A Comparison Between Metaheuristics in Production Scheduling Optimization

Mpourdakos, I.

Athens University of Economics and Business

July, 2025

Supervisor: Mourtos, I.

Department of Management Science and Technology

Contents

Abstract

Large manufacturing companies that are required to produce in mass, often use a distributed production plant with multiple consequent modules for their products. In the modern era, a production manager of these plants has to depend on a scheduling system in order to create the production schedules of the next days, based on the specific configurations of their respective plant. These systems incorporate various *scheduling optimization* algorithms based on optimization problems like the **Hybrid Flexible Flow-shop**. This research provides an empirical analysis, evaluation and comparison between a *Genetic Algorithm* and an *Adaptive Large Neighborhood Search* on this optimization problem. Various insights are generated from the respective experiments, like the behavior and performance of multiple *crossover operators* on different plant environments, and the value of parameter tuning tailored to each production configuration.
Keywords: *Scheduling Optimization, Metaheuristics, Genetic Algorithms, Adaptive Large Neighborhood Search*

I. INTRODUCTION

As the demand for complex manufactured products increases over the years, and as production technologies such as the Internet of Things, Artificial Intelligence, and Communications [1] continue to evolve at a rapid rate, large manufacturing companies, in an effort to catch up with new developments, will need to evolve in terms of the way they schedule their manufacturing operations [2] [3]. When using the word *product*, we refer to a result of many - *sequential* - internal operations of a manufacturer, which in turn require multiple resources (like materials, workers, machines, a production plant, etc.), and a product can be considered *complex* if its production procedure requires multiple restricted resources and is coupled by a large number of constraints. These type of production constraints can in turn come from the specifications of the product (e.g. different products might require a different sequence of operations, in the same manufacturing plant), or from the manufacturing plant in place (e.g. a manufacturer might have multiple machines that can be run in parallel, and a choice has to be made for the assignment of tasks to machines, ad-hoc) [3].

Because manufacturing has a crucial part in assisting the general development of a country's national economy, manufacturing companies will need to face the issue of minimization of the total manufacturing costs and increase the efficiency (or output) of their manufacturing plants [4]. As cost reduction is often tied to the level of detail the company is willing to invest in regards of their scheduling capabilities [5], *scheduling optimization* has become a relevant topic for manufacturers [4] [6].

I.1. Setting the Scene

Scheduling optimization is the process of finding the optimal manufacturing schedule, with respect to an objective (e.g. cost of production), given the constraints of the product and manufacturing plant in place [7]. Essentially, the goal of a manufacturer is not just to create a good production schedule once, but instead to have a mechanism, or more accurately a system, to do the job for them [7]. Such a system will be configured based on the requirements of the plant and can be later given some input (e.g., a set of customer orders), to produce a production schedule for the orders that i) satisfies all the predefined constraints of the

manufacturer (feasibility) and ii) optimizes the predefined objective (optimality) [3]. For example, the predefined requirements can be that the plant has a series of machines, m, and to manufacture the orders of the customers, each 'order' needs to be processed by all of the machines in the same sequence (first by m_1, then by m_2 and so on). A proposed schedule will then need to i) schedule every customer's order to pass through every machine in the correct sequence (if it did not, the schedule would be infeasible) and ii) have a satisfactory value on the objective score (e.g. have a low cost). For the development of such a system, scheduling algorithms capable of undertaking such problems which generate feasible and efficient manufacturing schedules will need to be utilized [8]. These algorithms have been researched in the past [9].

That last example is a good mental bridge for understanding what is the purpose of this paper. To be more precise, the aforementioned example with the sequential machines, and a set of '*jobs*' that need to be processed by them, is called a *Permutation Flow-shop* (PFSP) in the literature, with its earliest research on the subject published by Johnson in 1954 [10]. It is one of many scheduling problem variants [11], and it will be the base we will use to present the actual problem that is being tackled by this paper. Although the PFSP will not be considered on this paper, it would be in the convenience of the reader to define it in a little more detail, in order for us to set the scene before moving to the actual problem of the paper. The abstracted concept of a permutation flow-shop describes a setting where a set of $|N|$ jobs are issued for processing by a series of - sequential - machines, and each $n \in N$ has to pass through every machine, **in the same order**. In a permutation flow-shop, there exists a predefined $n - m$ mapping for the processing time each job $n \in N$ needs, on each machine $m \in M$. Of course, that results in a $|N| \times |M|$ matrix, because the same machine may need different amounts of time to process different jobs. Then, the actual problem is to find a starting sequence of the jobs (the sequence they will be processed in) that in turn minimizes the total time it took for all the jobs to finish. The output of a PFSP solver is what we will call a *schedule* (the processing sequence of the jobs) and the total production time required by the schedule is called *makespan* [12].

I.2. Scheduling Problems

Before turning to the scheduling optimization problem considered in this paper, let us take a step back and define the different types of *production scheduling* problems. The actual difference between production scheduling categories rests in the processing method used by the production plant [13] [14] as cited by [4]. If the production plant in question only has a single machine that is used to process jobs, we are looking at a *single machine* problem [4]. In contrast, if we have a manufacturing system in place where each job has to pass through the same order of machines before being 'completed', and all jobs have the same number of operations, we are witnesses of a flow-shop problem [15], one of the most researched scheduling problems [12]. By including the constraint that all jobs have to have the same processing order in a flow-shop system, we get the PFSP [16]. Finally, if we do not require the same processing sequence and number of operations for all the jobs, we have a job-shop problem [9]. Building on these constraints, if we have multiple stages in a sequence, where each stage contains multiple machines, and each job has to be assigned to a maximum one machine per stage, we have a ***Hybrid Flexible Flow-shop*** [11]. A generalization of the flow-shop problem that has been subjected to intensive research in the last few years [6]. In a Hybrid Flexible Flow-shop [11], a set of jobs N is issued to be processed by a sequence of stages S, where for each $n \in N$, there exists a set $S_n \subseteq S$, that contains the stages that job n has to pass through. That means that not all jobs have to pass through every stage. For each $s \in S$, there exists a set M_s of all available machines in stage s. The resulting schedule of an HFFS solver will contain a job-machine assignment at every eligible stage of every job while also allowing different jobs to be assigned to the same machine within a stage, which results in a job sequence. When multiple jobs are assigned to the same machine within a stage, we also need to solve the problem regarding the processing sequence, in addition to the assignments, as a machine cannot process more than one job at a given time, making the problem more complex [17]. Although many optimization objectives such as production cost, time, and quality appear in the literature [18] [6], the most common objective regarding the aforementioned problems, is the minimization of the makespan [12].

In summary, what this paper aims to achieve is providing an empirical numerical evalua-

tion of some scheduling optimization methods that will be used to solve the HFFS problem, with the addition of some case-specific constraints, under the makespan objective. The remainder of the paper is organized as follows: In **Section II** a literature review of the research space of optimization methods is conducted on the problem. **Section III** covers the detailed description of the problem. In **Section IV**, an introduction to the implemented methods is given while **Section V** describes the experimental procedure and summarizes the results of the comparative evaluation of the algorithms. Finally, **Section VI** concludes the findings of this analysis and **Section VII** highlights the direction for future research.

II. Literature Review

When faced with a difficult scheduling problem, the initial approach that should be considered when building an optimization program, is the use of mathematical models [19]. That is one of the reasons why **Mixed Integer Linear Programming (MILP)** solutions that are able to solve the problem to optimality have been fairly discussed in the literature [16]. Some approaches based on exact methodologies for flow-shop scheduling problems are applied by [20], [11], [21], [7] and [22]. [20] defines an MILP model for the hybrid flexible flow-shop problem with blocking constraints, which minimizes makespan and the total cost of the machines. Their problem is formulated based on probabilistic values for the parameters (e.g. for the processing and setup times), instead of fixed ones, in order to mimic the uncertainty of a real production environment. [11] proposes three sequence-based MILPs and a position-based MILP for the hybrid flexible flow-shop with makespan minimization. As all four models struggle on large problems, [11] also implemented a metaheuristic method. [21] acknowledged the effect of the human factor in production schedules and, in turn, developed an MILP that considers worker constraints on the hybrid flow-shop problem. In their work, they consider a fixed number of workers per stage that operate the machines, and they optimize the (multi-)objective of makespan and total job tardiness. [7] requires jobs and machines to use renewable resources, and also solves the problem of resource allocation through a mathematical model, in a flexible flow-shop manufacturing plant, to minimize makespan. Like [11], [7] uses a metaheuristic schema combined with a local search, to tackle large problems. Finally, [22] introduces a **Constraint Programming (CP)** model that

minimizes the makespan in flexible job shop systems with multiple resource constraints. The approach of CP methods has been proven as an effective solution to flexible job shops [23] and general scheduling problems with resource constraints [3].

