty = \inf\{r \geq 1 \,|\, \text{for some positive integer K, } R_A(I) \leq r \text{ for all instances of P satisfying OPT(I)} \geq K\}$$

The above formulas define a measure of the "goodness" of approximation algorithms. The closer $R_A^\infty$ is to 1, the better algorithm A performs. However, for some combinatorial problems it can be proved that there is no hope of finding an approximation algorithm of a specified accuracy, i.e. this question is as hard as finding a polynomial time algorithm for any NP-complete problem.

Analysis of the worst-case behavior of an approximation algorithm may be complemented by an analysis of its mean behavior. This can be done in two ways. The first consists in assuming that the parameters of instances of the considered problem P are drawn from a certain distribution D and then one analyzes the mean performance of algorithm A.

In such an analysis it is usually assumed that all parameter values are realizations of independent probabilistic variables of the same distribution function. Then, for an instance $I_n$ of the considered optimization problem (n being a number of generated parameters) a probabilistic value analysis is performed. The result is an asymptotic value OPT($I_n$) expressed in terms of problem parameters. Then, algorithm A is probabilistically evaluated by comparing solution values $A(I_n)$ it produces with OPT($I_n$) [41]. The two evaluation criteria used are *absolute error* and *relative error*. The **absolute error** is defined as a difference between the approximate and optimal solution values:

$$\alpha_n = A(I_n) - OPT(I_n)$$

On the other hand, the **relative error** is defined as the ratio of the absolute error and the optimal value:

$$\beta_n = \frac{A(I_n) - OPT(I_n)}{OPT(I_n)}$$

Usually, one evaluates the convergence of both errors to zero. Three types of convergence are distinguished. The strongest with almost sure convergence for a sequence of probabilistic variables $y_n$ which converge to constant c is defined as:

$$\Pr\{\lim_{n\to\infty} y_n = c\} = 1$$

The latter implies a weaker convergence in probability, which means that fpr every $\varepsilon > 0$:

$$\lim_{n\to\infty} Pr\{|y_n - c| > \varepsilon\} = 0$$

The above convergence implies the first one if the following additional condition holds for every $\varepsilon > 0$ :

$$\sum_{j=1}^{\infty} Pr\{|y_n - c| > \varepsilon\} < \infty$$

Finally, the third type of convergence which is *convergence in expectation* holds if

$$\lim_{n\to\infty} |E(y_n) - c| = 0$$

It follows from the above definitions, that an approximation algorithm A is the best from the probabilistic analysis point of view if its absolute error converges to 0. Algorithm A is then called *asymptotically optimal*.

It is rather obvious that the mean performance can be much better than the worst case scenario, justifying the use of a given approximation algorithm. A main obstacle is the difficulty of proofs of the mean performance for realistic distribution functions. Thus, the second way of evaluating the mean behavior of heuristic algorithms are computational experiments, which is still used very often. In the latter approach the values of the given criterion, constructed by the given heuristic algorithm and by an optimization algorithm are compared. This comparison should be made for a representative sample of instances. There are some practical problems which follow from the above statement and they are discussed with more details in [42].

#### 2.2.2.2 Heuristic Algorithms

In this section we will describe some polynomial heuristic algorithms to find approximate solutions to the flow shop problem. Solutions obtained from these heuristics can be used as upper bounds in an optimization algorithm or as an initial solution in a meta-heuristic algorithm. In addition, the computational effort so as to solve the problem to optimality with an exact optimization algorithm is restrictive in the majority of cases. Thus, heuristics can provide quality solutions at short computational time.

In general, we can classify most of the heuristic algorithms for the m-stage flow shop into three categories: the application of Johnson's two-machine algorithm, the use of a slope index for the job processing times, and the minimization of idle time on machines.

Optimizing techniques are often more practical for small size flow shop scheduling problems. The exceptions are Johnson's rule for the two-stage problem and its extension to optimize three-stage flow shop for minimizing makespan. Hence, Johnson's rule has been the basis of many m-stage flow shop scheduling heuristics. Palmer (1965) first proposed a heuristic for the flow shop scheduling problem to minimize makespan. The heuristic generates a slope index for jobs and sequences them in descending order of the index. Campbell, Dudek, and Smith (1970) developed a heuristic that is a generalization of Johnson's algorithm. The Campbell, Dudek, and Smith (CDS) heuristic performs better as compared to Palmer heuristic.

Hundal and Rajgopal (1988) made an improvement in the Palmer heuristic algorithm: they compute two additional sequences based on two modified versions of the slope index. Gupta (1972)

presented the minimum idle time (MINIT) algorithm based on the minimization of idle time at the last machine. Dannenbring (1977) proposed a variation of the CDS heuristic, the rapid access procedure. Yang et al. (1984) found that the rapid access with close order search heuristic compares favorably with the CDS heuristic. Also, they combined the CDS algorithm and Dannenbring rapid access procedure. This ensuing heuristic ranks higher than the CDS algorithm for makespan problems. Nawaz, Enscore, and Ham (1983) proposed that a job with larger total processing time should have higher priority in the sequence.They found their heuristic to be superior to the CDS algorithm. According to Shaukat Ali Brah et all, we considered the following five flow shop heuristics for a flow shop scheduling problem with makespan and mean flow time criteria [43].

#### 2.2.2.3 Campbell, Dudek, and Smith heuristic algorithm (CDS1)

The Campbell, Dudek, and Smith (1970) heuristic algorithm is a generalization of Johnson's algorithm. This algorithm generates a set of $m$-1 artificial, two-machine problems from the original $m$-machine problem. Next, it solves the two-machine problem using Johnson's algorithm; the best sequence becomes the solution to the $m$-machine problem.

#### 2.2.2.4 Nawaz, Enscore, and Ham heuristic algorithm (NEH)

Nawaz, Enscore, and Ham (1983) suggested that higher priority should be given in the sequence of a job with a larger total processing time. The heuristic algorithm arranges jobs in decreasing order of the total process time. The first two jobs form the initial schedule follow an adjacent pair-wise interchange to arrive at the best partial sequence. The algorithm inserts a third job into the previous partial sequence at the position that give the best makespan or mean flow time, depending on the criterion under consideration. Nevertheless, the relative position of the two previous jobs remains stable. The algorithm repeats the process for the remaining jobs according to the initial ordering of total process time.

#### 2.2.2.5 Hundal and Rajgopal modified Palmer heuristic algorithm (PAM)

Hundal and Rajgopal (1988) made an improvement in the Palmer algorithm: they compute two additional sequences based on two modified versions of the slope index. They claim their modification yields a better sequence for an odd number of machines and same or better otherwise. Also, they found the increase in computational time to be minimal. The algorithm generates three sequences by ordering the jobs in descending order of the slope index. We choose the sequence that optimizes the makespan criterion and the sequence that gives the best mean flow time.

#### 2.2.2.6 Yang, Pegden, and Enscore combined heuristic algorithm (CDS2)

Yang, Pegden, and Enscore (1984) combined the CDS heuristic and the Dannenbring rapid access procedure. The heuristic applies the weighting scheme for the rapid access procedure's slope index to the CDS heuristic. Similar to the CDS heuristic, it creates m-1 artificial two-machine sub-problems and solves them by Johnson's rule. The chosen sequence is the best one among the investigated sequences. This algorithm is similar in computation to CDS1 except that it calculates the process times differently, being adapted from the Dannenbring rapid access procedure.

#### 2.2.2.7 Ho heuristic algorithm (HO)

Ho (1995) developed a heuristic to minimize the mean flow time. It uses three sorting methods: *bubble-sort, insertion sort* and *non-adjacent pair-wise interchange.* Ho algorithm develops an initial solution based on a slope index similar to that of Palmer and Dannenbring. It forms an initial sequence by arranging jobs in ascending order of their slope index. Subsequently, it performs a bubble sort on the initial sequence and transposes adjacent jobs only if an improvement in makespan or mean flow time occurs for the respective criteria. The current solution goes through further sorting first by the insertion sort method and then by the non-adjacent pair-wise interchange method, and transpose jobs

if an improvement in the objective value occurs. The sorts terminate if the pass fails to make any improvement. This new solution goes through a bubble sort again. It repeats the whole process until the number of repetitions equals n-4, or the new objective function value is equal to the one noted earlier.

Panwalker and Iskander (1977) have carried out the most well known and comprehensive survey of scheduling heuristics. A total of 113 priority dispatching rules are presented, reviewed and classifies. They categorized these rules as per the specific area of application [44]. Finally we will introduce one of the most popular heuristic algorithm for flow shop scheduling problems called *Shifting Bottleneck (SB).*

#### 2.2.2.8 Shifting Bottleneck (SB)

The Shifting Bottleneck (SB) algorithm, developed by Adams et al. [15] is among the most popular and powerful heuristic solution method for the Job Shop Scheduling Problem ($J||C_{max}$). SB is an iterative algorithm that decomposes the job shop scheduling problem into single machine scheduling subproblems (i.e., the single machine scheduling problem with release dates and due dates, where the objective is to minimize the maximum lateness, $1|r_j|L_{max}$). These single machine scheduling problems are then solved and the machine with the largest maximum lateness or longest makespan called the *bottleneck machine*, is given priority in the schedule. Whenever a new machine is sequenced, the sequence of each antecedently sequenced machine is going to be submitted in a reoptimization process. The performance which entails the quality of solution and the computational complexity of the SB algorithm, is entirely dependent on the algorithm used for solving the emerging subproblems. The SB comprises of two subroutines: the first one (SB I) repetitively solves one-machine scheduling problems; the second one (SB II) builds a partial enumeration tree where each path from the root to a leaf is similar to an application of SB I. Flow shop ($F||C_{max}$)is a special case of the job shop scheduling problem, where the order in which the jobs must pass through machines is identical for all jobs. Therefore, the SB heuristic could be a promising method for solving flow shop scheduling problems [45, 46].

