UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ARTUR FERREIRA BRUM

Automatic Algorithm Configuration for Flow Shop Scheduling Problems

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. rer. nat. Marcus Ritt

Porto Alegre
August 2020

CIP — CATALOGING-IN-PUBLICATION

Ferreira Brum, Artur

Automatic Algorithm Configuration for Flow Shop Scheduling Problems / Artur Ferreira Brum. – Porto Alegre: PPGC da UFRGS, 2020.

130 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Marcus Ritt.

1. Flow shop scheduling problem. 2. Automatic algorithm configuration. 3. Iterated local search. 4. Iterated greedy algorithm. I. Ritt, Marcus. II. Título.

ABSTRACT

Scheduling problems have been a subject of interest to the optimization researchers for many years. Flow shop problems, in particular, are one of the most widely studied scheduling problems due to their application to many production environments. A large variety of solution methods can be found in the literature and, since many flow shop problems are NP-hard, the most frequently found approaches are heuristic methods.

Heuristic search methods are often complex and hard to design, requiring a significant amount of time and manual work to perform such a task, which can be tedious and prone to human biases. Automatic algorithm configuration (AAC) comprises techniques to automate the design of algorithms by selecting and calibrating algorithmic components. It provides a more robust approach which can contribute to improving the state of the art.

In this thesis we present a study on the permutation and the non-permutation flow shop scheduling problems. We follow a grammar-based AAC strategy to generate iterated local search or iterated greedy algorithms. We implement several algorithmic components from the literature in a parameterized solver, and explore the search space defined by the grammar with a racing-based strategy. New efficient algorithms are designed with minimal manual effort and are evaluated against benchmarks from the literature. The results show that the automatically designed algorithms can improve the state of the art in many cases, as evidenced by comprehensive computational and statistical testing.

Keywords: Flow shop scheduling problem. automatic algorithm configuration. iterated local search. iterated greedy algorithm.

Configuração Automática de Algoritmos para Problemas de Agendamento em Flow Shop

RESUMO

Problemas de agendamento tem sido assunto de interesse para pesquisadores em otimização por muitos anos. Problemas de *flow shop*, em particular, são alguns dos problemas de agendamento mais amplamente estudados devido à sua aplicação em muitos ambientes de produção. Uma grande variedade de métodos de resolução pode ser encontrada na literatura e, visto que muitos problemas de *flow shop* são NP-difíceis, as abordagens mais frequentemente encontradas são métodos heurísticos.

Métodos heurísticos de busca podem ser complexos e difíceis de projetar, requerendo uma significativa quantia de tempo e trabalho manual para realizar tal tarefa, que pode ser tediosa e propensa a viés humano. Configuração Automática de Algoritmos (CAA) compreende técnicas para automatizar o projeto de algoritmos, selecionando e calibrando componentes algorítmicos. Ela fornece uma abordagem mais robusta que pode contribuir para melhorar o estado da arte.

Nesta tese apresentamos um estudo sobre os problemas de agendamento em *flow shop* permutacional e não-permutacional. Nós seguimos uma estratégia de CAA baseada em gramática para gerar buscas locais iteradas ou algoritmos gulosos iterados. Nós implementamos vários componentes algorítmicos da literatura em um *solver* parametrizado, e exploramos o espaço de busca definido pela gramática com uma estratégia baseada em corridas. Novos algoritmos eficientes são obtidos com esforço manual mínimo e são avaliados em *benchmarks* da literatura. Os resultados mostram que os algoritmos projetados de maneira automatizada podem melhorar o estado da arte em muitos casos, conforme evidenciado por abrangentes testes computacionais e estatísticos.

Palavras-chave: Problema de agendamento em flow shop, Configuração automática de algoritmos, Busca local iterada, Algoritmo guloso iterado.

LIST OF ABBREVIATIONS AND ACRONYMS

PFSSP	Permutation Flow Shop Scheduling Problem
NPFSSP	Non-Permutation Flow Shop Scheduling Problem
MIP	Mixed-Integer Programming
B&B	Branch-and-Bound
IG	Iterated Greedy
CBFS	Cyclic Best-First Search
ILS	Iterated Local Search
ACO	Ant Colony Optimization
SA	Simulated Annealing
AAC	Automatic Algorithm Configuration
GP	Genetic Programming
GE	Grammatical Evolution
SGE	Structured Grammatical Evolution
SAT	Propositional Satisfiability Problem
VNS	Variable Neighborhood Search
GGA	Gender-Based Genetic Algorithm
EDA	Estimation of Distribution Algorithm
BNF	Backus-Naur Form
ARD	Average Relative Deviation

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The efficient solving of optimization problems helps to improve our lives, even though it often remains unnoticed. For example, we benefit from reduced delivery times for products bought online due to the improvement of solution methods for problems in logistics, or from reduced costs for products due to the efficiency gains in manufacturing processes. Shop scheduling problems, in particular, model many optimization problems that manufacturing and service industries face. These problems usually consist of efficiently allocating resources to jobs whose processing requires a certain time in such a way that minimizes a cost metric. Since the costs involved in the industry are often high, the research for more efficient resolution methods is highly desirable. However, optimization in production environments can involve challenging combinatorial problems. Most of them are NP-hard (COOK, 1971; LEVIN, 1973), i.e., no efficient polynomial-time algorithms for solving them are known. Besides, it is frequently impractical to solve large-sized problems that arise from the industry in a short time through exact approaches, such as mathematical programming. A common solution in these cases is the adoption of heuristic methods.

Heuristics often produce high-quality results, but on the other hand, they can be complex and hard to design. Furthermore, shop scheduling problems have a large number of variants to represent the differences in production processes in practice. The variants can include different objective functions, job sequence constraints, and machine configurations. However, most heuristics are explicitly designed to a single or a few variants, generating an increase in development cost when the methods have to be adapted to solve different problem variants. In addition, the methods usually have a set of parameters that have to be calibrated to ensure good results. The calibration is often performed manually, requiring a significant amount of time and effort, and being vulnerable to human error. The automation of the design and calibration processes has been a useful technique that can reduce the workload of designers and increase the robustness compared to a manual approach.

In this work, we address the permutation and the non-permutation flow shop scheduling problems. In these problems, each job is composed of a sequence of operations, and each operation has to be processed without interruption by a specific machine for a certain time. All jobs go through the same sequence of machines, one job at a time. We focus on minimizing the maximum completion time, i.e., the completion time of the

last job of the schedule, and the total completion time, i.e., the sum of the completion times of all jobs. We go beyond the typical goal of proposing a new algorithm. Instead, we build a solver that efficiently implements individual algorithmic components, and allows one to combine them into iterated local search or iterated greedy algorithms. We use an automated methodology to find efficient combinations of components, thus generating a set of new algorithms, and compare them to the state of the art.

1.1 Objectives

Our objective is to automate the design process of heuristics for flow shop problems based on a library of individual components that can be combined to generate full methods. We aim to reduce the manual effort required during the design process while producing equivalent algorithms. The specific objectives of this work are:

- Study if it is possible to automate the design process.
- Implement a library of algorithmic components in a solver.
- Automate the design process through automatic algorithm configuration techniques.
- Compare the obtained methods to the state of the art.

1.2 Contributions

The main contributions of this work are a solver that uses efficiently implemented algorithmic components from the literature, and a set of algorithms for each problem variant and objective function. Some components rely on an efficient implementation of non-trivial acceleration procedures, therefore providing such an implementation can be of high interest. We also studied non-permutation flow shops and provided more evidence in favor of its relevance. The source code of our solver is publicly available at <https://github.com/arturfb/FSSolver>.

Regarding the contributions to the literature, we have the following published papers:

- BRUM, A.; RITT, M. Automatic algorithm configuration for the permutation flow shop scheduling problem minimizing total completion time. In: LIEFOOGHE, A.;

LÓPEZ-IBÁÑEZ, M. (Ed.). **Evolutionary Computation in Combinatorial Optimization**. Cham: Springer International Publishing, 2018. p. 85–100. ISBN 978-3-319-77449-7. The European Conference on Evolutionary Computation in Combinatorial Optimization is classified by CAPES WebQualis as B1. The contributions of this paper are presented in Section 3.1.

- BRUM, A.; RITT, M. Automatic design of heuristics for minimizing the makespan in permutation flow shops. In: **IEEE Congress on Evolutionary Computation**. [S.l.: s.n.], 2018. p. 1–8. The IEEE Congress on Evolutionary Computation is classified by CAPES WebQualis as A1. The contributions of this paper are presented in Section 3.2.

And the following article which has been submitted:

- BRUM, A.; RUIZ, R.; RITT, M. Automatic generation of iterated greedy algorithms for the non-permutation flow shop scheduling problem with total completion time minimization. 2020, submitted to Computers & Operations Research. The contributions of this paper are presented in Section 4.1.

1.3 Overview

This thesis is organized as follows. Chapter 2 presents an introduction regarding flow shop problems (Section 2.1) and automatic algorithm configuration (Section 2.2). Chapter 3 presents our study on automatic algorithm configuration for permutation flow shop scheduling problems, and Chapter 4 presents our work on non-permutation flow shop scheduling problems. In both cases, we studied the minimization of the total completion time and the makespan, and describe our experiments and its results in respective sections. Finally, we present our conclusions and remarks in Chapter 5.

2 BACKGROUND

2.1 Flow Shop Scheduling Problems

Flow shop problems consist of scheduling a set of n jobs, each one comprising m operations, that have to be processed by a set of m machines so that the i^{th} machine processes the i^{th} operation. The machines are organized in a sequence, defined as $M_1, M_2, \ldots, M_m$, and each job goes through this sequence in that specific order, one machine at a time. Each job j has a processing time p_{ij} linked to its operation on machine i. Each machine can process only one job at a time, and preemption is not allowed. A solution for the problem consists of the assignment of a start time for the processing of each job on each machine, commonly referred to as a schedule. Schedules are usually represented as a sequence of jobs for each machine, in which each job is processed as soon as possible.

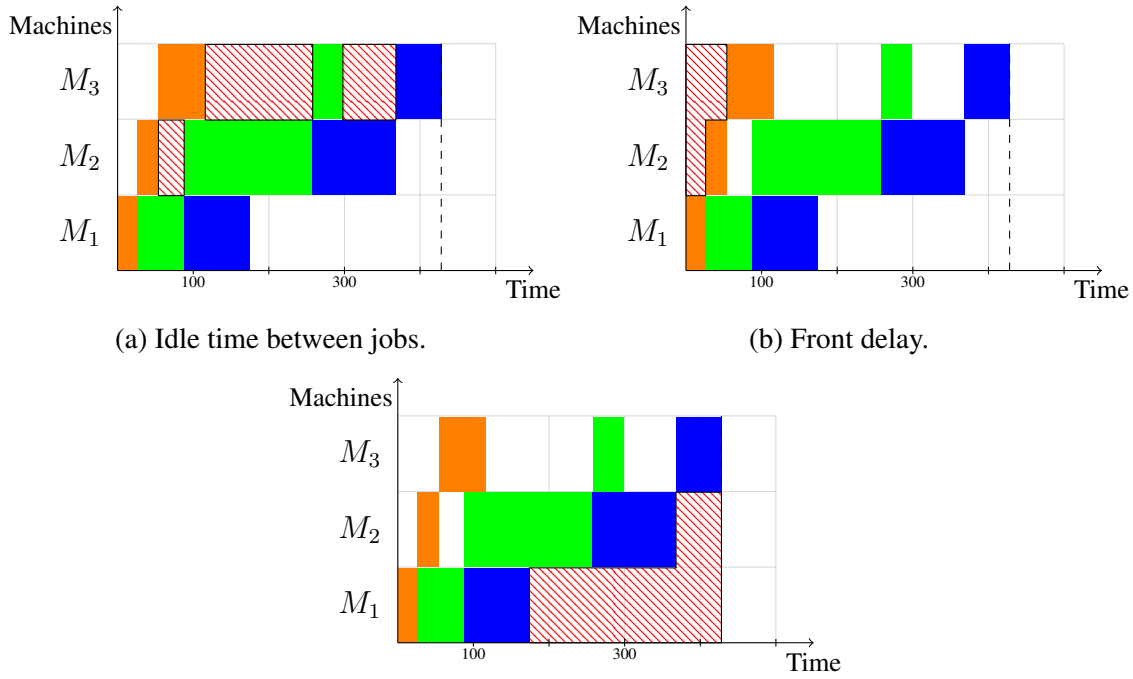
Let $[k]$ denote the job at position k in the schedule, and $C_{i[k]}$ denote the completion time of the job at position k on machine i. The following recurrence defines the completion times:

$$C_{i[k]} = \max\{C_{i-1,[k]}, C_{i,[k-1]}\} + p_{i[k]},\ \forall i = 1, \ldots, m,\ \ \forall k = 1, \ldots, n,$$

with $C_{i0} = 0, \forall i = 1, \ldots, m$ and $C_{0[k]} = 0, \forall k = 1, \ldots, n$. That is, a job is immediately processed on machine i once **(i)** its processing on the previous machine, if any, is finished, and **(ii)** machine i is available. If a machine i is available, but its next job is still being processed by machine $i-1$, then i has to wait and stays idle. Idle time can only occur on machines 2 to m as the first machine always has its next job available immediately. Besides, machines 2 to m have idle time before processing the first job in the schedule as they have to wait for the job to be processed by the other machines. This idle time is referred to as the *front delay*. Similarly, machines 1 to $m-1$ stay idle after processing the last job of the sequence until machine m finishes its last operation. This idle times is referred to as the *back delay*. Figure 2.1 shows an example of a problem instance with three jobs and three machines. The blocks represent the operations of each job on each machine, and the operations of a job share the same color. The hatched areas indicate the idle time between jobs, the front delay, and the back delay.

The most frequent objective functions in the literature consist of minimizing the maximum completion time, or *makespan*, and the total completion time. The makespan

Figure 2.1: Example of idle time, front delay and back delay in a schedule.



(a) Idle time between jobs.

(b) Front delay.

(c) Back delay.

is defined as

$$C_{max} = C_{m[n]},$$

which is the completion time of the last job on the last machine. The total completion time is defined as

$$C_{sum} = \sum_{j=1}^{n} C_{mj},$$

which is the sum of the completion times of all jobs on the last machine. Optimizing the total completion time is equivalent to optimizing the mean completion time, which is the total completion time divided by n. In some variants of flow shops, jobs have a release time r_j. This means that the processing of job j cannot start before time r_j. The *flow time* of j is defined by $f_j = C_{mj} - r_j$. When all jobs are available since the beginning, i.e., $r_j = 0$ for each job j, minimizing total flow time is equivalent to minimizing total completion time.

When introducing the problem, Johnson (1954) showed that the first two machines and the last two machines have the same sequences in optimal solutions when minimizing the makespan. Hence, optimal solutions have the same job order for all machines if $m \leq 3$. In this case, a solution can be represented by a single permutation of jobs. These are known as permutation schedules and are represented with the nota-

tion $\pi = (\pi_1, \ldots, \pi_n)$. When $m > 3$, however, machines can have different sequences in optimal solutions, therefore a solution representation requires multiple permutations. Schedules in which the job sequence is not identical in all machines are known as non-permutation schedules. The permutation flow shop scheduling problem (PFSSP) is a simplification that considers only permutation schedules, regardless of the number of machines. Note, however, that this simplification can exclude optimal solutions. The more general problem, known as the non-permutation flow shop scheduling problem (NPFSSP), allows non-permutation schedules.

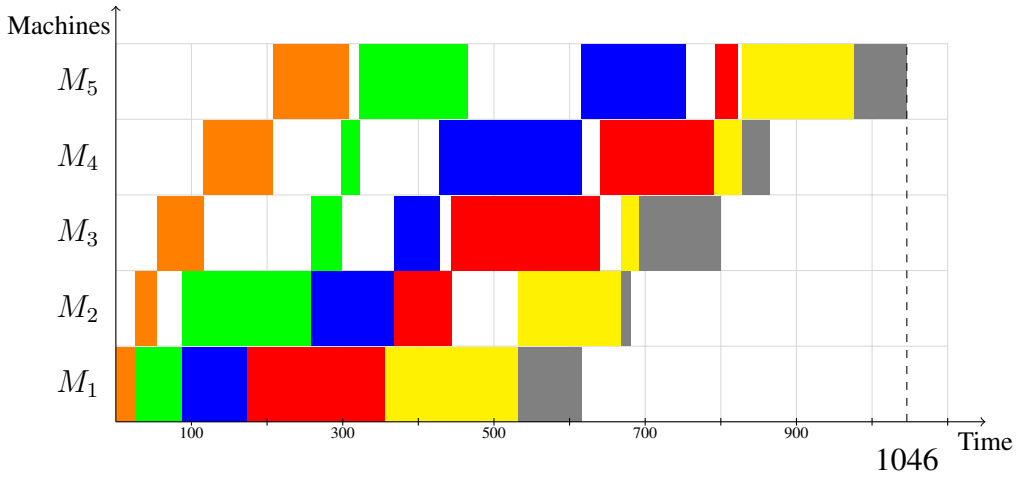
Figure 2.2 shows Gantt charts for two schedules for a problem instance with six jobs and five machines. Operations of a job have the same color. The top one is an optimal permutation schedule with makespan $C_{\max} = 1046$, and the bottom one is an optimal non-permutation schedule with slightly shorter makespan $C_{\max} = 994$. Note that the last two machines have a different sequence than the first three machines on the latter.

Flow shop problems can also be represented by a disjunctive graph, as shown in Figure 2.3, for a problem instance with three jobs and three machines. Let o_{ij} denote the operation of job j on machine i. This graph contains one vertex for each operation o_{ij}, with weight p_{ij}, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. It also contains two artificial nodes "0" and "$*$" with weight zero, respectively connected to the first operation and the last operation of each job. The arcs are classified as conjunctive or disjunctive. The former are directed arcs that define the sequence of operations of each job. The sequence is the same for all jobs in flow shops, i.e., from 1 to m. The latter are undirected arcs that connect vertices of operations that are processed on the same machine. They are the dashed arcs in Figure 2.3. The problem consists of defining a sequence for the operations on each machine through the attribution of a direction to each disjunctive arc. If the resulting graph is acyclic, then it represents a valid schedule. The topological ordering of the graph results in the sequence of operations. The objective is to minimize the longest path from "0" to "$*$". This path is called the *critical path* and its length is equivalent to the makespan of the schedule. A critical path can also be seen as the longest path of consecutive operations (without idle time) from the start to the end of a schedule.

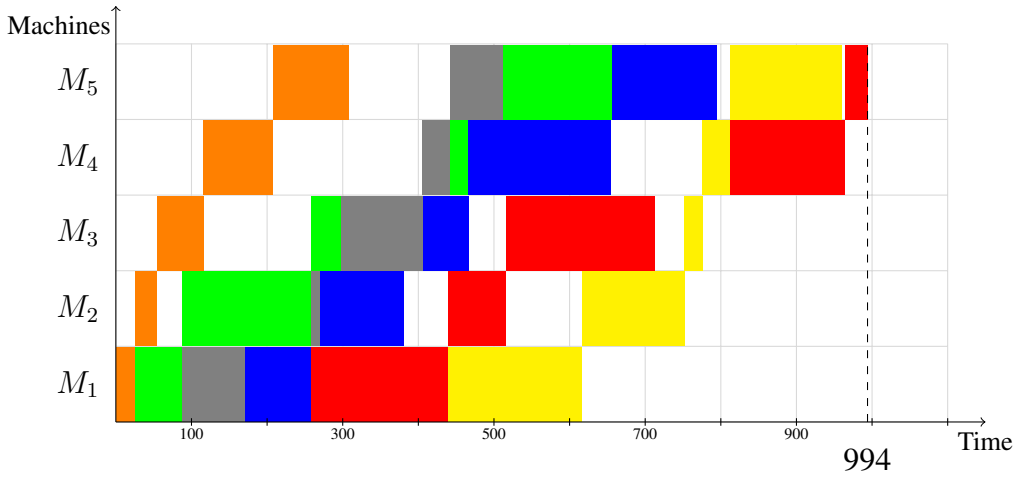
Figure 2.4 presents the optimal sequence for the considered instance, $\pi = (1, 3, 2)$, with makespan $C_{\max} = 251$. A critical path is highlighted in red.

Graham et al. (1979) introduced a three-field notation $\alpha|\beta|\gamma$ to classify scheduling problems. The α field contains information related to the machine environment. For flow shops with m machines, we have $\alpha = Fm$. The β field contains the characteristics

Figure 2.2: Example of a permutation and a non-permutation schedule.



(a) Permutation.



(b) Non-permutation.

of the jobs. Permutation and non-permutation flow shops have $\beta = prmu$ and $\beta = \circ$, respectively. The third field, γ, is the objective function, e.g., $\gamma = C_{\max}$ denotes makespan minimization and $\gamma = \sum C_j$ total completion time minimization.

Despite the simplicity to describe these problems, it has been shown that most flow shop problems are $\mathcal{NP}$-hard. For example, $Fm|prmu|C_{\max}$ is strongly $\mathcal{NP}$-hard for $m \geq 3$ (GAREY; JOHNSON; SETHI, 1976). The same is true for $Fm|prmu|\sum C_j$ with $m \geq 2$ (GAREY; JOHNSON; SETHI, 1976).

In the next subsections, we provide a literature review regarding both exact and heuristic methods for the variants that we address in this work: the PFSSP and the NPFSSP.

Figure 2.3: Example of a disjunctive graph.

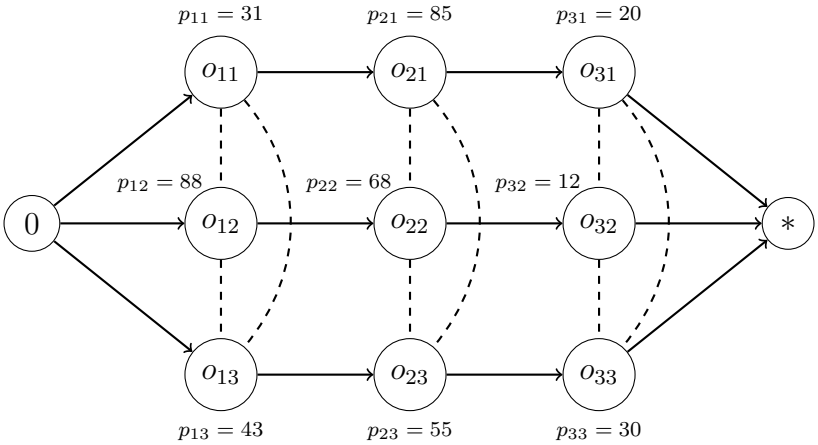
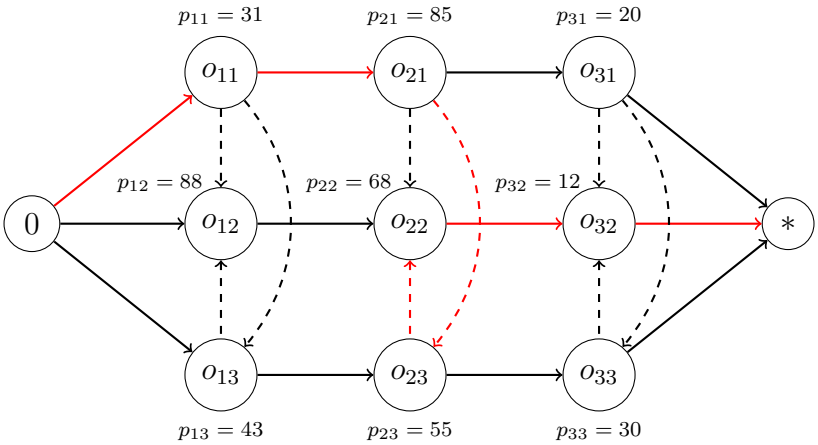


Figure 2.4: Optimal schedule and critical path for the example.



2.1.1 Permutation Flow Shops

One of the most frequently cited reasons to support the adoption of permutation schedules is the vast reduction of the search space from $n!^{\max\{m-2,1\}}$ (when minimizing the makespan, the first two and the last two machines have the same sequence) to $n!$ solutions. However, the size of the search space does not necessarily define how hard the problem is. For example, the assignment problem has a search space with $n!$ solutions, but there are algorithms to solve the problem in polynomial time (KUHN; YAW, 1955), while the PFSSP is $\mathcal{NP}$-hard. Despite that, the PFSSP is the most studied variant in the literature. Due to the vast number of papers addressing the problem (Fernández-Viagas, Ruiz and Framiñan (2017) report more than a hundred new algorithms over the last decade only), we focus on the most notable methods or those that are more related to our work.

2.1.1.1 Exact Methods

There are polynomial-time algorithms for the $F2|prmu|C_{\max}$ in the literature (JOHNSON, 1954), but the general case with m machines is $\mathcal{NP}$-hard. In this case, the problem has been addressed by mathematical programming, through mixed-integer programming (MIP) mathematical models and generic MIP solvers, or with branch-and-bound (B&B) algorithms, which can often outperform the MIP solvers, especially as the problem dimensions increase. In this section, we review two polynomial-time algorithms for two-machine problems and present some of the most common mathematical models and B&B methods for the m-machine problem.

Besides showing that the first two machines and the last two machines have the same sequence in optimal solutions, Johnson (1954) also showed that a job j is scheduled before a job k in optimal solutions if $\min\{p_{1j}, p_{2k}\} \leq \min\{p_{2j}, p_{1k}\}$ in a two-machine problem. This rule is known as Johnson's rule. A procedure to obtain the optimal schedule in polynomial time $O(n \log n)$ based on Johnson's rule can be summarized as follows. Start with an empty schedule and build a list containing p_{ij} for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$ sorted in non-decreasing order. Iterate through the sorted elements, starting from the first one. If $i = 1$, schedule job j at the leftmost available position and remove p_{2j} from the list. Otherwise, schedule j at the rightmost position and remove p_{1j} from the list. Repeat this step until all jobs have been scheduled. The procedure can also be applied to flow shops with three machines, provided that $\min p_{1j} \geq \max p_{2k}$ or $\min p_{3j} \geq \max p_{2k}$.

More recently, Silva (2010) proposed an algorithm for the $F2||C_{\max}$ with lower complexity $\Theta(n \log \kappa)$, where κ is the minimum number of cliques needed to cover a certain interval graph. The number of vertices in the graph is equal to n, therefore $\kappa \leq n$, resulting in an algorithm that is asymptotically faster than the procedure of Johnson (1954).

Regarding the general case with m machines, a few distinct mathematical models can be found in the literature. The most common models for the problem can be divided into the Wagner family and the Manne family. The former started with the integer formulation introduced by Wagner (1959) for the $F3||C_{\max}$, which was based on the classical assignment problem, i.e., it assigns jobs to positions of the schedule. Baker (1974) and Stafford (1988) extended the model to a mixed-integer formulation that supports m-machine permutation flow shops, which was later improved by Stafford and Tseng (2002). Other members of the Wagner family include the formulations of Wilson (1989), modeled around the starting times of job on each machine, and of Stafford, Tseng and Gupta (2005), modeled around the completion times.

Consider the following variables:

$$x_{jk} = \begin{cases} 1, & \text{if job } j \text{ is scheduled at the } k^{\text{th}} \text{ position,} \\ 0, & \text{otherwise.} \end{cases}$$

$P_{ik} =$ processing time of the job at the k^{th} position on machine i.

$I_{ik} =$ idle time on machine i between the completion of $(k-1)^{th}$ job and the start of k^{th} job.

$W_{ik} =$ waiting time of the k^{th} job between the completion on machine i and the start on machine $i+1$.

The Wagner model for the $F||C_{\max}$ is:

$$\text{Minimize} \qquad C_{\max} = \sum_{k\in[n]} I_{mk} + \sum_{k\in[n]} P_{mk} \tag{2.1}$$

subject to

$$\sum_{k\in[n]} x_{jk} = 1, \qquad j = 1,\ldots,n, \tag{2.2}$$

$$\sum_{j\in[n]} x_{jk} = 1, \qquad k = 1,\ldots,n, \tag{2.3}$$

$$P_{ik} = \sum_{j\in[n]} p_{ij}x_{jk}, \qquad i = 1,\ldots,m;\ k = 1,\ldots,n, \tag{2.4}$$

$$I_{i,k+1} + P_{i,k+1} + W_{i,k+1} = W_{ik} + P_{i+1,k} + I_{i+1,k+1}, \qquad i = 1,\ldots,m-1;\ k = 1,\ldots,n-1, \tag{2.5}$$

$$I_{i+1,1} = I_{i1} + W_{i1} + P_{i1}, \qquad i = 1,\ldots,m-1, \tag{2.6}$$

$$W_{i1} = 0, \qquad i = 1,\ldots,m-1, \tag{2.7}$$

$$I_{ik} \geq 0,\ \ x_{jk} \in \{0,1\}, \qquad i = 1,\ldots,m;\ j = 1,\ldots,n;\ k = 1,\ldots,n, \tag{2.8}$$

$$W_{ik} \geq 0, \qquad i = 1,\ldots,m-1;\ k = 1,\ldots,n. \tag{2.9}$$

The objective function (2.1) minimizes the makespan $C_{\max}$, obtained as the total idle time plus the total processing time of machine m. Constraints (2.2) and (2.3) ensure that each job is assigned to one position, and each position is occupied by one job. Constraints (2.4) define the processing times according to the job in each position. Constraints (2.5) and (2.6) ensure that **(i)** the job at $(k+1)^{\text{th}}$ position cannot be processed

on machine i until the previous job on the same machine has been processed, and **(ii)** the k^{th} job cannot be processed on machine $i+1$ until it has been processed on machine i. Constraints (2.7) ensure that the first job of the schedule will be immediately processed on each sucessive machine once it is completed on the current machine. Finally, constraints (2.8) and constraints (2.9) define the domains of the variables.

The model that gave the name to the Manne family is due to Manne (1960) and was initially proposed for the job shop scheduling problem. It was adapted for flow shops by Stafford and Tseng (1990) with a formulation that uses pairs of dichotomous constraints. Liao and You (1992) extend the job shop model of Manne (1960) with a formulation with fewer constraints but more variables, which was adapted to flow shops by Pan (1997). Despite being based on the model of Manne (1960), this formulation did not retain the dichotomous constraints that characterize the Manne family and uses surplus variables instead.