After exact methods, several methodologies are implemented in the literature based on metaheuristic approaches that are capable of solving even large problem instances, fast [24]. Some of the most popular ones for flow-shop systems are **Genetic Algorithms**, **Simulated Annealing**, **Particle Swarm Optimization** and **Ant Colony Optimization** [18] [25]. **Tabu Search** variants have also presented good performance on job-shop systems with multi-purpose machines [26]. Some empirical evidence of the efficient performance of these specific methods are presented by [15], [27], [28] and [29]. [15] studies the efficiency of simple variants of the search-based local search method, with the objective of minimizing the sum of completion times of all jobs, also known as the **total flow time minimization**, an optimization objective well studied in the flow-shop systems literature, because of its ability to result in schedules with a more stable utilization of machine resources [15]. The study presents four algorithms that are used to solve the flow-shop problem. First, a **Iterated Local Search (ILS)** is introduced that escapes local optima by changing the solution drastically (also called a perturbation of the solution) after an execution of a complete local search, in an iterative manner, until a termination criterion is activated. The **Iterated Greedy (IG)** method follows a similar logic, but instead of iteratively producing a perturbation of local optimum solutions, it performs a series of destructions and greedy recreations of parts of the solution. Finally, the authors also implement two variants of the aforementioned techniques, that use the respective method to produce a population of solutions, instead of just single schedules. [27] uses an **Ant Colony Optimization (ACO)** approach to solve the same problem, under the optimization (mutli-)objective of makespan, total flow time and total machine idle time. Their chosen method is based on the fact this technique has been used constantly in the literature to solve hard combinatorial optimization problems, as well as the fact that algorithms that adopt the mechanisms of ACO, have been shown to to produce good results on flow-shop scheduling [30]. The way this method operates, is by imitating the nature of ants and their methods of always finding the shortest path when in search of food [30], to create good flow-shop schedules. Coupled with ACO, they also use a local

search to intensify over the search space of good produced solutions. Finally, as evident from [28], metaheuristic methods can be even more powerful when coupled all together with other methods, providing hybrid approaches. More specifically, [28] implements three stochastic heuristics that incorporate the popular **NEH** method of [31] with a **Simulated Annealing (SA)** search technique, to solve the PFSP, by optimizing the objective of makespan. An interesting part of their SA implementation, is the use of both a *swap*, as well as a *shift* neighborhood, instead of just fixating on a single search space. [29] further reinforces the same evidence with their implementation of a **Hybrid Combinatorial Particle Swarm Optimization (HCPSO)** method in a flow-shop with blocking constraints and with the objective of makespan minimization. The PSO method is in the same class of metaheuristics as the ACO method, as they both use biological phenomena observed in nature to explore the search space of solutions [29]. Unlike the ACO algorithm, the PSO method tries to mimic the mechanisms of a flock of birds, and the way it travels while it tries to locate food [32]. What the authors actually present with their research, is a variant of the PSO that changes some of its basic properties as well as adding some new ones, coupled with an iterated local search that is invoked randomly on each solution of the swarm after every iteration. Their proposed implementation outperforms both CPSO and ILS when each algorithm is executed independently.

In parallel, [24] and [25] prove that Genetic Algorithms are efficient implementations for the job-shop systems. One of the many reasons researchers tend to focus on Genetic Algorithms in these type of problems, is that these methods do not have heavy mathematical requirements for their implementation, and instead can be changed quite easily to adopt to new problem specifications [4]. Similarly to Constraint Programming models, an evolutionary way of thinking has proven to produce good results on the flexible jobshop scheduling problem [33]. Other works on Evolutionary and Genetic Algorithms are held by [18], [4], [34], [35], [17] and [8]. [18] considers a (multi-)objective problem based on the minimization of the makespan, in combination of minimizing the total energy usage of the respective machines that process the jobs, in a flexible flow-shop environment. In their research, they enforce a chromosome encoding schema that models both the job assignments on each respective eligible stage, as well as the sequences of jobs that are assigned on common machines, using a

starting $|jobs| \times |stages|$ matrix, that is later transformed to a $1 \times (|jobs| \times |stages|)$ vector. For the population initialization phase, they use a random chromosome set, while a **Roulette-wheel** criterion is used for the chromosome selection procedure. To generate new off-springs for the reproduction phase, a two-point crossover is performed based on a probability, and a mutation operator is invoked, again based on a probability, in order to introduce new information into the next generation. Finally, in order to escape local-optima, [18] executes a simulated annealing search in the best schedule found by the GA. [4] implements a two-layer encoding schema for their GA approach in a hybrid flow-shop production plant. The first layer is used to model the job process sequences, while the second layer models the machine assignments for each job. By combining the two layers, a final production schedule is formed. In their proposed solution, the crossover operator used performs a single-point cut, while the rest of their implementation matches the one of [18]. [34] uses a GA in a hybrid flexible flow-shop plant with sequence-dependent setup times by generating a random initial population and also including a chromosome in the population that is constructed by an adaptation of the **NEH** [31] heuristic, called NEH Hybrid [36]. Unlike [18] and [4], the way the chromosome encoding scheme is defined by [34], is with a single permutation of jobs (where each chromosome defines a sequence of jobs), that represents the order in which the jobs will be processed, in the first stage. Instead of using an n-point crossover, the paper proposes four new operators for the reproduction procedure, which in turn give better results in makespan minimization when compared with other operators of the literature. In the final algorithm, when the best two proposed operators are each selected with a 50% chance, greater results can be achieved than when each operator is used independently. [35] optimizes the (multi-)objective of makespan and total weighted tardiness minimization in a group scheduling hybrid flexible flow-shop problem with sequence-dependent setup times. In their GA solver, they implement an encoding scheme similar to [18], but in order to satisfy the two sides of their objective, they first use a population to optimize the combination of the objective functions and later use the produced solutions to create a new and different population. [17], in a hybrid flow-shop problem, instead of relying on crossover operators that use multiple chromosomes to create new off-springs, they require the population to undergo a *self-evolution* phase to intensify over good solution areas in the search space. In this phase, a

solution that exists in the population evolves by performing three moves in the *insertion*, *swap* and *pairwise exchange* neighborhood. To choose solutions to evolve, their implementation employs a *tournament*, as well as a *elitist* selection procedure, and the targeted objective is the minimization of makespan. The encoding schema used is permutation-based, while they also define two decoding strategies. Before the termination of each self-evolution iteration, a tabu search is executed. [8] promotes the encoding used by [34] to solve the same problem as [17] and uses the *first-come-first-served (FCFS)* rule to define the processing order of the jobs for the rest of the stages. For the evolution of the population, [8] proposes a new crossover operator that produces off-springs that inherit a lot of characteristics from the parent solutions, and uses it in combination with five other classical operators. Finally, [24] and [25] encounter a job-shop and a flexible job-shop system, respectively, where they both use a genetic approach, and the latter also incorporates a search procedure.

Although Genetic Algorithms, as evident from the above literature, have great popularity in the field, they can also have negatives. Even the evolutionary approach they use can itself converge early and result in local optima spaces [18] [24]. To counter this early convergence, methods capable of drastically changing a given solution are needed, in order to help execution escape a search space that shows no promising results [37]. For this reason, research has been done on **Large Neighborhood Search (LNS)** and **Adaptive Large Neighborhood Search (ALNS)** methods [22]. These algorithms have been shown to be effective solvers for shop scheduling problems [38] but are also generally good solutions for problems that have complex constraints that appeal to real-world scenarios [39]. Research on these methodologies has been done by [33], [40], [41], [42], as well as [43]. [33] used an Integrated ALNS approach to solve the flexible job shop problem under the makespan objective function. Their proposed integrated algorithm removes a group of job operations on machines while keeping the others fixed (producing a partial schedule), and later recreates the complete schedule. [40] randomly uses a greedy rule and a roulette wheel to choose destruction and recreation operators, in a distributed heterogeneous hybrid flow-shop scheduling problem with a mixed-model assembly line. [41], in combination with an ALNS, uses an SA decision rule to accept or reject the new solutions, in a sustainable distributed permutation flow-shop. In their implementation, they use six destroy operators in addition to a local search after each

move, providing low optimality gaps on the makespan objective, when compared with exact solutions. [42] uses both swap and removal types of destruction operators in their ALNS implementation, using a roulette wheel. In their research, they employ their method to solve large instances of a Mixed Blocking Permutation Flow-shop, and they have competitive results in total flow time minimization. Finally, [43], in a parallel machine problem, invoke a hybrid method that couples a Differential Evolution algorithm with an ALNS, to minimize the energy consumption of the plant. In their respective work, they propose five different destruction methods and three reconstruction procedures.