In the original SB heuristic, a pure dispatching rule was used to solve the single machine scheduling subproblems. Later on, alternative extensions of the SB method have been produced. They have been principally concentrated on making the original SB heuristic applicable to problems with other objective functions, and simultaneously proposing solution methods for the subproblems in order to make the SB heuristic more efficient. Balas et al [47] proposed a heuristic subproblem solution method and used it in a modified SB method. What is more, Ovacik and Uzsoy [48, 49] proposed extensions to the SB heuristic for more complex job shop scheduling problems with applications to semiconductor testing.

Wenqi and Aihua [50] clearly state that two improved extensions of the SB heuristic method that guarantee feasible solutions for any given instances. Based on the conducted experimental analysis, the improved extensions of the SB method proved to be more efficient than the original SB method for most of the tested instances. Monch et al. [52] extended the SB method to a more complex job shop scheduling problem with parallel batching machines where the objective is to minimize the total weighted tardiness, and machines have sequence-dependent setup times. They used a genetic algorithm to solve the subproblems.

The performance (computational complexity and solution quality) of the SB heuristic has been investigated by several researchers. Holtsclaw and Uzsoy [53] conducted a performance analysis on the classic SB heuristic by testing 320 randomly generated instances for the problem of minimizing maximum lateness. Demirkol et al. [54] organised an extensive computational study on the performance of SB with minimization of the makespan as the objective. They studied the effect of different subproblem solution methods, bottleneck selection criteria, and reoptimization strategies on the performance of the SB heuristic. According to the computational results, one of the most decisive factors in the performance of the SB heuristic is the algorithm employed in solving the subproblems. These results designate that instead of solving subproblems by some dispatching rules, using more complex algorithms that give better quality solutions to subproblems yield a better quality solutions to the main problem. The main drawback of the algorithms, which are capable of finding good quality solutions,

is their high computational complexity. The outline of the SB I is depicted below.

**Algorithm 1** The shifting bottleneck procedure (SB I)

1. Set S = $\emptyset$ and make all machines unsequenced
2. Solve a one-machine scheduling problem for each unsequenced machine
3. Among the machines considered in Step 2, find the bottleneck machine and add its schedule to S. Make the machine sequenced.
4. Reoptimize all sequenced machines in S
5. Go to step 3 unless S is completed; otherwise stop

### 2.2.3 Metaheuristics

In this part we will refer to the metaheuristic content. In our time and era, more generally applicable heuristic algorithms for combinatorial optimization problems became known under the name *local search*. Principally, they are designed as universal global optimization methods operating on a highlevel solution space in order to guide heuristically lower-level local decision rules' performance to their best outcome. Therefore, local search heuristics are often called *Metaheuristics* or strategies with knowledge-engineering and learning capabilities reducing uncertainty while knowledge of the problem setting is exploited and acquired in order to improve and accelerate the optimization process.

As a matter of precision, we will insert some of the most popular and powerful metaheuristics developed for flow shop scheduling and in our case we will focus on $F_m|perm|C_{max}$. The $\beta$ field declares that we have a permutation flow shop problem which means that permutation schedules are taken into account. Those schedules in which the same job order is maintained at all machines. So the sequence in which the jobs are to be processed is the same for each machine. In addition for hybrid shops, the concept of a permutation schedule may be extended as follow: given an ordering or job list, each job is scheduled sequentially in list order to the successive work stations at the earliest feasible time that a machine at that stage becomes available. In the literature, this restriction to permutations is sometimes imposed when the general problem is intractable, to simplify finding a useful, even if it isn't the optimum, but a suboptimal solution. Also in some shops, it may be a technological necessity or a managerial requirement that no job ever overwhelm another.

Metaheuristics make extensive use of *neighborhood search*, as do iterative heuristics. However, they are also equipped with mechanisms that permit the searching procedure to deviate from local optima – temporarily accepting inferior solutions – so as to direct the search to other (hopefully more promising) areas of the search space. Metaheuristic algorithms are mimetic in nature and include *simulated annealing (SA), tabu search (TS), genetic algorithms (GA), ant colony optimization (ACO), Neural Networks* etc and also hybrids thereof.

#### 2.2.3.1 Simulated Annealing

According to G. Vairaktarakis, H. Emmons and J. Blazewicz et al. [19, 4] Simulated Annealing was proposed as a framework for the solution of combinatorial optimization problems by Kirkpatrick, Gelatt and Vecchi and, independently, by Cemy [56, 55]. It is based on a procedure originally devised by Metropolis et al. used in physics to simulate the annealing (or slow cooling) of solids, after they have been heated to their melting point. The current benchmark for SA algorithms is the implementation of Osman and Potts (1989). It utilizes two ways to improve upon an incumbent solution:

- interchanging the positions of a pair of jobs in the permutation
- moving one job in a pair to a position right after the other

Both types of job movements are examples of *descent* algorithms that seek to reduce the makespan value. A descent method repeatedly attempts to construct an improved sequence from a current

sequence. We refer to the first way as a *swap* and to the second as an *insertion*. Swaps and insertions may be done in a predetermined order that captures all $\binom{n}{2}$ possible pair combinations, or randomly. In their experiments on randomly generated problems with $m \leq 20$ and $n \leq 100$, Osman and Potts found that their SA outperforms CDS (Campbell,Dudek and Smith heuristic algorithm) and NEH (Nawaz,Enscore and Ham heuristic algorithm) when these two do not benefit by a decent method to improve the starting solution. When CDS and NEH are so improved, and the pairs of jobs are chosen in a predetermined order, it is found that the two methods are comparable in solution quality. Moreover, compared to NEH, the Osman and Pott's SA algorithm finds the better solution for 82.5% of the problems while for the remaining 17.5% there is a tie – thus giving a definite advantage to SA.

It is also found that this SA algorithm performs far better when insertions are used instead of swaps, and surprisingly, when the insertion pairs are chosen randomly. It was also found that, the starting solution used in SA makes a statistically significant difference. Rad et al. (2009) showed that using their $NEH1_L$ solution as the seed, yields on average about 2.3% better makespan performance for the problems in the Taillard suite compared to using the NEH solution as the seed.

Ogbu and Smith (1990a) developed a similar SA algorithm where the initial schedule is produced by Palmer's slope algorithm, and the RA, RACS and RAES heuristics of Dannenbring (1977). Ogbu and Smith (1990b) compared the performance of their SA algorithm against the one by Osman and Potts giving a slight advantage to the latter. Alternative SA implementations exhibiting robust performance with respect to temperature cooling are presented by Ishibuchi et al. (1995) who show that the solution quality is comparable to that of Osman and Potts. Zegordi et al. (1995) used a so-called move *desirability* index for each job within the SA framework, resulting in faster convergence. Still, the overall solution quality is inferior to the one of Osman and Potts for the problems tested.

In simulated annealing procedures, the sequence of solutions does not roll monotonically down towards a local optimum, as was the case with local search. Instead, the solutions fluctuate randomly through the feasible set $\mathcal{S}$, and this fluctuation is loosely guided in a "favorable" direction. To be more specific, we describe the $k^{th}$ iteration of a typical simulated annealing procedure, starting from a current solution $x$. First, a neighbor of x, say $y \in \mathcal{N}(x)$, is selected (usually, but not necessarily, at random). Then, based on the amplitude of $\Delta := \delta(x) - \delta(y)$, a transition from $x \rightarrow y$ (i.e., an update of x by y) is either accepted or rejected. This decision is made non-deterministically: the transition is accepted with probability $p_k(\Delta)$, where $p_k$ is a probability distribution depending on the iteration count k. The intuitive justification for this rule is as follows. In order to avoid getting trapped early in a local optimum, transitions implying a deterioration of the objective function (i.e. with $\Delta < 0$) should be occasionally accepted, but the probability of acceptance should nevertheless increase with $\Delta$. Moreover, the probability distributions are chosen so that $p_{k+1}(\Delta) \leq p_k(\Delta)$. In this way, escaping local optima is relatively easy during the first iterations, and the procedure explores the set $(S)$ freely. But, as the iteration count increases, only improving transitions tend to be accepted, and the solution path is likely to terminate in a local optimum. The procedure stops if the value of the objective function remains constant in T (a termination parameter) consecutive iterations, or if the number of iterations becomes too large. In most implementations, and by analogy with the original procedure of Metropolis et al. [57], the probability distributions $p_k$ take the form:

$$p_k(\Delta) = \begin{cases} 1 & \text{if } \Delta \geq 0 \\ e^{c_k \Delta} & \text{if } \Delta < 0 \end{cases}$$

where $c_{k+1} \geq c_k \geq 0$ for all k, and $c_k \rightarrow \infty$ when $k \rightarrow \infty$. A popular choice for the parameter $c_k$ is to hold it constant for a number L(k) of consecutive iterations, and then to increase it by a constant factor: $c_{k+1} = \alpha^{k+1} c_0$, where $c_0$ is a small positive number, and $\alpha$ is slightly larger than 1. The number T(k) of solutions visited for each value of $c_k$ is based on the requirement to achieve a quasi equilibrium state. Intuitively this is reached if a fixed number of transitions is accepted. Thus, as the acceptance probability approaches 0 we would expect T(k)$\rightarrow \infty$. Therefore T(k) is supposed to be bounded by some constant $\beta$ to avoid long chains of trials for large values of $c_k$. According to the above statement, it is clear that the choice of the termination parameter and the distributions $p_k$ strongly influences the performance of the procedure. For example if $\beta$ is small and $\alpha$ is large then simulated annealing tends to behave like local search, and gets trapped in local optima of poor quality.

Moreover the idea is not to accept transitions with a certain probability that changes over time but to accept a new solution if the amplitude $\Delta$ falls below a certain threshold which is lowered over time. To sum up, as a general rule, one may say that simulated annealing is a reliable procedure to use in situations where theoretical knowledge is scarce or appears difficult to apply algorithmically. Even for the solution of complex problems, simulated annealing is relatively easy to implement, and usually outperforms a hill-climbing (a powerful heuristic) procedure with multiple starts [4].