Consider a sufficiently large constant M, e.g., $M = \sum_{i\in[m]} \sum_{j\in[n]} p_{ij}$, and the following variables:

$$y_{jk} = \begin{cases} 1, & \text{if job } j \text{ is scheduled before job } k, \\ 0, & \text{otherwise.} \end{cases}$$

$$C_{ij} = \text{completion time of job } j \text{ on machine } i.$$

Manne's model is as follows.

$$\text{Minimize} \qquad C_{\max} \tag{2.10}$$

subject to:

$$C_{1j} \geq p_{1j}, \qquad j = 1, \ldots, n, \tag{2.11}$$

$$C_{i+1,j} - C_{ij} \geq p_{i+1,j}, \qquad i = 1, \ldots, m-1;\ j = 1, \ldots, n, \tag{2.12}$$

$$C_{ij} - C_{ik} + My_{jk} \geq p_{ij}, \qquad i = 1, \ldots, m;\ \ 1 \leq j < k \leq n, \tag{2.13}$$

$$C_{ik} - C_{ij} + M(1 - y_{jk}) \geq p_{ik}, \qquad i = 1, \ldots, m;\ \ 1 \leq j < k \leq n, \tag{2.14}$$

$$C_{\max} \geq C_{mj}, \qquad j = 1, \ldots, n. \tag{2.15}$$

Constraints (2.11) ensure the correct completion time of each job on the first machine, while constraints (2.12) ensure that a job is processed on the current machine before starting on the subsequent machine. Constraints (2.13) and (2.14) are the dichotomous pairs of constraints that determine the job sequence, allowing a job j to either precede or succeed a job k. Constraints (2.15) define the makespan as the maximum completion time.

Tseng, Stafford and Gupta (2004) presented an empirical analysis of formulations from both families. The computational experiment considered problem instances with $m \in \{5, 7, 9\}$, $n \in \{6, 7, 8, 9\}$ and $p_{ij} \in [1, 100]$, according to a uniform distribution. It was observed that the models from the Wagner family always requires less computational time than Manne models, with an increasing difference as m and n increased. In particular, the model of Wagner was the fastest to find the optimal solution in 52 out of the 60 problems solved.

Although all the cited papers addressed makespan minimization, the formulations can be adapted to total completion time minimization (STAFFORD, 1988; TSENG; STAFFORD, 2008).

Despite the increase of computational power and the improvement of mixed-integer programming solvers in recent years, only small instances are solvable in a reasonable time with mathematical programming formulations. For example, Tseng and Stafford (2008) reported an average of six hours to solve instances with 15 jobs and 10 machines.

Another type of exact approach are branch-and-bound (B&B) methods (LAND; DOIG, 1960), which can often tackle larger instances in less time. A B&B algorithm is an enumerative technique that recursively divides the problem into smaller subproblems according to a so-called branching rule. Bounds are used to avoid an exhaustive enumeration through the elimination of branches with a lower bound that is greater than an upper bound for the optimal solution (assuming a minimization problem). Dominance rules can also be used to further eliminate from consideration branches that are shown to never lead to a solution better than a solution of another branch. The tree is usually explored following a depth-first or a breadth-first strategy.

The first applications of B&B techniques to flow shop problems are dated from the 1960s. Lomnicki (1965) proposed an approach for three-machine flow shops with makespan minimization. At the same time, Ignall and Schrage (1965) presented procedures for makespan minimization in three-machine flow shop and mean completion time in two-machine flow shops. Both Lomnicki (1965) and Ignall and Schrage (1965) use

a branching rule that appends unscheduled jobs at the end of a partial schedule, known as forward branching, and use the same machine-based bound. Methods proposed subsequently included multiple bounds and alternate between forward and backward branching, in which unscheduled jobs are prepended to a partial schedule (POTTS, 1980; CARLIER; REBAÏ, 1996; LADHARI; HAOUARI, 2005).

The state-of-the-art method for the $Fm|prmu|C_{\max}$ is the cyclic best-first search algorithm proposed by Ritt (2016). The algorithm is initialized with a solution found by an iterated greedy (IG) algorithm (RUIZ; STÜTZLE, 2007) that runs for a set amount of time. The branching strategy considers both forward and backward directions, selecting the one with least subproblems, while the search follows a cyclic best-first search (CBFS) strategy (KAO; SEWELL; JACOBSON, 2008). Starting at the first level, CBFS selects the subproblem with the best lower bound. The subproblem is explored until its deepest level, always selecting the node with the smallest lower bound at each level. After that, the search returns to the first level and selects the next subproblem, repeating the whole process. CBFS can be seen as a hybrid that combines breadth-first and depth-first search. The method of Ritt (2016) implements two distinct lower bounds from the literature but no dominance rules. The computational experiments showed that the method could solve instances with 10 machines and up to 200 jobs in about one hour.

As for total completion time minimization, the number of papers about B&B algorithms in the literature is much smaller when compared to makespan minimization. The two-machine problem is addressed by Ignall and Schrage (1965), Croce, Narayan and Tadei (1996) and Croce, Ghirardi and Tadei (2002). Bansal (1977) extended the B&B method of Ignall and Schrage (1965) to the m-machine problem. Chung, Flynn and Kirca (2002) proposed a new machine-based lower bound, which was shown to dominate the lower bound of Bansal (1977), and introduced a dominance rule. Madhushini, Rajendran and Deepa (2009) also addressed the problem with m machines, although focusing on the minimization of weighted total flow time.

2.1.1.2 Heuristic Methods

Since flow shop problems are usually $\mathcal{NP}$-hard, many heuristic methods have been proposed. Framiñan, Gupta and Leisten (2004) proposed a classification for heuristics for the PFSSP according to the strategies used for **(i)** job ordering, **(ii)** solution construction, i.e., how jobs are selected and added to a partial solution, and **(iii)** solution improvement, e.g., with metaheuristics. We review some of the most relevant methods

for makespan and total completion time minimization in the following sections.

Makespan

One of the most known methods is the constructive heuristic NEH of Nawaz, Enscore and Ham (1983). The method has two phases. It starts by sorting the jobs in non-increasing order of their total processing time. This results in a priority order for the next phase, in which the jobs are inserted into a partial schedule one at a time at the position that minimizes the makespan. A solution is obtained when all the jobs have been inserted. The method generates schedules with an average relative deviation of about 3% from the optimal (VASILJEVIC; DANILOVIC, 2015). A naive implementation of NEH has polynomial-time complexity of $O(n^3m)$. However, Taillard (1990) introduced an acceleration procedure, to reduce that complexity. Consider the insertion of job l into a partial schedule with k jobs $\pi = \pi_1 \dots \pi_k$, and the following values:

- $e_{i,j}$, the earliest completion time (head) of job π_j on machine i, defined as:

$$e_{i,j} = \max\{e_{i-1,j}, e_{i,j-1}\} + p_{i,\pi_j}, \quad \forall i = 1, \dots, m; \ \forall j = 1, \dots, k, \tag{2.16}$$

  with $e_{0,j} = e_{i,0} = 0$. Figure 2.5a shows the head of the fourth job on the second machine for a given schedule.

- $q_{i,j}$, the time between the start of the processing of job π_j on machine i and the end of the processing of the last job of the schedule on the last machine (tail), defined as:

$$q_{i,j} = \max\{q_{i+1,j}, q_{i,j+1}\} + p_{i,\pi_j}, \quad \forall i = 1, \dots, m; \ \forall j = 1, \dots, k, \tag{2.17}$$

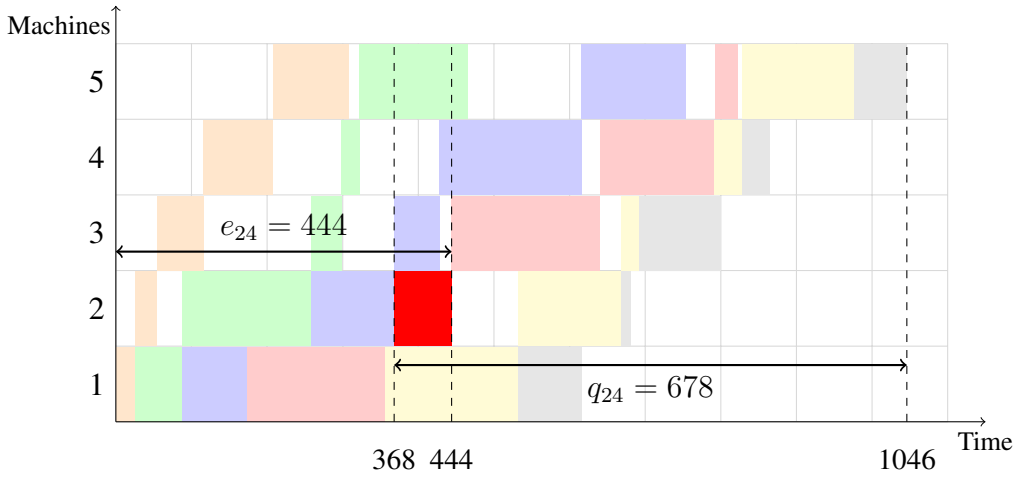
  with $q_{m+1,j} = e_{i,k+1} = 0$. Figure 2.5a shows the tail of the fourth job on the second machine for a given schedule.

- $f_{i,j}$, the earliest relative completion time on machine i of job π_j after the insertion of job l into some position, defined as:

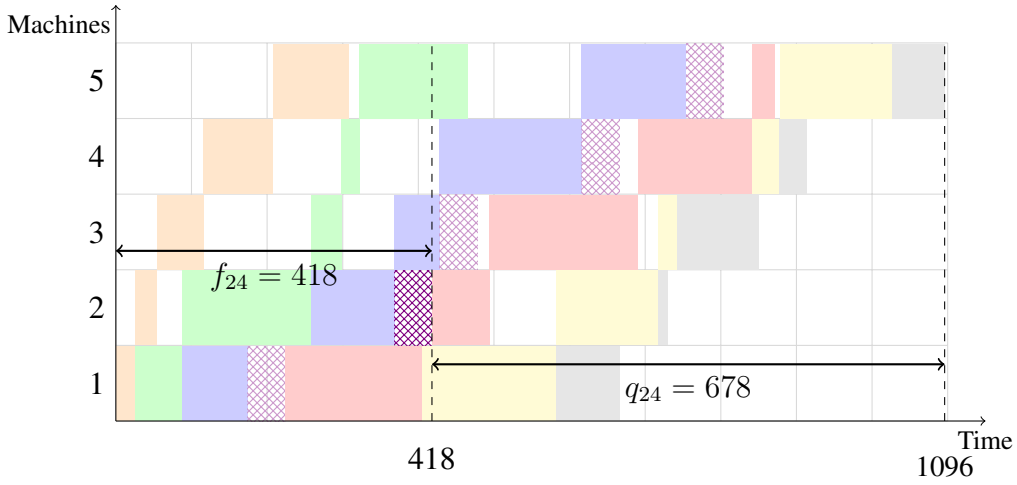
$$f_{i,j} = \max\{f_{i-1,j}, e_{i,j-1}\} + p_{i,l}, \quad \forall i = 1, \dots, m; \ \forall j = 1, \dots, k+1,$$

  with $f_{0,j} = e_{i,0} = 0$. Figure 2.5b shows the earliest relative completion time on the second machine of a new job inserted into the fourth position.

Figure 2.5: Taillard's accelerations.



(a) Head and tail.



(b) Earliest relative completion time.

The makespan M_j after the insertion of job l at position j is given by:

$$M_j = \max_{i \in [m]} \{f_{ij} + q_{ij}\}.$$

With these values, it is possible to evaluate the insertion of job l into all possible positions in time $O(nm)$. As this is repeated for every job, we have an overall complexity of $O(n^2m)$. This complexity makes NEH one of the most efficient methods for the PFSSP.

Many authors have proposed improvements for NEH, such as new priority orders for the first phase or new tie-breaking rules. Framiñan, Leisten and Rajendran (2003) tested 177 priority orders and concluded that the standard strategy is the best one when minimizing the makespan. Kalczynski and Kamburowski (2008) present a new order

based on two indexes:

$$a_j = \left[\binom{m-1}{2} + m\right] P_j - \sum_{i\in[m]} ip_{ij},$$
$$b_j = \left[\binom{m-1}{2} - 1\right] P_j + \sum_{i\in[m]} ip_{ij},$$

where P_j is the total processing time of job j. The jobs are sorted in non-increasing order according to $\min\{a_j, b_j\}$.

Kalczynski and Kamburowski (2009) present another priority order based on two different indexes:

$$a_j = T_j + U_j,$$
$$b_j = T_j - U_j,$$

where

$$T_j = \sum_{i=1}^{m} p_{ij},$$
$$U_j = \sum_{h=1}^{s} \left(\frac{h-3/4}{s-3/4} - \varepsilon\right)\left(p_{s+1-h,j} - p_{t+h,j}\right),$$
$$s = \lfloor m/2 \rfloor,$$
$$t = \lceil m/2 \rceil,$$

and ε is a small positive real number. Again, the jobs are sorted in non-increasing order according to $\min\{a_j, b_j\}$. The computational experiments showed that both priority orders improved over the one in the original NEH, while the resulting procedures, named NEHKK1 and NEHKK2, retain the same overall complexity.

Furthermore, in NEH and other insertion-based methods, we often observe ties when looking for a job's best insertion position. Several tie-breaking rules to decide among the multiple positions can be found in the literature. Kalczynski and Kamburowski (2008) and Kalczynski and Kamburowski (2009) use a_j and b_j indexes to decide ties, selecting the position of smallest index if $a_j \leq b_j$, or the position of biggest index otherwise.

Dong, Huang and Chen (2008) proposed a priority order called AvgDev that consists of sorting the jobs in non-decreasing order according to the sum of their average

processing time over the m machines, and their standard deviation. More formally:

$$\text{AvgDev}_j = \overline{p}_j + \sigma_j,$$

where $\overline{p_j}$ and σ are the mean and the standard deviation of the processing times of job j over the m machines. Dong, Huang and Chen (2008) also proposed a tie-breaking rule based on a measure D_{π_j} for each job j, defined as follows:

$$D_{\pi_j} = \sum_{i=1}^{m} \left(\frac{p_{i,\pi_j}}{q_{i,\pi_{j+1}} - e_{i,\pi_{j-1}}} - E_{\pi_j} \right)^2,$$
$$E_{\pi_j} = \frac{1}{m} \sum_{i=1}^{m} \frac{p_{i,\pi_j}}{q_{i,\pi_{j+1}} - e_{i,\pi_{j-1}}},$$

where π_j denotes the job scheduled at the j^{th} position, and $e_{i,j}$ and $q_{i,j}$ are defined as in Eqns. (2.16) and (2.17). If there is a tie, the position that minimizes D_{π_j} is chosen. The procedure adopting both this tie-breaking rule and the AvgDev order is named NEH-D and has the same complexity as NEH. The reported results show improvements of about 0.4% over NEH, and 0.2% over NEHKK1.

More recently, Fernández-Viagas and Framiñan (2014) presented a rule that focuses on minimizing the front delay of each machine, using estimations computed taking advantage of the Taillard's accelerations. The method, named NEHFF, uses the priority order of Dong, Huang and Chen (2008), and also the same complexity as NEH. NEHFF yields a small improvement of about 0.05% over NEH-D on average.

Lastly, Liu, Jin and Price (2017) introduced a priority rule in which jobs are sorted in non-increasing order of the sum $\text{AvgDev}_j + |\text{SKE}_j|$, where AvgDev_j is as defined in Dong, Huang and Chen (2008) and SKE_j is the skewness of job j. This priority order was coupled with a tie-breaking rule that aims to minimize the front delay and the partial idle time before the insertion position. The reported results showed an average improvement of 0.15% over NEHFF.

Moreover, Vasiljevic and Danilovic (2015) presented a study on the NEH heuristic and several tie-breaking rules found in the literature and concluded that repeatedly executing NEH with random tie-breaking for the priority order and the insertion phase yields results that are competitive with those obtained using the best tie-breaking rules. The proposed procedure was named NEHI, and the computational experiments showed an improvement of about 0.45% over NEHFF. To the best of our knowledge, this is the

best NEH variant regarding solution quality with complexity $O(n^2m)$ in the literature. However, it uses more time, as it performs NEH several times.

Rad, Ruiz and Boroojerdian (2009) present five new constructive heuristics that generate better solutions when compared to NEH, but at a higher computational cost. The method with the best results, FRB5, can be seen as an extension to NEH, in which a local search phase is added after each job insertion. The main steps are the same as in NEH, i.e., the jobs are sorted in non-increasing order of their total processing time, and added one at a time to a partial schedule. However, after each addition, a local search with an insertion neighborhood is performed. In this local search, each one of the jobs on the partial schedule is removed and reinserted into the position that minimizes the makespan. This is repeated until a local optimum is found.

Rossi, Nagano and Tavares Neto (2016) proposed new constructive heuristics that work similarly to FRB5. The difference lies in the local search phase: the methods of Rossi, Nagano and Tavares Neto (2016) reinsert pairs of adjacent jobs, while FRB5 reinserts single jobs only. The heuristic with the best performance is called G8, and evaluates the reinsertion of jobs at positions j and $j+1$, with $j = 1, 3, 5, \ldots, n-1$, and then repeats the same process with $j = 2, 4, 6, \ldots, n-2$. As opposed to FRB5, the local search is not repeated until a local optimum is found, possibly resulting in a shorter computational time at the expense of solution quality. G8 was tested with several priority orders and tie-breaking rules from the literature, and the results showed that the strategies of Kalczynski and Kamburowski (2009) yield the best solution quality on average. In general, G8 is competitive with other methods in the literature.

Fernández-Viagas, Ruiz and Framiñan (2017) compared 19 constructive heuristics from the literature, showing that FRB5 yields the best solution quality, but also requires the highest computational effort. In a comparison using a measure that also takes the computational time in consideration, NEHKK2 and NEHFF were the best-ranked methods.

Regarding metaheuristics, flow shop problems have been addressed with a variety of methods, such as simulated annealing (SA) by Osman and Potts (1989), Wodecki and Bożzejko (2002) and Low, Yeh and Huang (2004), tabu search by Taillard (1990), Reeves (1993), Nowicki and Smutnicki (1996b) and Grabowski and Wodecki (2004), and genetic algorithms by Chen, Vempati and Aljaber (1995), Murata, Ishibuchi and Tanaka (1996) and Ruiz, Maroto and Alcaraz (2006), just to cite a few. Many metaheuristics start from a solution obtained by the NEH heuristic and perform local searches with an insertion neighborhood. This is mainly because the accelerations can also be applied to find the

Algorithm 1 Iterative improvement insertion local search

Input: Solution π
Output: Best solution found π^*

```
1:  function INSERTION(π)
2:      repeat
3:          π' = π
4:          for k = 1 to n do
5:              j = k^th job in π'
6:              π' = REMOVE(j,π')
7:              π' = INSERTBESTPOSITION(j,π')
8:              if f(π') < f(π) then
9:                  π = π'
10:             end if
11:         end for
12:     until no improvement is found
13:     return π*
14: end function
```

best insertion position in such local searches at a reduced computational cost, resulting in a significantly increased performance. A common strategy is to remove each job and reinsert it into the position that minimizes the objective function value, repeating this process until a local optimum is found. Pseudocode for an iterative improvement insertion local search is presented in Algorithm 1. We consider that the best solution π^* is implicitly maintained in all algorithms presented in this thesis.

Another neighborhood used in methods for flow shops is the one proposed by Nowicki and Smutnicki (1996a). This is a reduced neighborhood based on blocks of jobs on the critical path. A block is a sequence of consecutive jobs on the same machine on the critical path. The neighborhood, hereafter referred to as NS, swaps the first two jobs or the last two jobs of each block, except the first two jobs of the first block and the last two jobs of the last block of the schedule. This strategy is used to discard swaps that will not immediately improve the makespan.

Fernández-Viagas, Ruiz and Framiñan (2017) review the state-of-the-art metaheuristics. Out of dozens, the 12 most promising methods were reimplemented and evaluated under the same conditions. The results showed that the best metaheuristics are based on the IG algorithm of Ruiz and Stützle (2007), IG$_{rs}$, one of the most prominent metaheuristics in the literature.

IG$_{rs}$ is an algorithm that repeatedly applies a two-phase procedure. In the first phase, called destruction, d randomly selected jobs are removed from the current schedule. During the second phase, called construction, the removed jobs are reinserted greed-

Algorithm 2 Metropolis acceptance criterion

Input: Current solution π, new solution π', temperature T
Output: Solution π

```
1: function METROPOLIS(π, π', T)
2:     r = RANDOM(0, 1)                          ▷ random number in [0,1)
3:     Δ = f(π') − f(π)
4:     if Δ ≤ 0 or r ≤ e^{−Δ/T} then
5:         π = π'
6:     end if
7:     return π
8: end function
```

ily into the schedule. Additionally, a local search procedure can be applied to optimize the resulting solution. After that, the new solution is accepted according to certain a criterion, and the whole process is repeated until a termination criterion is met. In particular, Ruiz and Stützle (2007) use the insertion local search presented in Algorithm 1, and the Metropolis acceptance criterion (METROPOLIS et al., 1953) with a constant temperature T, i.e., a solution is always accepted if it is better than the current one. Otherwise, it is accepted with a probability that decreases as the difference in the objective function value increases, as shown in Algorithm 2. The temperature value is defined by Ruiz and Stützle (2007) as:

$$T = \alpha \sum_{j \in [n]} P_j / 10nm,$$

where α is a parameter.

IG$_{rs}$ is presented in Algorithm 3, where $f(\pi)$ denotes the objective function value of solution π. We consider that the best solution visited so far π^* is implicitly maintained in all algorithms presented in this thesis. Note that the underlying idea is similar to an iterated local search (ILS) (LOURENÇO; MARTIN; STÜTZLE, 2003).

An ILS method repeatedly performs a local search, followed by a perturbation to escape from the current local optimum. The perturbation is done through a certain number of random movements, e.g., reinserting a job at a random position, or swapping a pair of randomly selected jobs. A solution is accepted according to a certain criterion. Pseudocode for an ILS method is presented in Algorithm 4.

The IG of Ruiz and Stützle (2007) is a particular ILS variant in which the perturbation is performed by a randomized greedy procedure, represented by "Destruct" and "Construct" in lines 5 and 6 of Algorithm 3, and the acceptance of solutions follows a

Algorithm 3 IG_{rs}

Input: Perturbation intensity d
Output: Best solution found π^*
1: **function** $\text{IG}_{rs}(d)$
2: π = INITIALSOLUTION
3: π = LOCALSEARCH(π)
4: **while** termination criterion not met **do**
5: π' = DESTRUCT(π, d)
6: π' = CONSTRUCT(π')
7: π' = LOCALSEARCH(π')
8: **if** ACCEPT(π, π') **then**
9: $\pi = \pi'$
10: **end if**
11: **end while**
12: **return** π^*
13: **end function**

Algorithm 4 ILS

Input: Perturbation intensity d
Output: Best solution found π^*
1: **function** $\text{ILS}(d)$
2: π = INITIALSOLUTION
3: π = LOCALSEARCH(π)
4: **while** termination criterion not met **do**
5: π' = PERTURBATION(π, d)
6: π' = LOCALSEARCH(π')
7: **if** ACCEPT(π, π') **then**
8: $\pi = \pi'$
9: **end if**
10: **end while**
11: **return** π^*
12: **end function**

Metropolis criterion.

Apart from providing solutions of high quality, this IG has several advantages. In particular, it is simple to implement and can be easily adapted to other flow shop variants. This has motivated a noticeable adoption of IG-based algorithms in the literature, e.g., by Pan and Ruiz (2012), Fernández-Viagas and Framiñan (2014), Benavides and Ritt (2016) and Fernández-Viagas, Valente and Framiñan (2018).

Dubois-Lacoste, Pagnozzi and Stützle (2017) proposed the addition of an iterative improvement insertion local search to optimize the partial schedule that is obtained after the destruction phase in IG_{rs}. It results in an average improvement of about $0.02\,\%$ for the Taillard benchmark, and about $0.33\,\%$ for the VRF benchmark. These benchmarks are

presented in Section 2.1.3.

Benavides and Ritt (2018) present two new constructive heuristics that extend NEH with the addition of local searches that reinsert pairs of consecutive jobs after each job is added to the partial schedule. The first constructive heuristic uses a local search that considers the full neighborhood, while the second considers a reduced neighborhood based on critical paths. Similarly, two new local search methods that reinsert pairs of consecutive jobs, one with the full neighborhood and the other with the reduced neighborhood, are proposed. A total of 21 IG algorithms combining the newly proposed methods with constructive heuristics and local search procedures from the literature are evaluated. The best algorithm uses the local search with the reduced neighborhood and the constructive heuristic with the full neighborhood.

Lastly, Fernández-Viagas and Framiñan (2019) proposed an IG that combines the constructive heuristic with the full neighborhood of Benavides and Ritt (2018), the local search applied to the partial solution after the destruction phase proposed by Dubois-Lacoste, Pagnozzi and Stützle (2017), the local search procedure with the reduced neighborhood of Benavides and Ritt (2018), and the tiebreaker of Fernández-Viagas and Framiñan (2014). The resulting method yielded the lowest overall ARD compared to the original methods on both Taillard and VRF benchmarks. To the best of our knowledge, this is the state-of-the-art heuristic for the $Fm|prmu|C_{\max}$.

Total Completion Time

Some of the best constructive heuristics for total completion minimization originated from the makespan literature, such as NEH and FRB5. Others were explicitly designed for this objective, such as LR (LIU; REEVES, 2001) and BSCH (FERNÁNDEZ-VIAGAS; FRAMIÑAN, 2017). The adaptation of NEH and FRB5 consists of replacing the non-increasing order of total processing time with a non-decreasing order. During the subsequent steps, the jobs are inserted into the positions that minimize the total completion time. The same is true for the insertion local search in FRB5. However, without Taillard's accelerations, which are specific to compute the makespan, evaluating the possible insertion positions for a job takes $O(n^2m)$ steps, leading to an overall complexity of $O(n^3m)$ for NEH. Li, Wang and Wu (2009) proposed an acceleration strategy that consists in reducing the number of updates regarding job completion times when the schedule changes. The strategy is based on the observation that, when inserting a job into a position k, the schedule remains unchanged from positions 1 to $(k-1)$. Therefore it is not

necessary to recompute the completion time of the jobs in those positions. Duan et al. (2013) further observed that the completion times of operations on a critical path depend only on other operations also on the critical path. Therefore only operations on the highest critical path and above have to be recomputed. Duan et al. (2013) reported that the average computational time of NEH was reduced by 30% when using both strategies.

The LR constructive heuristic starts by ordering the jobs according to a measure that considers both the weighted idle time induced when appending a job j at the end of a schedule and an estimation for the completion time of jobs to be appended after j. Then, x schedules are created, each starting with one of the first x jobs of this order, and the remaining jobs are appended one by one using the same measure. The final schedule is the best one out of the x generated schedules. The method has an overall complexity of $O(xn^2m)$. Framiñan, Leisten and Ruiz-Usano (2005) compared heuristics from the literature and combined them to generate composite methods that outperformed the former. Pan and Ruiz (2013) reviewed simple and composite constructive heuristics and proposed the LR-NEH method, which inserts the first d jobs into the schedule according to LR, and the remaining $n - d$ jobs according to NEH. Four other composite heuristics with higher complexity were also proposed. They could increase the solution quality at an increased computational effort.

Fernández-Viagas and Framiñan (2015) proposed a modified measure for LR, intended for reducing the method's complexity to $O(n^2m)$. The resulting procedure was named FF, and the computational experiments showed that it could replace LR in the composite heuristics presented by Pan and Ruiz (2013) for increased performance.

Finally, BSCH is a more recent beam search heuristic. Also similar to LR, the method appends jobs at the end of w partial sequences. The underlying idea of beam search (LOWERRE, 1976) is similar to a branch-and-bound algorithm, but only a reduced number of nodes is kept at each level. This number is defined by a parameter commonly called the beam width (w). In each level and for each node, a job is appended at the end of the partial sequence. To select such jobs, a forecast index that considers not only the current partial schedule but also the unassigned jobs is used. After evaluating all the candidates, only the w best nodes are kept, and this is repeated until the nodes are complete schedules, in which case the best one is selected. The computational experiments showed that BSCH significantly improves over the other constructive heuristics in the literature.

Regarding local search procedures, Rajendran and Ziegler (1997) proposed an approach with an insertion neighborhood. Starting with the first job of a given schedule,

each job is removed and reinserted into the position that minimizes the total completion time. Improvements are immediately accepted, and the search is resumed with the updated schedule.

Liu and Reeves (2001) introduced two swap-based local search procedures named FPE and BPE. FPE stands for forward pairwise exchange, and the procedure consists in evaluating the exchange of each job with all those scheduled after it. An exchange is immediately performed if it leads to an improvement. This is repeated until a local optimum is obtained. BPE stands for backward pairwise exchange and works similarly, but in a reversed direction, i.e., it evaluates swapping of a job with those preceding it in the schedule.

Jarboui, Eddaly and Siarry (2009) proposed an insertion-based and a swap-based procedure. The former is similar to the one of Rajendran and Ziegler (1997), and the latter swaps every pair of jobs, immediately accepting improvements, and iterating until a local optimum is found.

Tasgetiren et al. (2011) also adopts a swap-based and an insertion-based local search. Both are similar to the procedures of Jarboui, Eddaly and Siarry (2009), except that they restart the search from the beginning of the schedule when improvements are found. Additionally, a procedure that repeatedly applies the insertion local search followed by the swap local search is presented. Similarly, this is repeated until no improvements can be found. A fourth local search proposed by Tasgetiren et al. (2011) cyclically performs reinsertions, stopping when no improvements are found after n consecutive tries.

More recently, Benavides and Ritt (2015) introduce a swap-based based local search which exchanges the jobs in positions p and $p + q$, immediately accepting improving swaps. If none of the swaps improves the current permutation, q is incremented, and the search is restarted. Otherwise, when an improvement is found, q is set to a value of q_{min}. The search stops when $q > q_{max}$ or a total number of swaps s_{max} is performed.