Table I: Literature Review (2003-2025)

Reference	*objective*	*problem*	*Solution*	*Method*
[20]	C_{max}, TFT	HFFS	Exact	MILP
[11]	C_{max}	HFS	Exact, Metaheuristic	MILP, PSO
[21]	C_{max} and TJT	HFS	Exact	MILP
[7]	C_{max}	FFS	Exact, Metaheuristic	MILP, PSO
[22]	C_{max}	FJS	Exact	CP
[15]	C_{max}, TFT	FS	Metaheuristic	ILS, IG
[27]	C_{max} and TFT and $TMIT$	FS	Metaheuristic	ACO, LS
[28]	C_{max}	PFS	Hybrid Metaheuristic	SA, NEH
[29]	C_{max}	FS	Hybrid Metaheuristic	PSO, ILS
[26]	C_{max}	JS	Metaheuristic	TS
[18]	C_{max} and TE	FFS	Metaheuristic	GA, SA
[4]	C_{max}	HFS	Metaheuristic	GA
[34]	C_{max}	HFFS	Metaheuristic	GA
[35]	C_{max} and TWT	HFFS	Metaheuristic	GA
[17]	C_{max}	HFS	Metaheuristic	GA
[8]	C_{max}	HFS	Metaheuristic	GA
[24]	C_{max}	JS	Metaheuristic	GA
[25]	C_{max} and TMW and MMW	FJS	Metaheuristic	GA, TS
[33]	C_{max}	FFS	Metaheuristic	ALNS
[40]	C_{max}	HFS	Metaheuristic	ALNS
[41]	C_{max} and TWT	DPFS	Metaheuristic	ALNS, LS
[42]	TFT	PFS	Metaheuristic	ALNS
[43]	TE	PM	Hybrid Metaheuristic	DE, ALNS
Current	C_{max}	HFFS	Metaheuristic	GA, ALNS

MILP: Mixed-Integer Linear Programming, **CP**: Constraint Programming, **GA**: Genetic Algorithm, **SA**: Simulated Annealing, **TS**: Tabu Search, **LS**: Local Search, **ILS**: Iterated Local Search, **IG**: Iterated Greedy, **ACO**: Ant Colony Optimization, **PSO**: Particle Swarm Optimization, **NEH**: Nawaz-Enscore-Ham Algorithm, **ALNS**: Adaptive Large Neighborhood Search, **DE**: Differential Evolution **HFS**: Hybrid Flow-shop, **FFS**: Flexible Flow-shop, **FJS**: Flexible Job-shop, **FS**: Flow-shop, **JS**: Job-shop, **DPFS**: Distributed Permutation Flow-shop, **PM**: Parallel Machines, **TFT**: Total Flow Time, **TJT**: Total Job Tardiness, **TMIT**: Total Machine Idle Time, **TE**: Total Energy Usage, **TWT**: Total Weighted Tardiness, **TMW**: Total Machine workload, **MMW**: Maximum machine workload.

III. Hybrid Flexible Flow-shop

III.1. General Problem Definition

As previously stated, the scheduling optimization problem that is of issue in this paper is the Hybrid Flexible Flow-shop (HFFS). The detailed constraints and variables of a general HFFS manufacturing plant, as presented by [11], are as follows. First, sets S and J are defined that contain the stages of the plant $(1, \ldots, |S|)$ and the jobs that the plant will process $(1, \ldots, |J|)$. The set $S_j \subseteq S$ is the stages that job $j \in J$ is required to use for processing before being completed, with the corresponding set $J_s \subseteq J$ being the jobs that require stage $s \in S$ for processing. Finally, the set M_s represents the machines of stage $s \in S$ $(1, \ldots, |M_s|)$. Due to the multi-machine nature of the problem, $\exists s \in S : |M_s| > 1$, which means that at least one stage has more than 1 machine. Although we use sets to define the properties of the problem, it should be made clear that all jobs have the same, predefined, stage sequence and we do not have cycles in the processing, meaning that when a job gets processed by a machine of a stage, it can not get processed by the same stage again.

III.2. Unique Properties

Although these properties explain the general abstraction of an HFFS problem, and have been thoroughly studied in the past [6], they do not fully model the unique problem of this paper, and more constraints are needed. The HFFS variant considered here, derived from a real case of a *motor manufacturing plant*, adds the concept of *Job Batches*, instead of just single jobs, *Machine Dependent Transportation Times* between stages, as well as *Work in Progress Buffers* of limited capacity, placed before and after each machine (input and output buffers), providing a waiting space for the jobs that can not be processed by their next machine because of unavailability. More specifically, instead of a set of jobs, we are given a set of groups of jobs O as input $(1, \ldots, |O|)$, where all jobs $j \in o$ inside a group $o \in O$ have to be scheduled sequentially. Using this concept in mind, we define $J_o \in O$ as the jobs within group o. These job groups are *customer orders* of the manufacturer. By taking into account machine-dependent transportation times, we also factor in the fact that the processing machines between two stages may be far away from each other, and if a job

is assigned to a machine on both stages, it has to travel a great distance between processes, affecting the time required for the original schedule. In this specific case, specialized robots (AGVs) are programmed to transfer jobs from machine to machine between stages, but that is a technicality that is not relevant to the scheduling algorithms presented in the rest of the paper. It should finally be noted that in this HFFS, all machines and buffer capacities inside the same stage are *identical*, so our processing times, p_{sj} $s \in S, j \in J$, are *stage-dependent*, instead of machine-dependent. A simple graphical representation of the HFFS examined here is provided at figure I. In the figure, we have three stages (S_1, S_2, S_3), three jobs that are assumed to be part of the same order (j_1, j_2, j_3), and finally some machines and buffers at each stage. The lines between two machines represent distance, where larger distances result in larger transportation times, while the red lines between the machines of the first and last stage denote that a job can skip some stages.

III.3. Solution Representation & Implementations

An important part of every scheduling optimization method, is the way that the production schedules are formulated [44]. In simple flow-shop variants like the PFSP, each schedule can be presented using a single job sequence [16]. In contrast, in a job-shop, for a final solution to be communicated, we require m job sequences - one for each available machine [24]. For our specific HFFS case, we will use a representation similar to [34], but instead of having sequences of jobs in our schedules, we have sequences of job orders. A permutation provides the scheduling sequence of job orders in the first stage, and then machine assignments at every eligible stage are made using a *least-load* rule, which assigns a job to the machine with the minimum time load, considering the time required for the job. Having followed this assignment procedure, we have defined the machine assignments and sequences of every job in all stages, but we still have not used the input and output buffers on the machines. That is handled by a **constraint programming (CP)** model, which takes as input a job assignment and sequencing on machines, and it changes the starting and completion times of every job, to respect the existent buffers. Because the assignment and CP implementations fall outside the scope of this research, their details will be emitted from this paper, but their algorithmic implementations can be found in [2].

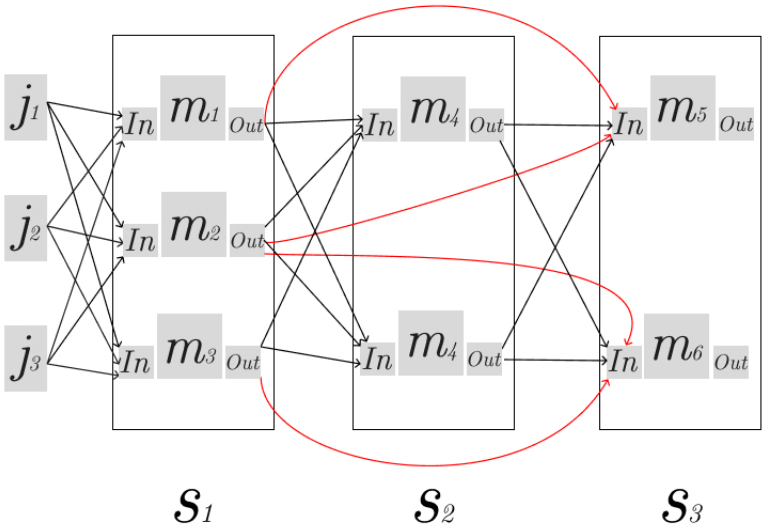

Figure I: A Hybrid Flexible Flow-shop

IV. Methodology

Before defining our algorithmic strategy to solve the HFFS problem, we should first take a step back and explain the reasoning behind the selection of this specific approach.

As evident from the available literature in the scheduling field and presented in II, popular solution methodologies include **exact methods** and **metaheuristics**. Since we aim at providing a methodology that is applicable to real-world scenarios, exact methods are not an attractive choice because our combinatorial problem is **NP-hard** [27] [11] [7] and thus, it would be inefficient to search for an exact solution, even on small instances [17] [6] [28]. Instead, on this type of problems, researchers typically enforce approximation methods like metaheuristics, that provide quicker but not necessarily optimal solutions, and are able to tackle larger problems [4]. That being said, if the respective metaheuristic is well-tuned, the non optimal solutions it produces can be really close to the global optimum [41].