#### 2.2.3.2 Tabu Search

Tabu search based algorithms have also attracted significant attention and is a general framework, which was originally proposed by Glover, and subsequently expanded in a series of papers [58, 59, 60, 61, 62]. The incumbent solution is transformed into another solution through appropriate *moves* like swaps, insertions, etc., chosen from a neighborhood. Some times we are stuck at a local minimum and allow to do some movements in which the objective value worsen. Unfortunately, this may lead to cycling and return to the same solution every few steps: $S^0 \rightarrow S^1 \rightarrow S^2 \rightarrow S^0$.... So to avoid this cycling problem, some solutions (moves) are forbidden,that is, they become ***tabu***. We build a tabu list with the recent solutions that are forbidden and it is not allowed to go back to those solutions. Usual parameter are the *size* of the tabu list and the *number* of iterations that a certain move is tabu. One of the central ideas in this proposal is to guide deterministically the local search process out of local optima (in contrast with the non-deterministic approach of simulated annealing). This can be done using different criteria, which ensure that the loss incurred in the value of the objective function in such an "escaping" step (a move) is not too important, or is somehow compensated for. The algorithm terminates when an iteration or time limit is reached.

A straightforward criterion for leaving local optima is to replace the improvement step in the local search procedure by a "least deteriorating" step. One version of this principle was proposed by Hansen under the name *steepest descent mildest ascent* [61]. In its simplest form, the resulting procedure replaces the current solution x by a solution $y \in \mathcal{N}(x)$ which maximizes $\Delta := \delta(x) - \delta(y)$. If during L (a termination parameter) iterations no improvements are found, the procedure stops. Notice that $\Delta$ may be negative, thus resulting in a deterioration of the objective function. Now, the major defect of this simple procedure is readily apparent. If $\Delta$ is negative in some transition from $x$ to $y$, then there will be a tendency in the next iteration of the procedure to reverse the transition, and go back to the local optimum $x$ (since $x$ improves on $y$). Such a reversal would cause the procedure to oscillate endlessly between $x$ and $y$ without any improvement. Therefore to deal with the above issue, throughout the search a (dynamic) list of forbidden transitions, called *tabu list* is maintained.

The purpose of this list is not to rule out cycling completely (this would in general result in heavy bookkeeping and loss of flexibility), but at least to make it improbable. In the framework of the steepest descent mildest ascent procedure, we may for instance implement this idea by placing solution $x$ in a tabu list TL after every transition away from x. In effect, this amounts to deleting x from $\mathcal{S}$. However, for reasons of flexibility, a solution would only remain in the tabu list for a limited number of iterations, and then should be freed again. To be more specific the transition to the neighbor solution, i.e. a move, may be described by one or more attributes. These attributes (when properly chosen) can become the foundation for creating a so-called *attribute based memory*. For example, in a 0-1 integer programming context the attributes may be the set of all possible value assignments (or changes in such assignments) for the binary variables. Then two attributes which denote that a certain binary variable is set to 1 or 0, may be called *complementary* to each other. A move may be considered as the assignment of the compliment attribute to the binary variable. That is, the complement of a move cancels the effect of the considered move. If a move and its complement are performed, the same solution is reached as without having performed both moves. Moves eventually leading to a previously visited solution may be stored in the tabu list and are hence forbidden. The tabu list may be derived from the running list (RL), which is an ordered list of all moves (or their attributes) performed throughout the search. That is, RL represents the trajectory of solutions encountered.

Each iteration consist of two parts: The guiding or tabu process and the application process. The tabu process updates the tabu list hereby requiring the actual RL; the application process chooses the best move that is not tabu and updates RL. For faster computation or storage reduction both

processes are often combined. The application process is a specification on, e.g., the neighborhood definition and has to be defined by the user. The tabu navigation method is a rather simple approach requiring one parameter l called tabu *list length*. The tabu navigation method disallows choosing any complement of the l most recent moves of the running list in order to establish the next move. Hence, the tabu list consists of a (complementary) copy of the last part of RL. Older moves are disregarded.

Tabu search encompasses many features beyond the possibility to avoid the trap of local optimality and the use of tabu lists. Due to space limitations we cannot discuss them all in this thesis. However I would like to mention two of them, which provide interesting links with artificial intelligence and with genetic algorithms. In order to guide the search, Glover suggests recording some of the salient characteristics of the best solutions found in some phase of the procedure (e.g., fixed values of the variables in all, or in a majority of those solutions, recurring relations between the values of the variables, etc.). In a subsequent phase, tabu search can then be restricted to the subset of feasible solutions presenting these characteristics. This enforces what Glover calls a "regional intensification" of the search in promising "regions" of the feasible set. An opposite idea may also be used to "diversify" the search. Namely, if all solutions discovered in an initial phase of the search procedure share some common features, this may indicate that other regions of the solution space have not been sufficiently explored. Identifying these unexplored regions may be helpful in providing new starting solutions for the search. Both ideas, of search intensification or diversification, require the capability of recognizing recurrent patterns within subsets of solutions.

As mentioned before, Tabu search can be applied in a more advanced way to incorporate theoretical foundations and another practical approaches. Due to wide use some other concepts have been created like the *reverse elimination method* or the *reactive tabu search* incorporating a memory employing simple reactive mechanisms that are activated when recurrence of solutions are discovered throughout the search [58]. My research in tabu search topic relied on a couple of paper and books and mainly on J.Blazewicz et al. [4], and G. Vairaktarakis et al. [19].

#### 2.2.3.3 Genetic Algorithm

Genetics is a biological term. Biologically, genes of a good parent produce better children (offspring). The same concept underlies the development of GA. Genetic algorithms based on principle of natural selection and belong to a bigger family of evolutionary algorithms, which create solutions using some techniques came from the nature development. Those techniques may rely on *evaluation, parent selection, crossover, mutation, population selection etc.* A GA begins by creating an initial population of chromosomes and tries to imitate the natural progress taking a finite population of solutions $S_1, S_2, ..., S_n$ and cross their individuals over so that new stronger and better individuals are born and at the same time the weaker die and removed. They have been designed as general search methods working on a pool of feasible solutions. Those algorithms can identify and work out the properties which good solutions have in common so as to create some even better.

A GA heuristic using some routines tries to generate useful solutions for optimization and in our case flow shop scheduling problems. The process begins with a set of individuals which is called a *Population*. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as *Genes*. Genes are joined into a string form a *Chromosome* (solution). When we have some feasible solutions generated by a heuristic algorithm, we can use a GA heuristic to develop new solutions using some genetics operations which they will be explained below. The new solution or "child" solution contain the strongest characteristics or genes from a pair of "parent" solutions. Every time that a new child is born, new parents are selected and the whole process continues until a new population is created [19].

Three popular types of genetic operators that are used to when a new population is constructed are: Selection, Crossover operators and Mutation operators.

1. **Selection** : to begin with this step the fitness of the individual is evaluated using an evaluation method. A new temporary population is created where is individual is a cope of the old population. As in the local search methods, a fitness function determines how fit an individual is. After the evaluation the individual get a fitness score. Based on its fitness score there is a

probability that the individual will be selected for reproduction. The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Individuals with high fitness have more chance to be selected for reproduction. With this way certain pairs of solutions (parents) can be selected if their score is large enough. In addition based on their fitness a whole new population can be generated replacing or adding to the old one. A major disadvantage in this process is that there isn't any new information to the children. This issue can be tackled by crossover operator.

2. **Crossover** : Crossover is a method for sharing information between chromosomes. In the literature there exist a crossover rate $P_c$ which actually is the number of crossover operations that allowed into the population [63]. Sometimes the population is randomly partitioned into pairs so as to crossover operator can be applied. Each pair of parents combines to produce one or more new solutions (or offspring) and the crossover operator is applied with a certain probability by randomly choosing a position in the string and exchanging the tails of the two strings (this is the simplest version of a crossover) [4]. To be more precise a way to do the crossover is by representing a scheduling solution S as a string $x_1, x_2...x_n$. Given two strings $x_1, x_2...x_n$ and $y_1, y_2...y_n$ there are three possible ways to do crossover:

   - 1-point crossover : In a scheduling problem choose an integer $p \in \{1, ..., r-1\}$ as a location in the permutation. Then all the jobs beyond this point swapped between the two parents and the two new children will be

     $$x_1...x_p y_{p+1}...y_r \quad \text{and} \quad y_1...y_p x_{p+1}...x_r$$

   - 2-point crossover: Similarly, choose integers $p, q \in \{1, ..., r-1\}$, with $p < q$. Then the two children will be

     $$x_1...x_p y_{p+1}...y_q x_{q+1}...x_r \quad \text{and} \quad y_1...y_p x_{p+1}...x_q y_{q+1}...y_r$$

   - Uniform crossover : The result is a child $u_1...u_r$ where each $u_i$ is uniformly chosen from $\{x_i, y_i\}$ [64].

   The effect of the crossover is that certain properties of the individuals chromosomes are combined to new ones or other properties are destroyed. The construction of a crossover operator should also take into consideration that fitness values of offspring are not too far from those of their parents, and that offspring should be genetically closely related to their parents.

3. **Mutation** : To prevent convergence to a local optimum point, the mutation operator can be used. This operator, which makes random changes to single elements, tries to maintain the diversity inside the population so some of the offspring are randomly modified. Like crossover operators, there is a range of mutation operators. In the literature, a classic approach of a mutation operator include a probability that an arbitrary position in a permutation will be changed from its original state [19]. Moreover the mutation operator can be defined as a simple 1-point mutation and below are quoted the steps [63]:

   (a) Define $\mathcal{S}$ the job sequence with length n, like the number of jobs

   (b) Generate two numbers between 1 and n randomly

   (c) Cut the job concerned with the first number

   (d) Paste the job in the position concerned with the second number

   So a common method of implementing the mutation operator involves generating a random variable for each position in the permutation. These random variables tell whether or not each position is modified through swaps or insertions. Thus, several changes would be made sequentially. These processes ultimately result in the next generation population of chromosomes (e.g. job permutations) that is different from the previous generation. Generally the average fitness will have increased by this procedure for the population, since only the best genes are selected for breeding. Finally, a small proportion of less fit solutions are included, for reasons of diversity so as to avoid getting trapped in local optima.