Regarding metaheuristics, many approaches have been applied, such as ant colony optimization (RAJENDRAN; ZIEGLER, 2004; RAJENDRAN; ZIEGLER, 2005), discrete differential evolution (PAN; TASGETIREN; LIANG, 2008), genetic algorithms (TSENG; LIN, 2009; ZHANG; LI; WANG, 2009; TSENG; LIN, 2010), discrete artificial bee colony (TASGETIREN et al., 2011), IG (PAN; RUIZ, 2012) and ILS (PAN; RUIZ, 2012; DONG et al., 2013; BENAVIDES; RITT, 2015).

Similar to what is observed for makespan minimization, the current best-performing methods include IG and ILS. Pan and Ruiz (2012) proposed an ILS and an IG algorithm

for total completion time minimization. Both methods are initialized with LR and repeatedly perform the insertion local search of Rajendran and Ziegler (1997) until a local optimum is found. The difference between the proposed IG and ILS algorithms is the perturbation procedure, which reinserts the removed jobs greedily in the former, and randomly in the latter. The computational experiments showed that both methods outperformed the other existing approaches, with a marginal advantage in favor of the IG. Population-based variants for the IG and ILS were also introduced. However, they were outperformed by the standard methods. Benavides and Ritt (2015) further improved the IG by using a local search that alternates between an insertion and a swap neighborhood.

Dong et al. (2013) introduced an ILS with a pool of solutions, called MRSILS. The pool maintains the best solutions visited so far and is used to restart the search when no improvements can be found for a certain number of consecutive iterations. Fernández-Viagas and Framiñan (2017) used BSCH to initialize MRSILS and improved the state of the art. To the best of our knowledge, this is the best performing method in the literature.

2.1.2 Non-permutation Flow Shops

Potts, Shmoys and Williamson (1991) showed that optimal permutation schedules can be worse than optimal non-permutation schedules by a factor $\Omega(\min\{\sqrt{m}, \sqrt{n}\})$ for a certain family of instances, and Nagarajan and Sviridenko (2009) later proved the bound to be $\Theta(\min\{\sqrt{m}, \sqrt{n}\})$. Despite that, the majority of published works nowadays still addresses only the PFSSP. Rossit, Tohmé and Frutos (2018) presented a literature review in which 72 papers addressing the NPFSSP were identified, constrasting with the several hundreds of papers concerning the permutational variant (FERNÁNDEZ-VIAGAS; RUIZ; FRAMIÑAN, 2017). About 65% of the papers on the NPFSSP were published after 2006, showing that the interest in this variant has grown recently.

The following subsections give an overview on the most important exact and heuristic methods for makespan and total completion time in the literature.

2.1.2.1 Exact Methods

Most of the literature on the NPFSSP is about heuristics. The exact approaches are limited to some of the mathematical models for the PFSSP, such as those of Wagner (1959), Wilson (1989), and Manne (1960), which can be adapted for non-permutation

flow shops, as shown by Pan (1997). Methods for the job shop scheduling problem (JSSP) can be used since the NPFSSP is a particular case of the JSSP in which all jobs go through the same machine sequence. However, to the best of our knowledge, these are the only works addressing the NPFSSP in specific.

2.1.2.2 Heuristic Methods

In this section we present the most important heuristic methods for the $F||C_{\max}$ and $F||\sum C_j$.

Makespan

Tandon, Cummings and LeVan (1991) performed computational experiments to compare permutation and non-permutation schedules. The solutions were obtained by an enumerative search algorithm and a simulated annealing method, which initially explores permutation schedules and transitions to non-permutation schedules when a certain temperature is reached. It was observed that non-permutation solutions have a considerably shorter makespan. The difference increases with the range of the processing times and the dimensions of the instance.

Koulamas (1998) proposed a two-phase constructive heuristic. The first phase generates a permutation schedule, which is used as input for the second phase. The second phase evaluates job passing by swapping pairs of adjacent jobs on some of the machines, generating non-permutation schedules. The method has an overall complexity of $O(n^2m^2)$ and was able to build solutions with a shorter makespan (about $2.5\,\%$ on average) than the permutation solutions obtained by NEH.

Jain and Meeran (2002) proposed a framework that aims to balance intensification and diversification. The primary intensification mechanism is based on tabu search with a reduced neighborhood proposed by Nowicki and Smutnicki (1996a) for the job shop scheduling problem, and scatter search and path relinking techniques perform the diversification. The method improved solution quality by about 2% when compared to the tabu search of Nowicki and Smutnicki (1996a).

Liu and Ong (2002) compared tabu search, simulated annealing, and threshold acceptance approaches applied to both the PFSSP and the NPFSSP. The methods for the PFSSP use an insertion neighborhood, while the methods for the NPFSSP use the reduced neighborhood of Nowicki and Smutnicki (1996a). Both methods run for the same number

of iterations, but the PFSSP took 25% longer to finish on average since the insertion neighborhood requires higher computational effort. The results showed that the methods for the PFSSP obtained solutions with shorter makespan by about 2% on average.

Liao, Liao and Tseng (2006) proposed a tabu search with an insertion local search and a dynamic tabu duration, and compare it on six objective functions to a version of the genetic algorithm of Reeves (1995) adapted to the NPFSSP. Both methods run for a certain time in order to find a good permutation schedule. In a second phase, some machines have their sequences fixed, and non-permutation schedules are evaluated through changes in the sequences of the remaining machines. The experiments considered instances with 10 machines, and 20 and 50 jobs. Regarding makespan minimization, the tabu search was superior in solving the smaller instances, while the genetic algorithm had better performance on the larger ones. Moreover, out of the different objectives evaluated, the makespan of non-permutation schedules improved the least over permutation schedules, with $0.09\,\%$ on average. The average improvement when minimizing total completion time was 0.25%.

Haq et al. (2007) introduced a scatter search with an insertion neighborhood and a diversification mechanism to avoid duplicate solutions. The method was able to reduce the makespan by about 5.5% and 1% on average when compared to the tabu search methods of Nowicki and Smutnicki (1996a) and Jain and Meeran (2002) on the benchmark of Demirkol, Mehta and Uzsoy (1998).

Ying and Lin (2007) address the NPFSSP with a multi-heuristic desirability ant colony system (MHD-ACS). The method works over the disjunctive graph representation of the problem. The ants attempt to find the shortest path that includes all vertices, guided by a heuristic desirability measure and the pheromone quantity. The computational experiments showed that the proposed method was able to obtain new best-known values for 32 out of the 40 instances of the benchmark of Demirkol, Mehta and Uzsoy (1998).

Ying (2007) proposed an IG to address a simplification of the NPFSSP, based on the observation of Conway, Maxwell and Miller (1967) that the first two machines and the last two machines have the same job sequence on optimal solutions when minimizing the makespan. The simplification consists in representing a schedule with three permutations. The first permutation defines the job sequence for the first and second machine, the second permutation for all machines except the first two and the last two, and the third permutation for the last two machines. This effectively reduces the search space from $n!^{\max\{m-2,1\}}$ to $n!^3$ solutions, but may prevent some candidates or even optimal solutions

from being found. The computational experiments showed an average improvement of almost 9% over MHD-ACS, although using a faster computer and different time limits.

Furthermore, Lin and Ying (2009) introduce a hybrid approach that combines simulated annealing and tabu search. The same simplification with three permutations is used. The results showed that the approach could outperform MHD-ACS, with an average improvement of about 8% in solution quality and using lower computational time.

Sadjadi, Bouquard and Ziaee (2008) used an ant colony optimization (ACO) algorithm. It is initialized with NEH and builds a permutation schedule, which is then improved with a local search that evaluates non-permutation schedules and runs for 10 seconds. This local search performs a pairwise exchange of jobs on the k first or the k last machines, with a maximum distance between the pair of jobs set to two. The computational experiments showed that the average makespan of the permutation schedules is about 0.35% better than the best ACO algorithm of Rajendran and Ziegler (2004) for the PFSSP, named PACO. The experiments with non-permutation schedules showed that the proposed local search could yield an average improvement of 0.12%.

Gharbi, Labidi and Louly (2014) presented new procedures to generate lower and upper bounds for the NPFSSP. The new lower bound procedure improved the typically used one-machine lower bound (ADAMS; BALAS; ZAWACK, 1988) in 14 out of 24 tested cases. However, it always required more time to be computed, sometimes up to three orders of magnitude. Regarding upper bounds, five heuristics named H1 to H5 were introduced. The average gaps to the proposed lower bound are $3.18\%, 3.26\%, 2.51\%, 3.33\%$, and 2.54%. Furthermore, there was a clear trade-off between quality and computational time, with H5 offering the best balance.

Rossi and Lanzetta (2014) proposed a non-permutation ant colony optimization algorithm. Instead of building an initial permutation and then evaluating job passing, the algorithm directly explores the non-permutation solution space. The method was compared to MHD-ACS of Ying and Lin (2007), showing an average improvement of about 3% in solution quality, but also running for almost five times longer on average.

Benavides and Ritt (2016) took into consideration that optimal non-permutation schedules often require only a few inversions of operations over a permutation schedule, and proposed a constructive heuristic and an IG algorithm to minimize the makespan on the NPFSSP. The constructive heuristic inserts jobs into a partial schedule considering anticipation or delay of operations after a certain machine. At the same time, the IG algorithm uses a local search also based on this idea, allowing job passing. In both cases, an

acceleration method that is a generalization of Taillard's procedure (TAILLARD, 1990) is used, reducing the computational complexity to find the best insertion position. The reported results show an average improvement of 0.75% over permutation schedules, which is substantial compared to the progress observed in the literature in recent years.

Benavides and Ritt (2018) presented new constructive heuristics and local search procedures that explore non-permutation schedules. A total of 28 IG algorithms combining the newly proposed methods is evaluated. The best method combines a constructive heuristic that extends NEH by adding a local search that reinserts pairs of consecutive jobs and a local search that swaps adjacent jobs at the beginning or at the end of blocks of operations on a critical path. The method reduces the overall average relative deviation by about 0.3% compared to the method of Benavides and Ritt (2016). To the best of our knowledge, this is the state-of-the-art method for makespan minimization in the literature.

Total Completion Time

Liao, Liao and Tseng (2006) compared the tabu search and the genetic algorithm of Reeves (1995) mentioned in the previous subsection under total completion time minimization. The tabu search had substantially better results, and the average improvement with non-permutation schedules was 0.25%.

Sadjadi, Bouquard and Ziaee (2008) applied an ACO algorithm to minimize total completion time. The same procedure used for makespan minimization is applied, i.e., an initial permutation is constructed with NEH, improved with the ACO approach, and a local search that explores non-permutation solutions is applied as the last step. Regarding permutation schedules, the approach yielded an average improvement of approximately 0.12% over the PACO method of Rajendran and Ziegler (2004), and the non-permutation schedules had an average improvement of about 0.8% over permutation schedules.

Benavides and Ritt (2015) proposed a two-phase IG. First, an ILS for the PFSSP is used to obtain a good initial permutation schedule. This ILS is initialized with a solution obtained by the constructive heuristic LR, and alternates between an insertion and a swap neighborhood. The second phase explores non-permutation schedules by iteratively removing random jobs and reinserting them, allowing delayed or anticipated operations on some machines. The computational experiments showed an average improvement close to 0.45% when allowing non-permutation schedules. To the best of our knowledge, this is the state-of-the-art method for total completion time minimization in the literature.

We summarize the methods mentioned through Section 2.1 and their respective

references in Tables 2.1 and 2.2.

Table 2.1: Summary of the mentioned methods for the PFSSP.

Method	Reference
Exact algorithm for $F2\|\|C_{\max}$	Johnson (1954), Silva (2010)
Mathematical model	Wagner (1959), Baker (1974), Stafford (1988), Wilson (1989), Stafford and Tseng (1990), Pan (1997), Stafford and Tseng (2002), Stafford, Tseng and Gupta (2005), Tseng and Stafford (2008)
B&B	Land and Doig (1960) Lomnicki (1965), Ignall and Schrage (1965), Bansal (1977), Potts (1980), Carlier and Rebaï (1996), Croce, Narayan and Tadei (1996), Croce, Ghirardi and Tadei (2002), Chung, Flynn and Kirca (2002), Ladhari and Haouari (2005), Madhushini, Rajendran and Deepa (2009), Ritt (2016)
Constructive heuristic	Nawaz, Enscore and Ham (1983), Liu and Reeves (2001), Kalczynski and Kamburowski (2008), Dong, Huang and Chen (2008), Kalczynski and Kamburowski (2009), Rad, Ruiz and Boroojerdian (2009), Pan and Ruiz (2013), Fernández-Viagas and Framiñan (2014), Fernández-Viagas and Framiñan (2015), Vasiljevic and Danilovic (2015), Rossi, Nagano and Tavares Neto (2016), Liu, Jin and Price (2017), Fernández-Viagas and Framiñan (2017), Benavides and Ritt (2018)
Simulated annealing	Osman and Potts (1989), Wodecki and Bożzejko (2002), Low, Yeh and Huang (2004)
Tabu search	Taillard (1990), Reeves (1993), Nowicki and Smutnicki (1996b), Grabowski and Wodecki (2004)
Genetic algorithm	Chen, Vempati and Aljaber (1995), Murata, Ishibuchi and Tanaka (1996), Ruiz, Maroto and Alcaraz (2006), Tseng and Lin (2009), Zhang, Li and Wang (2009), Tseng and Lin (2010)
IG	Ruiz and Stützle (2007), Pan and Ruiz (2012), Fernández-Viagas and Framiñan (2014), Benavides and Ritt (2016), Dubois-Lacoste, Pagnozzi and Stützle (2017), Fernández-Viagas, Valente and Framiñan (2018), Benavides and Ritt (2018)
ILS	Pan and Ruiz (2012), Dong et al. (2013), Benavides and Ritt (2015), Fernández-Viagas and Framiñan (2017)
Local search procedure	Rajendran and Ziegler (1997), Liu and Reeves (2001), Jarboui, Eddaly and Siarry (2009), Tasgetiren et al. (2011), Benavides and Ritt (2015), Benavides and Ritt (2018)
Ant colony optimization	Rajendran and Ziegler (2004), Rajendran and Ziegler (2005)
Discrete differential evolution	Pan, Tasgetiren and Liang (2008)
Artificial bee colony	Tasgetiren et al. (2011)

Table 2.2: Summary of the mentioned methods for the NPFSSP.

Method	Reference
Mathematical models	Pan (1997)
Lower bound procedure	Gharbi, Labidi and Louly (2014)
Constructive heuristic	Koulamas (1998) Benavides and Ritt (2018)
Simulated annealing	Tandon, Cummings and LeVan (1991), Liu and Ong (2002)
Hybrid heuristic	Jain and Meeran (2002), Lin and Ying (2009)
Threshold acceptance	Liu and Ong (2002)
Genetic algorithm	Liao, Liao and Tseng (2006)
Tabu search	Liu and Ong (2002), Liao, Liao and Tseng (2006)
Scatter search	Haq et al. (2007)
Ant colony optimization	Ying and Lin (2007), Sadjadi, Bouquard and Ziaee (2008), Rossi and Lanzetta (2014)
IG	Benavides and Ritt (2015), Benavides and Ritt (2016) Benavides and Ritt (2018)

2.1.3 Benchmarks

Recent works addressing the PFSSP or NPFSSP with makespan or total completion time minimization usually consider the benchmarks of Taillard (1993), and Vallada, Ruiz and Framiñan (2015) referred to as the VRF benchmark. The former contains 12 groups, each with 10 instances of the same dimensions. The dimensions range from 20 jobs and 5 machines to 500 jobs and 20 machines. The latter contains 48 groups, each one also with 10 instances with the same dimensions, which range from 10 jobs and 5 machines to 800 jobs and 60 machines.

We present upper bounds from the literature for $C_{\max}$ and C_{sum} minimization for each instance of the Taillard benchmark in Appendix A, in Table A.1. These are the values we used as a reference in our computational experiments. Regarding the VRF benchmark, we present the upper bounds in Table A.2. To the extent of our knowledge, there were no upper bounds for C_{sum} minimization in the literature before this thesis. The values in Table A.2 were obtained during our experiments.

2.2 Automatic Algorithm Configuration

The design of a heuristic search method often consists of a trial-and-error approach guided by intuition and previous knowledge about the problem. During the design process, one usually considers multiple options for algorithmic components, such as different local search procedures or perturbation strategies, and tries to identify the combination that yields the best performance through empirical experiments. This is no trivial task since **(i)** the components interact with themselves in different manners, **(ii)** the components can have parameters that need to be tuned, **(iii)** the evaluation of a candidate can take a long time, as it usually consists of running the candidate over a set of problem instances. The problem that arises is called algorithm configuration problem, and a solution for such a problem is commonly called a *configuration*. Moreover, the components can be seen as parameters of the algorithm. In this case, we want to find the optimal values for the parameters linked to the component selection, along with the parameters that determine the algorithm behavior, e.g., the perturbation intensity in an ILS, or the temperature in simulated annealing.

More formally, consider an algorithm $\mathcal{A}$ that we want to configure, with parameter configuration space Θ, a set of problem instances Π and a cost metric $\mathcal{C}$. Hutter et al. (2009) define the algorithm configuration problem as finding the best configuration θ^*, which minimizes $\mathcal{C}$ considering the instances in Π. That is:

$$\theta^* \in \arg\min_{\theta \in \Theta} \sum_{\pi \in \Pi} \mathcal{C}(\theta, \pi).$$

Since $\mathcal{A}$ can be stochastic, $\mathcal{C}(\theta, \pi)$ is often a random variable and its value has to be estimated through observations $c(\theta, \pi)$.

Since modern algorithms are often complex and have several parameters, a manual configuration can take a long time and require a large amount of manual effort. The automation of the design of algorithms, also referred to as automatic algorithm configuration (AAC), can significantly reduce the necessary manual effort, make the design process more efficient, and result in better algorithms (MARMION et al., 2013; KHUDABUKHSH et al., 2016). A more systematic approach also increases the robustness and makes the whole process less prone to human error and bias.

The first techniques to automate algorithm configuration in the literature are dated from as far back as the 1960s (BURKE et al., 2013), however the area has only gained

popularity more recently. Current relevant approaches include genetic programming (GP) (KOZA, 1992), grammatical evolution (GE) (RYAN; COLLINS; NEILL, 1998), ParamILS (HUTTER et al., 2009), SMAC (HUTTER; HOOS; LEYTON-BROWN, 2011) and irace (LÓPEZ-IBÁÑEZ et al., 2016).

GP was introduced by Koza (1992) and consists of applying genetic algorithms to configure programs, using a fitness that measures how well they solve a set of instances of a target problem. As in standard genetic algorithms, individuals are represented as a sequence of genes called genotype, and the evolution can be done with typical strategies, such as two-point crossover.

GE is a type of GP, in which a grammar is used to specify how individual algorithmic components can be combined to form a full heuristic search method. The search space defined by the grammar is explored with a genetic algorithm in which each individual corresponds to an instantiation of the grammar.

In the approach introduced by Ryan, Collins and Neill (1998), each gene has a value in the interval $[0, 255]$ and serves the purpose of defining how to expand the non-terminals from the grammar to generate an algorithm. The decodification procedure works as follows. The grammar is traversed with a leftmost derivation strategy, i.e., the leftmost non-terminal is expanded at each stage. When a non-terminal is expanded, the next available gene is consumed to select an expansion option. Each option is mapped to an integer value, starting from zero. The gene value modulo the number of options determines which one is to be selected.

For example, consider the grammar in Figure 2.6 and the genotype $g = (8, 122, 109)$. The first choice is associated with the start symbol `<START>`, which has two expansion options: `neh`, mapped to value 0, and `frb5`, mapped to value 1. The first gene, with value 8, is consumed to decide how to expand `<START>`, thus we have $8 \bmod 2 = 0$. This selects the option mapped to value 0, `neh`. Now, we have the `<ORDER>` and `<TIEBREAK>` non-terminals. We continue with the leftmost, `<ORDER>`, which has four expansion options. The next gene, with value 122, is consumed. We have $122 \bmod 4 = 2$, thus `kk1` is selected. The remaining non-terminal `<TIEBREAK>` has five expansion options and the third gene is 109, thus we have $109 \bmod 5 = 4$, resulting in `random` being selected.

This strategy has some issues, such as its high redundancy and low locality (MCKAY et al., 2010). High redundancy is an issue because several different genotypes can induce the same phenotype (many-to-one mapping), resulting in an unnecessarily large search space. Considering the previous example, `<START>` has two expansion options. Even

Figure 2.6: An example of a grammar in Backus-Naur Form.

```
1 <START>     ::= neh(<ORDER>,<TIEBREAK>) |
2                 frb5(<ORDER>,<TIEBREAK>)
3 <ORDER>     ::= noninc | nondec | kk1 | kk2
4 <TIEBREAK> ::= kk1 | kk2 | first | last | random
```

values for the first gene values will result in `neh` being selected, while odd values select `frb5`. Thus, since the gene value is in $[0, 255]$, we have 128 different values that will select each option.

The locality of a mapping between genotype and phenotype is determined by the extent of the changes induced to the phenotype when the genotype is modified. A mapping is said to have a high locality if a small change in the genotype leads to a small change in the phenotype. This is important to the efficiency of search methods, as a low locality can result in a search trajectory similar to a random walk. In the case of Ryan, Collins and Neill (1998), it is easy to see that a simple increment or decrement to the value of a gene may change the decisions to which the subsequent genes are linked to, possibly generating a highly dissimilar neighbor.

Structured grammatical evolution (SGE) is an approach proposed by Lourenço, Pereira and Costa (2016) that addresses the high redundancy and low locality of GE. In SGE, each non-terminal is linked to a specific gene, increasing locality. This allows the interval for the value of a gene to be adjusted according to the number of expansion options of the respective non-terminal, resulting in a less redundant mapping between genotype and phenotype (one-to-one mapping). If a non-terminal can be expanded multiple times, the respective gene contains a list of values, one for each possible expansion. We provide a detailed example of a genotype and its decoding in Section 3.1.

One of the biggest challenges in AAC is the high computational time required to evaluate the algorithms. Some of the most relevant AAC methods in the literature use adaptive capping mechanisms to reduce the overall length of the configuration process. The mechanisms typically include the early termination of candidates with poor performance. However, the implementation of this idea has been almost exclusively limited to runtime minimization problems, such as the propositional satisfiability problem (SAT). In this case, a candidate run is terminated early if it has been running for a time higher than an upper bound based on the best candidates obtained so far.

Furthermore, the high evaluation time issue has also been addressed through sur-

rogate models trained to estimate the candidates' empirical performance over a set of problem instances. The models can be trained in an offline step with data from previous configuration runs and are useful to reduce the time necessary to evaluate a candidate (EGGENSPERGER et al., 2018). Commonly used models include random forests (BREIMAN, 2001) and gaussian processes (RASMUSSEN; WILLIAMS, 2006). An example of a model-based method for AAC is SMAC. It builds regression models based on random forests to predict candidates' performance and select promising candidates to evaluate. The method repeatedly searches the model for configurations with a high-performance prediction and compares them to the incumbent over a set of instances. The model is then updated, and the whole process is repeated until a time budget is exhausted.

ParamILS is an AAC tool that applies ILS to configurations. It implements a first-improvement local search that modifies the value of one parameter at a time (one-exchange neighborhood) and includes a capping mechanism for runtime minimization that discards poorly performing configurations. The perturbation step performs a sequence of random moves. The acceptance criterion admits solutions of equal or better quality than the incumbent, with a probability to randomly reinitialize the search. Moreover, ParamILS implements two search strategies named BasicILS and FocusedILS. BasicILS evaluates each neighbor on a fixed number of instances, while FocusedILS adjusts this number according to the quality of the neighbor.

Cáceres and Stützle (2017) propose a variable neighborhood search (VNS) for configurations, on k-exchange neighborhoods N_k, with two search strategies. The first visits the neighborhoods in a round-robin manner in order of increasing k, the second in order of decreasing k. The computational experiments compared VNS and ILS for the tuning of 74 parameters of the MIP solver CPLEX and showed a slight improvement compared to ParamILS.

Ansótegui, Sellmann and Tierney (2009) addressed the algorithm configuration problem with a gender-based genetic algorithm (GGA). The so-called gender is an attribute of the candidates and is used to divide them into competitive and non-competitive subsets. Individuals in the former compete for the right to reproduce, while the individuals of the latter are selected at random as a diversification mechanism. Reproduction always considers one individual from each subset. Computational experiments showed that GGA was able to find better configurations than ParamILS for three out of four tested SAT solvers.

Ansótegui et al. (2015) investigated the use of regression models based on random

Algorithm 5 irace

Input: Set of instances I, parameter space S
Output: A set of elite candidates E

```
1:  function IRACE(I, S)
2:      C = INITIALCANDIDATES(S)
3:      E = RACE(C, I)
4:      while termination criterion not met do
5:          UPDATEDISTRIBUTIONS(E)
6:          C' = SAMPLECANDIDATES(E, S)
7:          C = C' ∪ E
8:          E = RACE(C, I)
9:      end while
10:     return E
11: end function
```

forests in GGA. They contributed a surrogate model to predict and improve offspring quality, and another model to select individuals for the crossover step. The resulting method was named GGA++, and the experimental results showed a small improvement over GGA when configuring two SAT solvers.

One of the best known tools for AAC is irace. It implements an elitist iterated racing procedure that repeatedly performs three main steps: **(i)** it generates new candidates according to certain sampling distributions for the parameters, **(ii)** evaluates the candidates on a subset of problem instances, discarding the inferior ones according to statistical tests, and **(iii)** and updates the sampling distributions to generate new candidates that are more similar to the best ones obtained so far. The main idea is similar to an estimation of distribution algorithm (EDA).

An EDA is an evolutionary method that builds a probabilistic model to sample promising candidates for a given problem. The sampled candidates are iteratively evaluated, and the model is updated to propagate the characteristics of good candidates to future generations. irace introduces racing (BIRATTARI et al., 2002), which eliminates candidates with poor performance early, reducing the overall length of the configuration process. Pseudocode for the iterated racing procedure implemented in irace is presented in Algorithm 5.

There are three main types of parameters in irace: categorical, numerical, and ordinal. Categorical parameters have a set of unrelated discrete values as the domain. Ordinal parameters are similar, but with an ordered set of values. An example of the categorical type is a parameter with values $\{yes, no\}$ that defines whether to apply a local search at some point during the execution of an algorithm. And an example of ordinal

parameter is the intensity of the perturbation to be applied to a solution, with the ordered set of values $\{low, medium, high\}$. Finally, numerical parameters are divided into integer and real types: integer parameters have integer values within a given interval, and real parameters can take floating-point values within a given interval.

Each parameter of the algorithm to be configured is linked to an independent distribution. Truncated normal distributions are used for numerical and ordinal parameters, and discrete distributions for categorical parameters. The initial set of candidates is generated uniformly at random (line 2 in Algorithm 5), and after that, the method iteratively performs races. Each race consists of evaluating the candidates over a set of instances, one instance at a time. In regular intervals, statistical tests are performed to verify the significance of the difference between candidates, and a candidate is discarded if it is statistically worse than at least another candidate. The statistical tests implemented in irace are the Friedman test and the paired *t*-test. The race continues until there is a certain number of surviving candidates or until a budget is exhausted. When a race ends, the best candidates obtained so far are kept in an elite set and are used to sample new candidates (lines 5 and 6 in Algorithm 5). This is done in such a way that each new candidate inherits distributions based on those of a certain elite candidate, which is selected at random with a probability proportional to its rank in the elite set. In normal distributions, the inherited mean is equal to the value of the respective parameter in the parent configuration. The standard deviation is inherited with a slight decrease in value in order to sample values closer to the elite configuration as the search progresses. Similarly, in discrete distributions, the discrete probability of selecting the same value as the parent configuration is inherited with a slight increase. The next race is started with the new candidates, in addition to those in the elite set (lines 7 and 8 in Algorithm 5). The whole process is repeated until a global time limit, or a budget of evaluations is exhausted. The output of irace is the ranked elite set.

Further contributions to irace include the adaptive capping strategy based on runtime of Cáceres et al. (2017), and the surrogate models to predict the performance of candidates of Cáceres, Bischl and Stützle (2017) and Dang et al. (2017).

Some solvers for hard combinatorial problems similar to the one we present in this thesis include SATzilla (XU et al., 2008) and SATenstein (KHUDABUKHSH et al., 2016), both for the SAT problem. The former uses an approach based on an algorithm portfolio, i.e., it selects one algorithm from a collection of fixed algorithms according to some of the characteristics of the problem instance to be solved. Identifying when to use

which algorithm is a challenging task since algorithms can have different performance across different instances. The related problem is called the algorithm selection problem (RICE, 1976). In contrast, SATenstein uses a similar strategy to the one we use in this work: individual components are combined to build the algorithms. In this latter case, ParamILS is used as the search method. Both solvers yield results that are competitive with the state of the art. Moreover, López-Ibáñez and Stützle (2012) proposed a generic framework to generate multi-objective ACO algorithms that facilitates the usage of automatic algorithm configuration. The framework was used to generate new algorithms for a bi-objective traveling salesman problem. The computational experiments showed that these algorithms significantly outperformed other multi-objective ACO algorithms in the literature.

Regarding the application of AAC techniques to flow shop problems, Vázquez-Rodríguez and Ochoa (2011) used GP to discover new priority orders for NEH for five objective functions. Computational experiments considering modified Taillard instances showed that the new priority orders could improve the original orders with statistical significance in 49 out of 60 tested cases. However, in an additional experiment with makespan minimization over the standard instances of Taillard, the new priority order was unable to improve the original one, yielding equivalent results on average.

The PFSSP has also been addressed by means of AAC by Mascia et al. (2013) and Marmion et al. (2013). Both works use a similar grammar-based approach to generate algorithms for weighted tardiness minimization. They use a parametric representation for the algorithms, in which each possible expansion of a non-terminal is linked to a parameter of a solver. The search space defined by the grammar is explored with irace. The grammar of Mascia et al. (2013) generates IG algorithms, whereas the grammar of Marmion et al. (2013) generates hybrid methods that can combine simulated annealing, variable neighborhood search, iterated greedy algorithms, among others. Mascia et al. (2013) compared the parametric representation to the one used in GE and showed that the former outperformed the latter. Marmion et al. (2013) compared three hybrid algorithms obtained with irace to a state-of-the-art IG. The reported results were mostly equivalent for instances with 50 jobs and 20 machines. For instances with 100 jobs and 20 machines, the hybrid algorithms outperformed the IG.