In regards of metaheuristics and as mentioned in II, **Genetic Algorithms** have been frequently seen on flow shop scheduling problems, because of their great results and efficiency [6] [15] [45] [17]. In parallel, search-based algorithms also have good potential for real-world scales [42], with the **Adaptive Large Neighborhood Search** method being a great example because of its competitive optimality gaps [41]. For these reasons, this paper investigates the

results and performance of a Genetic Algorithm and the Adaptive Large Neighborhood Search method on the Hybrid Flexible Flowshop problem of our unique case.

IV.1. Genetic Algorithm (GA)

Genetic and Evolutionary Algorithms were first introduced by [46], as a way to solve complex optimization problems, which other older methods were unable to handle. The way a Genetic Algorithm operates is by employing aspects of the **Darwinian Theory** and natural phenomena to simulate the genetic procedure of biology, in order to efficiently navigate the solution space of a problem [35]. Just like in biology, where parents with good genes are more likely to have children with good genes as well, parts of two good solutions of an optimization problem can be conjoined to produce other good solutions [45]. The following details of the GA procedure are derived by [45]. For starters, instead of relying on improving a single solution, a GA uses a population of solutions, and through an iterative process of selection and genetic computations, the population evolves over time, and good solutions are continuously passed onto the next generations. In order to formulate a GA, one needs to define some basic common properties of the algorithm [24], which can be categorized as follows. i) ***Chromosome Encoding/Decoding*** - the way a solution of the problem is formulated to define a *chromosome*, ii) ***Population Generation*** - the method used to generate the initial population of chromosomes, iii) ***Chromosome Fitness*** - the function used to evaluate each chromosome, iv) ***Chromosome Selection*** - the method used to select the parent chromosomes of the population to reproduce, based on their fitness, v) ***Crossover Operation*** - a function that takes as input some parent chromosomes and produces new offspring and finally, vi) ***Mutation Operation*** - a function that takes as input an offspring, and performs a mutation on it. In the above procedure, the selection operation is used to substitute bad chromosomes in the population with better-fitted ones, whereas the crossover operator intensifies the search over good solution spaces and the mutation operator offers diversification in the search [8]. The generic algorithmic flow of a GA is described in a simple chart at figure II.

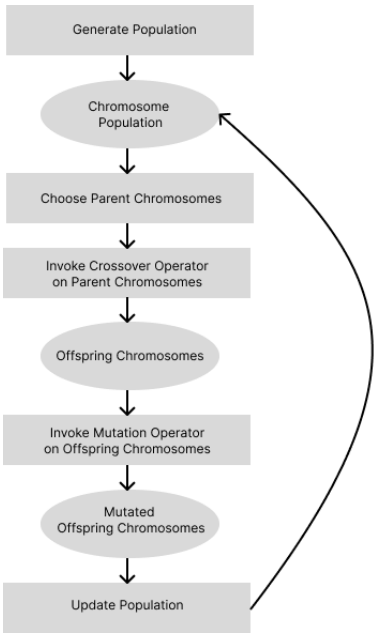

Figure II: Flow of a Genetic Algorithm

In our unique HFFS problem, as it has already been mentioned in III.3, we use a permutation of unique orders to communicate a valid solution. That means that the chromosome encoding and decoding method we employ is permutation-based [17]. Similar encoding strategies are enforced by [34], [17] and [8]. The rest of the GA properties mentioned above are defined for our specific problem in the following parts.

IV.1.1. Population Generation

The implemented method for generating a initial chromosome population for our HFFS uses a randomness factor to initialize chromosomes as random permutations of orders. Since all chromosomes have the same length (number of orders), this population can be generated by placing each available order in a random valid position of each chromosome. The choice to yield a random population of chromosomes for the start of the algorithm is made based on the fact that it is a common practice for the scheduling field, as many researches like [18], [35] and [17] have used it.

IV.1.2. Chromosome Fitness

In each iteration of a genetic algorithm, all chromosomes are evaluated using a fitness function in order to single out the fittest solutions of the population which will be used as the parents for the current generation [4]. For maximization problems, the fitness function and the objective function can be used interchangeably within the problem scope [45]. For minimization problems like the ones under the makespan objective, the fitness of a chromosome can be defined as the *inverse* of its objective value [18]. Using the same approach on our unique HFFS, if a chromosome has an objective of C_{max}, its fitness value is defined as C_{max}^{-1}, or $\frac{1}{C_{max}}$. This way, the smaller the makespan value of a chromosome, and thus the closer it is to the optima value, the higher its fitness.

IV.1.3. Chromosome Selection

Following the evaluation of all chromosomes within a population, the next step is to find which chromosomes will be chosen to become parents. As the goal of the selection procedure is to pick out the chromosomes that are likely to produce good offspring for the population of the next generation [17], we will use a *roulette-wheel* approach, that takes into account both the fitness of the individuals, but also a randomness factor, to give the highest selection probabilities to the chromosomes with the highest fitness [4]. So, at every iteration of the genetic algorithm, we give each chromosome, i, a $score_i$ value that is computed as:

$$score_i = \frac{fit_i}{\sum\limits_{c \in Pop} fit_c}$$

Where Pop is the population and fit_i is the fitness value of chromosome i. $Score_i$ is the probability that the chromosome i will be selected. Using this roulette-wheel, we pick out two parents at each iteration in our implementation. This type of selection method has been used in the past by [35], [4] and [18].

IV.1.4. Crossover

Having selected our two parents that we wish to use to produce new offspring for the next generation, we now need to define the way that these two chromosomes will be combined, in order to retain the good features of each parent into the resulting chromosomes [8]. The crossover operator used in a genetic algorithm is an essential part of the implementation as it provides the logic that will be used to exploit the good characteristics of older generations, in order to pass them into the next ones [8]. In this research, we implement and compare four crossover operators that appear to have good performance on flow-shop scheduling problems. These literature operators are taken from [34], [35], [6] and [8]. From [34], we enforce the two best proposed operators, ***LJMPX* and *MPOBX***, in our genetic search, while we also use the ***PUX*** (or uniform) operator from [6] and [35]. Finally, we also test our algorithm using the **Swap Crossover (SX)** of [8]. [34] proposes these operators instead of using other classical options because the latter methods do not have the expected performance when employed in the HFFS problem, as they do when employed on PFS problems. LJMPX works by inserting a random block of orders from the first parent into a random position of the first offspring, and then by using a *left* and a *right* list containing the unscheduled orders from the other parent, to place the rest of the orders greedily using a makespan estimation of the complete sequence. In the other hand, MPOBX uses four random crossover points and keeps the longest order block from one parent, before inserting the rest of the jobs from the second parent, using a similar makespan estimation approach. These two methods appear to behave better on our problem, because they manage to retain large blocks of jobs from the parents, instead of breaking them [34]. As proposed by the authors, to achieve the best possible results in our final schema, we use the two operators at the same time, by randomly choosing one of the two in each generation, instead of fixating on just one. Finally, the Swap Crossover operator is promoted by [8] as a way to make sure that the resulting offspring retains as much information from the parent chromosomes as possible. Something that other previously reported crossovers fail to do [8]. To make this possible, this operator tries to produce an offspring that is similar to the second parent, by performing a series of swaps and only terminating when the result is of better quality than the first parent chromosome.

In our implementation, all of the crossover operators are adapted to produce two children chromosomes, from two parent chromosomes.

IV.1.5. Mutation

The final step of a genetic algorithm, before replacing some of the chromosomes of the population with the newly created offspring, and iterating over the next generation, is to *mutate* the resulting offspring of the crossover to change a small part of it, to explore new solution spaces [8] [24]. The reason the genetic procedure includes this step, is because by only using a crossover to replace chromosomes of the population, we cannot introduce new information into the next generation [18] and we run the risk of early convergence of the population into local optima spaces [8]. The mutation operator that will be promoted in our final genetic algorithm is the ***Local Swap (LS)*** of [8], that aims at changing a small random gene range of the offspring, by doing swaps inside that range.