# 3 Problem Description

We can segment the BRC factory into distinctive parts where different processes take place. Looking at figure (1) that displays the factory layout we see that it is segmented vertically into the left and right part responsible for the production of *coils* and *bars* respectively. Additionally distinct places are:

- A : Stock Coil (left) and Stock Bars (right) is stored in different places relevant to diameter
- B : For the 3 bays different cranes transfer the raw material from A to any other of the three B's in order to always have stock to feed in the machines. When a crane operator observes that in any B there is a shortage of raw material, either coils or bars, he/she proceeds to move to A, pick up respective raw material and deposit to respective B
- C: For the 3 bays this place denotes the theoretical position where the trailers that will carry the final products to the end customer(s) are located in order to be filled with completed products towards forming completed orders
- D: Once a C is considered as full it proceeds to location D in order to be ready to leave the factory towards delivery to the end customer

Below we present the figure (1) which shows the whole production line. We can observe all the 3 Bays and each production stage for either coils or bars material. Moreover we will explain in more detail the raw material procedure and how the final products are moved until they are ready for shipment.

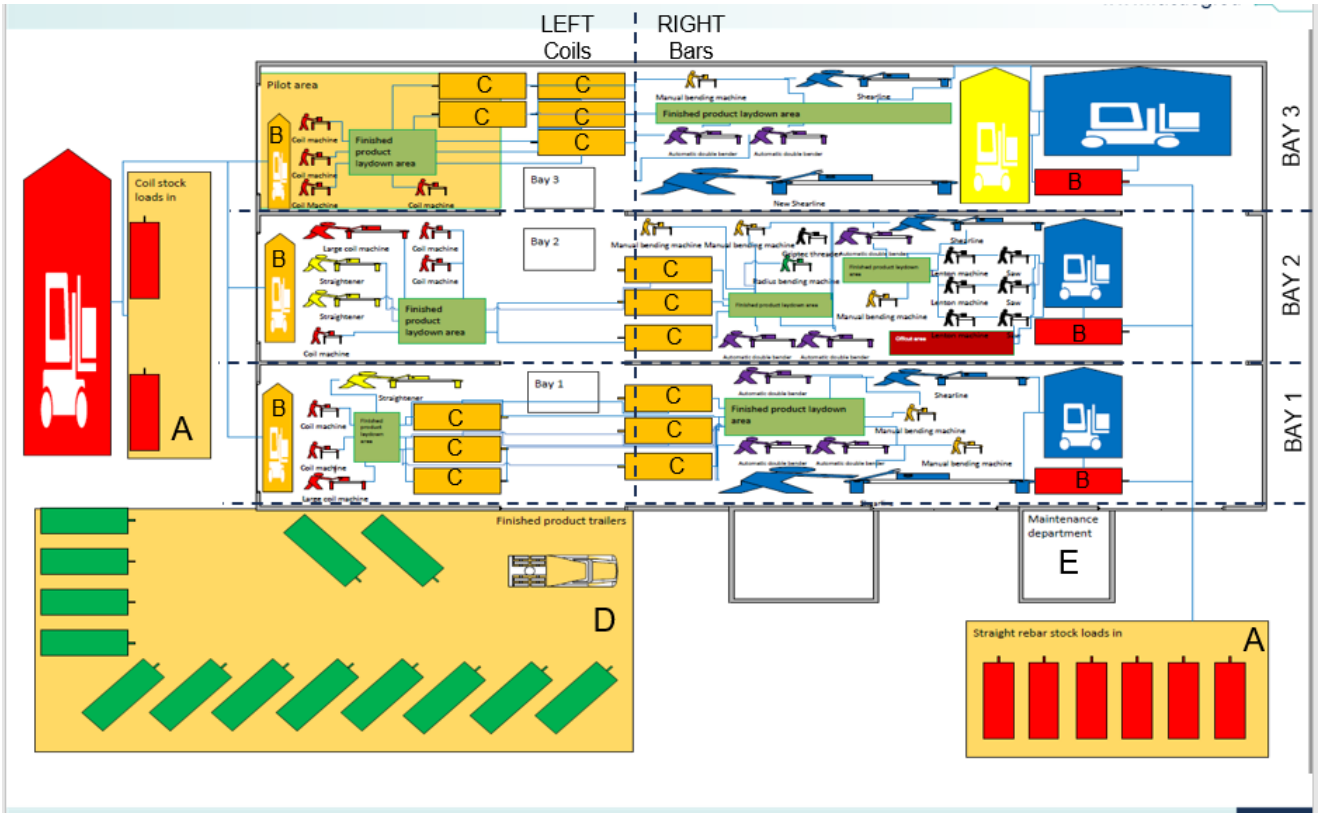


Figure 1: BRC Facility Layout

In the above figure (1) there are three different types of trailers like Red, Yellow and Green, in a sense they correspond to the production cycle of BRC. The stock is piled in the Red trailers. Production transforms the stock into products which are placed by cranes in the "finished product area" (green block in the layout diagram). In the yellow trailers, the various orders (batches) are fulfilled by placing the various finished products that these products are composed of. Scanning takes

place at this phase. This means the product is given a time that was scanned in the yellow trailers. When the order is complete then the trailer becomes green, which means that the order can be shipped.

**Material Transportation and products notes**

- From A to B transportation of raw material is conducted by cranes.
- From B to machine, machine to finished product area (or C), machine to another machine (with exceptions of automated), and finished product area to C area all transportation is conducted by cranes that are dedicated on bays and move on rails.

After consultation with the company this dissertation will focus on bay 3, and more precisely on the flow shop scheduling problems for both coil and bar area. So given the orders in a specific time horizon, our goal will be to find this schedule that optimize the specific *Key Performance Indicators (KPIs)*. In our case we will try to minimize the makespan $C_{max}$ and the total lateness of the jobs $L_j$.

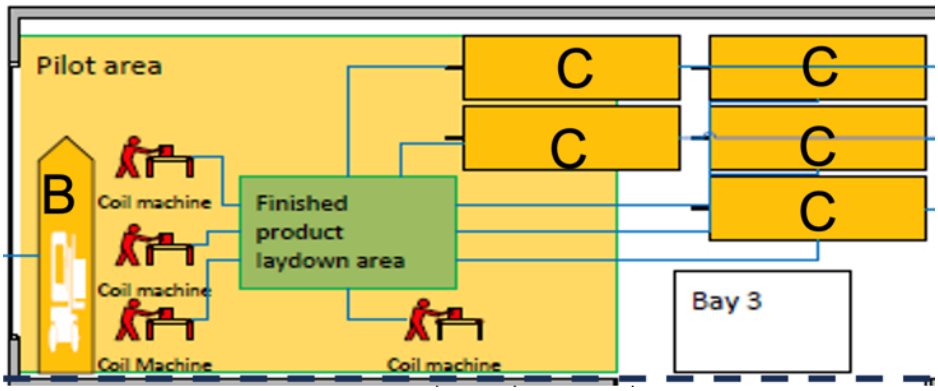


Figure 2: Coil Area at Bay 3

In figure 2 we can observe that the coil area in bay 3 has four coil machines and every finished product placed on the finished product laydown area. Cranes feed location B from stock coil location A with coils of appropriate diameter. We consider location B that is the location from where the input of the raw coils is conducted into machines as having always the appropriate raw coils and always are available. Then from area B, and in relation to the transfer of the raw coils to the coil machines, one of the fixed movement cranes is responsible for loading them on the machines. Currently the coil machine operator identifies an available crane and signals for a request to load. The installation of cranes' sensors by BRC, at later stage, will allow for relevant data collection to be used optimizing the cranes' movements. Also below we give figure (3) from the bar area on bay 3. Similarly with the coil area, there are cranes which feed the B area with raw material. In this area there are two stages of machines: two shearlines and three bending machines respectively.

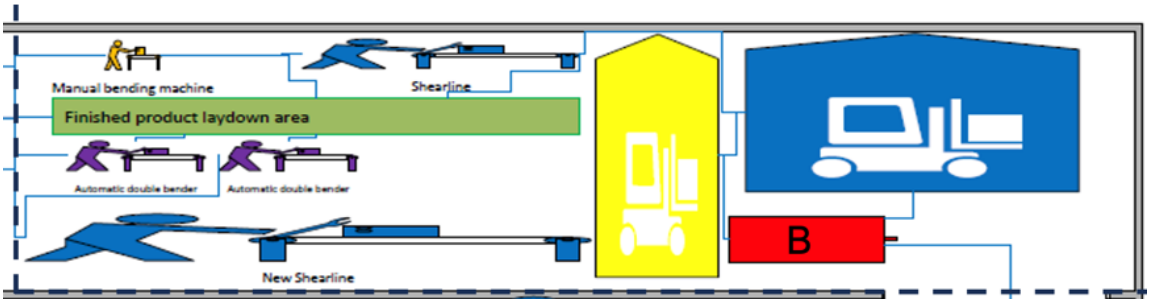


Figure 3: Bar Area at Bay 3

Once the machine is loaded the machine operator "scans" the initiation of processing towards the creation of a product. When the machine finishes the creation of the product a crane is needed to

remove the product from the machine (signaled by the bar machine operator or noticed by the crane operator). Most of products coming out of the machines are considered "finished products" (with the exception of particular shape code that needs further processing which will be ignored now). Therefore the products can be placed by the crane operator either on the "product finished laydown area" in order to wait for completion and/or the other components of the order or they can be placed directly onto the trailer C. Once a trailer has a complete order (or a series of completed orders) it is flagged by the crane operator as ready and it is moved to location D to await departure towards the customers.

### 3.1 Notation

At this section we present the basic notation that will be used in our optimization model. We note that each order has a number of different jobs that is required, namely every job in our case could be a specific product (i.e. a product with a *Shape Code* which may denote a bar with diameter $\phi = 12$ mm and 4 m length etc.). The factory receives the orders from the customers giving a unique order ID which is composed of different jobs' IDs. So using the IDs we can keep track the relationships between orders and jobs.