Pagnozzi and Stützle (2019) and Pagnozzi (2019) presented a solver for the PFSSP that implements several individual algorithmic components and combined them through AAC to obtain efficient methods. In general, the solver was built with similar goals and

used techniques similar to the one presented in this thesis, such as the grammar-based approach, the parametric representation for the algorithms, and the application of irace as the search method. The main difference to our work is that Pagnozzi and Stützle (2019) and Pagnozzi (2019) focused on representing a broader range of heuristics. They particularly focused on metaheuristics and on generating hybrid recursive local search methods that combine tabu search, variable neighborhood descent, ILS, and IG algorithms. In contrast, we focus exclusively on non-recursive ILS and IG algorithms based on their known efficiency for solving flow shop problems. Pagnozzi (2019) also conducted computational experiments to compare algorithms with one or two recursion levels to non-recursive algorithms. The results showed that they have similar performance. Finally, besides studying the classical PFSSP, Pagnozzi (2019) studied additional constraints for the PFSSP, such as sequence-dependent setup times. We took a different direction and studied the non-permutation variant of the problem.

We refer the reader to the review of Branke et al. (2016) for a more extensive description of the literature regarding the AAC of heuristics for production scheduling problems.

3 AUTOMATED DESIGN OF HEURISTICS FOR PERMUTATION FLOW SHOPS

In this section, we present our approach and the results we obtained in our work on automating the design process of heuristics for the PFSSP through AAC techniques. We addressed makespan and total completion time minimization, the two most common objectives in the literature.

For each problem and objective function, we defined a context-free grammar that establishes how individual algorithmic components can be combined to form full heuristics. These algorithmic components are, for example, constructive heuristics, local search procedures, and tiebreakers. The grammar and its components were implemented in a parameterized solver, in such a way that a heuristic can be instantiated with a set of parameter values. To convert a grammar to parameters, we use an approach similar to the one in SGE in which categorical parameters are linked to the non-terminal symbols. More precisely, we use one parameter for each time each non-terminal can be expanded (we provide a detailed example in the next section). The value of each parameter determines which component is to be selected. Finally, the solver instantiates the corresponding algorithm and uses it to solve a given problem instance.

The search space defined by the grammar is explored with the parameter configuration tool irace. Our goal is to obtain new algorithms by combining individual components, seeing how well they perform compared to the state of the art, and identifying good combinations that were unknown.

Overall, our approach is similar to Mascia et al. (2013), Marmion et al. (2013), Pagnozzi and Stützle (2019). However, instead of using an approach that aims to represent a broader range of heuristic strategies and generate hybrid methods, we focus on IG and ILS algorithms only. Although simpler, our algorithms are fine-grained. For example, we implement generic tie-breaking rules for constructive heuristics, perturbation functions, and local search procedures. We chose these types of algorithms based on their known efficiency to solve flow shop problems, as shown in previous sections.

In the following sections, we present the grammar, describe the algorithmic components, and the computational experiments and its results for total completion time and makespan minimization.

Figure 3.1: A grammar of iterated local search and iterated greedy algorithms. For simplicity, the numerical parameters have been omitted.

```
 1 <START>     ::= IG(<INI_SOL>,<TIEBREAK>,<PARTIAL>,<LS>) |
 2                 ILS(<INI_SOL>,<LS>,<PERTURB>)
 3 <INI_SOL>   ::= NEHCsum(<TIEBREAK>) | LR | BSCH |
 4                 FRB5(<TIEBREAK>)
 5 <TIEBREAK>  ::= KK1 | KK2 | first | last | random
 6 <PARTIAL>   ::= ϵ | insertion(<TIEBREAK>)
 7 <LS>        ::= <LS_PROC> | alternate(<LS_PROC>,<LS_PROC>)
 8 <LS_PROC>   ::= insertLS(<TIEBREAK>) | swapTasgetiren |
 9                 swapInc | insertTasgetiren |
10                 lsTasgetiren | fpe | bpe | iRZ |
11                 riRZ | raiRZ | insertFPR | swapFirst |
12                 swapBest | swapR
13 <PERTURB>   ::= swap | shift
```

3.1 Methods to Minimize the Total Completion Time

3.1.1 Grammar and Components

The grammar we used is presented in Backus-Naur Form (BNF) in Figure 3.1. We omitted numerical parameters for the sake of simplicity. They are presented in the following sections. The algorithmic components comprise metaheuristic search strategies, constructive heuristics to generate initial permutations, tie-breaking rules, local search procedures, and perturbation strategies.

The algorithms generated by derivations of such a grammar have the form of two state-of-the-art metaheuristics for the PFSSP: they can be either an IG algorithm, as the one proposed by Ruiz and Stützle (2007) (see Algorithm 3), or an iterated local search with a pool of solutions, as proposed by Dong et al. (2013).

Regarding the IG, Dubois-Lacoste, Pagnozzi and Stützle (2017) proposed the addition of a local search to improve the partial solution during the perturbation step. The insertion local search presented in Algorithm 1 is applied to optimize the solution obtained after the destruction and before the start of the construction phase. The results showed that such a strategy could improve the makespan and contributed to the improvement of the state of the art. We, therefore, evaluate it for total completion time minimization. The strategy is represented by the `<PARTIAL>` non-terminal in the grammar, which is optional and specific to the IG.

Furthermore, ties are often found when looking for the best insertion for a job.

This can occur, for example, during the perturbation step in the IG, in which each removed job is inserted into the best position in the schedule. We implemented five tiebreakers to decide among tied positions. The tiebreakers are linked to the <TIEBREAK> symbol and are as follows: **(i)** *first* selects the position of smallest index among the tied positions, **(ii)** *last* selects the position of biggest index, **(iii)** *KK1* is the tie-breaking rule of Kalczynski and Kamburowski (2008), **(iv)** *KK2* is the tie-breaking rule of Kalczynski and Kamburowski (2009), and **(v)** *random* selects a random position among the tied ones.

Finally, the temperature for the Metropolis acceptance criterion in the IG is defined as: $T = \alpha\overline{p}/10n$, where $\overline{p} = \sum_{i\in[m]} \sum_{j\in[n]} p_{ij}/nm$ and α is a parameter.

The second metaheuristic in our grammar is similar to the multi-restart iterated local search (MRSILS) proposed by Dong et al. (2013). MRSILS is an ILS that includes a pool of solutions used to restart the search once it is unable to improve the incumbent solution for a certain number of iterations. Algorithm 6 presents the method in pseudocode. The functions in the algorithm perform the following: **(i)** *Perturb* applies a perturbation to a given solution. The perturbation performs p random movements and is represented by the <PERTURB> non-terminal in the grammar. The two derivation options determine if the movements performed are shift or swap movements. A shift movement reinserts a given job in a random position, while a swap movement exchanges the positions of two random jobs. **(ii)** *InsertIntoPool* adds a solution to the pool, and **(iii)** *RemoveWorstFromPool* removes the worst solution from the pool. **(iv)** *PoolSize* returns the number of solutions currently in the pool, **(v)** *ClearPool* empties the pool of solutions, and **(vi)** *NotInPool* returns true if a solution is not in the pool and false otherwise. Finally, **(vii)** *SelectFromPool* determines the current solution. If the solution pool is not full (parameter *ps* determines the maximum pool size) then the best one in it is returned, otherwise a randomly selected solution from the pool is returned. Note that the pool size is equivalent to the number of iterations without improvement after which restarts from random solutions from the pool are done.

Both the IG and the ILS start from a solution built by a constructive heuristic. We implemented four constructive heuristics: *LR* (LIU; REEVES, 2001), *BSCH* (FERNÁNDEZ-VIAGAS; FRAMIÑAN, 2017), and adaptations of *NEH* (NAWAZ; ENSCORE; HAM, 1983) (here named *NEHCsum*) and *FRB5* (RAD; RUIZ; BOROOJERDIAN, 2009) to minimize the total completion time. The initial solution is linked to the <INI_SOL> non-terminal.

As mentioned in Section 2.1.1.2, *NEH* and *FRB5* can be adapted for total com-

Algorithm 6 MRSILS for the PFSSP

Input: Perturbation intensity p, maximum pool size *ps*
Output: Best solution found π^*

```
1:  function MRSILS(p, ps)
2:      π = INITIALSOLUTION
3:      π = LOCALSEARCH(π)
4:      while termination criterion not met do
5:          π′ = PERTURB(π, p)
6:          π′ = LOCALSEARCH(π′)
7:          if UPDATEDINCUMBENT then
8:              CLEARPOOL
9:          end if
10:         if NOTINPOOL(π′) then
11:             INSERTINTOPOOL(π′)
12:             if POOLSIZE > ps then
13:                 REMOVEWORSTFROMPOOL
14:             end if
15:         end if
16:         π = SELECTFROMPOOL
17:     end while
18:     return π*
19: end function
```

pletion time minimization by simply sorting the jobs in non-decreasing order according to the sum of their processing times across all m machines. The other steps remain the same. Both *NEHCsum* and *FRB5* break the ties according to one of the rules presented earlier. *LR* is the constructive heuristic proposed by Liu and Reeves (2001). We fix the number of schedules $x = \lceil n/m \rceil$, the best value found by Liu and Reeves (2001). *BSCH* is the beam search method proposed by Fernández-Viagas and Framiñan (2017). We set the beam width $w = n$, as it performed best according to Fernández-Viagas and Framiñan (2017).

Benavides and Ritt (2015) observed that some local search procedures often complement each other, and adopted a strategy in which two different procedures are applied alternately. In their proposed ILS method, a swap-based local search is performed if the current iteration is even, and a shift-based local search is performed otherwise. We implemented an analogous strategy in this work. The `<LS>` non-terminal in Figure 3.1 can be derived into a single local search or an alternation between two local search procedures. The `<LS_PROC>` non-terminal determines the procedures to be performed. We present a brief description of each of the 14 local search procedures we implemented. In addition, we provide algorithms in pseudocode for each method in Appendix B.

(i) *iRZ* (RAJENDRAN; ZIEGLER, 1997) is an insertion local search in which, starting with the first job in the schedule, each job is removed and reinserted into the best position. The method follows a first-improvement strategy and iterates until no improvements are found after a full neighborhood evaluation, limited to n_{ls} full evaluations. **(ii)** *riRZ* is the same as *iRZ*, except that jobs are removed in reverse order, *i.e.*, it starts from the last job in the schedule, while in **(iii)** *raiRZ* jobs are removed in random order. **(iv)** *insertLS* is an insertion local search that removes and reinserts each job into the best position, similarly to *iRZ*. However, it was implemented to break ties according to one of the five rules presented subsequently. All the other procedures were implemented as described in the original papers. When no tie-breaking strategy is specified, we select the first position by default. **(v)** *insertFPR* (TASGETIREN et al., 2011) is also similar to *iRZ*, except that the search immediately stops after n iterations without improvement, whereas *iRZ* only checks the stopping criterion after reinserting the last job in the schedule. **(vi)** *insertTasgetiren* (TASGETIREN et al., 2011) is an insertion local search in which a job is removed, and its insertion is evaluated only after its previous position. **(vii)** *swapTasgetiren* (TASGETIREN et al., 2011) is a standard swap local search that swaps all pairs of jobs with a first-improvement strategy. Improvements restart the search, which continues until no improvements are found after a full neighborhood evaluation. **(viii)** *lsTasgetiren* (TASGETIREN et al., 2011) applies *insertTasgetiren*, followed by *swapTasgetiren*. This is repeated while improvements are found. **(ix)** *swapFirst* is a first-improvement adjacent swap local search that cyclically swaps adjacent jobs, while **(x)** *swapBest* is a best-improvement adjacent swap local search that evaluates all possible swaps of adjacent jobs and performs the best one if it improves the current solution. **(xi)** *fpe* (LIU; REEVES, 2001) exchanges each job with its x successors with a first-improvement strategy. We set $x = \lceil n/m \rceil$, the best value according to Liu and Reeves (2001). **(xii)** *bpe* (LIU; REEVES, 2001) works similarly to *fpe*, except that the jobs are exchanged with their predecessors. **(xiii)** *swapInc* (BENAVIDES; RITT, 2015) is similar to *fpe*, but uses an incremental value for x. It starts with $x = 1$ and increments x by one each time improvements are not found after a full neighborhood evaluation, up to $x = n$. Both the search and the value for x are restarted when an improvement is found. **(xiv)** And finally, in the *swapR* (DEROUSSI; GOURGAND; NORRE, 2006) method, for each pair of job positions (i, j), the j^{th} job is swapped with the i^{th} job, which is then reinserted into the best possible position. The method follows a first-improvement strategy.

We use the n_{ls} parameter to limit the number of times the full neighborhood is

Table 3.1: Categorical parameters required to instantiate algorithms from the grammar.

Parameter	Decision
alg	Metaheuristic (IG or ILS).
ini_sol	Constructive heuristic to generate the initial solution.
tb_ini_sol	Tiebreaker for constructive heuristic (*NEHCsum* or *FRB5*).
tb_ig	Tiebreaker for the IG.
partial	If the partial solution is to be improved with a local search.
tb_partial	Tiebreaker for the local search to be applied to the partial solution.
ls	If one or two local search procedure are to be applied.
ls_proc1	First local search procedure.
ls_proc2	Second local search procedure.
tb_ls_proc1	Tiebreaker for the first local search procedure.
tb_ls_proc2	Tiebreaker for the second local search procedure.
pert	Perturbation strategy for the ILS.

evaluated in *insertLS*, *iRZ*, *riRZ*, *raiRZ*, *swapFirst* and *swapBest*. Low values for n_{ls} can reduce the running time of the local search, but at a possible cost in solution quality (DUBOIS-LACOSTE, 2014).

We now describe how to transform the grammar into a parametric representation through an example. Given the grammar in Figure 3.1 we have 7 non-terminals (`<START>`, `<INI_SOL>`, `<TIEBREAK>`, `<PARTIAL>`, `<LS>`, `<LS_PROC>` and `<PERTURB>`). We observe that `<START>`, `<INI_SOL>`, `<PARTIAL>`, `<LS>`, and `<PERTURB>` can be expanded only once. The `<TIEBREAK>` symbol can be expanded up to five times: one if `iga` is selected in line 1, one if the initial solution is generated with `FRB5`, another if the partial solution is optimized, and other two times if two local search procedures are selected and derived to `insertLS`. Lastly, the `<LS_PROC>` non-terminal can be expanded up to two times, if two local search procedures are to be applied. As each possible expansion is linked to a parameter, we therefore have a total of 12 parameters. The parameters and their respective decisions are summarized in Table 3.1.

The value of each parameter defines which option is to be selected, e.g. *ini_sol* = *LR* selects the *LR* option, and defines that the *LR* constructive heuristic is used to build the initial solution. The solver takes the parameter values as input and instantiates the corresponding algorithm. Moreover, some parameters are conditional to certain options being selected, e.g., the parameter *pert*, which decides the perturbation strategy for the ILS, is ignored if the algorithm is an IG (*alg* = *IG*).

Having established a grammar that determines how individual components can be combined to generate an algorithm and having defined the parametric approach to represent instantiations of the grammar, we use the parameter configuration tool irace to

find good configurations. Besides the 12 categorical parameters, other parameters linked to the components must be configured as well. If the algorithm is an IG, then the number of jobs removed in the destruction phase d and the temperature multiplier α have to be calibrated. For the ILS, we have the pool size *ps* and the number of moves performed by the perturbation procedure p. We used ranges including typical values used in the literature: $d \in [1, 20]$, $\alpha \in [0.01, 1]$, $p \in [1, 20]$, $ps \in [2, 20]$ and $n_{ls} \in [1, 20]$. We present a list with all the parameters that were configured with irace in Table 3.2. Column "Parameter" presents the parameter name, and "Type" is the parameter type specified in irace. Column "Values" defines the values each parameter can assume and column "Conditions" lists for each parameter the conditions under which it is enabled. If no conditions are specified, then the parameter is always enabled.

Table 3.2: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
alg	Categorical	*IG*, *ILS*	
ini_sol	Categorical	*NEHCsum*, *LR*, *BSCH*, *FRB5*	
tb_ini_sol	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in \{$*NEHCsum*, *FRB5*$\}$
tb_ig	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*alg* $=$ *IG*
partial	Categorical	*yes*, *no*	*alg* $=$ *IG*
tb_partial	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*partial* $=$ *yes*
ls	Categorical	$1, 2$	
ls_proc_1	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	
ls_proc_2	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	*ls* $= 2$
tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $=$ *insertLS*
tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $=$ *insertLS*
pert	Categorical	*swap*, *shift*	*alg* $=$ *ILS*
d	Integer	$[1, 20]$	*alg* $=$ *IG*
α	Real	$[0.01, 1]$	*alg* $=$ *IG*
p	Integer	$[1, 20]$	*alg* $=$ *ILS*
ps	Integer	$[2, 20]$	*alg* $=$ *ILS*
n_{ls}	Integer	$[1, 20]$	*ls_proc_1* $\in \{$*insertLS*, *iRZ*, *riRZ*, *raiRZ*$\}$ or *ls_proc_2* $\in \{$*insertLS*, *iRZ*, *riRZ*, *raiRZ*$\}$

The columns contain the name of the parameter, the type specified in irace, the domain, and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

3.1.2 Computational Experiments

In this work we considered the benchmarks of Taillard (1993) and Vallada, Ruiz and Framiñan (2015). We ran irace 10 times, each with its own set of randomly generated training instances to avoid overtuning. Each set contains 120 instances with the same dimensions as in the Taillard benchmark and with uniform random processing times in the interval $[1, 99]$. Furthermore, each irace run was restricted to a budget of 2400 candidate runs, and each candidate run was limited to $10nm$ milliseconds. The best algorithm found in each run was then evaluated on the Taillard benchmark.

The experiments were conducted on a PC with an Intel Core i7 930 processor, and 12 GB of main memory, running Ubuntu 16.04.3. Our method was implemented in C++ and compiled with `g++` 5.4.0 using `-O3` flag. In irace (version 2.4.1844), we allowed the parallel evaluation of candidates, limited to four threads.

First, we present the best algorithm of each irace run in Table 3.3. The columns present the algorithm name ("Alg."), the chosen metaheuristic ("M"), the constructive heuristic to generate an initial permutation ("CH"), the tiebreaker for the construction phase in IG ("Tie"), whether the local search for partial solutions is applied or not ("Partial"), the first and the second local search procedures ("Proc. 1", "Proc. 2"), and the numerical parameters d, n_{ls} and α.

Analyzing the obtained algorithms, we observed that all of them are IG algorithms. All did select BSCH as the constructive heuristic, which was to be expected since the results reported in Fernández-Viagas and Framiñan (2017) show a significant advantage over other constructive heuristics. Another choice common to all algorithms was the exclusion of the local search for partial permutations, which suggests that this strategy is less useful for total completion time minimization. The most frequently selected local search procedures were *iRZ* and its variants (*riRZ* and *raiRZ*) and *insertFPR*. The choice of tiebreakers had a significant variation: *KK1* and *KK2* were selected more frequently, but all rules were selected at least once. Regarding the numerical parameters, the values for d in $[5, 11]$ confirm previous findings that removing and reinserting a higher number of jobs is beneficial when minimizing completion time (BENAVIDES; RITT, 2015), in comparison to makespan minimization. The high values for n_{ls} are mostly equivalent since the local search usually terminates earlier in a local minimum. The α-values were very similar, except when the local searches included non-adjacent swaps.

We evaluated algorithms A_0 to A_9 on the Taillard benchmark and compared them

Table 3.3: Automatically designed algorithms for minimizing the total completion time on the PFSSP.

			Perturbation			Local search			
Alg.	M	CH	Tie	d	Partial	Proc. 1	Proc. 2	n_{ls}	α
A_0	IG	*BSCH*	*KK1*	6	No	*riRZ*	-	9	0.2
A_1	IG	*BSCH*	*KK1*	11	No	*insertFPR*	-	-	0.1
A_2	IG	*BSCH*	*First*	8	No	*riRZ*	*fpe*	19	0.1
A_3	IG	*BSCH*	*KK2*	9	No	*raiRZ*	-	15	0.1
A_4	IG	*BSCH*	*KK2*	8	No	*iRZ*	-	15	0.1
A_5	IG	*BSCH*	*Random*	7	No	*fpe*	-	-	0.2
A_6	IG	*BSCH*	*KK2*	5	No	*swapTasgetiren*	*riRZ*	16	0.5
A_7	IG	*BSCH*	*KK1*	7	No	*swapInc*	*insertFPR*	-	0.3
A_8	IG	*BSCH*	*First*	6	No	*fpe*	-	-	0.1
A_9	IG	*BSCH*	*Last*	7	No	*bpe*	*iRZ*	7	0.1

Each row contains the components and parameter values for one of the algorithms obtained with irace, named A_0 to A_9 in the first column. All methods are IG algorithms, as indicated in column "M", and all use BSCH to build the initial solution, as seen in column "CH". The following three columns refer to the perturbation step and contain the tiebreaker, perturbation intensity d, and whether or not to optimize the partial solution between destruction and construction phases. The next three columns contain the first local search procedure, the second local search procedure, and the maximum number of full neighborhood scans n_{ls}. The last column is the value for α, used in the Metropolis acceptance criterion.

to the state-of-the-art metaheuristic MRSILS(BSCH) of Fernández-Viagas and Framiñan (2017). The results are presented in Table 3.4 as the average relative deviation (ARD) in percent from the upper bounds reported by Pan and Ruiz (2012) (see Appendix A). For a fair comparison, we reimplemented MRSILS(BSCH) and presented the results in column "MRSILS". The parameter values used for MRSILS(BSCH) were the same as in Dong et al. (2013). For each algorithm, ten replications per instance were performed, each with a time limit of $30\,nm$ milliseconds. Columns "A_0" to "A_9" show the results for each algorithm. Since algorithms' performance often varies according to the dimensions of the instances, we present a combination of the best results in column "Best". This provides a theoretical estimate of the best results obtained with the components from the grammar if we were able to combine the best characteristics of the ten algorithms in a single method.

Table 3.4: ARD for the Taillard benchmark.

Inst.	MRSILS	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
20×5	0.007	0.002	0.001	0.004	0.006	0.006	**0.000**	**0.000**	**0.000**	0.006	0.005	0.000
20×10	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.001	**0.000**	**0.000**	0.005	**0.000**	0.000
20×20	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.002	**0.000**	**0.000**	0.010	**0.000**	0.000
50×5	0.173	0.219	0.178	0.152	0.173	0.162	0.177	0.248	0.190	**0.141**	0.153	0.141
50×10	0.551	0.464	0.536	0.564	0.534	0.536	0.503	0.397	**0.313**	0.705	0.556	0.313
50×20	0.462	0.435	0.462	0.483	0.449	0.458	0.483	**0.345**	0.379	0.645	0.481	0.345
100×5	0.089	0.105	0.101	0.085	0.100	0.098	0.105	0.099	0.102	**0.070**	0.082	0.070
100×10	0.251	0.255	0.244	0.246	0.241	0.238	0.256	0.278	0.248	0.237	**0.234**	0.234
100×20	0.473	0.462	0.452	0.473	0.452	0.458	0.463	0.503	**0.437**	0.512	0.461	0.437
200×10	-0.657	-0.661	-0.663	-0.670	-0.668	-0.665	-0.657	-0.657	-0.666	**-0.673**	-0.672	-0.673
200×20	-0.846	-0.841	-0.856	-0.851	-0.860	-0.860	-0.837	-0.837	**-0.862**	-0.849	-0.855	-0.862
500×20	-1.889	-1.888	**-1.891**	-1.890	**-1.891**	-1.890	-1.885	-1.888	-1.890	-1.887	**-1.891**	-1.891
Avg.	-0.115	-0.121	-0.120	-0.117	-0.122	-0.122	-0.116	-0.126	**-0.146**	-0.090	-0.121	-0.157

Instances have been grouped according to their dimensions in the first column. The results presented in this table are the ARD from the upper bounds in Table A.1. Column "MRSILS" contains the results for the state-of-the-art method MRSILS(BSCH) proposed by Fernández-Viagas and Framiñan (2017). A_0 to A_9 are the algorithms found with irace. The last column contains a combination of the best results obtained by algorithms A_0 to A_9. The best results for each instance group are highlighted in bold.

In general, we can see that the results obtained with the automatically designed algorithms and those obtained with MRSILS(BSCH) were similar. The algorithm with the best overall ARD, A_7, is an IG that alternates between *insertFPR* and *swapInc*. Since *insertFPR* is similar to the strategy in MRSILS(BSCH), the observed improvement is probably due to the choice of an IG algorithm instead of an ILS, and due to the use of a second, swap-based local search. Algorithm A_8 had slightly higher ARD, while all the other methods, including MRSILS(BSCH), had similar results with a maximum difference of $0.01\,\%$ among them. Furthermore, the ARD of the automatically designed methods is only $0.01\,\%$ to $0.07\,\%$ higher than the combination of the best results. This shows that the algorithms have consistent performance across the different groups of instances, and are not overfitted for specific dimensions.

We applied a Wilcoxon signed-rank test between MRSILS(BSCH) and each of the algorithms from A_0 to A_9 to verify whether the differences are statistically significant. For a significance level of 0.01, with Bonferroni correction, the results indicated that the difference is significant for A_3, A_4, A_7 and A_9 ($p < 0.0006$ in all cases). We have also computed the average solution quality of a random derivation of the grammar over 50 samples and found it to be $1.04\,\%$. This shows that irace was effective in selecting good algorithms.

We conducted an additional experiment to evaluate A_7 on the VRF benchmark and compared it to MRSILS(BSCH). The results are presented as the ARD from MRSILS(BSCH) in Table 3.5. The instances are divided into "Small" and "Large" according to the number of jobs. Ten replications per instance were performed for both algorithms. We note an improvement of 0.064% on average for the smaller instances, while the results are very close for the larger instances. A Wilcoxon signed-rank test confirmed that the difference between the two methods is statistically significant ($p < 2.2 \times 10^{-16}$). It is also worth mentioning that the best algorithm obtained via AAC never performed worse than MRSILS(BSCH).

Overall, the results showed that the automatically designed algorithms are competitive compared to the state of the art, and one of them, in particular, has a slightly superior performance.

Table 3.5: ARD to MRSILS(BSCH) for the VRF benchmark.

Small				Large			
Inst.	ARD	Inst.	ARD	Inst.	ARD	Inst.	ARD
10×5	0.000	40×5	-0.126	100×20	-0.043	500×20	-0.002
10×10	0.000	40×10	-0.147	100×40	-0.042	500×40	-0.005
10×15	0.000	40×15	-0.111	100×60	-0.020	500×60	-0.003
10×20	0.000	40×20	-0.109	200×20	-0.006	600×20	-0.001
20×5	0.000	50×5	-0.042	200×40	-0.010	600×40	-0.003
20×10	0.000	50×10	-0.182	200×60	-0.013	600×60	-0.004
20×15	0.000	50×15	-0.090	300×20	0.000	700×20	-0.002
20×20	-0.000	50×20	-0.143	300×40	-0.010	700×40	-0.002
30×5	-0.085	60×5	-0.005	300×60	-0.011	700×60	-0.004
30×10	-0.069	60×10	-0.076	400×20	-0.001	800×20	-0.002
30×15	-0.034	60×15	-0.153	400×40	-0.002	800×40	-0.001
30×20	-0.024	60×20	-0.129	400×60	-0.005	800×60	-0.004
Avg.			-0.064				-0.008

3.2 Methods to Minimize the Makespan

In this section, we present the grammar, describe the algorithmic components, and present the computational experiments and its results for makespan minimization on the PFSSP.

3.2.1 Grammar and Components

We propose the grammar presented in Figure 3.2. For simplicity, the figure shows only the main algorithmic components and not the numerical parameters of the individual components. The numerical parameters are presented at the end of the section.

Our methods are based on ILS. We refer to them as ILS instead of IG because some of the perturbation options are purely random, without the greedy element. To generate the initial solution for the ILS, we have implemented three constructive heuristics. The first one is the well-known *NEH* constructive heuristic (NAWAZ; ENSCORE; HAM, 1983). The second one is *FRB5* (RAD; RUIZ; BOROOJERDIAN, 2009). For both, *NEH* and *FRB5*, our grammar considers a priority order and a tiebreaker for the job insertion phase. The ordering options we implemented are a non-increasing and a non-decreasing order of total processing time, and the *KK1* (KALCZYNSKI; KAMBUROWSKI, 2008) and *KK2* (KALCZYNSKI; KAMBUROWSKI, 2009) orders. The third constructive heuristic, named *RC*, repeatedly constructs solutions with the same

Figure 3.2: A grammar of iterated local search algorithms. For simplicity, the numerical parameters have been omitted.

```
 1 <START>     ::=  ILS(<INI_SOL>,<PERTURB>,<LS>,
 2                  <ACCEPT>)
 3 <INI_SOL>   ::=  NEH(<ORDER>,<TIEBREAK>) |
 4                  FRB5(<ORDER>,<TIEBREAK>) |
 5                  RC
 6 <ORDER>     ::=  noninc | nondec | KK1 | KK2
 7 <TIEBREAK>  ::=  KK1 | KK2 | first | last |
 8                  random
 9 <PERTURB>   ::=  ri | gi(<TIEBREAK>) | rs | gs |
10                  ras | gi_asls(<TIEBREAK>) |
11                  ils_gi(<TIEBREAK>) |
12                  gi_ils(<TIEBREAK>)
13 <LS>        ::=  ϵ | <LS_PROC> |
14                  alternate(<LS_PROC>,<LS_PROC>)
15 <LS_PROC>   ::=  insertLS(<TIEBREAK>) | NS |
16                  Pc(<TIEBREAK>)
17 <ACCEPT>    ::=  met | lac | rrt
```

strategy as *NEH*, although it breaks ties both in the priority order and in the insertion phase at random. The method builds r solutions and returns the best one. As in Section 3.1.1, the tiebreakers (lines 7 and 8 in Figure 3.2) are **(i)** *first* which selects the position of smallest index, **(ii)** *last* which selects the position of biggest index, **(iii)** *KK1* which is the tiebreaker of Kalczynski and Kamburowski (2008), **(iv)** *KK2* which is the tiebreaker of Kalczynski and Kamburowski (2009) and **(v)** *random* which selects a random position.