IV.2. Adaptive Large Neighborhood Search (ALNS)

The ***Adaptive Large Neighborhood Search (ALNS)*** was first introduced by [47], as an enhancement to the older search method, ***Large Neighborhood Search (LNS)*** of [39]. Similarly to the original algorithm, its first implementation was on vehicle routing problems where, as mentioned in II, proved to be an efficient optimization procedure. The reason why these methods emersed, when compared with other algorithms, is their unique way of navigating through the solution space of a problem [38]. More specifically, as stated by [47] and [38], when a problem contains very tight constraints, local search heuristics and other metaheuristic approaches can struggle to move from a promising solution space to the next, because the neighboring solutions produced often fall very close to the already examined spaces. The way that large neighborhood search methods counter this is by temporally allowing the solutions to fall outside the feasibility space of the problem [47] by using moves that relax and re-optimize the problem constraints, resulting in drastically changed solutions [39]. In an LNS, we iteratively employ two operators in a current solution. The first operator is called the *destruction operator* and, as the name suggests, ruins a large part of the current solution by using some predefined method, while the predefined *recreation operator*, takes

the destructed result and fixes it to produce a feasible solution [37]. An (A)LNS algorithm follows a similar logic, but instead of using a single destroy and a single recreate operator, in a deterministic way, it uses two sets of operators which are *adaptively* chosen using a nondeterministic approach, based on their performance in previous iterations during the search [41] [42]. The *performance* of an operator is recorded using a respective *weight* for each destroy and recreate operator, which is updated in every iteration of the execution, based on the quality of the solution in which the entire destruction and recreation phase resulted [47]. A famous way to update the weight of the two operators employed at each iteration is the following formula, as suggested by [47] and [40]:

$$w_{i,j+1} = w_{i,j} * (1 - r) + r * (p_i / u_i)$$

In the formula above, i is the operator that has been used in an iteration j, p_i is the quality of the produced solution after the reconstruction procedure, u_i is the number of times operator i has been used in the past, and $r \in [0, 1]$ is a predefined constant used to control the effect that the past weight ($w_{i,j}$) and p_i/u_i have in the updated weight of the operator. For example, with $r = 0.5$, the previous weight of the operator and the performance it had in the current iteration have the same weight in affecting the updated weight, while when $r = 0$, the new weight of an operator is not based on its performance in previous iterations [47]. For our implementation, we use $r = 0.9$ based on [40]. Given these weights at each iteration, a roulette-wheel is used for the selection of both a destruction and a recreation operator [38] [41]. That is the reason we want to include u_i in the formula, in order to minimize the overuse of any single operator [42]. The abstract flow of an ALNS can be seen in figure III.

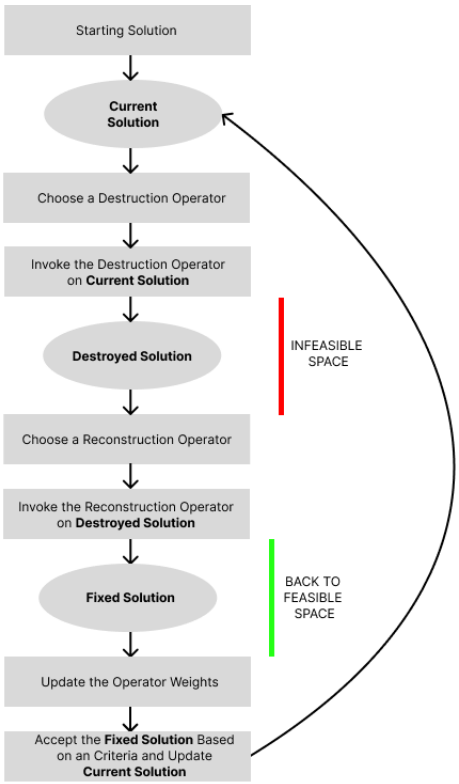

Figure III: Flow of an Adaptive Large Neighborhood Search

Just like in the GA, in order to construct an ALNS procedure for our problem, we decided to use and combine multiple operators taken from different parts of the literature, that appear to give promising results on this type of problems. These operators are depicted in the following paragraphs.

IV.2.1. Destruction Operators

The first destruction operator that was implemented in our ALNS is the basic ***N-random*** heuristic as mentioned in [48]. This simple rule picks N random positions from the original sequence and removes the elements placed in those positions. From [48], we also adopted the ***N-shortest greedy***, the ***N-longest greedy*** and the ***N-two-tails*** removal heuristics. All of these operators have a common property that incorporates a randomness factor regarding the decision of the destruction size. In parallel, our procedure enforces the ***random block removal*** of [42], which can be thought of as a variation of the N-random removal, but where the removed elements are positioned in a sequence. The last destruction operator that will be

used in our adaptive search is an adaptation of the dissimilarity removal heuristic as defined in [49]. This adaptation uses a ***dissimilarity matrix*** and tries to remove pairs of orders that are the most dissimilar with each other.

IV.2.2. Reconstruction Operators

Having invoked one of the mentioned destruction operators of IV.2.1, the next step, as previously mentioned, is to transform the destroyed sequence into a feasible schedule. To do that, we take as input the positions and the orders that were removed from the original sequence during the destruction procedure, and invoke a recreation operator. The simplest operator that our optimization procedure uses to recreate a feasible order sequence, is the ***random recreation*** of [42]. With this method, we randomly allocate the removed orders in the available empty positions as provided by the destruction procedure. In addition, since the set of destruction operators contains the block removal of [42], we also need to implement the ***best block insertion*** in the set of reconstruction operators, as proposed by the same author. To retain consistency, if the algorithm chooses the random block removal operator in the destruction phase of an iteration, then the best block insertion will be used in the reconstruction procedure with 100% probability. Finally, we also use the ***cyclic construction*** of [48] and a custom made operator that we will call ***greedy swaps recreation***. This operator performs all swaps for each removed order in the destroyed solution, with all other removed orders and chooses the one with the minimum makespan, before locking in place the two swapped orders and moving on to the rest non-locked removed orders. For the reconstructed solutions produced, we use the acceptance criteria of [48], which can be considered a variation of the Simulated Annealing criterion, but it does not require an initial temperature or a cooling rate.

V. Experiments

This chapter covers the description of the experimentation procedure and analyzes the respective results of our empirical investigation on the two metaheuristic schemas, to finally present and compare their efficiency in various HFFS plants.

Before continuing with the procedure, it should first be made clear what *efficiency* means

for our case. In order to evaluate a specific execution of one of the methods, we use a set of metrics, with respect to the metaheuristic used. For the GA, we use the following performance measures: i) **maximum fitness** found throughout the generations, ii) **number of generations** needed to find the chromosome with the maximum fitness, iii) **time needed** to find the chromosome with the maximum fitness and finally, iv) **total execution time** of the algorithm. In turn, we calculate the following statistics for the ALNS: i) **minimum makespan** found throughout the search, ii) **number of iterations** needed to find the solution with the minimum makespan, iii) **time needed** to find the the solution with the minimum makespan and finally, iv) **total execution** time of the search. In addition, we also use the *Relative Percentage Difference* for both the GA and ALNS, which denotes the percentage distance of a given solution, from the best solution found in each instance. RPD is calculated as $\frac{\text{SOL - BEST\_SOL}}{\text{BEST\_SOL}}$ [41] [34] [35] [17] [29] [11] [7] [6], where, for our case, *sol* is the makespan value of an algorithm and *best_sol* is the minimum makespan found by any execution of the same algorithm, in the same problem type (same orders and stages). The reason we do not use the best solution from **both** the algorithms for *best_sol*, is because we aim to first analyze each algorithm separately, before comparing them together. Thus, in our analysis, the RPD value provides an indicator of the average makespan deviation from the respective minimum of each problem. Using these metrics, we derive conclusions on which method provides a more suitable solver for the HFFS problem.

V.1. Instance Generation

To test the methods, we generated problem instances, comprised of benchmark data. As our unique HFFS case has yet to be presented in the literature, our generation procedure combines multiple other generation procedures that were found in the literature. For the buffers, orders and transportation times, where the literature fails to provide us with enough benchmarks, we use custom method for the generation. As proposed by [11], we use number of stages $|S| = \{3, 6, 9\}$ and number of machines on each stage s, $|M_s| \in U(1, 3)$. For the orders, we use $|O| = \{10, 30, 50\}$ where each order o, contains $|J_o| \in U(1, 3)$ jobs. The buffer capacities in the machines of each stage are taken from $U(1, 2)$. Moving to the processing times of the jobs on the stages, we use $p_{sj} \in U(2, 15) \forall s \in S, j \in J$ and the probability of a

job skipping a stage is set to 20% as proposed by [11]. Since our case also contains machine-dependent transportation times, we use the following formula: $t_{mm\prime} = r \in U(1,4) \cdot (s - s\prime)$, where m is a machine of stage s and $m\prime$ is a machine on stage $s\prime$. This ensures that stages far away from each other will have greater transportation times for their machines, while also incorporating a randomness factor. It should also be noted that for each instance type (orders and stages), three versions of that instance are generated (with different machines, processing times etc), in order to not overfit our methods. Taking into account all of the defined ranges for the problem properties, we are looking at a total of $|S| \times |O| \times 3 = 3 \times 3 \times 3 = 27$ different instance inputs.