**Sets**

- $\mathcal{S}$ : set of stages ($s = |S|$)
- $\mathcal{M}(s)$ : set of machines in stage $s \in \mathcal{S}$, ($m(s) = |M(s)|$)
- $\mathcal{I}$ : set of jobs
- $\hat{\mathcal{B}}$ : set of forbidden (job,machine) combinations
- $\hat{\mathcal{M}}$ : set of non-existing processing paths

**Parameters**

- $T_i^d$ : due date of job $i \in \mathcal{I}$
- $T_i^r$ : release date of job $i \in \mathcal{I}$
- $T_m^s$ : setup time for machine $m \in \mathcal{M}(s)$
- $T_{mi}^p$ : processing time in machine $m \in \mathcal{M}(s)$ for job $i \in \mathcal{I}$
- $U$ : a positive big number, $U = \sum_i \max_m T_{mi}^p$
- $\lambda$ : weight coefficient between makespan-lateness

**Decision Variables**

- $y_{mi}$ : 1 iff on machine $m \in M(s)$ we schedule job $i \in I$ and 0 otherwise
- $x_{ii's}$ : 1 iff we assign job $i$ before job $i'$ at stage $s \in S$ and 0 otherwise
- $c_{is}$ : completion time of job $i \in I$ at stage $s \in S$
- $C_{max}$ : makespan, the time that is required to finish the last job
- $L_i$ : lateness of job $i \in I$, $L_i = c_{is} - T_i^d$
- $T_i$ : 1 iff job $i \in I$ is tardy and 0 otherwise

### 3.2 Assumptions

After consultation with BRC people in charge we will create a deterministic model which represents in high level Bay 3. Currently, there are no crane sensors installed to have transportation data, however our model will be ready made to be compatible when they will be available. With regards to maintenance there are some historic data in paper format, but there is no periodic planning so the maintenance is based on empirical rules. However, BRC will apply in the future a periodic maintenance plan based on the specifications of each different machine.So, at this phase we consider the maintenance as input and we incorporate this aspect by a parameter providing whether the machine is available or not. To continue with an Mixed Integer Programming (MIP) formulation it is essential to make some assumptions which are mainly due to shortage of data or some simplification so as to make the problem more manageable. The assumptions made for the development of the present MIP are as follows:

- All jobs are available at the start of time horizon
- All jobs follow the same predefined order of stages
- No preemption/ interruption is allowed
- No job can be processed by more than one machine at the same time and no machine can process more than one job at the same time (i.e. job slitting is not allowed)
- There should be no waiting time between consecutive tasks of a job
- Processing time is independent of the schedule
- Machines never breakdown and are available throughout the scheduling period unless a failure is reported in the beginning of the time time horizon
- Setup up times for each machine and for each different product that is going to be produced are known in advance and they are not included in processing time
- The machines are parallel unrelated which implies that the machines are not uniform and might have different processing and setup times for the same product
- If a product is flagged as finished, then it cannot be processed again. So reproduction is not allowed

### 3.3 Mathematical Formulation

Following the above notation and after considering the assumptions we made, we can proceed in a Mixed Integer Linear Programming (MILP) formulation for bay 3. We note that at every stage the factory can process only a specific set of jobs. There are three stages $s \in \{1, 2, 3\}$ and three different kind of jobs $i \in \{coil, cutting, bending\}$. We know in advance that the job 'coil' is processed in stage s=1, the job 'cutting' is processed in stage s=2 and finally the job 'bending' at stage s=3. To be consistent with the factory's production line we constructed the set $\hat{B}$ whose members are all the infeasible combinations of jobs and machines. With this way we avoid to process a bar which need cutting in a coil machine or a coil job in a bending machine etc.

An order may have multiple jobs however we can omit an additional order index in our MIP since with a postprocess procedure and using the job and order IDs we can get all the desired information. Furthermore the reason we cannot decompose the problem into some smaller subproblems related only with a specific kind of job is the two stage process for the bending jobs. The bending jobs need to be processed firstly in stage s=2 and then move to s=3. To sum up the flow shop problem on behalf of BRC Ltd can be formulated as follow:

$$minimize \quad \lambda C_{max} + (1-\lambda)\sum_{i\in I} L_i \tag{3.1}$$

$s.t.$

$$\sum_{m\in M(s)} Y_{mi} = 1 \qquad \forall i \in I, s \in S \tag{3.2}$$

$$C_{max} \geq c_{is} \qquad \forall i \in I, s \in S \tag{3.3}$$

$$c_{is} \geq \sum_{m\in M(s)} y_{mi}(T_i^r + T_m^s + T_{mi}^p) \qquad \forall i \in I, s \in S \tag{3.4}$$

$$c_{is} \leq c_{i,s+1} - \sum_{m\in M(s+1)} y_{mi}(T_{mi}^p + T_m^s) \qquad \forall i \in I, s < |S| \tag{3.5}$$

$$c_{i's} \geq c_{is} + T_{mi'}^p + T_m^s - U(3 - y_{mi} - y_{mi'} - x_{ii's}) \qquad \forall i, i' \in I, i < i', \forall s \in S, m \in M(s) \tag{3.6}$$

$$c_{is} \geq c_{i's} + T_{mi}^p + T_m^s - U(2 - y_{mi} - y_{mi'} + x_{ii's}) \qquad \forall i, i' \in I, i < i', \forall s \in S, m \in M(s) \tag{3.7}$$

$$y_{mi} = 0, \qquad \forall i \in I, m \in M, (i,m) \in \hat{B} \tag{3.8}$$

$$y_{mi} + y_{m'i} \leq 1 \qquad \forall i \in I, (m,m') \in \hat{M} \tag{3.9}$$

$$L_i = \max\{c_{is} - T_i^d, 0\} \qquad \forall i \in I, s \in S \tag{3.10}$$

$$L_i \leq T_i U \qquad \forall i \in I \tag{3.11}$$

$$T_i \leq L_i U \qquad \forall i \in I \tag{3.12}$$

$$\begin{aligned}\sum_{i\in I} y_{mi}(T_{mi}^p + T_m^s) \leq & \max_{i\in I}\{T_i^d - \sum_{s'\in S, s'>s} \min_{m'\in M(s'),\ (i,m')\notin \hat{B}} (T_{m'i}^p + T_{m'}^s)\} - \\ & \min_{i\in I}\{T_i^r + \sum_{s'\in S, s'<s} \min_{m'\in M(s'), (i,m')\notin \hat{B}} (T_{m'i}^p + T_{m'}^s)\}, \forall s \in S, m \in M(s)\end{aligned} \tag{3.13}$$

$$x_{ii's}, y_{mi}, T_i \in \{0,1\},\ 0 \leq \lambda \leq 1, c_{is} \geq 0 \tag{3.14}$$

Based on discussions with BRC the objective is to maximize the Throughput. However due to the shortage of data (such as transportation) we agreed that in the first place we will try to minimize the maximum completion time of a job (makespan criterion) and the total lateness of tardy jobs. To achieve that we created an objective which tries to determine a schedule that minimizes a convex sum of makespan and the total lateness of tardy jobs, see (3.1).

We define $Y_{mi} = 1$ if job $i$ is assigned on machine $m$. The set of constraints (3.2) ensures that every job is assigned to a machine at each stage, respecting the relationship between jobs and stages that mentioned before the mathematical model. Constraints (3.3) link the makespan decision variable with the completion time of the last job to finish its processing. The next four sets of constraints (3.4) - (3.7) are related to the completion time of a job. More precisely constraint (3.4) imposes that the completion time of a job $i$ in a stage $s$ should be at least the sum of release date, the processing time and machines' setup time as well. The next constraint (3.5) guarantees the precedence sequence where each job cannot start its processing at stage $s$ before it finishes at stage $s-1$. This set of constraints is active only for those jobs which need cutting and bending operations.

The next two set of constraints (3.6) - (3.7) prevent any two jobs from overlapping in a common machine. The difference of completion times between job $i$, which precedes job $i'$ should be at least the setup time plus the processing time of the first. From these two set of constraints only one set will be active and the other will be redundant. At this point a small example could be helpful for the reader. Firstly we remind that the $x_{ii's} = 1$ if job $i$ precedes job $i'$. We can observe that we do not need the machine index $m$ in the $x_{ii's}$ decision variables because the nature of the constraints and the relationship that exist between $y_{mi}$ and $x_{ii's}$, hence these restrictions make sense only when we have jobs in a common machine. Let's assume that job $i$ precedes job $i'$ on machine $m$ at stage $s$ so $y_{mi} = 1,\ x_{ii's} = 1$. If we substitute these values in the above constraints we will take:

$$(3.6): c_{i's} - T^p_{mi'} - T^s_m \geq c_{is}, \text{ so}$$
$$\text{starting time } i' \geq \text{finishing time } i \Rightarrow \text{True}$$

$$(3.7): c_{is} - T^p_{mi} - T^s_m \geq c_{i's} - U \Rightarrow \text{trivial}$$

We can see that the first constraint implies that the starting time of job $i'$ ($c_{i's} - T^p_{mi'} - T^s_m$) is at least the completion time of job $i$. However the second constraint is redundant. So the initial assumption which job $i$ precedes job $i'$ holds.

Forbidden assignments are specified in (3.8), where $\hat{B}$ is a set of forbidden (job, machine) combinations. Using this constraint we ensure that every job is going to be processed in the correct stage. Also every job specification that cannot be processed on a machine can be incorporated using this set of constraints. Similarly, constraint (3.9) prohibits any job $i$ to go from machine $m$ to $m'$, if these are part of the set of non-existing processing paths $\hat{M}$.