Moreover, Lomnicki (1965) showed that solutions for the problem with reversed machine order, i.e., with machine m as the first machine and machine 1 as the last machine, have reversed job order compared to solutions for the problem with the regular machine order. Lomnicki (1965) states that in some cases, it is more efficient to explore the problem with the reversed machine order and revert the job order of the obtained solution at the end. We implemented this strategy for the construction of the initial solution. Parameter *rev* determines if the reversed instance is to be considered. If so, the constructive heuristic is first used to build a solution for the regular instance, and then for the instance with reversed machine order. The best out of the two solutions is kept.

Regarding perturbation functions, we implemented eight strategies (FERNÁNDEZ-VIAGAS; VALENTE; FRAMIÑAN, 2018). All methods perform d moves every iteration but differ in the types of moves performed. The strategies are presented in lines 9 to 12

in Figure 3.2 and are briefly described as follows: **(i)** random insertion (*ri*) removes d randomly selected jobs and inserts them into random positions. **(ii)** Greedy insertion (*gi*) removes d randomly selected jobs and inserts them one at a time, and in the same order they were removed into the best position. This is the destruction-construction operator in IG_RS. **(iii)** Random swap (*rs*) swaps the positions of two randomly selected jobs. **(iv)** Random adjacent swap (*ras*) swaps a randomly selected job in positions 1 to $n-1$ with its immediate successor. **(v)** Greedy swap (*gs*) randomly selects a job and swaps its position with the job that results in the best objective function value. **(vi)** Greedy insertion followed by an adjacent swap local search (*gi_asls*) works similarly to greedy insertion with the addition of an adjacent swap local search performed after each removed job is reinserted. This local search evaluates the swapping of all adjacent jobs that are scheduled after the newly inserted job. **(vii)** Insertion local search followed by greedy insertion (*ils_gi*) removes d randomly selected jobs, applies an insertion-based local search to the partial solution and then reinserts the removed jobs, one at a time and in the same order they were removed. The local search removes each job of the partial solution and inserts it into the best position, immediately accepting improvements. The search is stopped when there are no improvements after a full neighborhood evaluation, limited to n_{ls} full evaluations. **(viii)** Greedy insertion followed by an insertion local search (*gi_ils*) removes d randomly selected jobs and inserts them one at a time, and in the same order they were removed into the best position. After each insertion, both the predecessor and successor of the newly inserted job, if any, are removed and reinserted into the best positions.

During the local search phase, we can apply none, one or two local search procedures (lines 13 and 14 in Figure 3.2). If two procedures are selected, they are performed in alternation, i.e., the first procedure is performed if the current iteration is an even number, and the second procedure is performed otherwise. The implemented procedures are: **(i)** *insertLS*, which is an insertion local search in which each job is removed and reinserted greedily. This is repeated until a local minimum is reached, i.e., no better neighbors are found, or up to n_{ls} full neighborhood scans. **(ii)** *NS*, which is a local search procedure with the NS neighborhood. It evaluates swapping the first two or the last two jobs of each block of jobs on the critical path (except the first two and last two jobs of the schedule) (NOWICKI; SMUTNICKI, 1996a). The method follows a best-improvement strategy. **(iii)** *Pc*, which is similar to *NS*, but instead of being swapped, the jobs are removed and reinserted greedily (BENAVIDES; RITT, 2018).

Finally, we implemented three options for the acceptance criterion (line 17 in Fig-

Table 3.6: Categorical parameters required to instantiate algorithms from the grammar.

Parameter	Decision
ini_sol	Constructive heuristic to build the initial solution.
order	Criterion for job ordering during the construction of the initial solution.
tb_ini_sol	Tiebreaker for the construction of the initial solution.
perturb	Perturbation strategy.
tb_perturb	Tiebreaker for the perturbations that perform greedy insertions.
ls	Number of local search procedures.
ls_proc_1	First local search procedure.
ls_proc_2	Fecond local search procedure.
tb_ls_proc_1	Tiebreaker rule for the first local search procedure.
tb_ls_proc_2	Tiebreaker rule for the second local search procedure.
accept	acceptance criterion.

ure 3.2). The first one is the Metropolis criterion (*met*) (METROPOLIS et al., 1953), in which a new solution is always accepted if it improves the current solution. Otherwise it is accepted with probability $e^{-\Delta/T}$, where Δ is the difference between the objective function value of the new and the current solution, and T is a parameter called the temperature. The second option is late acceptance (*lac*) (BURKE; BYKOV, 2017), in which a new solution is accepted if it improves the current solution or the solution visited l iterations before. The third option is record-to-record travel (*rrt*) (DUECK, 1993), in which a new solution is accepted if its objective function value is smaller than the value of the current solution plus a deviation *rrtd*.

Having described the grammar and the individual components, we now present the parametric representation for instantiations of such a grammar. Our strategy links each possible expansion of a non-terminal in the grammar to a categorical parameter. Each parameter value determines which of the expansion options is to be selected for the respective non-terminal. We can see that <INI_SOL>, <PERTURB>, <LS>, <ACCEPT> and <ORDER> are expanded at most once, while <LS_PROC> can be expanded twice and <TIEBREAK> up to four times: one if either *NEH* or *FRB5* are selected to build the initial solution, one if the perturbation performs greedy insertions, and two more times if two local search procedures are used and both perform greedy insertions. Thus, we have a set with 11 parameters to represent the instantiations of the grammar. The parameters and their respective decisions regarding component selection are summarized in Table 3.6. Additionally, some parameters are conditional to others, e.g., *tb_ini_sol* is not needed if *ini_sol* = *RC*, as *RC* always uses a random tiebreaker.

We used the parameter configuration tool irace to search for good algorithms. As

mentioned earlier, numerical parameters are not shown in Figure 3.2, but are calibrated by irace as well. These parameters are the number of moves performed by the perturbation function d, the number of solutions r built by the *RC* constructive heuristic, whether or not to consider the reversed machine order when building the initial solution (parameter *rev*), the length of the list for late acceptance l and the deviation for record-to-record travel $rrtd$. The temperature T in the Metropolis acceptance criterion is defined as $T = \alpha \times \overline{p}/10$, where $\overline{p} = \sum_{i \in [m]} \sum_{j \in [n]} p_{ij}/nm$ and α is a parameter which we calibrate. The last parameter is the limit of full neighborhood evaluations n_{ls}, which is applied to all local search methods. This includes the local searches performed in *FRB5*, in the perturbation *ils_gi*, and in the procedures *insertLS*, *NS*, and *Pc*.

A summary is presented in Table 3.7, where column "Parameter" presents the parameter name, and "Type" is the parameter type specified in irace. Column "Values" defines the values each parameter can assume. For the numerical parameters, these values were defined according to the typical values used in the literature. Column "Conditions" lists for each parameter the conditions under which it is enabled. If no conditions are specified, then the parameter is always enabled.

Table 3.7: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
ini_sol	Categorical	*NEH*, *FRB5*, *RC*	
order	Categorical	*noninc*, *nondec*, *KK1*, *KK2*	*ini_sol* $\in \{$*NEH*, *FRB5*$\}$
tb_ini_sol	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in \{$*NEH*, *FRB5*$\}$
tb_perturb	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*perturb* $\in \{$*gi*, *gi_asls*, *ils_gi*, *gi_ils*$\}$
tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $\in \{$*insertLS*, *Pc*$\}$
tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $\in \{$*insertLS*, *Pc*$\}$
perturb	Categorical	*ri*, *gi*, *rs*, *gs*, *ras*, *gi_asls*, *ils_gi*, *gi_ils*	
ls	Categorical	$0, 1, 2$	
ls_proc_1	Categorical	*insertLS*, *NS*, *Pc*	*ls* $\in \{1, 2\}$
ls_proc_2	Categorical	*insertLS*, *NS*, *Pc*	*ls* $= 2$
accept	Categorical	*met*, *lac*, *rrt*	
rev	Categorical	no, yes	
r	Ordinal	$1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 1000$	*ini_sol* $=$ *RC*
d	Integer	$[1, 20]$	
n_{ls}	Ordinal	$1, 2, 3, 4, \infty$	*ini_sol* $=$ *FRB5* or *ls* $\in \{1, 2\}$ or *perturb* $=$ *ils_gi*
α	Real	$[0.01, 1]$	*accept* = *met*
l	Ordinal	$1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000$	*accept* = *lac*
rrtd	Ordinal	$0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500$	*accept* = *rrt*

The columns contain the name of the parameter, the type specified in irace, the domain, and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

3.2.2 Computational Experiments

In this section, we report the results of the automatic configuration and compare them to the state-of-the-art. To evaluate our algorithms we considered the benchmarks of Taillard (1993) and Vallada, Ruiz and Framiñan (2015) (VRF-large).

Our experiments were performed on a PC with an AMD FX-8150 processor running at 3.6 GHz, and 32 GB of main memory, under Ubuntu 16.04.3. Our solver was implemented in C++ and compiled with `g++ 5.4.0` using the optimization flag `-O3`. We used version $2.4.1844$ of irace with the option of parallel execution set to eight threads.

We ran irace ten times, with a budget of 30000 candidate runs and a time limit of $10nm$ milliseconds per run. The training set contained 120 randomly generated instances with the same dimensions of those in the Taillard benchmark. The instances were generated according to Taillard (TAILLARD, 1993), with uniform random processing times in the interval $[1, 99]$.

We present the best algorithm for each of the 10 runs in Table 3.8. Column "Alg." presents the name of the algorithm. For the initial solution, columns "CH", "Order", and "Tie" report the chosen constructive heuristic, job priority order, and tiebreaker for the constructive heuristic. Column "*rev*" presents whether or not the reversed instance is also solved by the constructive heuristic, while column "r" contains the value for the respective parameter when *RC* was selected. The next three columns contain the perturbation method, its tie-breaking rule, and the perturbation intensity d. The next five columns refer to the local search phase and present the first local search procedure, the tiebreaker for the first procedure, the second local search procedure, the tiebreaker for the second procedure, and the value for n_{ls}. The last two columns contain the acceptance criterion and its respective parameter.

Table 3.8: Automatically designed algorithms for minimizing the makespan on the PFSSP.

	Initial solution					Perturbation			Local search					Acceptance	
Alg.	CH	Order	Tie	r	*rev*	Method	Tie	d	Proc. 1	Tie 1	Proc. 2	Tie 2	n_{ls}	Crit.	α
A_0	*RC*	-	-	30	yes	*ils_gi*	*KK1*	1	*insertLS*	*first*	-	-	4	*met*	0.6
A_1	*RC*	-	-	60	yes	*gi*	*random*	6	*insertLS*	*KK1*	-	-	4	*met*	0.8
A_2	*NEH*	*noninc*	*last*	-	yes	*ils_gi*	*KK2*	1	*insertLS*	*last*	-	-	∞	*met*	0.8
A_3	*NEH*	*noninc*	*last*	-	no	*ils_gi*	*last*	1	*insertLS*	*KK2*	-	-	4	*met*	0.7
A_4	*NEH*	*KK1*	*last*	-	no	*ils_gi*	*KK2*	1	*Pc*	*KK2*	*insertLS*	*last*	4	*met*	0.9
A_5	*RC*	-	-	1	yes	*ils_gi*	*last*	1	*insertLS*	*KK1*	-	-	2	*met*	0.6
A_6	*RC*	-	-	50	no	*ils_gi*	*random*	2	*insertLS*	*KK1*	-	-	3	*met*	0.3
A_7	*FRB5*	*nondec*	*KK1*	-	no	*gi_ils*	*KK1*	3	*insertLS*	*KK2*	-	-	4	*met*	0.5
A_8	*RC*	-	-	60	no	*gi*	*first*	9	*insertLS*	*KK1*	-	-	∞	*met*	0.2
A_9	*RC*	-	-	40	no	*ils_gi*	*KK2*	1	*insertLS*	*random*	-	-	∞	*met*	0.8

Each row contains the components and parameter values for one of the ten algorithms obtained with irace, named A_0 to A_9 in the first column. The following five columns refer to the initial solution and contain, respectively, the constructive heuristic, the ordering criteria for the constructive heuristic, the tiebreaker, and the values for parameters r and *rev*. The following three columns contain the perturbation method, its tiebreaker, and the perturbation intensity d. The following columns refer to the local search phase and contain the procedures, the tiebreakers, and the maximum number of full neighborhood scans n_{ls}. The last two columns present the acceptance criterion and the values for its parameter α.

Analyzing the results, we observe that regarding the construction of the initial solution, *RC* is the most frequently selected option, with low values for r in all cases. Algorithm A_5, in particular, has $r = 1$ which is equivalent to selecting the *NEH* heuristic with the *noninc* order and the *random* tiebreaker. Four algorithms use the reversed instance, but there was no clear pattern of choice.

Regarding the perturbation function, the most frequently selected strategy was *ils_gi* with seven occurrences, always with low intensity. In the tiebreakers, however, there was a substantial variation, and four different options were selected. The *gi* strategy was chosen two times, both with the *KK1* tiebreaker, but with different values for d.

The choices concerning the local search were very homogeneous: all algorithms use *insertLS*, mostly with *KK1* or *KK1* tiebreakers. Algorithm A_4, in particular, also performs the *Pc* procedure. Another choice common to all algorithms is the Metropolis acceptance criterion, although there was substantial variation in the value of α.

In summary, we observe that the automatically designed algorithms resemble either the method of Ruiz and Stützle (2007) or the method of Dubois-Lacoste, Pagnozzi and Stützle (2017). A_4 and A_7 are exceptions: A_4 alternates between two local search procedures, and A_7 has a different perturbation strategy.

We have evaluated algorithms A_0 to A_9 on the Taillard benchmark and present the results in Table 3.9. For each group of instances (column "Inst."), we report the ARD in percent from the values in Table A.1. For the VRF-large benchmark, we use the same values as in Dubois-Lacoste, Pagnozzi and Stützle (2017). Five replications per instance were performed. Each run had a time limit of $30nm$ milliseconds.

To compare our results to the literature, we instantiate in our solver the algorithms IG$_{rs}$ of Ruiz and Stützle (2007) and the state-of-the-art IG$_{lsps}$ of Dubois-Lacoste, Pagnozzi and Stützle (2017). Since the authors do not specify the tie-breaking rules, we evaluated all our implemented rules and used the one with the best results, which was *KK2*. The results are presented in columns "IG$_{rs}$" and "IG$_{lsps}$". Columns "A_0" to "A_9" contain the individual results for the algorithms obtained via AAC. Column "Best" contains the combined best results.

Table 3.9: ARD for the Taillard benchmark.

Inst.	IG_{rs}	IG_{lsps}	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
20×5	0.000	0.024	0.040	**0.000**	0.016	**0.000**	0.016	0.016	0.016	0.016	**0.000**	0.040	0.000
20×10	0.000	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.008	**0.000**	**0.000**	**0.000**	**0.000**	0.000
20×20	0.008	0.003	0.023	**0.000**	0.012	0.012	0.006	0.021	0.014	0.005	0.008	0.006	0.000
50×5	0.003	0.003	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.000
50×10	0.425	0.487	0.390	0.482	0.410	0.416	0.428	**0.380**	0.356	0.420	0.409	0.429	0.380
50×20	0.550	0.520	0.478	0.499	**0.381**	0.404	0.427	0.428	0.571	0.395	0.587	0.450	0.381
100×5	0.003	0.002	**0.000**	0.004	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.004	**0.000**	**0.000**	0.000
100×10	0.109	0.085	0.038	0.061	0.051	0.040	0.055	0.043	0.041	0.059	0.059	**0.034**	0.034
100×20	0.925	0.881	0.667	0.842	0.620	**0.603**	0.707	0.658	0.848	0.760	0.896	0.667	0.603
200×10	0.045	0.045	0.040	0.043	**0.037**	0.043	0.044	0.041	0.041	0.038	0.043	0.040	0.037
200×20	1.142	1.006	0.788	1.073	0.807	0.822	0.874	0.868	1.004	1.048	1.079	**0.770**	0.770
500×20	0.429	0.402	0.292	0.391	0.303	0.332	0.317	0.328	0.341	0.374	0.382	**0.285**	0.285
Avg.	0.303	0.288	0.230	0.283	**0.220**	0.223	0.240	0.233	0.270	0.260	0.289	0.227	0.207

Instances have been grouped according to their dimensions in the first column. The results presented in this table are the ARD over the upper bounds from Dubois-Lacoste, Pagnozzi and Stützle (2017). IG_{rs} is method proposed by Ruiz and Stützle (2007), and IG_{lsps} is the state-of-the-art IG proposed by Dubois-Lacoste, Pagnozzi and Stützle (2017). A_0 to A_9 are the algorithms found with irace. The last column contains a combination of the best results obtained by algorithms A_0 to A_9. The best results for each instance group are highlighted in bold.

All ten algorithms have lower ARD than IG$_{rs}$. When compared to IG$_{lsps}$, A_0, A_2 to A_7 and A_9 have better performance, while A_1 and A_8 are equivalent. Note that A_1 and A_8 are the algorithms with a greedy insertion (*gi*) perturbation, while the others (except A_7) have the same perturbation strategy as IG$_{lsps}$. The algorithm with the best overall ARD is A_2, which is similar to IG$_{lsps}$. The main differences are the smaller value for d, which was recurrent in similar algorithms, and the different combinations of tie-breaking rules. These differences reduced the ARD by about 0.07% compared to IG$_{lsps}$, and by about 0.08% compared to IG$_{rs}$. Furthermore, the ARD of A_2 is only 0.01% higher than the combination of the best results. Using the best results per instance group as reference, we can see that the algorithms perform well over different dimensions, and there was no overfitting for specific groups.

We applied a Wilcoxon signed-rank test between IG$_{lsps}$ and each one of the algorithms from A_0 to A_9 to verify the statistical significance of the differences. For a significance level of 0.01 with Bonferroni correction, the test indicates that the difference is statistically significant for A_0, A_2, A_3, A_4, A_5 and A_9 ($p < 1.3 \times 10^{-8}$ in all cases).

We have repeated the previous experiments for the VRF-large benchmark. The results are presented in Table 3.10, where we compare the algorithms obtained via AAC to IG$_{rs}$ and IG$_{lsps}$.

Table 3.10: ARD for the VRF-large benchmark.

Inst.	IG_{rs}	IG_{lsps}	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
100×20	0.225	0.350	0.114	0.293	**0.033**	0.115	0.156	0.142	0.431	0.160	0.503	0.126	0.033
100×40	0.271	0.097	0.007	0.086	-0.050	**-0.072**	0.045	0.025	0.436	0.118	0.484	0.025	-0.072
100×60	0.279	0.018	0.052	0.062	-0.034	-0.021	**-0.039**	0.014	0.447	0.098	0.423	-0.018	-0.039
200×20	0.461	0.475	0.227	0.476	**0.126**	0.164	0.209	0.259	0.493	0.505	0.582	0.201	0.126
200×40	0.419	0.195	0.041	0.057	**-0.138**	0.016	-0.071	0.185	0.463	0.217	0.438	-0.007	-0.138
200×60	0.246	0.028	-0.052	0.002	**-0.226**	-0.134	-0.157	-0.010	0.303	0.116	0.395	-0.162	-0.226
300×20	0.274	0.243	0.124	0.299	0.130	0.110	0.141	0.167	0.219	0.328	0.326	**0.088**	0.088
300×40	0.510	0.257	0.060	0.176	0.004	0.050	**-0.075**	0.144	0.367	0.305	0.457	-0.033	-0.075
300×60	0.504	0.217	0.069	0.184	**-0.078**	0.011	-0.072	0.136	0.331	0.256	0.443	-0.040	-0.078
400×20	0.233	0.204	0.073	0.248	0.131	0.103	0.129	0.145	0.176	0.224	0.221	**0.062**	0.062
400×40	0.636	0.249	0.115	0.223	0.089	0.033	**-0.112**	0.120	0.373	0.387	0.303	0.075	-0.112
400×60	0.678	0.333	0.097	0.321	-0.030	0.064	**-0.044**	0.111	0.302	0.267	0.495	0.008	-0.044
500×20	0.192	0.145	0.062	0.170	0.088	0.085	0.101	0.136	0.140	0.175	0.165	**0.042**	0.042
500×40	0.388	0.196	0.017	0.098	0.044	-0.074	**-0.175**	0.003	0.216	0.293	0.143	-0.027	-0.175
500×60	0.732	0.480	0.253	0.504	0.079	0.167	**0.071**	0.234	0.366	0.481	0.615	0.194	0.071
600×20	0.108	0.088	0.025	0.117	0.044	0.063	0.048	0.072	0.059	0.105	0.090	**0.024**	0.024
600×40	0.253	0.106	-0.026	0.125	0.050	-0.068	**-0.133**	-0.005	0.169	0.358	0.107	-0.003	-0.133
600×60	0.877	0.538	0.224	0.520	0.161	0.129	**0.122**	0.174	0.350	0.482	0.558	0.235	0.122
700×20	0.108	0.092	0.040	0.126	0.047	0.074	0.075	0.102	0.067	0.087	0.117	**0.029**	0.029
700×40	0.061	0.030	-0.216	0.016	-0.017	-0.232	**-0.255**	-0.137	0.011	0.177	0.003	-0.126	-0.255
700×60	0.938	0.542	0.242	0.531	0.139	0.150	**0.081**	0.126	0.321	0.447	0.530	0.213	0.081
800×20	0.067	0.046	0.022	0.055	0.027	0.031	0.029	0.034	0.033	0.047	0.067	**0.008**	0.008
800×40	0.004	-0.018	**-0.257**	-0.006	-0.095	-0.197	-0.202	-0.125	-0.046	0.170	0.011	-0.135	-0.257
800×60	1.032	0.542	0.370	0.528	0.254	0.185	**0.166**	0.293	0.468	0.542	0.548	0.393	0.166
Avg.	0.396	0.227	0.070	0.217	0.032	0.031	**0.002**	0.098	0.271	0.264	0.334	0.049	-0.031

Instances have been grouped according to their dimensions in the first column. The results presented in this table are the ARD from the values in Table A.2. IG_{rs} is algorithm proposed by Ruiz and Stützle (2007), and IG_{lsps} is the method proposed by Dubois-Lacoste, Pagnozzi and Stützle (2017). A_0 to A_9 are the algorithms found with irace. The last column contains a combination of the best results obtained by algorithms A_0 to A_9. The best results for each instance group are highlighted in bold.

As in the previous experiments, all algorithms have lower ARD when compared to IG$_{rs}$, and seven algorithms have lower ARD when compared to IG$_{lsps}$. In general, there is a more substantial variation of the results, although they are still mostly better than the state of the art. Even though the training was carried out using instances more similar to those of the Taillard benchmark, the performance of the algorithms scales well to larger instances.

Overall, the results are similar to those of the first experiment. However, the algorithm with the best overall ARD is A_4. The best results per instance group were obtained mainly by three methods. A_2 had the best results for most of the instances with up to 200 jobs, A_9 was the best method when solving instances with 20 machines, and A_4 was the best when solving instances with 40 and 60 machines. We also noticed that, even though A_3 had the best results for only one instance group, its average results are equivalent to those of A_2. These four methods are similar, with the main differences lying in the initial solution construction and the combinations of tiebreakers. Moreover, A_4 is the only method that alternates two local search procedures. This indicates that such a strategy is more helpful when solving large instances such as those in this benchmark.

Once again, we applied a Wilcoxon signed-rank test between IG$_{lsps}$ and each one of the algorithms from A_0 to A_9. For a significance level of 0.01 with Bonferroni correction, the test indicates that the difference is statistically significant for A_0, A_2, A_3, A_4, A_5, A_8 and A_9 ($p < 4.6 \times 10^{-13}$), and A_7 ($p < 2 \times 10^{-5}$).

4 AUTOMATED DESIGN OF HEURISTICS FOR NON-PERMUTATION FLOW SHOPS

In this section, we present our methods for the NPFSSP. For both total completion time and makespan minimization, we implemented two-phase IG algorithms: during the first phase, the algorithms build and improve a permutation solution. In contrast, during the second phase, the search space is explored with procedures that evaluate job passing and generate non-permutation solutions. Pseudocode is presented in Algorithm 7.

Algorithm 7 IG for the NPFSSP.

Input: Perturbation intensities d, d^{NP}, time percentage *npf* allocated to the non-permutation phase
Output: Best solution found s^*

```
1:  function IG_NPFSSP
2:      π = ILS_PFSSP(d,npf)
3:      s = (π, ..., π)
4:      repeat
5:          s' = s
6:          Remove d^NP random jobs from s'
7:          for each removed job j do
8:              for each position k ∈ 1, ..., n do
9:                  Evaluate inserting j into position k without job passing
10:                 Evaluate inserting j into position k with anticipation
11:                 Evaluate inserting j into position k with delay
12:             end for
13:             Perform the best insertion
14:         end for
15:         s' = LocalSearch(s')
16:         if Accept(s, s') then
17:             s = s'
18:         end if
19:     until global time limit is reached
20:     return s*
21: end function
```

An intuitive approach is to start from a permutation solution and then evaluate non-permutation solutions through job passing. Likely, assigning very different sequences for the machines does not result in short schedules. Therefore, we focus on starting with a high-quality permutation schedule and then evaluating job passing with inversions on the order of pairs of adjacent jobs on some machines. Since our IG algorithms run for a certain time, we introduced the parameter *npf* to determine what fraction of the global time limit is to be allocated to the non-permutation phase, *e.g.* $npf = 0.65$ determines that

the permutation phase will take 35% of the total time, while the non-permutation phase will take 65% of the total time.

To obtain the permutation solution, we run an ILS for the PFSSP for some time and keep the best solution (line 2 in Algorithm 7). We then assign this solution to all m machines before starting the non-permutation phase (line 3 in Algorithm 7). The ILS for the PFSSP has its configurable components: the initial solution generator, perturbation function, the local search, and the acceptance criterion. We refer to it as an ILS and not IG because some of the perturbation options are not greedy procedures.

Regarding the non-permutation phase, the perturbation strategy consists of removing d^{NP} randomly selected jobs and reinserting them without job passing, with anticipation after a certain machine, or with delay after a certain machine (BENAVIDES; RITT, 2015), according to which one results in the best objective function value. In more detail, after removing the randomly selected jobs, we reinsert them into the solution in the same order they were removed, one at a time, into the best position for each one. First, we evaluate the insertion of the job j into position $k \in [1, n]$ on all machines. Then, for each machine $i = 2, \ldots, m-1$, we keep j at position k on machines 1 to i, and insert j into position $k-1$ on machines $i' > i$, *i.e.*, we anticipate j on machines after i. After that, for each machine $i = 2, \ldots, m-1$, we evaluate inserting j into position k on machines 1 to i, and into position $k+1$ on machines $i' > i$, *i.e.*, we delay the job on machines after i. We keep the same sequence of jobs for the first two machines, as optimal solutions have this characteristic (JOHNSON, 1954). When there are ties between different insertions for j, we prioritize those without anticipation nor delay, then insertions with anticipation. If there are ties for different k values, we use a configurable tiebreaker.

After the perturbation step, a local search is performed, and the acceptance criterion defines whether to accept or discard the new solution. Both the local search and the acceptance criterion are configurable components. We present the grammar and components for total completion time and makespan minimization in the following sections.

4.1 Methods to Minimize the Total Completion Time

4.1.1 Grammar and Components

The grammar for IG algorithms for total completion time minimization is presented in Figure 4.1. The numerical parameters were omitted, but are presented later in

this section. The symbol <ILS_PFSSP> represents the ILS for the PFSSP that is used to build the permutation solution in the first phase. Note that the part of the grammar concerning the PFSSP shares most of the components with the grammar of Section 3.1.1. As our work on the NPFSSP succeeded the one on the PFSSP, some additional components that were not present in the previous grammar (see Figure 3.1) were added later. These components are **(i)** the *RC* constructive heuristic, which has been mentioned in Section 3.2.1, although for makespan minimization. In the case of total completion time, *RC* builds r solutions with *NEHCsum* with a random tiebreaker for both the priority order and the insertion phase, and returns the best solution. **(ii)** We added the eight perturbation strategies presented in Section 3.2.1, adapted for total completion time minimization, and **(iii)** we implemented four acceptance criteria: Metropolis, late acceptance, record–to-record travel, and threshold acceptance. The first three were explained in Section 3.2.1. Threshold acceptance (*thr* in Figure 4.1) is similar to record-to-record travel, although with a deviation that is relative to the current solution. The deviation is calculated by multiplying a parameter *thres* $\in [0, 1]$ by the objective function value of the current solution. Since the addition of these components significantly increases the number of possible algorithms the grammar can generate, we repeated the computational experiments for the PFSSP with total completion time minimization, including these new components. We present the results in Appendix C.

Regarding the non-permutation phase, we chose a fixed perturbation strategy, described in the previous section. The configurable components are the local search and the acceptance criterion.