V.2. Design of Experiment

Having defined both the evaluation metrics we will use for the metaheuristics, and the generation method for our input test data, the underline of the experimental procedure should be now made clear. In order to test the methods, a *Design of Experiment* (DOE) is employed to find the parameters that will be used for the algorithms. These parameters, as already discussed, are the population size (pop_size), the number of generations (G_{max}) and the mutation probability (P_m) for the GA, while we use a time limit parameter for the ALNS, that specifies the seconds that need to be passed before the algorithm terminates. Since these parameters are quantitative and unbounded variables and thus can take an infinite number of values, a *fractional factorial design* is performed on a logical subset of these values. After a preliminary experimentation on a random sample of the generated instances, we chose the following values for the parameters of the GA: $pop\_size = 30$, $G_{max} = 100$ and $P_m = 0.3$. Since the goal of this research is to provide insights on the differences between a GA and an ALNS on the same problem, in order to perform a fair comparison we set the time limit of each ALNS run to be equal to the total execution time of the corresponding GA. This way, both the GA and the ALNS will run for the same amount of time. For the same reason we also use the best solution of the initial GA population as the starting solution for our ALNS.

As we aim at investigating the performance of multiple GA crossover operators on this problem, the first part of the experiment is to conduct a comparison between all the crossover methods that were discussed in this paper, and then use the best one for the rest of the

experiments. To do that, we start by taking another random instance sample, and then by executing the GA with every one of the crossovers, using the parameters we just mentioned. It should finally be noted that each method was executed a total of three times on each instance and only the averages are considered, to reduce the stochastic effect of each metaheuristic on the final results. This gives us a total of $3 \times |instances| = 3 \times 27 = 81$ executions for each metaheuristic.

All experiments have been performed on a machine with Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40GHz and 8GB RAM, running the Windows 10 operating system. All the algorithms were written and run using Python 3.10.11.

V.3. Crossover Performance

As already stated, in order to make the choice of the crossover operator that will be used for the final comparisons to the ALNS, an independent comparison between the operators will first be performed, based on a sample of the input instances. The detailed results of the crossover operators on the sample are summarized in the following table. As mentioned in V.2, each method on an instance was executed three times and the results represent the averages.

Table II: Crossover Operators Performance

Operator	**Metric**	**10x9**	**30x6**	**50x3**
LJMPX-MPOBX	Fitness	0.005736	0.004666	0.003509
	RPD	0.001916	0.001558	0.003521
	Generation found	42.666667	86.333333	53.666667
	Time found (s)	481.383927	7241.910230	7399.017616
	Total time (s)	1086.425840	8779.482048	14413.573158
PUX	Fitness	0.005609	0.004546	0.003492
	RPD	0.024904	0.028037	0.008216
	Generation found	58.666667	82.333333	72.000000
	Time found (s)	113.611025	118.432500	100.966750
	Total time (s)	175.363301	146.774660	138.570770
SX	Fitness	0.005672	0.004567	0.003501
	RPD	0.013410	0.023364	0.005869
	Generation found	41.666667	90.000000	61.333333
	Time found (s)	174.861736	395.250073	409.900998
	Total time (s)	322.833921	420.989540	700.045008

Fitness: maximum fitness reached in all generations. **RPD**: relative percentage deviation from the best solution in the same instance from all operators. **Generation found**: the generation where the best solution was discovered. **Time found**:

computational time (seconds) when best solution was found. **Total time**: the algorithm execution time (seconds). Column headers represent problem instance sizes where the first number denotes the number of orders in the instance and the second number, the number of stages. All numbers are averages taken on three executions of the respective combination.

From these results, it is evident that the **LJMPX-MPOBX** combination provides the most promising results in most metrics, when compared with the **PUX** and **SX** crossovers. In contrast, a large difference in the execution times can be noticed between the first and the other two operators. In more details, **LJMPX-MPOBX** achieves the best average RPD and fitness across all instances, while also having the ability to find its best solution in earlier generations than the other two operators, with an average of 60 generations needed. Since the total number of generations was limited at 100, an average number of 60 generations required to find the best solution can be considered a positive trait and not an indication that **LJMPX-MPOBX** converges early. Even though **LJMPX-MPOBX** provides promising results in terms of solution quality, the computational costs of the method are substantial, as evident from the times in the last two rows of its metrics. In fact, this operator can require up to **35**$\times$ more computational time than the other two operators, in average. When we move our attention to the **PUX** crossover, the results are polar-opposites. That is true because the method achieves the worst average fitness and RPD and requires more generations to produce its best results, when compared to the other crossover schemas. Instead, what this crossover is providing us, is a method that is fast and not computational expensive, while its execution speed seems to be independent of the respective problem size, as the time values on the different columns do not have great variance. Finally, the **SX**'s results seem to fall in the middle of the results we just analyzed. Its average produced solution quality, generations needed to find the best solution and computational times, are the second best after **LJMPX-MPOBX**.

Even though we managed to locate the best operator by performing an comparative evaluation, some other insights can also be derived from the numerical results. Based on our empirical analysis, we propose the use of the **LJMPX-MPOBX** for HFFS problems when the solution quality is of critical importance, while on cases where fast computational times are required (e.g. for a fast-changing plant environment where multiple production schedules have to be created in short periods of time), the **PUX** crossover should be preferred. It should

also be noted that since **LJMPX-MPOBX** proves that it can produce good solutions in early generations, if we take advantage of computing techniques like **multi-threading** or by investing in more computational power, its results can become even more attractive. At last, the **SX** operator can be used when the production plant management is looking for a middle ground between schedule quality and computational time.

Since we aim at providing a method capable of handling realistic production environments, for the rest of the experiments will be using the **SX** operator as it produces schedules with good (but not the best) quality, without requiring the long computational times of the **LJMPX-MPOBX** crossover.

V.4. GA Performance

Having singled out the operator that seems to outperform the others, the next step is to use this operator on all available problem inputs, to measure its performance in detail. The respective results of the complete experiments using the SX operator are summarized in the following two tables. Table III provides the results per problem instance, while table IV informs on the averages per orders and stages. As already stated, for each problem size (stages and orders), we generated three problem input instances. Thus, in all the following tables, the results represent the respective averages of the problem sizes.

Table III: SX Operator Performance Results

$\|O\| \times \|S\|$	**Fitness**	**Makespan**	**Time Found (s)**	**Total Time (s)**	**Generation Found**	**RPD**
10x3	0.011922	83.878	60.919310	103.521518	48.777778	0.298507
30x3	0.002103	475.511	51.541258	511.398040	10.111111	0.084091
50x3	0.003488	286.697	635.079203	815.437473	72.000000	0.002331
10x6	0.008242	121.329	63.459572	145.499153	39.444444	0.408187
30x6	0.003636	275.027	441.134882	564.115404	73.555556	0.338384
50x6	0.001836	544.662	926.177251	1469.711723	60.555556	0.571759
10x9	0.005496	181.950	73.407896	156.733121	44.555556	0.046616
30x9	0.002621	381.533	777.975271	892.358886	81.888889	0.438534
50x9	0.002038	490.677	2268.227733	2518.904566	91.555556	0.390794

Table IV: SX Operator Average Performance by Configuration

	Fitness	Makespan	Time Found (s)	Total Time (s)	Generation Found	RPD
Orders						
10	0.008553	116.91	65.929	135.251	44.259	0.251103
30	0.002787	358.80	423.550	655.957	55.185	0.287003
50	0.002454	407.50	1276.495	1601.351	74.704	0.321628
Stages						
3	0.005838	171.29	249.180	476.786	43.630	0.128310
6	0.004571	218.77	476.924	726.442	57.852	0.439443
9	0.003385	295.42	1039.870	1189.332	72.667	0.291981

Based on the empirical results presented on the two tables, we can make some starting comments. For starters, we notice that as the problem grows in size, there is an increase on the generations and time needed to find the best solution, but also on the RPD value. The first insight tells us that, as we would expect, larger problems require greater computational times. The increasing number of generations required to find optimal solutions indicates that larger problem instances pose greater challenges for the algorithm in producing high-quality schedules, which delays the discovery of the best solution during the search process. This pattern may also suggest that the SX crossover operator demonstrates superior performance on smaller order sequences. In parallel, large numbers in the required generations suggest that the the risk of early convergence does not increase when the problems gets bigger. From table IV we notice that as the orders increase, the RPD value does as well. This indicates that as we include more orders in our production schedule, the GA struggles more in giving makespan values that are near the respective minimum of the problem category. From table III and IV, it is notable that the maximum deviations from the respective category minimum, is noticed when our production plant has a total of six stages.

V.5. ALNS Performance

Moving to the ALNS experiments, as already stated in V.2, we use information produced by the GA runs, in order to set limits to the search for a fair comparison. More specifically, the ALNS has a starting solution equal to the best solution produced on the initial population of the GA, and it runs for the same amount of seconds. All the numerical results of the ALNS executions are presented and summarized in tables V and VI.