Furthermore constraint sets (3.10) - (3.12) are referred to the job's lateness and tardiness. In more detail, constraint (3.10) calculate the lateness of a job specifying the positive lateness as tardiness ($L_i = \max\{c_{is} - T^d_i, 0\}$). Constraint (3.11) links the tardiness with the decision variable that indicates the number of tardy jobs, namely, if lateness is greater than zero ($L_i > 0$), then the job is tardy $T_i = 1$. Moreover constraint (3.12) reacts additionally with the previous one and ensures that if we do not have lateness for a job ($L_i = 0$) then the job cannot be tardy ($T_i = 0$). Hence with constraints (3.11) - (3.12) we ensure:

$$\text{If } L_i > 0 \Rightarrow T_i = 1$$
$$\text{If } L_i = 0 \Rightarrow T_i = 0$$

$$\text{So if } L_i > 0: \begin{cases} (3.11): L_i \leq T_i U & \Rightarrow \ T_i = 1 \\ (3.12): T_i \leq L_i U & \Rightarrow \ \text{trivial} \end{cases}$$

$$\text{otherwise if } L_i = 0: \begin{cases} (3.11): 0 \leq T_i U & \Rightarrow \ \text{trivial} \\ (3.12): T_i \leq 0 & \Rightarrow \ T = 0 \end{cases}$$

Last but not least the next constraint will reduce the search space by adding some logical cutting plane. Constraint (3.13) reduces the search domain by making sure that the total processing time of the jobs on a machine will fit between 1) the maximum due date subtracted with the shortest processing and setup times of all later stages, and 2) the minimum release date plus the shortest processing times of all earlier stages. Finally constraints (3.14) ensure the integrallity constraints and the non-negativity as well.

### 3.3.1 Formulation challenges

During this project we faced a lot of challenges beyond the size of the problem. In this section we will try to mention some major and important difficulties and how we managed to address them giving examples where is necessary. One of biggest challenge was the level of information we need to keep track for being consistent with the problem. Jobs' and machines' specifications as well the sequence for a given job through the production line where also obstacles we had to overcome during this project. So giving some small examples we will try to imitate the process at Bay 3.

Firstly we have three different types of jobs, every job can be considered as a steel product in our case. Every job is related with the raw material like coil or bars, and some other specifications like if the bar job needs cutting, or cutting and then bending e.t.c. So we can construct the elements of the set of jobs like below:

- C : for coil related jobs
- B_C : for bars related jobs which need only cutting
- B_C_B : for bars related jobs which after the cutting procedure also need to be processed on the bending machines

Another aspect of the formulation was the number of machines in each stage which is different and as we have already defined in the mathematical formulation section the set of stages is $S = \{0, 1, 2\}$. We can give some extra clarifications for the stages:

- 0 : coil stage where there exist 4 parallel and unrelated coil machines
- 1 : bar stage where there are 2 parallel and unrelated cutting machines which they cut the bars
- 2 : bar stage where there are 3 parallel and unrelated bending machines which they perform the bending operations

Then the set of machines at each stage $s$ can be defined as: $M(s) = \{\{0, 1, 2, 3\}, \{4, 5\}, \{6, 7, 8\}\}$. It's very difficult to determine the indices for each variable or even for the parameters however it is more challenging to figure out the sets that every index lie on. We can give a small example to demonstrate the above statement. Let's assume a set $O$ of orders which contains all the orders' IDs that the factory receives and another set $J^o$ of jobs that compose an order like:

$$O = \{\text{id\_1}, \text{id\_2}, ..., \text{id\_n}, \}$$

$$J^o = \{\{C, B\_C\}, \{B\_C\_B\}, ..., \{C, B\_C, B\_C\_B\}\} \subset I$$

We can consider the $J^o$ structure as a set of sets and given a value for their indices we can retrieve a subset of them. For instance if $o = \text{id\_n}$ then $j \in J^{\text{id\_n}} = \{\text{C, B\_C, B\_C\_B}\}$. However our goal is to allocate in a optimal way all the jobs for all orders that the factory has. Hence our MIP model take into consideration the total number of jobs and then using orders' ID, jobs' ID and a post process procedure we present the solution as BRC wants. Using the postprocess method then we have a mathematical model with fewer indexes (because we omit the index of order) so as a consequence the number of decision variables is significantly smaller.

Another difficulty was the way we made the connection between the assignment variables $y_{mi}$ and those that are referred to precedence sequence, namely $x_{ii's}$. As we mentioned very briefly in the previous section we do not need the index of stage $s$ on the $y_{mi}$ variable because $m \in M(s)$ so we can retrieve the information regarding the stage. In the same manner we do not need the machine index $m$ on the $x_{ii's}$ due to the relationship between the assignment and sequence variables. Omitting those indexes there are fewer possible variable combinations so the number of decision variables decreases. Hence we hope the tree that is created, when B&B algorithm is applied, will be smaller.

A small example will show the reason we omitted the above indexes. Constraint set (3.6 - 3.7) specify the sequence of jobs in a common machine. The variable $x_{ii's}$ takes values, which are meaningful, when $y_{mi} = y_{mi'} = 1$ which means the job $i$ and $i'$ are processed in the same machine. If this is not the case, namely at least one of $y_{mi/i'} = 0$, then $x_{ii's}$ doesn't make any difference whether is 0 or 1 since both constraints are redundant. Let's assume $y_{mi} = 0$ then:

$$c_{i's} \geq c_{is} + T^p_{mi'} + T^s_m - U(3 - y_{mi'} - x_{ii's}) \Rightarrow \text{ redundant}$$
$$c_{is} \geq c_{i's} + T^p_{mi} + T^s_m - U(2 - y_{mi'} + x_{ii's}) \Rightarrow \text{ redundant}$$

### 3.3.2 Preliminary computational analysis

In this section, we will provide all the material related with the computational research. We implemented the prototype MIP model in Python using Pyomo library and cplex solver for solving the optimization problem. We will mention the computational challenges we faced during the implementation part and how we dealt with them. Then we will discuss the results for some instances that we made so as to demonstrate the complexity and how difficult it is to get the optimal solution even for relative small cases. Regarding the data we fed the model, due to shortage of them we fabricated the major parameters by observing some raw data and then estimating the required parameters like the processing time, the due dates e.t.c. The declaration for other parameters, like the type of each job (C,B_C,B_C_B) was generated randomly. We created an array of integers from 0..2, and we assumed 0 for coil (C) jobs, 1 for cutting jobs (B_C) and 2 for bending jobs (B_C_B). This array was constructed following the uniform distribution.

#### 3.3.2.1 Implementation challenges

The main difficulty, like in section 3.2.1 about the formulation challenges, is the level of information we have to be considered and being consistent with the jobs which belong in a customer's order. Here we will also explain how the post-process and the omission of some indexes, we discussed in the formulation challenges, also helps the implementation. We will give more details for the previous statement by giving a small comparison between the mathematical notation and the implementation part. Let's examine the decision variable $x_{ii's}$ assuming that we need to keep track the job-order relationship explicitly in our MIP model. So in the mathematical model we should define the variable by saying $\forall s \in S$, job $i \in J^o$, $o \in O$ and similarly for job $i' \in J^{\bar{o}}$, $\bar{o} \in O$ where $o$ and $\bar{o}$ are different orders. However during the implementation we should define the sets and/or ranges which the variable will be declared given explicitly the path of information for job $i$ and $i'$. So is essential to add extra indices to keep track the jobs of different orders, and in our case the $x_{ii's}$ declaration in our implementation would actually be:

$$x_{oi\bar{o}i's}$$

Now it's obvious that the order index omission will reduce the number of variables dramatically by allowing us to declare the variable as $x_{ii's}$ knowing that $i$ and $i'$ are different jobs.

Moreover another issue arises when we try to declare the variables. The length of indices is not the same and may differ according by another index. Now let's take as an example the $y_{mi}$ decision variable if we had not omitted the stage index s (something like $y_{mis}$) and it was required to keep track the job-order relationship explicitly in the model. Then from programming point of view this variable would be defined as:

$$y_{mois}$$

Hence the $m$ index is depended by the stage $s$ because at every stage we have different number of machines $m(s)$. So if $s = 0 \rightarrow m \in \{0..3\}$ but if $s = 1 \rightarrow i \in \{4, 5\}$. Similarly with $i$ index which is dependent by $o$ index sine $i \in J^o$ e.g. if order $o = \text{id\_1} \rightarrow i \in \{\text{C, C, B\_C\_B}\}$ but if $o = \text{id\_2} \rightarrow i \in \{\text{B\_C}\}$. By cutting out the order and stage indices an easy, but not very efficient, way to declare those variables is to define them according the length of machines' and jobs' set. Nevertheless following this way we will create variables which we will never make use of them since they are not exist in the real model e.g. the assignment of a coil job on a cutting machine at s = 1.

Another more efficient and elegant way to declare as many variables as we need is to take advantage of the *dynamic* variable declaration (using the VarList class for Pyomo) which allow us to define variables according on our need. Following this way we will define only those variables that are coherent between the stages and the number of machines. Again a small example will be beneficial for the reader's understanding. Keeping the example as simple as possible, we will focus on the machines' index. So for a given job $i \in I$:

if $i = \text{C} \rightarrow$ being processed at s=1 $\rightarrow m \in \{0, 1, 2, 3\}$, then the variables which actually we need are:

$$y_{0i}\ ,\ \ y_{1i}\ ,\ y_{2i},\ \ y_{3i}$$

so we can omit the decision variables that are not consistent with the relationship between stage and machines namely :

$$y_{\{4..8\}i}$$

because the specific coil job i cannot be processed on cutting or bending machines we do not actual need the above variables. Our aim is to use as fewer as possible decision variables so as the algorithm will need fewer iterations for an optimal solution.

Another challenging part of the implementation was the declaration of $\hat{B}$ set. As a reminder this set is all the infeasible combinations for the variable $y_{mi}$. Using this set we are consistent with job assignments making sure that every job and accordingly with its type is worked out on a valid machine subset. In addition whatever jobs' or machines' specification occur we can incorporate it by adding a (job,machine) combination in this set knowing that we are compatible with the case. For example

if job i cannot be produced on machine m due to machine's capabilities then the pair (i,m) will be append in the list. Then according to constraint (3.8) for every pair (job,machine) $y_{mi} = 0$.

Constraint (3.13) reduces dramatically the search domain by adding some logical cutting planes regarding the jobs' assignments for all machines m at each stage s. According to our mathematical model it is allowed to have tardy jobs. However due to constraint's structure, if we have very tight due dates then it provokes infeasibility. This is contrary to our goal so we updated the constraint by adding an additional check. The infeasibility was occurred when the due dates were too tight and then the right hand side (RHS) was negative or quite small so all the jobs' assignments $y_{mi}$ (left hand side) were fixed to 0 which it's in conflict with constraint (3.2).