The local search of the non-permutation phase consists of a single procedure or an alternation between two local search procedures. We introduced the parameter $f_{\text{ls}}^{\text{NP}}$ to determine the frequency to apply the second procedure, *e.g.* $f_{\text{ls}}^{\text{NP}} = 10$ determines that the second local search procedure is to be applied every ten iterations. This was introduced with the different complexity of the methods in mind, as the insertion local search has a higher complexity and requires more computational time. The idea is that applying it with a low frequency may increase the general efficiency of the method. We have implemented five procedures as follows: **(i)** *insertNP* is a standard insertion local search with the addition of insertions with job passing. Similarly to the strategy used in the perturbation step, we evaluate insertions without job passing, with anticipation after a machine i, and with delay after a machine i, for all $i \in [2, m-1]$. We give priority to insertions without job passing, then with anticipation when breaking ties. The method also

Figure 4.1: A grammar of iterated greedy algorithms for the NPFSSP. For simplicity, the numerical parameters have been omitted.

```
 1 <START>       ::=  IG(<ILS_PFSSP>,<TIEBREAK>,<LS_NP>,
 2                    <ACCEPT>)
 3 <ILS_PFSSP>   ::=  ILS_PFSSP(<INI_SOL>,<PERTURB>,<LS>,
 4                    <ACCEPT>)
 5 <INI_SOL>     ::=  NEHCsum(<TIEBREAK>) | LR |
 6                    FRB5(<ORDER>,<TIEBREAK>) | RC | BSCH
 7 <ORDER>       ::=  noninc | nondec | KK1 | KK2
 8 <TIEBREAK>    ::=  KK1 | KK2 | first | last | random
 9 <PERTURB>     ::=  ri | gi(<TIEBREAK>) | rs | gs |
10                    ras | gi_asls(<TIEBREAK>) |
11                    ils_gi(<TIEBREAK>) |
12                    gi_ils(<TIEBREAK>)
13 <LS>          ::=  ϵ | <LS_PROC> |
14                    alternate(<LS_PROC>,<LS_PROC>)
15 <LS_PROC>     ::=  insertLS(<TIEBREAK>) | swapTasgetiren |
16                    swapInc | insertTasgetiren |
17                    lsTasgetiren | fpe | bpe | iRZ |
18                    riRZ | raiRZ | insertFPR | swapFirst |
19                    swapBest | swapR
20 <ACCEPT>      ::=  met | lac | rrt | thr
21 <LS_NP>       ::=  ϵ | <LS_PROC_NP> |
22                    alternate(<LS_PROC_NP>,<LS_PROC_NP>)
23 <LS_PROC_NP>  ::=  insertionNP(<TIEBREAK>) | RNASLS |
24                    ASLS | ASLS_r | ASLS_G8
```

uses a given tiebreaker for deciding between different insertion positions. The complexity of this method is a factor of m higher in comparison to *insertLS* (see Section 3.1.1), as it evaluates job passing on some of the machines. **(ii)** *ASLS* is an adjacent swap local search that swaps each job with its successor if any. It starts from the first job of the schedule and swaps each pair of jobs on all machines, then only on machines from 1 to i, and finally only on machines after i, for all $i \in [2, m-1]$. The method follows a best-improvement strategy, giving preference to neighbors without job passing and then with anticipation of jobs. It iterates until no improvements are found after a full neighborhood scan or after n_{ls}^{NP} full neighborhood scans. **(iii)** *ASLS_r* is a variant of *ASLS* in which the jobs are swapped in reverse order, *i.e.* each job is swapped with its predecessor, if any, starting from the last job of the schedule. **(iv)** *ASLS_G8* is another variant that works similarly. In this case, the jobs are swapped in the same order as in the G8 heuristic proposed by Rossi, Nagano and Tavares Neto (2016): first it swaps the jobs at positions k and $k+1$, for $k = 1, 3, 5, \ldots, n-1$, and then for $k = 2, 4, 6, \ldots, n-2$. Finally, **(v)** *RNASLS* is

similar to *ASLS*, but uses a reduced neighborhood instead. This method swaps adjacent jobs only at the beginning or at the end of blocks of consecutive operations on the critical path of a schedule, except the first two and the last jobs of the schedule, as in the NS neighborhood for makespan minimization. When minimizing the total completion time, we aim to minimize the completion time of every job, instead of only the last job in the schedule as happens when minimizing the makespan. Therefore, at each iteration j, we consider the j^{th} job as the last job in the schedule and apply a local search with the described reduced neighborhood to this partial schedule.

Finally, the implemented acceptance criteria are the same ones previously presented for the ILS to build the permutation solution, *i.e.*, Metropolis, record-to-record travel, threshold and late acceptance. There are different numerical parameters linked to the acceptance criteria of the permutation and non-permutation phases: the multiplier α for the Metropolis criterion, the threshold *thres*, the record-to-record travel deviation *rrtd*, and the list length l for late acceptance refer to the permutation phase, while the equivalent $\alpha^{\text{NP}}, thres^{\text{NP}}, rrtd^{\text{NP}}$ and l^{NP} refer to the non-permutation phase.

When using the parametric representation for the grammar, we have 18 categorical parameters. The parameters and their respective decisions regarding component selection are summarized in Table 4.1.

The parameters to be configured by irace are summarized in Table 4.2. The columns contain the name of the parameter, the type, which can be categorical, integer, ordinal or real. We discretized $r, d, d^{\text{NP}}, n_{\text{ls}}, n_{\text{ls}}^{\text{NP}}, l, l^{\text{NP}}, rrtd, rrtd^{\text{NP}}, thres, thres^{\text{NP}}$, and $f_{\text{ls}}^{\text{NP}}$ and defined them as ordinal parameters to reduce the search space for irace. Finally, the last two columns contain the domain and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

4.1.2 Computational Experiments

In the computational experiments, we used irace to configure the components and numerical parameters. First, we ran irace ten times and selected the best algorithm of each run, and then we evaluated the algorithms on Taillard's (TAILLARD, 1993) and VRF-large (VALLADA; RUIZ; FRAMIÑAN, 2015) benchmarks.

Each irace run had a budget of 10^5 candidate evaluations, and each candidate had a time limit of $60nm$ milliseconds. The training set contained 120 randomly generated instances with the same dimensions as those of the Taillard benchmark, with processing

Table 4.1: Categorical parameters required to instantiate algorithms from the grammar.

Parameter	Decision
	Permutation phase
ini_sol	Constructive heuristic to build the initial solution.
order	Criterion for job ordering during the construction of the initial solution.
tb_ini_sol	Tiebreaker for the construction of the initial solution.
tb_perturb	Tiebreaker for the perturbation functions that perform greedy insertions.
tb_ls_proc_1	Tiebreaker for the first local search procedure.
tb_ls_proc_2	Tiebreaker for the second local search procedure.
perturb	Perturbation strategy.
ls	Number of local search procedures to be applied.
ls_proc_1	First local search procedure.
ls_proc_2	Second local search procedure.
accept	Acceptance criterion.
	Non-permutation phase
np_tb_perturb	Tiebreaker for the perturbation.
np_tb_ls_proc_1	Tiebreaker for the first local search procedure.
np_tb_ls_proc_2	Tiebreaker for the second local search procedure.
np_ls	Number of local search procedures to be applied.
np_ls_proc_1	First local search procedure.
np_ls_proc_2	Second local search procedure.
np_accep	Acceptance criterion.

times in the interval $[1, 99]$. We used version 3.0 of irace, with all parameters set to default values.

The best algorithm of each of the ten runs is presented in Table 4.3, named IG_0 to IG_9. Overall, we can see that the algorithms have many similarities. Regarding the permutation phase, all algorithms use *BSCH* to build an initial permutation solution, which was expected since the results in Fernández-Viagas and Framiñan (2017) indicate that the method considerably outperforms the other options. For the perturbation, *gi* was selected seven times, with d between five and seven, except for IG_5. There was a significant variation in the tiebreakers, and most options were selected at least once, except for KK2. This indicates that they yield similar results during the perturbation step. For the local search, four algorithms perform a single procedure, while the other six alternate between two procedures. The pairwise exchange methods (*fpe* and *bpe*), and *iRZ* and its variants (*raiRZ* and *insertFPR*) were selected more frequently. Note that when there is an alternation between two procedures, they are always an insertion-based and a swap-based procedure, suggesting that they have complementary characteristics. Finally, record-to-record travel is the acceptance criterion of seven algorithms, with a deviation between 50 and 80, except for IG_2 and IG_3. As record-to-record travel does not use a relative value

for the deviation, the low values indicate that the algorithms will accept worse solutions more easily for small instances, since they have a smaller objective function value than large instances. When the search space is smaller, more diversification yields better results, while it is better to intensify the search when the search space is larger. Moreover, these low deviation values are mostly equivalent to zero when solving large instances due to the much bigger magnitude of the objective function value.

Regarding the non-permutation phase, we can see that all algorithms allocate more time for this phase, with IG_0 allocating the least amount, 58%, and IG_5 the biggest amount, 81%. All algorithms have a similar perturbation intensity, and the tiebreaker is usually *KK1* or *KK2*, except for IG_3, IG_5 and IG_6. The local search phase is also similar for all algorithms, since they all perform the *ASLS* method, although with different orders. Furthermore, seven of them alternate between different orders. Finally, record-to-record travel is the most frequently selected acceptance criterion, with low deviation values, as in the permutation phase, except for IG_6, which never accepts worse solutions.

Table 4.2: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
ini_sol	Categorical	*LR*, *NEHCsum*, *FRB5*, *BSCH*, *RC*	
order	Categorical	*noninc*, *nondec*, *KK1*, *KK2*	*ini_sol* $\in \{$*NEHCsum*, *FRB5*$\}$
tb_ini_sol	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in \{$*NEHCsum*, *FRB5*$\}$
tb_perturb	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*perturb* $\in \{$*gi*, *gi_asls*, *ils_gi*, *gi_ils*$\}$
tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $\in \{$*insertLS*$\}$
tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $\in \{$*insertLS*$\}$
perturb	Categorical	*ri*, *gi*, *rs*, *gs*, *ras*, *gi_asls*, *ils_gi*, *gi_ils*	
ls	Categorical	$0, 1, 2$	
ls_proc_1	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	*ls* $\in \{1, 2\}$
ls_proc_2	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	*ls* $= 2$
accept	Categorical	*met*, *lac*, *rrt*, *thr*	
np_tb_perturb	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	
np_tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*np_ls_proc_1* $=$ *insertionNP*
np_tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*np_ls_proc_2* $=$ *insertionNP*
np_ls	Categorical	$0, 1, 2$	
np_ls_proc_1	Categorical	*insertionNP*, *ASLS*, *ASLS_r*, *ASLS_G8*, *ARNASLS*	*np_ls* $\in \{1, 2\}$
np_ls_proc_2	Categorical	*insertionNP*, *ASLS*, *ASLS_r*, *ASLS_G8*, *ARNASLS*	*np_ls* $= 2$
np_accep	Categorical	*met*, *lac*, *rrt*, *thr*	
npf	Real	$[0, 1.0]$	
r	Ordinal	$1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 1000$	*ini_sol* $=$ *RC*
d, d^{NP}	Integer	$[1, 20]$	
$n_{\text{ls}}, n_{\text{ls}}^{\text{NP}}$	Ordinal	$1, 2, 3, 4, \infty$	*ini_sol* $=$ *FRB5* or *ls* $\in \{1, 2\}$ or *perturb* $=$ *ils_gi*, *np_ls* $\in \{1, 2\}$
$\alpha, \alpha^{\text{NP}}$	Real	$[0.01, 1.0]$	*accept* $=$ *met*, *np_accept* $=$ *met*
l, l^{NP}	Ordinal	$1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000$	*accept* $=$ *lac*, *np_accept* $=$ *lac*
$rrtd, rrtd^{\text{NP}}$	Ordinal	$0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500$	*accept* $=$ *rrt*, *np_accept* $=$ *rrt*
$thres, thres^{\text{NP}}$	Ordinal	$0, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5$	*accept* $=$ *thr*, *np_accept* $=$ *thr*
$f_{\text{ls}}^{\text{NP}}$	Ordinal	$2, 3, 4, 5, 10$	*np_ls* $= 2$

The columns contain the name of the parameter, the type specified in irace, the domain and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

Table 4.3: Automatically designed algorithms for minimizing the total completion time on the PFSSP.

	Permutation phase									Non-permutation phase								
Alg.	Init.	Pert.	d	Tie	Proc.1	Proc.2	n_{ls}	Accep.	$\alpha, rrtd, l$	Time	d^{NP}	Tie	Proc.1	Proc.2	$f_{\text{ls}}^{\text{NP}}$	$n_{\text{ls}}^{\text{NP}}$	Accep.	$rrtd^{\text{NP}}, l^{\text{NP}}$
IG_0	*BSCH*	*gi_asls*	6	*KK1*	*fpe*	*raiRZ*	∞	*rrt*	80	0.58	5	*KK2*	*ASLS*	*ASLS_r*	10	∞	*lac*	25
IG_1	*BSCH*	*gi*	7	*KK1*	*fpe*	-	-	*lac*	200	0.60	4	*KK2*	*ASLS_r*	*ASLS*	3	∞	*rrt*	30
IG_2	*BSCH*	*gi*	7	*random*	*fpe*	-	-	*rrt*	200	0.60	4	*KK1*	*ASLS*	*ASLS_r*	3	4	*rrt*	70
IG_3	*BSCH*	*gi*	5	*random*	*raiRZ*	*bpe*	3	*rrt*	500	0.74	5	*first*	*ASLS*	*ASLS_G8*	3	∞	*rrt*	60
IG_4	*BSCH*	*gs*	3	-	*raiRZ*	*bpe*	4	*rrt*	80	0.76	5	*KK2*	*ASLS_G8*	*ASLS*	3	∞	*rrt*	30
IG_5	*BSCH*	*gi*	11	*last*	*bpe*	*insertFPR*	-	*met*	0.24	0.81	5	*last*	*ASLS_G8*	*ASLS_r*	4	4	*rrt*	10
IG_6	*BSCH*	*ras*	4	-	*iRZ*	*bpe*	∞	*rrt*	80	0.62	5	*last*	*ASLS_G8*	*ASLS_r*	5	∞	*rrt*	0
IG_7	*BSCH*	*gi*	6	*KK1*	*fpe*	-	-	*rrt*	50	0.73	5	*KK2*	*ASLS*	-	-	4	*rrt*	60
IG_8	*BSCH*	*gi*	7	*last*	*swapInc*	-	-	*rrt*	60	0.78	5	*KK2*	*ASLS*	-	-	∞	*rrt*	40
IG_9	*BSCH*	*gi*	5	*first*	*swapTasgetiren*	*insertFPR*	-	*met*	0.19	0.79	4	*KK1*	*ASLS_r*	-	-	∞	*rrt*	40

Each row contains the components and parameter values for one of the algorithms obtained with irace, named IG_0 to IG_9 in the first column. The following nine columns refer to the permutation phase and contain the constructive heuristic to build the initial solution, perturbation strategy and intensity d, the tiebreaker, first and second local search procedures, n_{ls} value, the acceptance criterion, and its respective parameter value, depending on the acceptance criterion. The other columns refer to the non-permutation phase and contain the percentage of time allocated for the non-permutation phase, the perturbation intensity d^{NP}, the tiebreaker, first and second local search procedures, the values for $f_{\text{ls}}^{\text{NP}}$ and $n_{\text{ls}}^{\text{NP}}$, the acceptance criterion, and the value for its respective parameter.

We evaluated algorithms IG_0 to IG_9 against the benchmark of Taillard (1993). The algorithms were implemented in C++ and compiled with the GNU C++ compiler version 7.4 with an optimization level of 3. Each algorithm ran for $60nm$ milliseconds with 10 replications on a computer with two Intel Xeon E5-2697 v2 processors (12 physical cores each) at 2.7 GHz, running Ubuntu 18.04.3. We present the results grouped by instance dimensions as the average relative deviation (ARD) in percent from the values in Table A.1.

First, we use the best method for the PFSSP from Section 3.1.2 (algorithm A_7) as a baseline for the comparison with non-permutation schedules. The results are presented in column "A_7". Regarding the NPFSSP, we compare the algorithms to the state-of-art IG of Benavides and Ritt (2015), which we instantiated in our solver. This IG starts with a permutation schedule built by an ILS that runs for half of a global time limit, while this ILS itself is initialized with a schedule built with the *LR* constructive heuristic. Then the destruction phase consists of removing d randomly selected jobs, whereas the construction phase evaluates the insertion of each job without job passing, with anticipation after some machine i, or with delay after some machine i, with $i = 2, \ldots, m-1$. The results are presented in column "$B\&R$". Furthermore, since the recently proposed *BSCH* outperformed other similar methods, we evaluated replacing *LR* with *BSCH* in the method of Benavides and Ritt (2015) to make clear what portion of the improvements are due to simply using *BSCH*, and what is due to the new algorithms found with irace. We refer to the resulting algorithm as "$B\&R'$" in Table 4.4. All these algorithms were run for the same time limit as the algorithms found with irace, which are presented in columns "IG_0" to "IG_9". In addition, column "Best" contains a combination of the best results per instance group to provide a theoretical estimate of the best results obtained with the components from the grammar if we were able to combine the best characteristics of the ten algorithms in a single method.

Table 4.4: ARD for the Taillard benchmark.

Inst.	A_7	$B\&R$	$B\&R'$	IG_0	IG_1	IG_2	IG_3	IG_4	IG_5	IG_6	IG_7	IG_8	IG_9	Best
20×5	0.000	-0.667	-0.662	-0.673	-0.723	-0.774	-0.787	-0.735	**-0.796**	-0.661	-0.777	-0.765	-0.755	-0.796
20×10	0.000	-1.288	-1.290	-1.667	-1.681	-1.785	-1.800	-1.734	**-1.869**	-1.600	-1.814	-1.755	-1.746	-1.869
20×20	0.000	-1.129	-1.211	-1.964	-1.876	-1.998	-2.152	-2.059	**-2.173**	-1.954	-2.048	-2.066	-2.000	-2.173
50×5	0.182	0.081	0.039	-0.001	0.010	0.007	0.025	-0.009	0.013	0.018	-0.009	-0.011	**-0.016**	-0.016
50×10	0.225	-0.031	-0.065	0.057	0.187	0.061	**-0.192**	0.018	-0.150	0.046	0.127	0.096	-0.019	-0.192
50×20	0.313	-0.398	-0.346	-0.616	-0.452	-0.487	**-0.742**	-0.582	-0.720	-0.553	-0.441	-0.512	-0.609	-0.742
100×5	0.105	0.520	0.031	**-0.004**	0.005	0.001	0.017	0.002	0.025	0.012	0.002	0.003	0.012	-0.004
100×10	0.234	0.526	0.036	0.001	0.015	0.003	0.015	0.009	0.021	0.005	0.021	0.009	**-0.004**	-0.004
100×20	0.406	0.389	-0.193	-0.331	-0.333	-0.318	**-0.394**	-0.350	-0.370	-0.374	-0.287	-0.339	-0.393	-0.394
200×10	-0.671	0.422	-0.760	-0.792	-0.784	-0.791	-0.791	-0.784	-0.779	-0.785	-0.787	**-0.793**	-0.787	-0.793
200×20	-0.875	0.043	-1.169	-1.257	-1.266	-1.262	-1.267	-1.281	-1.279	**-1.289**	-1.254	-1.265	-1.274	-1.289
500×20	-1.892	0.124	-1.977	-2.025	-2.022	-2.022	-2.024	-2.026	-2.023	**-2.029**	-2.023	-2.023	-2.025	-2.029
Avg.	-0.164	-0.117	-0.631	-0.773	-0.743	-0.780	-0.841	-0.794	**-0.842**	-0.764	-0.774	-0.785	-0.801	-0.858

A_7 is the best method for the PFSSP from Section 3.1.2, $B\&R$ is the method for the NPFSSP proposed by Benavides and Ritt (2018), and $B\&R'$ is the modified version of $B\&R$. IG_0 to IG_9 are the algorithms found with irace. The last column contains a combination of the best results obtained by algorithms IG_0 to IG_9. The best results for each instance group are highlighted in bold.

Most of the values are negative because the results improve over the upper bounds from Table A.1, which are bounds for the PFSSP. We can see that most methods for the NPFSSP have considerably lower ARD than the method for the PFSSP. The exception is $B\&R$, which has higher ARD than A_7 even though it only considers permutation schedules. This is due to *BSCH*, as we can see that A_7 has a much lower ARD on the three largest instance groups, where *BSCH* is superior in comparison to *LR*. The overall ARD improved by about $0.5\,\%$ when we replaced *LR* with *BSCH* in the reference method.

All algorithms found with irace had lower ARD than the reference methods, with improvements between $0.62\,\%$ and $0.72\,\%$ over the method for the PFSSP. Overall, all generated algorithms had a very similar performance for the three largest groups of instances. Using the best results as a reference, we did not identify algorithms overfitted for certain instance dimensions. We applied Wilcoxon signed-rank tests with Bonferroni correction between $B\&R'$ and each one of the obtained methods, and the tests indicated that the difference is statistically significant in all cases ($99\,\%$ confidence, $p < 1.12 \times 10^{-10}$ in all cases).

IG_3 and IG_5 are the algorithms with the best results on average and good performance across the different instance dimensions. They have many similarities: during the permutation phase, both perform greedy insertions, although IG_5 has a stronger perturbation and both alternate between insertion and swap local search procedures. As for the non-permutation phase, they use similar time allocation, same perturbation intensity, although with different tiebreakers, both alternate between different orders for the *ASLS* procedure on similar intervals, and both accept solutions with a record-to-record travel criterion with low deviation values. They have similar performance all instance groups. Furthermore, we can see that the advantage IG_3 and IG_5 have over the other algorithms is gained on 20×20, 50×10 and 50×20 instances, *i.e.*, they perform considerably better than the others for medium-sized instances with a lower n/m ratio.

We noticed that although IG_4 is very similar to IG_3, one of the best methods, it performed slightly worse. The main differences are its perturbation strategy during the permutation phase, which greedily swaps three pairs of jobs, its significantly lower $rrtd$ for the permutation phase, and the tiebreaker and slightly lower $rrtd$ for the non-permutation phase. We investigated the solutions obtained at the end of the permutation phase by both methods and noticed that the IG_4 found better permutation solutions for about $44\,\%$ of the instances, while IG_3 for about $31\,\%$ of the instances. Therefore, the difference in performance is probably due to a more efficient non-permutation phase in

IG_3, with $KK2$ tiebreaker and the slightly higher $rrtd$.

We performed experiments with the VRF-large benchmark to see how well the algorithms perform when solving larger instances than those of the Taillard benchmark. The results are shown in Table 4.5 as the ARD from the values in Table A.2.

Again, we can see that the methods for the NPFSSP had considerably lower ARD, except for $B\&R$, which performed worse than A_7 due to *BSCH* significantly outperforming *LR* when solving large instances. When replacing *LR* with *BSCH*, the overall ARD decreased by more than 2%. In this benchmark, all the algorithms found with irace had similar ARD and consistent performance across instance dimensions, as evidenced by the small differences to the best results per group. The improvements ranged from $0.44\,\%$ to $0.47\,\%$ over A_7, and all had lower ARD than $B\&R'$. The results showed that even though the training instances were similar to those of Taillard's benchmark, the algorithms generalized well to the larger instances. Again, we applied Wilcoxon signed-rank tests with Bonferroni correction between $B\&R'$ and each one of the obtained methods, and the tests indicated that the difference is statistically significant in all cases ($99\,\%$ confidence, $p < 2.2 \times 10^{-16}$ in all cases).

Table 4.5: ARD for the VRF-large benchmark.

Inst.	A_7	$B\&R$	$B\&R'$	IG_0	IG_1	IG_2	IG_3	IG_4	IG_5	IG_6	IG_7	IG_8	IG_9	Best
100×20	0.046	0.037	-0.521	-0.692	-0.668	-0.684	**-0.750**	-0.708	-0.714	-0.709	-0.648	-0.678	-0.733	-0.750
100×40	0.062	-0.241	-0.606	-0.786	-0.846	-0.847	-0.926	**-0.927**	-0.887	-0.910	-0.815	-0.886	-0.923	-0.927
100×60	0.063	-0.282	-0.683	-0.835	-0.887	-0.894	-0.950	-0.965	**-0.968**	-0.961	-0.855	-0.900	-0.949	-0.968
200×20	0.010	1.196	-0.321	-0.417	-0.426	-0.428	-0.435	-0.441	**-0.443**	-0.442	-0.432	-0.427	-0.439	-0.443
200×40	0.028	0.892	-0.535	-0.666	-0.709	-0.713	-0.723	-0.728	-0.728	**-0.735**	-0.702	-0.710	-0.725	-0.735
200×60	0.019	0.723	-0.580	-0.693	-0.731	-0.751	-0.763	**-0.777**	-0.775	-0.762	-0.749	-0.727	-0.758	-0.777
300×20	0.008	1.682	-0.207	-0.281	-0.282	-0.289	-0.291	-0.291	-0.294	**-0.297**	-0.291	-0.286	-0.292	-0.297
300×40	0.016	1.625	-0.461	-0.620	-0.641	-0.646	-0.650	-0.651	**-0.657**	**-0.657**	-0.641	-0.638	**-0.657**	-0.657
300×60	0.010	1.425	-0.499	-0.651	-0.662	-0.682	-0.685	-0.689	**-0.698**	-0.677	-0.669	-0.662	-0.674	-0.698
400×20	0.005	2.030	-0.123	-0.178	-0.179	-0.176	-0.178	-0.182	-0.179	**-0.183**	-0.180	-0.179	-0.181	-0.183
400×40	0.009	2.174	-0.326	-0.487	-0.502	-0.503	-0.500	**-0.511**	-0.507	**-0.511**	-0.502	-0.502	-0.510	-0.511
400×60	0.020	2.023	-0.367	-0.554	-0.558	-0.565	-0.565	**-0.579**	-0.578	-0.573	-0.561	-0.559	-0.568	-0.579
500×20	0.003	2.253	-0.091	-0.146	-0.148	-0.146	-0.148	**-0.151**	-0.146	**-0.151**	-0.146	-0.150	**-0.151**	-0.151
500×40	0.011	2.666	-0.251	-0.434	-0.435	-0.422	-0.441	-0.442	-0.429	-0.441	-0.416	-0.433	**-0.444**	-0.444
500×60	0.007	2.579	-0.299	-0.544	-0.548	-0.507	-0.552	-0.553	-0.517	**-0.554**	-0.504	-0.546	-0.550	-0.554
600×20	0.002	2.178	-0.060	-0.101	-0.104	-0.102	-0.100	-0.103	-0.101	**-0.105**	-0.101	-0.101	-0.102	-0.105
600×40	0.006	2.693	-0.192	-0.390	-0.390	-0.362	-0.397	-0.394	-0.368	**-0.398**	-0.360	-0.392	-0.394	-0.398
600×60	0.008	2.831	-0.214	-0.458	-0.464	-0.418	-0.489	**-0.492**	-0.438	-0.476	-0.417	-0.481	-0.487	-0.492
700×20	0.002	2.084	-0.042	-0.078	-0.076	-0.078	-0.077	**-0.079**	-0.077	**-0.079**	-0.078	-0.076	-0.075	-0.079
700×40	0.003	2.854	-0.138	-0.341	-0.343	-0.291	**-0.347**	-0.344	-0.300	-0.344	-0.290	-0.345	-0.345	-0.347
700×60	0.008	3.096	-0.150	-0.377	-0.369	-0.326	**-0.403**	-0.402	-0.338	-0.387	-0.329	-0.394	**-0.403**	-0.403
800×20	0.002	1.839	-0.035	-0.071	-0.067	-0.068	-0.070	-0.069	-0.069	**-0.072**	-0.068	-0.065	-0.067	-0.072
800×40	0.003	2.853	-0.095	-0.290	-0.290	-0.235	**-0.311**	-0.301	-0.242	-0.292	-0.235	-0.301	-0.310	-0.311
800×60	0.007	3.197	-0.117	-0.344	-0.339	-0.290	**-0.371**	-0.366	-0.304	-0.349	-0.290	-0.364	-0.368	-0.371
Avg.	0.015	1.850	-0.288	-0.435	-0.444	-0.434	-0.463	**-0.464**	-0.448	-0.461	-0.428	-0.450	-0.463	-0.469

A_7 is the best method for the PFSSP from Section 3.1.2, $B\&R$ is the method for the NPFSSP proposed by Benavides and Ritt (2018), and $B\&R'$ is the modified version of $B\&R$. IG_0 to IG_9 are the algorithms found with irace. The last column contains a combination of the best results obtained by algorithms IG_0 to IG_9. The best results for each instance group are highlighted in bold.

4.2 Methods to Minimize the Makespan

The algorithms for makespan minimization in the NPFSSP have the same overall form as those presented in Section 4.1, i.e., a two-phase IG in which a permutation solution is built during the first phase, and methods that consider job passing are applied during the second phase. The algorithmic components and the computational experiments are presented in the following sections.

4.2.1 Grammar and Components

The grammar is presented in Figure 4.2. Note that the grammar for ILS methods for the PFSSP from Section 3.2.1 is contained within this one. In a comparison to the grammar in Section 3.2.1, two new components were added, namely the *FF* tiebreaker of Fernández-Viagas and Framiñan (2014), and threshold acceptance (*thr* in the grammar). These components can be used in any of the two phases. Moreover, we introduce a parameter f_{ls} to determine the frequency to apply the second local search procedure, if any, during the permutation phase. We repeated the same experiments as in Section 3.2.2 with these additions, and present the results in Appendix D.

As mentioned earlier, we used a fixed approach for the perturbation in which d^{NP} randomly selected jobs are removed and reinserted without job passing, with anticipation, or with delay after machine i, for all $i \in [2, m-1]$. Regarding the local search, we apply a single procedure or alternate between two local search procedures. Moreover, we allow procedures with tiebreakers to be selected twice if with different rules. Parameter $f_{\text{ls}}^{\text{NP}}$ determines the frequency to apply the second procedure, if any, and $n_{\text{ls}}^{\text{NP}}$ limits the number of full neighborhood evaluations. We implemented the following four local search procedures: **(i)** *insertionNP* is an insertion-based local search in which each job is removed and reinserted into the position that minimizes the makespan. The jobs are reinserted as in the perturbation step, i.e., without job passing, with anticipation, or with delay. **(ii)** *RNB* is a swap-based local search with a reduced neighborhood (BENAVIDES; RITT, 2018). The reduced neighborhood is the same one used in the *NS* local search for the PFSSP (see Section 3.2.1), i.e., it contains pairs of adjacent jobs that have consecutive operations at the beginning or at the end of blocks of jobs on the critical path. The method evaluates swapping each pair of jobs in the reduced neighborhood on all machines, before machine i, and after machine i, for all $i \in [2, m-1]$. **(iii)** *PcNP*: similar to *RNB*. However, instead

Figure 4.2: A grammar of iterated greedy algorithms for the NPFSSP. For simplicity, the numerical parameters have been omitted.

```
 1 <START>       ::=  IG(<ILS_PFSSP>,<TIEBREAK>,<LS_NP>,
 2                    <ACCEPT>)
 3 <ILS_PFSSP>   ::=  ILS_PFSSP(<INI_SOL>,<PERTURB>,<LS>,
 4                    <ACCEPT>)
 5 <INI_SOL>     ::=  NEH(<TIEBREAK>) | RC |
 6                    FRB5(<ORDER>,<TIEBREAK>)
 7 <ORDER>       ::=  noninc | nondec | KK1 | KK2
 8 <TIEBREAK>    ::=  FF | KK1 | KK2 | first | last | random
 9 <PERTURB>     ::=  ri | gi(<TIEBREAK>) | rs | gs |
10                    ras | gi_asls(<TIEBREAK>) |
11                    ils_gi(<TIEBREAK>) |
12                    gi_ils(<TIEBREAK>)
13 <LS>          ::=  ϵ | <LS_PROC> |
14                    alternate(<LS_PROC>,<LS_PROC>)
15 <LS_PROC>     ::=  insertion(<TIEBREAK>) | NS |
16                    Pc(<TIEBREAK>)
17 <ACCEPT>      ::=  met | lac | rrt | thr
18 <LS_NP>       ::=  ϵ | <LS_PROC_NP> |
19                    alternate(<LS_PROC_NP>,<LS_PROC_NP>)
20 <LS_PROC_NP>  ::=  insertionNP(<TIEBREAK>) | RNB |
21                    PcNP(<TIEBREAK>) | ASLS
```

of swapping each pair of jobs, they are removed and reinserted into the best positions, considering insertions without job passing, with anticipation, and with delay. **(iv)** *ASLS*: also similar to *RNB*, but with the full adjacent-swap neighborhood, i.e., all pairs of adjacent jobs are swapped on all machines, only on machines before i, and only on machines after i, for all $i \in [2, m-1]$.