Table V: ALNS Performance Results

$\|O\| \times \|S\|$	**Makespan**	**Iter. Found**	**Total Iter.**	**Time Found (s)**	**Total Time (s)**	**RPD**
10x3	87.89	85.33	314.44	27.71	103.99	0.331
30x3	477.00	7.11	238.33	17.34	515.16	0.084
50x3	287.56	32.11	106.22	245.43	838.36	0.005
10x6	134.11	78.00	360.11	32.46	146.05	0.396
30x6	293.22	92.22	147.78	331.72	576.49	0.357
50x6	604.11	60.22	101.11	872.06	1540.05	0.585
10x9	181.56	44.44	314.44	20.88	157.30	0.025
30x9	408.22	46.11	154.44	311.02	897.82	0.422
50x9	528.67	54.22	94.44	1661.01	2579.83	0.38

Table VI: ALNS Average Performance by Configuration

	Makespan	**Iter. Found**	**Total Iter.**	**Time Found (s)**	**Total Time (s)**	**RPD**
Orders						
10	134.52	69.26	329.66	27.02	135.78	0.25
30	392.81	48.81	180.18	220.03	663.16	0.287
50	473.44	48.18	100.59	926.17	1652.74	0.323
Stages						
3	284.15	41.52	219.67	96.83	485.84	0.14
6	343.81	76.81	203.00	412.08	754.20	0.437
9	372.81	48.26	187.78	664.31	1211.65	0.275

The first note that should be taken from these results, is that the ALNS appears to produce solutions with worse quality than the GA, on almost all instance types. We see that when inspecting the two *makespan* columns from the two methods. Although in some cases this increase is not substantial, we can point out a pattern of worse ALNS results, as the problem size gets bigger. That means that as we have more stages and more orders in our input problem, the use of the GA over the ALNS becomes more and more attractive. In parallel, when taking a closer look into the results, we notice a large difference between the total iterations executed and the iterations needed to find the best solution, which can be evidence of two things. It either means that the solution reached has such good quality, that the next iterations fail to improve upon it (which is a positive), or it is an indication that the search gets trapped into a local optima (which is a negative). Given that the GA is able to produce better solutions, we can confidently reach the conclusion that the second case is true, and our ALNS converges early. In contrast, by using this type of search we are able to find our best solution way faster than the GA, where in instance types (orders, stages) where the quality difference between the two algorithms is not large, can drive our final choice to

be the use of an ALNS. Finally, given the RPD results of the method, we can also make note at the fact that as the problem grows in size, the makespan results have a larger deviation from the minimum of the respective problem category. This and the fact that the maximum RPD is noticed when the stages are six, is consistent with the results of the GA.

In summary, the ALNS execution results suggest that this type of search should be used for producing schedules fast, when the final schedule quality is not a critical factor. These results also point out that when using this method, one should place a more constrained time limit on the search, as the method does not need great computational times to return a final production schedule. Of course, these results are heavily dependent on the operators used in our implementation, and may vary on different destruction and reconstruction operator sets.

V.6. Parameter Performance

Up to this point, we have executed all of the genetic algorithms using the same set of parameters, as mentioned in V.2. Having analyzed the respective results of these parameters on our experiments, it would also be of value to examine the performance of different GA parameter sets on our problem. More specifically, by trying different execution parameters on the same problem inputs, we can shine some light on the added value of *parameter tuning* in our specific problem. Additionally, by using the same execution parameters on all problem inputs, we run the risk of having parameters that favor certain problem types (e.g., small-scale problems) and underperform on others (e.g., large-scale problems). To overcome this risk, we tuned our GA to all problem inputs separately to find the best performing parameters on each problem size, and recorded the results. This methodology creates instance-specific parameter configurations instead of relying on a one-size-fits-all parameter set.

To conduct the parameter tuning, instead of using a *brute-force* search, we employ a **Tree-structured Parzen Estimator**, that is provided to us through the *optuna* library of Python. We use this approach to avoid the long execution times of the former approach, and instead of trying all different parameter combinations inside a range, we run three TPE iterations and record the best parameters found in those iterations. The parameter ranges that were used are $pop\_size \in [30, 100], G_{max} \in [100, 150], P_m \in \{0.5, 0.6, \ldots, 1\}$. The following two tables present the final fitness results per problem type, as well as the averages

per orders and stages.

Table VII: Instance Performance Results

$\|O\| \times \|S\|$	**Fitness**	**Makespan**	**RPD**
10x3	0.011997	83.354	0.313131
30x3	0.002103	475.511	0.084091
50x3	0.003505	285.306	0.001170
10x6	0.008303	120.438	0.371134
30x6	0.003669	272.553	0.334855
50x6	0.001845	542.005	0.570931
10x9	0.005551	180.147	0.036398
30x9	0.002638	379.075	0.390046
50x9	0.002042	489.715	0.376091

Table VIII: Average Performance by Configuration

	Fitness	**Makespan**	**RPD**
Orders			
10	0.008617	116.049	0.240221
30	0.002803	356.760	0.269664
50	0.002464	405.844	0.316064
Stages			
3	0.005868	170.415	0.132797
6	0.004606	217.108	0.425640
9	0.003411	293.169	0.267512

When comparing the results of tables III and IV with tables VII and VIII, we observe a small but consistent improvement in both average fitness and RPD values. While this improvement is not substantial across any individual instance, it nonetheless indicates that tuning our GA can potentially yield better results. As we just mentioned, we conducted only three trials when searching for optimal GA parameters, therefore, these results suggest that greater performance improvements could be achieved by increasing the total number of trials.

VI. CONCLUSION

In this research, we conducted an empirical analysis on the performance of two metaheuristic methods, in a **Hybrid Flexible Flow-shop**. Using a Design of Experiment approach, we managed to conclude that an **Genetic Algorithm** might be a more suitable solver for the HFFS, over an **Adaptive Large Neighborhood Search**. In contrast, the latter method can be an attractive choice if the goal of the manufacturer is to produce production schedules in mass, or when they have a more flexible production plant and they want to produce their schedules fast, ad-hoc. In parallel, we proved that the former method should instead be used on critical production environments, when the total production time should be as low as possible. In cases like that, the production manager will have to execute the scheduler well in advance before production (preferably overnight), and schedule multiple productions ahead of time, because of the great computational times needed. In our work, we also provided insights on the performance of multiple **crossover operators**, as well as the added value of a tree-structured Parzen estimator for **parameter tuning** and the use of different execution parameters on different production environments, instead of relying on a constant parameter set.

VII. Future Work

This paper serves as a stepping stone for more advanced analysis of hybrid flexible flow-shop solvers. Based on the empirical results, several interesting directions for future work emerge: analyzing additional production metrics such as *throughput* and *machine utilization*, evaluating different destruction and reconstruction operators on the same problem instances, and implementing more sophisticated tree-structured Parzen estimator configurations.

References

[1] Markus Sommer, Josip Stjepandić, Sebastian Stobrawa, and Moritz von Soden. Automated generation of digital twin for a built environment using scan and object detection as input for production planning. *Journal of Industrial Information Integration*, 33:100462, 2023.

[2] Stavros Vatikiotis, Ilias Mpourdakos, Dimitrios Papathanasiou, and Ioannis Mourtos. Makespan minimisation in hybrid flexible flowshops with buffers and machine-dependent transportation times. In Matthias Thürer, Ralph Riedel, Gregor von Cieminski, and David Romero, editors, *Advances in Production Management Systems. Production Management Systems for Volatile, Uncertain, Complex, and Ambiguous Environments*, pages 258–273, Cham, 2024. Springer Nature Switzerland.

[3] L.J. Zeballos, O.D. Quiroga, and G.P. Henning. A constraint programming model for the scheduling of flexible manufacturing systems with machine and tool limitations. *Engineering Applications of Artificial Intelligence*, 23(2):229–248, 2010.

[4] W Xu, HY Sun, AL Awaga, Y Yan, and YJ Cui. Optimization approaches for solving production scheduling problem: A brief overview and a case study for hybrid flow shop using genetic algorithms. *Advances in Production Engineering & Management*, 17(1):45–56, 2022.

[5] Victor Fernandez-Viagas and Jose M. Framinan. Exploring the benefits of scheduling with advanced and real-time information integration in industry 4.0: A computational study. *Journal of Industrial Information Integration*, 27:100281, 2022.

[6] Mostafa Zandieh, Eghbal Rashidi, et al. An effective hybrid genetic algorithm for hybrid flow shops with sequence dependent setup times and processor blocking. 2009.

[7] N. Abbaszadeh, E. Asadi-Gangraj, and S. Emami. Flexible flow shop scheduling problem to minimize makespan with renewable resources. *Scientia Iranica*, 28(3):1853–1870, 2021.