To overcome this obstacle we enforced an extra check where the RHS should be at least the minimum sum of processing and setup times over all machines in each stage. So for a given machine if this is not the case then we make the constraint redundant declaring that $\sum_{i \in I} y_{mi}(T^p_{mi} + T^s_m) \leq U$ which is always true. We tried to find some logical cuts getting better LP relaxations. However when you have more than one criteria to optimize it's quite difficult to find cuts which are valid and at least we managed to keep the cut referred in (3.13) valid even when we have tardy jobs.

Last but not least, maybe the most challenging part of the implementation was the definition of the ranges and settings over the constraints needed to be applied. The term consistency and precision is something quite difficult to be followed in the mathematical notation however in the implementation is much more challenging. As we referred due to the previous challenges we always have to check the indices and then to decide the correct ranges where the constraints should be applied respecting the stages and the number of machines in each of them, the jobs and the subset of machines that can be processed e.t.c. To sum up with this section we tried to capture the challenges in the coding part for generating some numerical results.

#### 3.3.2.2 Computational results

A numerical result analysis using an exact algorithm is rare because it is difficult to find the optimum in a flow shop scheduling problem let alone for a multistage flexible one. The majority of authors after the IP formulation use a heuristic method for solving their instances. For the purposes of this project we calculate some preliminaries results using an exact method as presented in section 3.3.

We performed all tests on a machine with Intel(R) Xeon(R) E5-2650 v2 2.6 GHz, 16 GB RAM, Windows 2007, using CPLEX solver. We want to emphasize on the statistics stuff as the scale of the instances raises. Every batch of test instances consist of 10 problems randomly generated. We fixed the number of machines as the production line of Bay 3 work with, and we will investigate how the mathematical formulation reacts while we increase the number of jobs in relationship the objective goal. The percentage near the solution time is the proportion on how many problems were solved within the time limit condition which is 30 minutes in our case.

To begin with in our instances we have m = 9 machines and we increase the number of jobs by two at a time starting from 10 up to 20. Then in our results we can observe how the model behaves as a consequence of the above changes regarding the optimization criterion as well. Below we summarize all the information in a table.

| | makespan (sec) | lateness (sec) | makespan-lateness (sec) |
|---|---|---|---|
| (n=10, m=9) | 1.15 (100%) | 2.72 (100%) | 1.74 (100%) |
| (n=12, m=9) | 1.21 (100%) | 103.79 (100%) | 62.76 (90%) |
| (n=14, m=9) | 35.34 (60%) | 12.26 (60%) | 21.77 (60%) |
| (n=16, m=9) | 303.70 (80%) | 251.23 (40%) | 78.44(40%) |
| (n=18, m=9) | 243.8 (40%) | 743.87 (30%) | 54.36(20%) |
| (n=20, m=9) | 57.63 (40%) | 365.62 (30%) | - (0%) |

Table 1: Instances' solution times according to objective criterion, $C_{max} - \sum_i L_i$

We can see every time we add two extra jobs the complexity is increased and the solution time

tends to grow since the percentage of solved problems within the time limits seems to deescalate. Moreover the makespan criterion seems to be the easiest one for be optimized having also the better solution times. The lateness criterion has smaller solution percentages however it takes much more time to be optimized. If we think about maybe the lateness statistics are clear mainly due to criterion's nature. The solver does not only try to avoid tardiness but also tries to find a schedule that minimizes the lateness. So if we cannot avoid a tardy job at least we should deliver the job as soon as possible minimizing the time that exceed the due date.

During my literature review the model of J. Jungwattanakita et al [51] caught my attention. The objective function of this model was a convex sum minimization between makespan and the number of tardy jobs. At first glance it was reasonable objective combination. Making some extra modifications but still using the same input data as above we present our results.

| | makespan (sec) | tardy jobs (sec) | makespan-tardy (sec) |
|---|---|---|---|
| (n=10, m=9) | 3.54 (100%) | 0.35 (100%) | 1.53 (100%) |
| (n=12, m=9) | 2.57 (100%) | 0.85 (100%) | 171.45 (100%) |
| (n=14, m=9) | 232.96 (70%) | 1.16 (100%) | 52.06 (60%) |
| (n=16, m=9) | 435.48 (60%) | 8.54 (100%) | 323.4(50%) |
| (n=18, m=9) | 446.53 (50%) | 12.19 (100%) | 1087,48(20%) |
| (n=20, m=9) | 145.33 (40%) | 68.59 (100%) | - (0%) |

Table 2: Instances' solution times according to objective criterion, $C_{max} - \sum_i T_i$

We can see that the results are similar to the previous table. However when we have the number of tardy jobs as optimization criterion then we can have the solution in a short amount having also 100% solvability. Nevertheless after further thought we believe that the convex sum of makespan and the number of tardy jobs is not accurate. Firstly those criteria have different units of measurement namely the makespan has time units while the number of tardy jobs is an integer. Because $C_{max}$ will be greater than $T_i$ so the solution will be bias on makespan independent of the objective weight $\lambda$. To sum up we believe this convex combination is not accurate and we have some doubts whether the B&B algorithm takes into consideration the $\lambda$ coefficient or focus on $C_{max}$ due to the largest unit of measurement it has.

# 4 Summary and conclusion

In this study, a multistage flexible flow shop scheduling problem with parallel and unrelated machines was investigated, motivated by a real-world planning problem. The system includes three processing stages, and each of them can perform a specific job only. One or more jobs compose an order and thus precedence relationships between jobs of different stages regulate the predecessor process of a job to be finished, as well as the setup time, if any, to pass before the process can continue at the next stage.

The formulation and implementation challenges we faced during our effort to imitate the real production line were reported. Furthermore, we outlined ideas for tackling a few issues that arose during this project. Ultimately, we developed an MIP model including all the desired level of information that was available.

The preliminary computational study presented, demonstrates vividly the difficulty of the problem. Still, the beta version of our exact algorithm that was developed in python. Although the implementation part is in a beta version, we managed to derive results for our instances.

## 4.1 Future research

While developing our model we came up with some ideas that could make it more accurate in the cost of increasing its complexity. We made an assumption that the time a job needs to alter onto a different machine or different location is negligible. This is something that does not happen in practise and in order to incorporate this aspect we can define a parameter $l^c_{ikl}$ as the maximum lag time is needed to

transfer job $i$ from location $k$ to location $l$ using crane $c$. This parameter can be used additionally in the constraints set (3.5 - 3.7) determining that beyond the operating and setup time, an additional lag time should be taken into account. Adding the above parameter in the model will be compatible for feeding with transportation data.

Furthermore at this phase we can also suggest to incorporate machine availability constraints which provide whether the machine is available or need to be in maintenance mode. With regards to maintenance, there are some historic data in paper format. Until BRC applies a periodic maintenance plan based on the manual of each different machine we will consider the maintenance as input and will incorporate this aspect by a parameter providing whether the machine is available or not. Future research may address, for example, the possibility to include machine breakdowns as an extension of our approach to deal with a dynamic environment. Depending on the nature of such breakdowns, this could be accomplished by scheduling preventive maintenance intervals. To incorporate the latter statement we need extra binary variables which can be translated into higher complexity.

Future research may address, for example, the possibility to include machine breakdowns as an extension of our approach to deal with a dynamic environment. Depending on the nature of such breakdowns, this could be accomplished by scheduling preventive maintenance intervals.

A way to get solutions for bigger instances could be the major field for further study. We should evolve our implementation to a stable version. Something else that is crucial for combinatorial optimization problems is finding out better formulations which give tighter feasible areas and consequently better linear programming (LP) relaxations. Hopefully by having a bigger and more difficult formulation, by adding extra or more advanced constraints, we will be able to deteriorate the number of possible optimal solutions. We are currently working to produce specific valid cuts for the BRC case as well. Moreover other tricks with the number of indices that a variable can have can significantly restrict the variable combinations so hopefully less subproblems must be solved.

Another popular method to deescalate the computational complexity is a decomposition approach as Harjunkoski and Grossman [65] used. That's because scheduling problems can be decomposed into an Assignments and a Sequence subproblem. The decomposition technique is an efficient way of combining the complementary strengths of MILP and Constraint programming(CP). A hybrid MILP/CP model combines MILP to model the assignment part and CP for modeling the sequencing part. The subproblems are solved sequentially since the MILP produces job assignments and CP tries to find feasible sequences for the given assignments. If no feasible solution is found, then CP adds integer cuts to the first MILP to generate new assignments. CP performs very well for feasibility problems since its cutting planes produce tighter LPs. Our aim is to demonstrate a hybrid MILP/CP model and examine how we can improve our solution times.

To the best of our knowledge the most relevant previous work appears in [**?**]. The authors proposed an elegant methodology for solving large flow shop scheduling instances. The authors proposed a tree-based priority rule in terms of a well-performing decision tree (DT) for dispatching jobs. The proposed DT relies on high quality solutions, obtained using a constraint programming (CP) formulation. Novel aspects include a unified representation of job sequencing and machine assignment decisions, as well as the generation of random forests (RF) to face overfitting behaviour.

Finally a machine learning approach by Benda et al (2019) [66] which uses decision trees and random forest have been trained by a MILP or CP model has caught our interest. The authors proposed a tree-based priority rule in terms of a well-performing decision tree (DT) for dispatching jobs. The proposed DT relies on high quality solutions, obtained using a constraint programming (CP) formulation. Novel aspects include a unified representation of job sequencing and machine assignment decisions, as well as the generation of random forests (RF) to face overfitting behaviour. The tree's training is becoming offline while online you can have job schedules very quickly. Using this methodology is possible to solve large instances. In conclusion we'd say that a real world problem is very tough to be solved to optimality therefore we can examine to construct a heuristic or other hybrid methodologies which will give us a production plan in a reasonable time.

# References


[1] *BBC News (2016).*[Online] By Tim Bowler [Retrieved on July 10, 2020] https://www.bbc.co.uk/newsround/35355882.