The acceptance criterion options are the same four as in the permutation phase: Metropolis, late acceptance, record-to-record travel, and threshold acceptance. Each criterion is linked to a numerical parameter to control the acceptance, namely α^{NP}, $thres^{\text{NP}}$, $rrtd^{\text{NP}}$ and l^{NP}. Finally, the time distribution between the two phases is controlled by the parameter *npf*.

The set with the necessary categorical parameters to instantiate an algorithm from the grammar contains 18 parameters. We show a summary and the decision linked to each parameter in Table 4.6. The list with all the parameters configured with irace, including numerical parameters, is shown in Table 4.7.

Table 4.6: Categorical parameters required to instantiate algorithms from the grammar.

Parameter	Decision
	Permutation phase
ini_sol	Constructive heuristic to build the initial solution.
order	Criterion for job ordering during the construction of the initial solution.
tb_ini_sol	Tiebreaker for the construction of the initial solution.
tb_perturb	Tiebreaker for the perturbation functions that perform greedy insertions.
tb_ls_proc_1	Tiebreaker for the first local search procedure.
tb_ls_proc_2	Tiebreaker for the second local search procedure.
perturb	Perturbation strategy.
ls	Number of local search procedures to be applied.
ls_proc_1	First local search procedure.
ls_proc_2	Second local search procedure.
accept	Acceptance criterion.
	Non-permutation phase
np_tb_perturb	Tiebreaker for the perturbation.
np_tb_ls_proc_1	Tiebreaker for the first local search procedure.
np_tb_ls_proc_2	Tiebreaker for the second local search procedure.
np_ls	Number of local search procedures to be applied.
np_ls_proc_1	First local search procedure.
np_ls_proc_2	Second local search procedure.
np_accep	Acceptance criterion.

4.2.2 Computational Experiments

In this section, we present the computational experiments on the NPFSSP with makespan minimization. We ran irace ten times and selected the best algorithm of each run. Each algorithm was evaluated against the benchmarks of Taillard (1993) and Vallada, Ruiz and Framiñan (2015) (VRF-large).

Each irace run had a budget of 10^5 candidate evaluations, and each candidate had a time limit of $30nm$ milliseconds. The training set contained 120 randomly generated instances with the same dimensions as those in the Taillard benchmark, with processing times in the interval $[1, 99]$. We used version 3.0 of irace, with all parameters set to default values.

We present the best algorithm of each run in Table 4.3, named IG_0 to IG_9. Starting with the permutation phase, we can see that all the constructive heuristics were selected, with varying orders and tiebreakers. For the perturbation step, six different strategies were selected, mostly with the *FF* tiebreaker. The algorithms with the perturbation that applies the insertion local search to the partial solution followed by the greedy insertion of he removed jobs (*ils_gi*) have a low perturbation intensity ($d = 1$ or $d = 4$), a pattern that

can also be seen in Section 3.2.2 (see Table 3.8). The other strategies have perturbation intensities $d \in [4, 8]$, which are compatible with values in the literature. All algorithms perform the *insertLS* local search. Three of them alternate between different tiebreakers, and a single algorithm uses *NS* in addition to *insertLS*. The *FF* tiebreaker was selected more frequently, while *KK1* and *KK2* were the other selected tiebreakers. Metropolis and late acceptance were the most selected acceptance criteria.

Regarding the non-permutation phase, all algorithms allocate most of the time for this latter phase, with *npf* ranging from 0.52 to 0.81. For the perturbation step, *random* was selected more frequently, and the values for d^{NP} were very close, ranging from two to four. Regarding the local search, all algorithms use *RNB* as the single procedure. Finally, Metropolis is the acceptance criterion of seven algorithms, while the other three use late acceptance.

Table 4.7: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
ini_sol	Categorical	*NEH*, *FRB5*, *RC*	
order	Categorical	*noninc*, *nondec*, *KK1*, *KK2*	*ini_sol* $\in \{$*NEH*, *FRB5*$\}$
tb_ini_sol	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in \{$*NEH*, *FRB5*$\}$
tb_perturb	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*perturb* $\in \{$*gi*, *gi_asls*, *ils_gi*, *gi_ils*$\}$
tb_ls_proc_1	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $\in \{$*insertLS*, *Pc*$\}$
tb_ls_proc_2	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $\in \{$*insertLS*, *Pc*$\}$
perturb	Categorical	*ri*, *gi*, *rs*, *gs*, *ras*, *gi_asls*, *ils_gi*, *gi_ils*	
ls	Categorical	$0, 1, 2$	
ls_proc_1	Categorical	*insertLS*, *NS*, *Pc*	*ls* $\in \{1, 2\}$
ls_proc_2	Categorical	*insertLS*, *NS*, *Pc*	*ls* $= 2$
accept	Categorical	*met*, *lac*, *rrt*, *thr*	
np_tb_perturb	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	
np_tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*np_ls_proc_1* $\in \{$*insertionNP*, *PcNP*$\}$
np_tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*np_ls_proc_2* $\in \{$*insertionNP*, *PcNP*$\}$
np_ls	Categorical	$0, 1, 2$	
np_ls_proc_1	Categorical	*insertionNP*, *RNB*, *ASLS*, *PcNP*	*np_ls* $\in \{1, 2\}$
np_ls_proc_2	Categorical	*insertionNP*, *RNB*, *ASLS*, *PcNP*	*np_ls* $= 2$
np_accep	Categorical	*met*, *lac*, *rrt*, *thr*	
npf	Real	$[0, 1.0]$	
r	Ordinal	$1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,$ $300, 400, 500, 1000$	*ini_sol* $=$ *RC*
d, d^{NP}	Integer	$[1, 20]$	
$n_{\text{ls}}, n_{\text{ls}}^{\text{NP}}$	Ordinal	$1, 2, 3, 4, \infty$	*ini_sol* $=$ *FRB5* or *ls* $\in \{1, 2\}$ or *perturb* $=$ *ils_gi*, *np_ls* $\in \{1, 2\}$
$\alpha, \alpha^{\text{NP}}$	Real	$[0.01, 1.0]$	*accept* $=$ *met*, *np_accept* $=$ *met*
l, l^{NP}	Ordinal	$1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600,$ $700, 800, 900, 1000$	*accept* $=$ *lac*, *np_accept* $=$ *lac*
$rrtd, rrtd^{\text{NP}}$	Ordinal	$0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,$ $300, 400, 500$	*accept* $=$ *rrt*, *np_accept* $=$ *rrt*
$thres, thres^{\text{NP}}$	Ordinal	$0, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.02,$ $0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,$ $0.2, 0.3, 0.4, 0.5$	*accept* $=$ *thr*, *np_accept* $=$ *thr*
$f_{\text{ls}}, f_{\text{ls}}^{\text{NP}}$	Ordinal	$2, 3, 4, 5, 10$	*ls* $= 2$, *np_ls* $= 2$

The columns contain the name of the parameter, the type specified in irace, the domain, and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

Table 4.8: Automatically designed algorithms for minimizing the makespan on the NPFSSP.

	Permutation phase															Non-permutation phase						
	Initial solution					Perturbation			Local search						Acceptance		Perturbation		Local search		Acceptance	
Alg.	CH	Order	Tie	r	*rev*	Method	Tie	d	Proc.1	Tie1	Proc.2	Tie2	f_{ls}	n_{ls}	Accep.	$\alpha, rrtd, l$	*npf*	Tie	d^{NP}	Proc.	$n_{\text{ls}}^{\text{NP}}$	Accep.	$\alpha^{\text{NP}}, l^{\text{NP}}$
IG_0	*NEH*	*noninc*	*FF*	-	yes	*gs*	-	8	*insertLS*	*FF*	*insertLS*	*KK2*	5	∞	*lac*	50	0.57	*KK2*	3	*RNB*	∞	*lac*	300
IG_1	*NEH*	*nondec*	*last*	-	no	*ils_gi*	*FF*	1	*insertLS*	*KK1*	-	-	-	2	*met*	0.85	0.61	*rand*	4	*RNB*	4	*lac*	300
IG_2	*FRB5*	*noninc*	*random*	-	yes	*gi_asls*	*FF*	7	*insertLS*	*FF*	*insertLS*	*KK1*	10	∞	*lac*	700	0.68	*rand*	4	*RNB*	∞	*met*	0.30
IG_3	*RC*	-	-	5	no	*ils_gi*	*FF*	2	*insertLS*	*KK1*	-	-	-	3	*lac*	25	0.79	*last*	4	*RNB*	∞	*met*	0.49
IG_4	*RC*	-	-	80	yes	*gi_asls*	*KK2*	5	*insertLS*	*FF*	-	-	-	∞	*met*	0.66	0.52	*rand*	2	*RNB*	∞	*met*	0.65
IG_5	*RC*	-	-	70	no	*gs*	-	6	*insertLS*	*FF*	-	-	-	3	*lac*	25	0.56	*rand*	3	*RNB*	∞	*met*	0.48
IG_6	*RC*	-	-	5	no	*gi*	*FF*	8	*insertLS*	*FF*	*insertLS*	*KK2*	3	4	*lac*	10	0.67	*rand*	3	*RNB*	3	*met*	0.59
IG_7	*NEH*	*noninc*	*FF*	-	no	*gs*	-	8	*insertLS*	*FF*	-	-	-	4	*rrt*	10	0.81	*KK1*	3	*RNB*	4	*met*	0.50
IG_8	*FRB5*	*KK1*	*KK2*	-	no	*gi_ils*	*FF*	6	*insertLS*	*FF*	-	-	-	∞	*met*	0.95	0.65	*rand*	4	*RNB*	3	*lac*	700
IG_9	*NEH*	*KK2*	*random*	-	yes	*ras*	-	4	*insertLS*	*FF*	*NS*	-	10	4	*met*	0.27	0.80	*rand*	3	*RNB*	∞	*met*	0.31

The rows contain the components and parameter values for the algorithms obtained with irace, named IG_0 to IG_9 in the first column. The following 16 columns refer to the permutation phase. They contain the constructive heuristic to build the initial solution and its ordering criteria and tiebreaker, the value for parameter r when *RC* is selected, the value for parameter *rev*, the perturbation strategy and its tiebreaker and intensity d, the first and second local search procedures with their tiebreakers, the values for f_{ls} and n_{ls}, the acceptance criterion, and the value for parameters $\alpha, rrtd$ or l, depending on the acceptance criterion. The following seven columns refer to the non-permutation phase and contain, the percentage of time allocated for the non-permutation phase, the tiebreaker for the perturbation and the intensity d^{NP}, the local search procedure (all methods perform a single procedure), the value for $n_{\text{ls}}^{\text{NP}}$, the acceptance criterion, and the value for its respective parameter.

Next, we evaluated the algorithms on the Taillard and VRF-large benchmarks. Each algorithm ran for $60nm$ milliseconds with 10 replications per instance on a computer with two Intel Xeon E5-2697 v2 processors (12 physical cores each) at 2.7 GHz, running Ubuntu 18.04.3. The algorithms were implemented in C++ and compiled with the GNU C++ compiler version 7.4 with an optimization level of 3.

We present the results for the Taillard benchmark in Table 4.9 as the ARD in percent from the values in Table A.1. The algorithms found with irace are presented in columns "IG_0" to "IG_9". Column "Best" contains a combination of the best results per instance group to provide a theoretical estimate of the best results that could be obtained with the components from the grammar. We compare our results to the A_2 algorithm for the PFSSP from Section 3.2.2, as this was the best method for the Taillard benchmark in that section. We also compare our results to the best IG algorithm for the NPFSSP proposed by Benavides and Ritt (2018) (column "$B\&R$"). This IG uses the same perturbation strategy as the one we used, with a random tiebreaker, the *RNB* local search, and a Metropolis acceptance criterion. It is, therefore, similar to the second phase in IG_2, IG_4, IG_5, IG_6, and IG_9. The main difference, besides numerical parameter values, is that this IG algorithm uses a constructive heuristic that builds a non-permutation initial solution. In contrast, our algorithms build a permutation solution and improve it with methods for the PFSSP before evaluating non-permutation solutions. Moreover, one of the most important contributions in Benavides and Ritt (2018) is a different solution representation. This representation allowed the implementation of acceleration procedures similar to those of Taillard (1990) that reduced the complexity of job insertions and the *RNB* local search by a factor of n when compared to our approach. Instantiating the method in solver without its solution representation and acceleration procedures would hinder its performance. Therefore, to establish a fair comparison, we used the publicly available source code of Benavides and Ritt (2018) to reproduce their results.

Most of the values in Table 4.9 are negative because the results improve over the upper bounds from Table A.1, which are bounds for the PFSSP. We can see that all algorithms had considerably lower ARD compared to the state-of-the-art method for the PFSSP. However, all algorithms performed worse than $B\&R$, although with ARD's higher by only 0.08% to 0.17%.

The improvements obtained by algorithms IG_0 to IG_9 over the method for the PFSSP range from 0.43% to 0.51%, and out of these algorithms, IG_4 has the best ARD. This algorithm allocates the time between the two phases almost evenly and has the lowest

Table 4.9: ARD for the Taillard benchmark.

Inst.	A_2	$B\&R$	IG_0	IG_1	IG_2	IG_3	IG_4	IG_5	IG_6	IG_7	IG_8	IG_9	Best
20×5	0.004	-0.396	-0.351	-0.376	**-0.393**	-0.388	-0.367	-0.369	-0.383	-0.358	-0.379	-0.381	-0.393
20×10	0.000	-1.535	-1.157	-1.258	-1.507	-1.585	-1.536	-1.543	**-1.601**	-1.588	-1.226	-1.517	-1.601
20×20	0.007	-2.335	-1.827	-1.843	-2.079	-2.202	-2.101	-2.061	**-2.230**	-2.157	-1.885	-2.032	-2.230
50×5	0.000	-0.165	-0.162	-0.162	-0.164	**-0.165**	**-0.165**	**-0.165**	**-0.165**	**-0.165**	-0.162	**-0.165**	-0.165
50×10	0.368	-0.040	0.103	0.121	**0.035**	0.123	0.075	0.073	0.108	0.098	0.174	0.036	0.035
50×20	0.317	-0.841	-0.341	-0.526	-0.436	-0.314	**-0.530**	-0.408	-0.428	-0.437	-0.408	-0.415	-0.530
100×5	0.000	-0.123	-0.111	-0.117	-0.133	-0.131	-0.129	**-0.134**	-0.130	-0.131	-0.119	-0.132	-0.134
100×10	0.033	-0.097	-0.035	-0.051	-0.066	-0.023	-0.069	-0.026	-0.034	-0.002	**-0.073**	-0.039	-0.073
100×20	0.524	0.115	0.291	**0.191**	0.409	0.505	0.232	0.328	0.346	0.368	0.304	0.352	0.191
200×10	0.036	-0.044	-0.042	-0.049	-0.041	-0.035	-0.040	-0.052	-0.044	-0.049	**-0.053**	-0.052	-0.053
200×20	0.713	0.373	**0.488**	0.521	0.667	0.691	0.524	0.567	0.641	0.601	0.591	0.660	0.488
500×20	0.275	0.210	0.248	**0.237**	0.284	0.274	0.259	0.286	0.259	0.303	0.267	0.276	0.237
Avg.	0.190	-0.407	-0.241	-0.276	-0.285	-0.271	**-0.320**	-0.292	-0.305	-0.293	-0.247	-0.284	-0.352

The results presented in this table are the ARD over the upper bounds in Table A.1. A_2 is the state-of-the-art method for the PFSSP presented in Section 3.2.2, $B\&R$ is the state-of-the-art method for the NPFSSP proposed by Benavides and Ritt (2018), and IG_0 to IG_9 are the algorithms found with irace. The best results obtained by IG_0 to IG_9 for each instance group are highlighted in bold, and column "Best" contains a combination of the best results.

perturbation intensity in the non-permutation phase. IG_5 and IG_6 were the second and third best-performing methods. Both have a similar non-permutation phase compared to IG_4, with the same tiebreaker and acceptance criterion, and the value for d^{NP} higher by only one. Regarding the permutation phase, however, the similarities are limited to the usage of *RC* constructive heuristic, and the *insertLS* local search with *FF* tiebreaker. IG_7 yielded an ARD that is close to IG_5 and IG_6, although with a different tiebreaker in the non-permutation phase and a significantly higher value for *npf*. Regarding the permutation phase, this algorithm is considerably different from the other mentioned methods.

In general, IG_4, IG_5, IG_6, and IG_7 have similar performance across all instance dimensions, and $B\&R$ has the best results for most instance groups. We applied Wilcoxon signed-rank tests with Bonferroni correction between $B\&R$ and each one of the obtained algorithms, and the tests indicated that the difference is statistically significant in all cases ($99\,\%$ confidence, $p < 1.03 \times 10^{-8}$ in all cases).

The following experiment considered the VRF-large benchmark. The results are presented in Table 4.10 as the ARD from the upper bounds of Vallada, Ruiz and Framiñan (2015) (see Appendix A). All algorithms had considerably higher ARD compared to $B\&R$. The difference in the performance due to the lower complexity of $B\&R$ is more noticeable in this benchmark since it contains larger instances than the previous one. In comparison to the method for the PFSSP, all algorithms had lower ARD, except for IG_9, which did not generalize as well as the other methods to this benchmark. In this experiment, IG_5 had the lowest ARD, followed by IG_0 and IG_4, with similar results. IG_4 and IG_5 had also performed well in the previous experiment. They were two of the best-

Table 4.10: ARD for the VRF benchmark.

Inst.	A_2	$B\&R$	IG_0	IG_1	IG_2	IG_3	IG_4	IG_5	IG_6	IG_7	IG_8	IG_9	Best
100×20	-0.102	-0.861	-0.427	**-0.566**	-0.322	-0.169	-0.490	-0.278	-0.184	-0.201	-0.364	-0.156	-0.566
100×40	-0.235	-0.860	-0.286	-0.707	-0.516	-0.404	**-0.918**	-0.554	-0.517	-0.559	-0.301	-0.459	-0.918
100×60	-0.260	-1.120	-0.094	-0.550	-0.457	-0.492	**-0.912**	-0.524	-0.530	-0.519	-0.164	-0.508	-0.912
200×20	0.009	-0.346	**-0.221**	-0.220	-0.084	0.013	-0.185	-0.051	-0.005	0.100	-0.135	0.086	-0.221
200×40	-0.430	-0.953	-0.592	-0.668	-0.493	-0.516	**-0.764**	-0.628	-0.353	-0.436	-0.323	-0.175	-0.764
200×60	-0.505	-0.939	-0.386	-0.595	-0.440	-0.599	**-0.883**	-0.751	-0.584	-0.613	-0.209	-0.369	-0.883
300×20	0.041	-0.292	**-0.241**	-0.168	-0.089	-0.034	-0.135	-0.106	-0.078	-0.024	-0.067	-0.038	-0.241
300×40	-0.350	-0.869	**-0.673**	-0.609	-0.551	-0.519	-0.609	-0.620	-0.400	-0.227	-0.486	-0.117	-0.673
300×60	-0.423	-0.953	-0.589	-0.506	-0.466	-0.575	-0.722	**-0.824**	-0.483	-0.547	-0.311	-0.277	-0.824
400×20	0.048	-0.148	-0.108	**-0.123**	-0.034	0.015	-0.052	-0.018	-0.042	0.027	-0.063	-0.004	-0.123
400×40	-0.264	-1.006	**-0.639**	-0.548	-0.533	-0.555	-0.500	-0.566	-0.472	-0.100	-0.599	-0.115	-0.639
400×60	-0.368	-1.175	-0.725	-0.531	-0.575	-0.611	-0.661	**-0.886**	-0.551	-0.413	-0.503	-0.276	-0.886
500×20	0.028	-0.071	**-0.064**	-0.054	0.009	0.002	-0.007	0.024	0.002	0.031	-0.016	0.063	-0.064
500×40	-0.231	-0.871	-0.701	-0.557	-0.585	-0.577	-0.460	-0.537	-0.569	-0.032	**-0.718**	-0.150	-0.718
500×60	-0.225	-1.076	-0.599	-0.367	-0.512	-0.468	-0.469	**-0.729**	-0.462	-0.272	-0.507	-0.138	-0.729
600×20	-0.003	-0.120	**-0.100**	-0.087	-0.024	-0.046	-0.031	-0.012	-0.024	0.008	-0.016	0.016	-0.100
600×40	-0.206	-0.746	-0.628	-0.515	-0.501	-0.492	-0.373	-0.446	-0.516	-0.019	**-0.697**	-0.098	-0.697
600×60	-0.130	-1.094	-0.647	-0.360	-0.565	-0.521	-0.441	**-0.704**	-0.563	-0.264	-0.627	-0.172	-0.704
700×20	-0.007	-0.018	**-0.045**	-0.043	-0.003	-0.015	-0.002	-0.002	-0.016	0.036	-0.015	0.030	-0.045
700×40	-0.268	-0.830	-0.747	-0.592	-0.540	-0.555	-0.456	-0.507	-0.619	-0.079	**-0.755**	-0.263	-0.755
700×60	-0.106	-1.144	-0.627	-0.362	-0.568	-0.536	-0.440	-0.690	-0.656	-0.242	**-0.740**	-0.221	-0.740
800×20	-0.008	-0.046	-0.030	**-0.037**	-0.004	-0.019	-0.018	-0.003	-0.023	0.010	-0.022	-0.006	-0.037
800×40	-0.283	-0.775	**-0.675**	-0.572	-0.509	-0.504	-0.439	-0.442	-0.575	-0.079	-0.739	-0.227	-0.675
800×60	0.053	-0.941	-0.487	-0.202	-0.435	-0.419	-0.349	-0.528	-0.558	-0.074	**-0.650**	-0.105	-0.650
Avg.	-0.176	-0.719	-0.430	-0.397	-0.367	-0.358	-0.430	**-0.433**	-0.366	-0.187	-0.376	-0.153	-0.565

The results presented in this table are the ARD over the upper bounds in Table A.2. A_2 is the state-of-the-art method for the PFSSP presented in Section 3.2.2, $B\&R$ is the state-of-the-art method for the NPFSSP proposed by Benavides and Ritt (2018), and IG_0 to IG_9 are the algorithms found with irace. The best results obtained by IG_0 to IG_9 for each instance group are highlighted in bold, and column "Best" contains a combination of the best results.

performing methods in the previous experiment, as opposed to IG_0, with the highest ARD among the algorithms obtained with irace. We can see that IG_4 has lower ARD for instances with up to 200 jobs, and instances with 20 machines, while IG_5 performed better on the rest. When comparing IG_0 to IG_5, we can see that the former yields better results for instances with $m = 20$ when the number of machines $n = 100$ or $n = 200$, and for instances with $m = 20$ and $m = 40$ when $n > 200$. On the other hand, IG_5 always had an advantage when $m = 60$, especially on the instances with a smaller number of jobs. Finally, IG_1, IG_2, IG_3, IG_6 and IG_8 have close ARD values, while IG_7 and IG_9 yielded the worst results. Again we applied Wilcoxon signed-rank tests with Bonferroni correction between $B\&R$ and each one of the obtained algorithms, and the tests indicated that the difference is statistically significant in all cases ($99\,\%$ confidence, $p < 2.2 \times 10^{-16}$ in all cases).

5 CONCLUSIONS

Flow shop scheduling problems have many applications and consequently have been studied for decades, with hundreds of new articles being published every year. Due to the considerable number of solving methods that have been proposed, establishing a comprehensive comparison is an onerous task. Moreover, the methods in the literature are often complex and hard to reproduce, which is a challenge in comparative studies.

In this thesis, we built a solver for flow shop problems that implements many algorithmic components and allows one to combine them to generate new algorithms quickly. By integrating multiple algorithmic components in a single solver, we **(i)** facilitate the reproduction of the literature, **(ii)** simplify the comparison to many published methods, including the state of the art, and **(iii)** allow researchers to focus on the development of new algorithmic components.

Furthermore, we used an automated approach to explore combinations of components and generate high-performing algorithms. This approach aims to reduce the high level of human effort required during the design process and increase its robustness, since the it often consists of trial-and-error approaches and can be biased by previous experience. Previous experience with the problem is valuable in guiding the design process towards promising directions. However, the bias can also be detrimental, e.g., certain design options can be prematurely discarded without systematic experimentation that evaluates its interactions with other components.

We focus on ILS and IG algorithms, which are known to be efficient for flow shop problems, and automate the configuration of its components and their related parameters. We present a total of 40 automatically designed algorithms (and 20 more in the appendices), and evaluate them on two benchmarks. The computational experiments show that the automatically designed algorithms are competitive with the state of the art and can improve it in some cases.

Our experiments with the NPFSSP showed that despite involving methods with higher computational complexity, exploring non-permutation solutions can yield better schedules within the same time limit, leading to significant reductions of the makespan and the total completion time. We hope that our evidence in favor of using non-permutation schedules encourages further research in the area.

As for future research, an interesting topic is the study of total tardiness minimization in flow shops. This objective function is of high interest since they model many

current real-world scenarios in which completing a job after a due date is undesirable and can result in a certain cost. The recent literature review of Vallada, Ruiz and Minella (2008) supports the relevance of this objective function. Moreover, Liao, Liao and Tseng (2006) showed that the improvements when adopting non-permutation schedules are more substantial when minimizing due-date-based objectives. More precisely, the average improvement was $2.28\,\%$ for total tardiness minimization, which is considerable and can motivate the adoption of non-permutation schedules and further studies in this direction. Despite that, the only published paper addressing the $Fm||\sum T_j$ we were able to find is due to Liao and Huang (2010). Thus, we believe that there is room for further improvements, which may be a promising line of research.

Another topic to be considered for future research are flow shops with missing operations, also referred to in the literature as flowline-based manufacturing systems. In this variant, some jobs can have operations only on some of the machines. The study of this scenario is highly relevant since it models many real-world environments (RAJENDRAN; ZIEGLER, 2001). Missing operations can introduce forced idleness in permutation schedules, as a job with a missing operation cannot pass another job even if a machine is available, resulting in poor solution quality. In this case, the use of non-permutation schedules can alleviate such an issue (PUGAZHENDHI et al., 2003). Additionally, Potts, Shmoys and Williamson (1991) showed that optimal permutation schedules are worse than non-permutation schedules by a factor of $\sqrt{m}/2$ for a certain set of instances with missing operations, further motivating a study on the topic.

REFERENCES

ADAMS, J.; BALAS, E.; ZAWACK, D. The shifting bottleneck procedure for job shop scheduling. **Management Science**, INFORMS, v. 34, n. 3, p. 391–401, 1988.

ANSÓTEGUI, C.; MALITSKY, Y.; SAMULOWITZ, H.; SELLMANN, M.; TIERNEY, K. Model-based genetic algorithms for algorithm configuration. In: **Proceedings of the 24th International Conference on Artificial Intelligence**. [S.l.]: AAAI Press, 2015. (IJCAI'15), p. 733–739.

ANSÓTEGUI, C.; SELLMANN, M.; TIERNEY, K. A gender-based genetic algorithm for the automatic configuration of algorithms. In: GENT, I. P. (Ed.). **Principles and Practice of Constraint Programming - CP 2009**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 142–157. ISBN 978-3-642-04244-7.

BAKER, K. R. **Introduction to sequencing and scheduling**. [S.l.]: John Wiley & Sons, 1974.

BANSAL, S. P. Minimizing the sum of completion times of n jobs over m machines in a flowshop - A branch and bound approach. **AIIE Transactions**, Taylor & Francis, v. 9, n. 3, p. 306–311, 1977.

BENAVIDES, A. J.; RITT, M. Iterated local search heuristics for minimizing total completion time in permutation and non-permutation flow shops. In: **Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling**. [S.l.]: AAAI Press, 2015. (ICAPS'15), p. 34–41.

BENAVIDES, A. J.; RITT, M. Two simple and effective heuristics for minimizing the makespan in non-permutation flow shops. **Computers & Operations Research**, v. 66, p. 160–169, 2016.

BENAVIDES, A. J.; RITT, M. Fast heuristics for minimizing the makespan in non-permutation flow shops. **Computers & Operations Research**, v. 100, p. 230–243, 2018.

BIRATTARI, M.; STÜTZLE, T.; PAQUETE, L.; VARRENTRAPP, K. A racing algorithm for configuring metaheuristics. In: **Proceedings of the Genetic and Evolutionary Computation Conference**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. (GECCO '02), p. 11–18. ISBN 1-55860-878-8.

BRANKE, J.; NGUYEN, S.; PICKARDT, C. W.; ZHANG, M. Automated design of production scheduling heuristics: A review. **IEEE Transactions on Evolutionary Computation**, v. 20, n. 1, p. 110–124, 2016.

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001.

BRUM, A.; RITT, M. Automatic algorithm configuration for the permutation flow shop scheduling problem minimizing total completion time. In: LIEFOOGHE, A.; LÓPEZ-IBÁÑEZ, M. (Ed.). **Evolutionary Computation in Combinatorial Optimization**. Cham: Springer International Publishing, 2018. p. 85–100. ISBN 978-3-319-77449-7.

BRUM, A.; RITT, M. Automatic design of heuristics for minimizing the makespan in permutation flow shops. In: **IEEE Congress on Evolutionary Computation**. [S.l.: s.n.], 2018. p. 1–8.

BRUM, A.; RUIZ, R.; RITT, M. Automatic generation of iterated greedy algorithms for the non-permutation flow shop scheduling problem with total completion time minimization. 2020, submitted to Computers & Operations Research.