[8] Yuxiang Guan, Yuning Chen, Zhongxue Gan, Zhuo Zou, Wenchao Ding, Hongda Zhang, Yi Liu, and Chun Ouyang. Hybrid flow-shop scheduling in collaborative manufacturing with a multi-crossover-operator genetic algorithm. *Journal of Industrial Information Integration*, 36:100514, 2023.

[9] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser, editors, *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 343–362. Elsevier, 1977.

[10] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[11] B. Naderi, Sheida Gohari, and M. Yazdani. Hybrid flexible flowshop problems: Models and solution methods. *Applied Mathematical Modelling*, 38(24):5767–5780, 2014.

[12] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[13] Ghita Lebbar, I El Abbassi, Abdelouahhab Jabri, A El Barkany, and M Darcherif. Multi-criteria blocking flow shop scheduling problems: Formulation and performance analysis. *Advances in Production Engineering & Management*, 13(2):136–146, 2018.

[14] Vladimír Modrák and R Sudhakara Pandian. Flow shop scheduling algorithm to minimize completion time for n-jobs m-machines problem. *Tehnički vjesnik*, 17(3):273–278, 2010.

[15] Quan-Ke Pan and Rubén Ruiz. Local search methods for the flowshop scheduling problem with flowtime minimization. *European Journal of Operational Research*, 222(1):31–43, 2012.

[16] B. Naderi and Rubén Ruiz. The distributed permutation flowshop scheduling problem. *Computers Operations Research*, 37(4):754–768, 2010.

[17] Jiaxin Fan, Yingli Li, Jin Xie, Chunjiang Zhang, Weiming Shen, and Liang Gao. A hybrid evolutionary algorithm using two solution representations for hybrid flow-shop scheduling problem. *IEEE Transactions on Cybernetics*, 53(3):1752–1764, 2023.

[18] Min Dai, Dunbing Tang, Adriana Giret, Miguel A. Salido, and W.D. Li. Energy-efficient scheduling for a flexible flow shop using an improved genetic-simulated annealing algorithm. *Robotics and Computer-Integrated Manufacturing*, 29(5):418–429, 2013.

[19] Chao-Hsien Pan. A study of integer programming formulations for scheduling problems. *International Journal of Systems Science*, 28(1):33–41, 1997.

[20] Amir Mollaei, Mohammad Mohammadi, and Bahman Naderi and. A bi-objective milp model for blocking hybrid flexible flow shop scheduling problem: robust possibilistic programming approach. *International Journal of Management Science and Engineering Management*, 14(2):137–146, 2019.

[21] Wenwu Han, Qianwang Deng, Guiliang Gong, Like Zhang, and Qiang Luo. Multi-objective evolutionary algorithms with heuristic decoding for hybrid flow shop scheduling problem with worker constraint. *Expert Systems with Applications*, 168:114282, 2021.

[22] Gregory A. Kasapidis, Dimitris C. Paraskevopoulos, Ioannis Mourtos, and Panagiotis P. Repoussis. A unified solution framework for flexible job shop scheduling problems with multiple resource constraints. *European Journal of Operational Research*, 320(3):479–495, 2025.

[23] Gregory A Kasapidis, Dimitris C Paraskevopoulos, Panagiotis P Repoussis, and Christos D Tarantilis. Flexible job shop scheduling problems with arbitrary precedence graphs. *Production and Operations Management*, 30(11):4044–4068, 2021.

[24] Byung Joo Park, Hyung Rim Choi, and Hyun Soo Kim. A hybrid genetic algorithm for the job shop scheduling problems. *Computers & industrial engineering*, 45(4):597–613, 2003.

[25] Hamid Reza Mollaei, Samira Zeynali, and Nasser Shahsavari Pour. Solving the multi

objective flexible job shop problem using combinational meta heuristic algorithm based on genetic algorithm and tabu-search. *J Basic Appl Sci Res*, 3(9):713–720, 2013.

[26] Eugeniusz Nowicki and Czesław Smutnicki. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8(2):145–159, 4 2005.

[27] Betul Yagmahan and Mehmet Mutlu Yenisey. Ant colony optimization for multi-objective flow shop scheduling problem. *Computers Industrial Engineering*, 54(3):411–420, 2008.

[28] Dipak Laha and Uday K. Chakraborty. An efficient stochastic hybrid heuristic for flowshop scheduling. *Engineering Applications of Artificial Intelligence*, 20(6):851–856, 2007.

[29] Mansour Eddaly, Bassem Jarboui, and Patrick Siarry. Combinatorial particle swarm optimization for solving blocking flowshop scheduling problem. *Journal of Computational Design and Engineering*, 3(4):295–311, 05 2016.

[30] Thomas Stützle et al. An ant approach to the flow shop problem. In *Proceedings of the 6th European Congress on Intelligent Techniques & Soft Computing (EUFIT'98)*, volume 3, pages 1560–1564. Verlag Mainz, Wissenschaftsverlag Aachen, 1998.

[31] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[32] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[33] C. Anand Deva Durai, M. Azath, and J. S. C. Jeniffer. Integrated search method for flexible job shop scheduling problem using HHS–ALNS algorithm. *SN Computer Science*, 1(2):83, 3 2020.

[34] Aymen Sioud, Marc Gravel, and Caroline Gagné. A genetic algorithm for solving a hybrid flexible flowshop with sequence dependent setup times. In *2013 IEEE Congress on Evolutionary Computation*, pages 2512–2516, 2013.

[35] M. Zandieh and N. Karimi. An adaptive multi-population genetic algorithm to solve the multi-objective group scheduling problem in hybrid flexible flowshop with sequence-dependent setup times. *Journal of Intelligent Manufacturing*, 22(6):979–989, 12 2011.

[36] Rubén Ruiz and Concepción Maroto. A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. *European Journal of Operational Research*, 169(3):781–800, 2006.

[37] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[38] Achmad P. Rifai, Huu-Tho Nguyen, and Siti Zawiah Md Dawal. Multi-objective adaptive large neighborhood search for distributed reentrant permutation flow shop scheduling. *Applied Soft Computing*, 40:42–57, 2016.

[39] Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. *APES Group, Dept of Computer Science, University of Strathclyde, Glasgow, Scotland, UK*, 46, 1997.

[40] Weishi Shao, Zhongshi Shao, and Dechang Pi and. A parallel deep adaptive large neighbourhood search algorithm for distributed heterogeneous hybrid flow shops with mixed-model assembly scheduling. *Engineering Optimization*, 57(2):543–570, 2025.

[41] Amir M. Fathollahi-Fard, Lyne Woodward, and Ouassima Akhrif. A scenario-based robust optimization model for the sustainable distributed permutation flow-shop scheduling problem. *Annals of Operations Research*, 4 2024.

[42] Damla Kızılay and Zeynel Abidin Çil. Adaptive large neighborhood search heuristic for mixed blocking flowshop scheduling problem. *International Journal of Pure and Applied Sciences*, 7(1):152–162, 2021.

[43] Rujapa Nanthapodej, Cheng-Hsiang Liu, Krisanarach Nitisiri, and Sirorat Pattanapairoj. Hybrid differential evolution algorithm and adaptive large neighborhood search

to solve parallel machine scheduling to minimize energy consumption in consideration of machine-load balance problems. *Sustainability*, 13(10), 2021.

[44] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. Equivalence between schedule representations: Theory and applications. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 237–247, 2008.

[45] Chuen-Lung Chen, Venkateswara S. Vempati, and Nasser Aljaber. An application of genetic algorithms for flow shop problems. *European Journal of Operational Research*, 80(2):389–396, 1995.

[46] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 04 1992.

[47] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[48] Rujapa Nanthapodej, Cheng-Hsiang Liu, Krisanarach Nitisiri, and Sirorat Pattanapairoj. Hybrid differential evolution algorithm and adaptive large neighborhood search to solve parallel machine scheduling to minimize energy consumption in consideration of machine-load balance problems. *Sustainability*, 13(10), 2021.

[49] Ioannis Avgerinos, Ioannis Mourtos, Nikolaos Tsompanidis, and Georgios Zois. Pickup delivery with time windows and transfers: combining decomposition with metaheuristics, 2025.

Appendix

Code Availability

The complete source code for all algorithms (GA, ALNS and all respective operators) implemented in this study is publicly available on GitHub at:

`https://github.com/HliasMpGH/hffs-optimization-thesis`

The repository includes:

- Crossover and Mutation Operators
- Implementation of the ALNS with destruction and reconstruction operators
- Problem instance generators for hybrid flexible flow-shop scheduling
- Experimental setup and parameter configurations
- Result analysis
- Documentation for reproducibility

Data Availability

All experimental data used in this study, including problem instances and detailed results, are available in the same repository under the `src/input/` directory. The dataset comprises:

- Problem instances for all tested configurations
- Raw experimental results for all trials of both algorithms
- Notebooks to reproduce all the numerical results