[2] *BBC (2016.*[Online] [Retrieved on July 10, 2020] https://www.bbc.co.uk/newsround/35355882

[3] *Steel production in the UK -Statistics & Facts.*[Online] By Melania Scerra *statista (2020)* https://www.statista.com/topics/4939/steel-production-in-the-uk/

[4] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz (2007) *Handbook on Scheduling, From Theory to Applications.* International Handbook on Information System: Springer

[5] C.C Hsu, K.C. Huang & F.J. Wang (2011) *Online Scheduling of Workflow Applications in Grid Environments*, Future Generation Computer System 27, pp 860-870

[6] C. Steiger, H. Walder, M. Platzner & L. Thiele (2003) *Online scheduling and placement of real-time tasks to partially reconfigurable devices* 24th ieee real-time systems symposium, pp.224–225

[7] Z. Li & M. Ierapetritou (2007) *Process scheduling under uncertainty: Review and challenges* Computers and Chemical Engineering 32, pp.715-727

[8] J. Carlier and E. Pinson (1989) *An algorithm for solving the job-shop problem*, Management Science, pp. 164-176

[9] P.Brucker, J. Bernd and S. Bernd (1992) *A branch and bound algorithm for the job-shop scheduling problem*, Discrete Applied Mathematics, p.107-127

[10] J.K. Lenstra, P.Brucker & A.H.G Rinnooy Kan (1977) *Complexity of machine scheduling problems in: Annals of Discrete Mathematics 1* (North Holland, Amsterdam) p. 343-362

[11] J.F. Muth and G.L Thompson (1963) *Industrial Scheduling*, Prentice-Hall. Englewood Cliffs

[12] M.R. Garey, D.S. Johnson & R. Sethi (1978) *"Strong" NP-Completeness Results: Motivation, Examples and Implications*, Journal of the ACM 25, no. 3, pp. 499-508

[13] K. Ploydanai and A. Mungwattana (2010), *Alogorithm for Solving Job Shop Scheduling Problem Based on machine availability constraint*, International Journal on Computer Science and Engineering

[14] G. McMahon and M. Florian (1975), *ON scheduling with ready times and due dates to minimize maximum lateness*, Oper. Res, p.475-482

[15] J. Adams et al (1988), *The shifting bottleneck procedure for job-shop scheduling*, Management Science, p.391-401

[16] Gupta JND (1988). *Two-stage hybrid flow-shop scheduling problem*, Journal Opl Res Soc 39: p.359-364

[17] S. Mahale (2017), PhD Thesis, *Developing a real time online scheduling system for a manufacturing service company: Achieving visibility*, Lamar University pp.22-24

[18] E.Taillard (1993), *Benchmarks for basic scheduling problems*, European Journal of Operational Research Volume 64, p. 278-285

[19] G. Vairaktarakis & H. Emmons (2013), *Flow Shop Scheduling Theoretical Results, Algorithms and Applications*, Springer, p.98-115

[20] S.M. Johnson (1954), *Optimal two and three stage production schedules with setup times included*, Naval Research Logistics, p.61-68

[21] C.A. Floudas and X. Lin (2004), *Continuous-time versus discrete-time approaches for scheduling of chemical processed: a review*, Computers and Chemical Engineering, p.2116-2120

[22] Sahinidis, N. V. & Grossmann I.E (1991), *Reformulation of multiperiod MILP models for planning and scheduling of chemical process*, Computers and Chemical Engineering, p.255-272

[23] Shah et al (1993), *A general algorithm for short-term scheduling of batch operations, part II,*Computers and Chemical Engineering, p.229-244

[24] Yee, Shah et all (1998), *Improving the efficiency of discrete time scheduling formulation*, Computers and Chemical Engineering

[25] Denopoulos I.T & Shah N. (1995), *Optimal short term scheduling of maintenance and production for multipurpose plants*, Industrial and Engineering Chemistry Research, p. 192-201

[26] M. H. Basset, J. F. Pekny & G. V. Reklaitis (1996), *Decomposition techniques for the solution of large scale scheduling problems*, Aiche Journal 42, p. 3373-3387

[27] A. Elkamel et al (1997), *A decomposition heuristic for scheduling the general batch chemical plant*, Engineering Optimization, p. 299-330

[28] P. Brucker (2006), *Scheduling Algorithms*, 5th Edition, Springer

[29] R.L. Graham, E.L. Lawler, J.K. Lenstra and G. Rinnooy Kan (1979), *Optimization and approximation in deterministic sequencing and scheduling theory: a survey*, Ann Discrete Math 5, p.287-326

[30] M.L. Pinedo (2015), *Scheduling Theory, Algorithm, and Systems*, Fifth Edition, Springer USA

[31] S. Reza Hejazi and S. Saghafian (2005), *Flowshop-scheduling problems with makespan criterion: a review*, International Journal of Production Research 43, no 14, p. 2895-2929

[32] C. Koulamas (1998), *A new construcrive heuristic for the flowshop scheduling problem*, European Journal of Operational Research 105, p, 66-71

[33] K.Baker (1974), *Introduction to Sequencing and Scheduling*, J.Wiley New York

[34] J.K. Lenstra (1977), *Sequencing by Enumerative Methods*, Mathematical Centre

[35] G. Rinnooy Kan (1976), *Machine Scheduling Problems: Classification, Comlexity and Computations*, Martinus Nijhoff, The Hague

[36] R. Bellman (1957), *Dynamic Programming*, Princeton University Press, NJ

[37] JI Ung Sum (2006), *Dynamic Programming Approach to a Two-Machine Flow Shop Sequencing with Two-Step-Prior-Job Dependent Setup Time*

[38] S. Dreyfus, A. Law (1979), *The Art and Theory of Dynamic Programming*, Academic Press, New York

[39] Potts CN (1980), *An adaptive branching rule for the permutation flow-shop problem*, European Journal of Operational Research, p.19-25

[40] M. Garey, D. Johnson (1979), *Computers and Intractability: A guide to the theory of NP-Completeness*, W.H. Freeman, San Francisco

[41] A. H. G. Rinnooy Kan (1987), *Probabilistic analysis of approximation algorithms*, Annual Discrete Maths 31, p. 365-384

[42] E. Silver, R. Vidal, D. de Werra (1980), *A tutorial on heuristic methods*, European Journal Operational Research 5, p. 153-162

[43] S.A. Brah and L.L. Loo (1999), *Heuristics for scheduling in a flow shop with multiple processors*, European Journal of Operational Research, p.1-13

[44] S. Panwalker and W. Iskander (1977), *A survey of scheduling rules*, Operational Research 25, p.45-61

[45] T. Yamada and R. Nakano (1997), *Job Scheduling*, IEE control engineering series 55, p.134-160

[46] H. Badri (2019), *A parallel randomized approximation algorithm for single machine scheduling with applications to flow shop scheduling*, MSc thesis, Wayne State University, Detroit Michigan

[47] E. Balas, J.K. Lenstra and A. Vazacopoulos (1995), *The one machine problem with delayed precedence constraints and its use in job shop scheduling*, Management Science 41, p. 94-109

[48] I.M. Ovacik and R. Uzsoy (1992), *A shifting bottleneck algorithm for scheduling semiconductor testing operations*, Journal of Electronics Manufacturing 2, p. 119-134

[49] I.M. Ovacik and R. Uzsoy (2012), *Decomposition methods for complex factory scheduling problems*, Spinger Science & Business Media

[50] H. Wengi and Y. Aihua (2004), *An improved shifting bottleneck procedure for the job shop scheduling problem*, Computers and Operations Research 31, no 12, p.2093-2110

[51] Jitti Jungwattanakita, Manop Reodechaa, Paveena Chaovalitwongsea, Frank Wernerb (2007), *A comparison of scheduling algorithms for flexible flow shop problems with unrelated parallel machines, setup times,and dual criteria*, Computers and Operations Research 36, no 12, p.358-378

[52] Lars Monch et al (2004), *Genetic algorithm-based subproblems solutionprocedures for a modified shifting bottleneck heuristic for complex job shops*, European Journal of Operational Research 177, p. 2100-2118

[53] H. Holtsclaw and R. Uzsoy (1996), *MAchine criticaly measures and subproblem solution procedures is shifting bottleneck methods: a computational study*, Journal of Operational Research 47, p.666-667

[54] E. Demirkol, S. Mehta and R. Uzsoy (1997), *A computational study of shifting bottleneck procedures for shop scheduling problems*, Journal of Heuristics 3, p. 111-137

[55] V.Cerny (1985), *Thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm*, J. Oprimization Theory and Applications 45, p. 41-51

[56] S. Kirpatrick, C.D. Gelatt, M.P. Vecchi (1983), *Optimization by simulated annealing*, Science 220, p. 671-680

[57] M. Metropolis, A. Rosenbluth, M.Rosenbluth, A. Teller, E. Teller (1953), *Equation of state calculations by fast computing machines*, J. Chemical Physics 21, p. 1087-1092

[58] F.Glover, M.Laguna (1997), *Tabu Search*, Kluwer Academic Oubl, Boston

[59] F.Glover (1977), *Heuristic for integer programming using surrogate constraints*, Decision Sciences, p.156-160

[60] F.Glover (1986), *Future paths for integer programming and links to artificial intelligence*, Computer and Oper. Res., p.533-549

[61] F.Gloverna (1989), *Tabu Search- Part I*

[62] F.Glover (1991), *Tabu Search and embedded search neighborhoods for the traveling salesman problem*, Working paper, University of Colorado, Boulder

[63] D. Eroglu and H. Ozmutlu (2014), *Genetic algorithm with local search for the unrelated parallel machine scheduling problem with sequence-dependent set-up times*, International Journal of Production Research

[64] *Integer and Combinatorial Optimization*, Course Notes, University of Edinburgh, academic year 2019-20

[65] Iiro Harjunkoski and Ignacio E. Grossmann (2002), *Decomposition Techniques for Multistage Scheduling Problems Using Mixed Integer and Constraint Programming Methods*, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh

[66] Frank Benda, Roland Braune, Karl Doerner, Richard Hartl (2019), *A machine learning approach for flow shop scheduling problems with alternative resources, sequence-dependent setup times, and blocking*, OR Spectrum 41, p.871-893, Springer