BURKE, E. K.; BYKOV, Y. The late acceptance hill-climbing heuristic. **European Journal of Operational Research**, v. 258, n. 1, p. 70–78, 2017.

BURKE, E. K.; GENDREAU, M.; HYDE, M.; KENDALL, G.; OCHOA, G.; ÖZCAN, E.; QU, R. Hyper-heuristics: a survey of the state of the art. **Journal of the Operational Research Society**, v. 64, n. 12, p. 1695–1724, 2013.

CÁCERES, L. P.; BISCHL, B.; STÜTZLE, T. Evaluating random forest models for irace. In: **Proceedings of the Genetic and Evolutionary Computation Conference Companion**. New York, NY, USA: ACM, 2017. (GECCO '17), p. 1146–1153. ISBN 978-1-4503-4939-0.

CÁCERES, L. P.; LÓPEZ-IBÁÑEZ, M.; HOOS, H.; STÜTZLE, T. An experimental study of adaptive capping in irace. In: BATTITI, R.; KVASOV, D. E.; SERGEYEV, Y. D. (Ed.). **Learning and Intelligent Optimization**. Cham: Springer International Publishing, 2017. p. 235–250. ISBN 978-3-319-69404-7.

CÁCERES, L. P.; STÜTZLE, T. Exploring variable neighborhood search for automatic algorithm configuration. **Electronic Notes in Discrete Mathematics**, v. 58, p. 167–174, 2017, 4th International Conference on Variable Neighborhood Search.

CARLIER, J.; REBAÏ, I. Two branch and bound algorithms for the permutation flow shop problem. **European Journal of Operational Research**, v. 90, n. 2, p. 238–251, 1996.

CHEN, C.-L.; VEMPATI, V. S.; ALJABER, N. An application of genetic algorithms for flow shop problems. **European Journal of Operational Research**, v. 80, n. 2, p. 389–396, 1995.

CHUNG, C.-S.; FLYNN, J.; KIRCA, O. A branch and bound algorithm to minimize the total flow time for m-machine permutation flowshop problems. **International Journal of Production Economics**, v. 79, n. 3, p. 185–196, 2002.

CONWAY, R.; MAXWELL, W.; MILLER, L. **Theory of scheduling**. [S.l.]: Addison-Wesley Pub. Co., 1967.

COOK, S. A. The complexity of theorem-proving procedures. In: **Proceedings of the Third Annual ACM Symposium on Theory of Computing**. New York, NY, USA: Association for Computing Machinery, 1971. (STOC '71), p. 151–158. ISBN 9781450374644.

CROCE, F. D.; GHIRARDI, M.; TADEI, R. An improved branch-and-bound algorithm for the two machine total completion time flow shop problem. **European Journal of Operational Research**, v. 139, n. 2, p. 293–301, 2002.

CROCE, F. D.; NARAYAN, V.; TADEI, R. The two-machine total completion time flow shop problem. **European Journal of Operational Research**, v. 90, n. 2, p. 227–237, 1996.

DANG, N.; CÁCERES, L. P.; CAUSMAECKER, P. D.; STÜTZLE, T. Configuring irace using surrogate configuration benchmarks. In: **Proceedings of the Genetic and Evolutionary Computation Conference**. New York, NY, USA: ACM, 2017. (GECCO '17), p. 243–250. ISBN 978-1-4503-4920-8.

DEMIRKOL, E.; MEHTA, S.; UZSOY, R. Benchmarks for shop scheduling problems. **European Journal of Operational Research**, v. 109, n. 1, p. 137–141, 1998.

DEROUSSI, L.; GOURGAND, M.; NORRE, S. **New effective neighborhoods for the permutation flow shop problem**. [S.l.], 2006.

DONG, X.; CHEN, P.; HUANG, H.; NOWAK, M. A multi-restart iterated local search algorithm for the permutation flow shop problem minimizing total flow time. **Computers & Operations Research**, v. 40, n. 2, p. 627–632, 2013.

DONG, X.; HUANG, H.; CHEN, P. An improved NEH-based heuristic for the permutation flowshop problem. **Computers & Operations Research**, v. 35, n. 12, p. 3962–3968, 2008, part of Special Issue: Telecommunications Network Engineering.

DUAN, J.; YANG, Y.; GAO, K.; LI, J.; PAN, Q. A speed-up method for calculating total flowtime in permutation flow shop scheduling problem. In: **2013 25th Chinese Control and Decision Conference (CCDC)**. [S.l.: s.n.], 2013. p. 2755–2758.

DUBOIS-LACOSTE, J. **Anytime Local Search for Multi-Objective Combinatorial Optimization: Design, Analysis and Automatic Configuration**. Thesis (PhD) — IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2014.

DUBOIS-LACOSTE, J.; PAGNOZZI, F.; STÜTZLE, T. An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. **Computers & Operations Research**, v. 81, p. 160–166, 2017.

DUECK, G. New optimization heuristics: The great deluge algorithm and the record-to-record travel. **Journal of Computational Physics**, v. 104, n. 1, p. 86–92, 1993.

EGGENSPERGER, K.; LINDAUER, M.; HOOS, H. H.; HUTTER, F.; LEYTON-BROWN, K. Efficient benchmarking of algorithm configurators via model-based surrogates. **Machine Learning**, v. 107, n. 1, p. 15–41, 2018.

FERNÁNDEZ-VIAGAS, V.; FRAMIÑAN, J. M. On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. **Computers & Operations Research**, v. 45, p. 60–67, 2014.

FERNÁNDEZ-VIAGAS, V.; FRAMIÑAN, J. M. A new set of high-performing heuristics to minimise flowtime in permutation flowshops. **Computers & Operations Research**, v. 53, p. 68–80, 2015.

FERNÁNDEZ-VIAGAS, V.; FRAMIÑAN, J. M. A beam-search-based constructive heuristic for the PFSP to minimise total flowtime. **Computers & Operations Research**, v. 81, p. 167–177, 2017.

FERNÁNDEZ-VIAGAS, V.; FRAMIÑAN, J. M. A best-of-breed iterated greedy for the permutation flowshop scheduling problem with makespan objective. **Computers & Operations Research**, v. 112, p. 104767, 2019.

FERNÁNDEZ-VIAGAS, V.; RUIZ, R.; FRAMIÑAN, J. M. A new vision of approximate methods for the permutation flowshop to minimise makespan: State-of-the-art and computational evaluation. **European Journal of Operational Research**, v. 257, n. 3, p. 707–721, 2017.

FERNÁNDEZ-VIAGAS, V.; VALENTE, J. M.; FRAMIÑAN, J. M. Iterated-greedy-based algorithms with beam search initialization for the permutation flowshop to minimise total tardiness. **Expert Systems with Applications**, v. 94, p. 58–69, 2018.

FRAMIÑAN, J. M.; GUPTA, J. N. D.; LEISTEN, R. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. **Journal of the Operational Research Society**, Taylor & Francis, v. 55, n. 12, p. 1243–1255, 2004.

FRAMIÑAN, J. M.; LEISTEN, R.; RAJENDRAN, C. Different initial sequences for the heuristic of nawaz, enscore and ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. **International Journal of Production Research**, Taylor & Francis, v. 41, n. 1, p. 121–148, 2003.

FRAMIÑAN, J. M.; LEISTEN, R.; RUIZ-USANO, R. Comparison of heuristics for flowtime minimisation in permutation flowshops. **Computers & Operations Research**, v. 32, n. 5, p. 1237–1254, 2005.

GAREY, M. R.; JOHNSON, D. S.; SETHI, R. The complexity of flowshop and jobshop scheduling. **Math. Oper. Res.**, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 1, n. 2, p. 117–129, 1976.

GHARBI, A.; LABIDI, M.; LOULY, M. A. The nonpermutation flowshop scheduling problem: Adjustment and bounding procedures. **Journal of Applied Mathematics**, v. 2014, 2014.

GRABOWSKI, J.; WODECKI, M. A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. **Computers & Operations Research**, v. 31, n. 11, p. 1891–1909, 2004.

GRAHAM, R.; LAWLER, E.; LENSTRA, J.; KAN, A. Optimization and approximation in deterministic sequencing and scheduling: a survey. In: HAMMER, P.; JOHNSON, E.; KORTE, B. (Ed.). **Discrete Optimization II**. [S.l.]: Elsevier, 1979, (Annals of Discrete Mathematics, v. 5). p. 287–326.

HAQ, A. N.; SARAVANAN, M.; VIVEKRAJ, A. R.; PRASAD, T. A scatter search approach for general flowshop scheduling problem. **The International Journal of Advanced Manufacturing Technology**, v. 31, n. 7, p. 731–736, 2007.

HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K. Sequential model-based optimization for general algorithm configuration. In: **LION-5**. [S.l.: s.n.], 2011. (LNCS), p. 507–523.

HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K.; STÜTZLE, T. ParamILS: An automatic algorithm configuration framework. **Journal of Artificial Intelligence Research**, v. 36, p. 267–306, 2009.

IGNALL, E.; SCHRAGE, L. Application of the branch and bound technique to some flow-shop scheduling problems. **Operations Research**, v. 13, n. 3, p. 400–412, 1965.

JAIN, A. S.; MEERAN, S. A multi-level hybrid framework applied to the general flow-shop scheduling problem. **Computers & Operations Research**, v. 29, n. 13, p. 1873–1901, 2002.

JARBOUI, B.; EDDALY, M.; SIARRY, P. An estimation of distribution algorithm for minimizing the total flowtime in permutation flowshop scheduling problems. **Computers & Operations Research**, v. 36, n. 9, p. 2638–2646, 2009.

JOHNSON, S. M. Optimal two- and three-stage production schedules with setup times included. **Naval Research Logistics Quarterly**, v. 1, n. 1, p. 61–68, 1954.

KALCZYNSKI, P. J.; KAMBUROWSKI, J. An improved NEH heuristic to minimize makespan in permutation flow shops. **Computers & Operations Research**, v. 35, n. 9, p. 3001–3008, 2008.

KALCZYNSKI, P. J.; KAMBUROWSKI, J. An empirical analysis of the optimality rate of flow shop heuristics. **European Journal of Operational Research**, v. 198, n. 1, p. 93–101, 2009.

KAO, G. K.; SEWELL, E. C.; JACOBSON, S. H. A branch, bound, and remember algorithm for the $1 \mid r_i \mid \sum T_i$ scheduling problem. **Journal of Scheduling**, v. 12, n. 2, p. 163, 2008.

KHUDABUKHSH, A. R.; XU, L.; HOOS, H. H.; LEYTON-BROWN, K. SATenstein: Automatically building local search SAT solvers from components. **Artificial Intelligence**, v. 232, p. 20–42, 2016.

KOULAMAS, C. A new constructive heuristic for the flowshop scheduling problem. **European Journal of Operational Research**, v. 105, n. 1, p. 66–71, 1998.

KOZA, J. R. **Genetic Programming: On the Programming of Computers by Means of Natural Selection**. Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-11170-5.

KUHN, H. W.; YAW, B. The hungarian method for the assignment problem. **Naval Research Logistics Quarterly**, p. 83–97, 1955.

LADHARI, T.; HAOUARI, M. A computational study of the permutation flow shop problem based on a tight lower bound. **Computers & Operations Research**, v. 32, n. 7, p. 1831–1847, 2005.

LAND, A. H.; DOIG, A. G. An automatic method of solving discrete programming problems. **Econometrica**, [Wiley, Econometric Society], v. 28, n. 3, p. 497–520, 1960.

LEVIN, L. A. Universal sequential search problems. **Problemy Peredachi Informatsii**, Russian Academy of Sciences, Branch of Informatics, Computer Equipment and Automatization, v. 9, n. 3, p. 115–116, 1973.

LI, X.; WANG, Q.; WU, C. Efficient composite heuristics for total flowtime minimization in permutation flow shops. **Omega**, v. 37, n. 1, p. 155–164, 2009.

LIAO, C. J.; LIAO, L. M.; TSENG, C. T. A performance evaluation of permutation vs. non-permutation schedules in a flowshop. **International Journal of Production Research**, Taylor & Francis, v. 44, n. 20, p. 4297–4309, 2006.

LIAO, C.-J.; YOU, C.-T. An improved formulation for the job-shop scheduling problem. **Journal of the Operational Research Society**, v. 43, n. 11, p. 1047–1054, 1992.

LIAO, L.-M.; HUANG, C.-J. Tabu search for non-permutation flowshop scheduling problem with minimizing total tardiness. **Applied Mathematics and Computation**, v. 217, n. 2, p. 557–567, 2010.

LIN, S.-W.; YING, K.-C. Applying a hybrid simulated annealing and tabu search approach to non-permutation flowshop scheduling problems. **International Journal of Production Research**, Taylor & Francis, v. 47, n. 5, p. 1411–1424, 2009.

LIU, J.; REEVES, C. R. Constructive and composite heuristic solutions to the $P \mid\mid \sum C_i$ scheduling problem. **European Journal of Operational Research**, v. 132, n. 2, p. 439–452, 2001.

LIU, S. Q.; ONG, H. L. A comparative study of algorithms for the flowshop scheduling problem. **Asia Pacific Journal of Operational Research**, World Scientific Publishing Co. Pte. Ltd., v. 19, n. 2, p. 205–222, 2002.

LIU, W.; JIN, Y.; PRICE, M. A new improved NEH heuristic for permutation flowshop scheduling problems. **International Journal of Production Economics**, v. 193, p. 21–30, 2017.

LOMNICKI, Z. A. A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. **Journal of the Operational Research Society**, v. 16, n. 1, p. 89–100, 1965.

LÓPEZ-IBÁÑEZ, M.; DUBOIS-LACOSTE, J.; CÁCERES, L. P.; BIRATTARI, M.; STÜTZLE, T. The irace package: Iterated racing for automatic algorithm configuration. **Operations Research Perspectives**, v. 3, p. 43–58, 2016.

LÓPEZ-IBÁÑEZ, M.; STÜTZLE, T. The automatic design of multiobjective ant colony optimization algorithms. **IEEE Transactions on Evolutionary Computation**, v. 16, n. 6, p. 861–875, 2012.

LOURENÇO, H. R.; MARTIN, O. C.; STÜTZLE, T. **Iterated Local Search**. Boston, MA: Springer US, 2003. 320–353 p.

LOURENÇO, N.; PEREIRA, F. B.; COSTA, E. Unveiling the properties of structured grammatical evolution. **Genetic Programming and Evolvable Machines**, v. 17, n. 3, p. 251–289, 2016.

LOW, C.; YEH, J.-Y.; HUANG, K.-I. A robust simulated annealing heuristic for flow shop scheduling problems. **The International Journal of Advanced Manufacturing Technology**, v. 23, n. 9, p. 762–767, 2004.

LOWERRE, B. T. **The Harpy Speech Recognition System.** Thesis (PhD), Pittsburgh, PA, USA, 1976. AAI7619331.

MADHUSHINI, N.; RAJENDRAN, C.; DEEPA, Y. Branch-and-bound algorithms for scheduling in permutation flowshops to minimize the sum of weighted flowtime/sum of weighted tardiness/sum of weighted flowtime and weighted tardiness/sum of weighted flowtime, weighted tardiness and weighted earliness of jobs. **Journal of the Operational Research Society**, Taylor & Francis, v. 60, n. 7, p. 991–1004, 2009.

MANNE, A. S. On the job-shop scheduling problem. **Operations Research**, v. 8, n. 2, p. 219–223, 1960.

MARMION, M.-E.; MASCIA, F.; LÓPEZ-IBÁÑEZ, M.; STÜTZLE, T. **Automatic Design of Hybrid Stochastic Local Search Algorithms**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. 144–158 p.

MASCIA, F.; LÓPEZ-IBÁÑEZ, M.; DUBOIS-LACOSTE, J.; STÜTZLE, T. From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In: **Revised Selected Papers of the 7th International Conference on Learning and Intelligent Optimization - Volume 7997**. New York, NY, USA: Springer-Verlag New York, Inc., 2013. (LION 7), p. 321–334.

MCKAY, R. I.; HOAI, N. X.; WHIGHAM, P. A.; SHAN, Y.; O'NEILL, M. Grammar-based genetic programming: a survey. **Genetic Programming and Evolvable Machines**, v. 11, n. 3, p. 365–396, 2010.

METROPOLIS, N.; ROSENBLUTH, A. W.; ROSENBLUTH, M. N.; TELLER, A. H.; TELLER, E. Equation of state calculations by fast computing machines. **The Journal of Chemical Physics**, v. 21, n. 6, p. 1087–1092, 1953.

MURATA, T.; ISHIBUCHI, H.; TANAKA, H. Genetic algorithms for flowshop scheduling problems. **Computers & Industrial Engineering**, v. 30, n. 4, p. 1061–1071, 1996.

NAGARAJAN, V.; SVIRIDENKO, M. Tight bounds for permutation flow shop scheduling. **Mathematics of Operations Research**, v. 34, n. 2, p. 417–427, 2009.

NAWAZ, M.; ENSCORE, E. E.; HAM, I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. **Omega**, v. 11, n. 1, p. 91–95, 1983.

NOWICKI, E.; SMUTNICKI, C. A fast taboo search algorithm for the job shop problem. **Management Science**, v. 42, n. 6, p. 797–813, 1996.

NOWICKI, E.; SMUTNICKI, C. A fast tabu search algorithm for the permutation flow-shop problem. **European Journal of Operational Research**, v. 91, n. 1, p. 160–175, 1996.

OSMAN, I.; POTTS, C. Simulated annealing for permutation flow-shop scheduling. **Omega**, v. 17, n. 6, p. 551–557, 1989.

PAGNOZZI, F. **Automatic Design of Hybrid Stochastic Local Search Algorithms – analysis and application**. Thesis (PhD) — Université Libre de Bruxelles, Ecole Polytechnique de Bruxelles — Informatique, 2019.

PAGNOZZI, F.; STÜTZLE, T. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. **European Journal of Operational Research**, 2019.

PAN, C.-H. A study of integer programming formulations for scheduling problems. **International Journal of Systems Science**, Taylor & Francis, v. 28, n. 1, p. 33–41, 1997.

PAN, Q.-K.; RUIZ, R. Local search methods for the flowshop scheduling problem with flowtime minimization. **European Journal of Operational Research**, v. 222, n. 1, p. 31–43, 2012.

PAN, Q.-K.; RUIZ, R. A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime. **Computers & Operations Research**, Elsevier Science Ltd., Oxford, UK, UK, v. 40, n. 1, p. 117–128, 2013.

PAN, Q.-K.; TASGETIREN, M. F.; LIANG, Y.-C. A discrete differential evolution algorithm for the permutation flowshop scheduling problem. **Computers & Industrial Engineering**, v. 55, n. 4, p. 795–816, 2008.

POTTS, C. An adaptive branching rule for the permutation flow-shop problem. **European Journal of Operational Research**, v. 5, n. 1, p. 19–25, 1980.

POTTS, C. N.; SHMOYS, D. B.; WILLIAMSON, D. P. Permutation vs. non-permutation flow shop schedules. **Operations Research Letters**, v. 10, n. 5, p. 281–284, 1991.

PUGAZHENDHI, S.; THIAGARAJAN, S.; RAJENDRAN, C.; ANANTHARAMAN, N. Performance enhancement by using non-permutation schedules in flowline-based manufacturing systems. **Computers & Industrial Engineering**, v. 44, n. 1, p. 133–157, 2003.

RAD, S. F.; RUIZ, R.; BOROOJERDIAN, N. New high performing heuristics for minimizing makespan in permutation flowshops. **Omega**, v. 37, n. 2, p. 331–345, 2009.

RAJENDRAN, C.; ZIEGLER, H. An efficient heuristic for scheduling in a flowshop to minimize total weighted flowtime of jobs. **European Journal of Operational Research**, v. 103, n. 1, p. 129–138, 1997.

RAJENDRAN, C.; ZIEGLER, H. A performance analysis of dispatching rules and a heuristic in static flowshops with missing operations of jobs. **European Journal of Operational Research**, v. 131, n. 3, p. 622–634, 2001.

RAJENDRAN, C.; ZIEGLER, H. Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. **European Journal of Operational Research**, v. 155, n. 2, p. 426–438, 2004, Financial Risk in Open Economies.

RAJENDRAN, C.; ZIEGLER, H. Two ant-colony algorithms for minimizing total flowtime in permutation flowshops. **Computers & Industrial Engineering**, v. 48, n. 4, p. 789–797, 2005, selected papers from the 30th International Conference on Computers & Industrial Engineering.

RASMUSSEN, C.; WILLIAMS, C. **Gaussian Processes for Machine Learning**. Cambridge, MA, USA: MIT Press, 2006. 248 p. (Adaptive Computation and Machine Learning). ISBN 0-262-18253-X.

REEVES, C. R. Improving the efficiency of tabu search for machine sequencing problems. **Journal of the Operational Research Society**, v. 44, n. 4, p. 375–382, 1993.

REEVES, C. R. A genetic algorithm for flowshop sequencing. **Computers & Operations Research**, v. 22, n. 1, p. 5–13, 1995.

RICE, J. R. The algorithm selection problem. In: RUBINOFF, M.; YOVITS, M. C. (Ed.). [S.l.]: Elsevier, 1976, (Advances in Computers, v. 15). p. 65–118.

RITT, M. A branch-and-bound algorithm with cyclic best-first search for the permutation flow shop scheduling problem. **2016 IEEE International Conference on Automation Science and Engineering (CASE)**, p. 872–877, 2016.

ROSSI, A.; LANZETTA, M. Native metaheuristics for non-permutation flowshop scheduling. **Journal of Intelligent Manufacturing**, v. 25, n. 6, p. 1221–1233, 2014.

ROSSI, F. L.; NAGANO, M. S.; TAVARES NETO, R. F. Evaluation of high performance constructive heuristics for the flow shop with makespan minimization. **The International Journal of Advanced Manufacturing Technology**, v. 87, n. 1, p. 125–136, 2016.

ROSSIT, D. A.; TOHMÉ, F.; FRUTOS, M. The non-permutation flow-shop scheduling problem: A literature review. **Omega**, v. 77, p. 143–153, 2018.

RUIZ, R.; MAROTO, C.; ALCARAZ, J. Two new robust genetic algorithms for the flowshop scheduling problem. **Omega**, v. 34, n. 5, p. 461–476, 2006.

RUIZ, R.; STÜTZLE, T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. **European Journal of Operational Research**, v. 177, n. 3, p. 2033–2049, 2007.

RYAN, C.; COLLINS, J.; NEILL, M. O. Grammatical evolution: Evolving programs for an arbitrary language. In: BANZHAF, W.; POLI, R.; SCHOENAUER, M.; FOGARTY, T. C. (Ed.). **Genetic Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 83–96.

SADJADI, S.; BOUQUARD, J.; ZIAEE, M. An ant colony algorithm for the flowshop scheduling problem. **Journal of Applied Sciences**, v. 8, p. 3938–3944, 2008.

SILVA, D. S. P. d. **On the Permutation Flow Shop Scheduling Problem**. Thesis (PhD) — Pontifícia Universidade Católica do Rio de Janeiro, 2010.

STAFFORD, E. F. On the development of a mixed-integer linear programming model for the flowshop sequencing problem. **The Journal of the Operational Research Society**, Palgrave Macmillan Journals, v. 39, n. 12, p. 1163–1174, 1988.

STAFFORD, E. F.; TSENG, F. T. On the Srikar-Ghosh MILP model for the NxM SDST flowshop problem. **International Journal of Production Research**, Taylor & Francis, v. 28, n. 10, p. 1817–1830, 1990.

STAFFORD, E. F.; TSENG, F. T. Two models for a family of flowshop sequencing problems. **European Journal of Operational Research**, v. 142, n. 2, p. 282–293, 2002.

STAFFORD, E. F.; TSENG, F. T.; GUPTA, J. N. D. Comparative evaluation of MILP flowshop models. **Journal of the Operational Research Society**, v. 56, n. 1, p. 88–101, 2005.

TAILLARD, E. Some efficient heuristic methods for the flow shop sequencing problem. **European Journal of Operational Research**, v. 47, n. 1, p. 65–74, 1990.

TAILLARD, E. Benchmarks for basic scheduling problems. **European Journal of Operational Research**, v. 64, n. 2, p. 278–285, 1993.

TANDON, M.; CUMMINGS, P.; LEVAN, M. Flowshop sequencing with non-permutation schedules. **Computers & Chemical Engineering**, v. 15, n. 8, p. 601–607, 1991.

TASGETIREN, M. F.; PAN, Q.-K.; SUGANTHAN, P.; CHEN, A. H.-L. A discrete artificial bee colony algorithm for the total flowtime minimization in permutation flow shops. **Information Sciences**, v. 181, n. 16, p. 3459–3475, 2011.

TSENG, F. T.; STAFFORD, E. F. New MILP models for the permutation flowshop problem. **Journal of the Operational Research Society**, v. 59, p. 1373–1386, 2008.

TSENG, F. T.; STAFFORD, E. F.; GUPTA, J. N. An empirical analysis of integer programming formulations for the permutation flowshop. **Omega**, v. 32, n. 4, p. 285–293, 2004.

TSENG, L.-Y.; LIN, Y.-T. A hybrid genetic local search algorithm for the permutation flowshop scheduling problem. **European Journal of Operational Research**, v. 198, n. 1, p. 84–92, 2009.

TSENG, L.-Y.; LIN, Y.-T. A genetic local search algorithm for minimizing total flowtime in the permutation flowshop scheduling problem. **International Journal of Production Economics**, v. 127, n. 1, p. 121–128, 2010.

VALLADA, E.; RUIZ, R.; FRAMIÑAN, J. M. New hard benchmark for flowshop scheduling problems minimising makespan. **European Journal of Operational Research**, v. 240, n. 3, p. 666–677, 2015.

VALLADA, E.; RUIZ, R.; MINELLA, G. Minimising total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics. **Computers & Operations Research**, v. 35, n. 4, p. 1350–1373, 2008.

VASILJEVIC, D.; DANILOVIC, M. Handling ties in heuristics for the permutation flow shop scheduling problem. **Journal of Manufacturing Systems**, v. 35, p. 1–9, 2015.

VÁZQUEZ-RODRÍGUEZ, J. A.; OCHOA, G. On the automatic discovery of variants of the NEH procedure for flow shop scheduling using genetic programming. **Journal of the Operational Research Society**, v. 62, n. 2, p. 381–396, 2011.

WAGNER, H. M. An integer linear-programming model for machine scheduling. **Naval Research Logistics Quarterly**, v. 6, n. 2, p. 131–140, 1959.

WILSON, J. M. Alternative formulations of a flow-shop scheduling problem. **Journal of the Operational Research Society**, v. 40, n. 4, p. 395–399, 1989.

WODECKI, M.; BOŻZEJKO, W. Solving the flow shop problem by parallel simulated annealing. In: WYRZYKOWSKI, R.; DONGARRA, J.; PAPRZYCKI, M.; WAŚNIEWSKI, J. (Ed.). **Parallel Processing and Applied Mathematics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 236–244.

XU, L.; HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K. SATzilla: Portfolio-based algorithm selection for SAT. **Journal of Artificial Intelligence Research**, AI Access Foundation, El Segundo, CA, USA, v. 32, n. 1, p. 565–606, jun. 2008.

YING, K.-C. Solving non-permutation flowshop scheduling problems by an effective iterated greedy heuristic. **The International Journal of Advanced Manufacturing Technology**, v. 38, n. 3, p. 348, 2007.

YING, K.-C.; LIN, S.-W. Multi-heuristic desirability ant colony system heuristic for non-permutation flowshop scheduling problems. **The International Journal of Advanced Manufacturing Technology**, v. 33, n. 7, p. 793–802, 2007.

ZHANG, Y.; LI, X.; WANG, Q. Hybrid genetic algorithm for permutation flowshop scheduling problems with total flowtime minimization. **European Journal of Operational Research**, v. 196, n. 3, p. 869–876, 2009.

APPENDIX A — BENCHMARKS

Table A.1: Upper bounds for the Taillard benchmark.

Inst.	n	m	$C_{\max}$	C_{sum}	Inst.	n	m	$C_{\max}$	C_{sum}	Inst.	n	m	$C_{\max}$	C_{sum}
ta001	20	5	1278	14033	ta041	50	10	2991	87114	ta081	100	20	6202	365463
ta002	20	5	1359	15151	ta042	50	10	2867	82820	ta082	100	20	6183	372449
ta003	20	5	1081	13301	ta043	50	10	2839	79931	ta083	100	20	6271	370027
ta004	20	5	1293	15447	ta044	50	10	3063	86446	ta084	100	20	6269	372393
ta005	20	5	1235	13529	ta045	50	10	2976	86377	ta085	100	20	6314	368915
ta006	20	5	1195	13123	ta046	50	10	3006	86587	ta086	100	20	6364	370908
ta007	20	5	1234	13548	ta047	50	10	3093	88750	ta087	100	20	6268	373408
ta008	20	5	1206	13948	ta048	50	10	3037	86727	ta088	100	20	6401	384525
ta009	20	5	1230	14295	ta049	50	10	2897	85441	ta089	100	20	6275	374423
ta010	20	5	1108	12943	ta050	50	10	3065	87998	ta090	100	20	6434	379296
ta011	20	10	1582	20911	ta051	50	20	3850	125831	ta091	200	10	10862	1046314
ta012	20	10	1659	22440	ta052	50	20	3704	119247	ta092	200	10	10480	1034195
ta013	20	10	1496	19833	ta053	50	20	3640	116459	ta093	200	10	10922	1046902
ta014	20	10	1377	18710	ta054	50	20	3723	120261	ta094	200	10	10889	1030481
ta015	20	10	1419	18641	ta055	50	20	3611	118184	ta095	200	10	10524	1034027
ta016	20	10	1397	19245	ta056	50	20	3681	120586	ta096	200	10	10329	1006195
ta017	20	10	1484	18363	ta057	50	20	3704	122880	ta097	200	10	10854	1053051
ta018	20	10	1538	20241	ta058	50	20	3691	122489	ta098	200	10	10730	1044875
ta019	20	10	1593	20330	ta059	50	20	3743	121872	ta099	200	10	10438	1026137
ta020	20	10	1591	21320	ta060	50	20	3756	123954	ta100	200	10	10675	1030299
ta021	20	20	2297	33623	ta061	100	5	5493	253266	ta101	200	20	11195	1227733
ta022	20	20	2099	31587	ta062	100	5	5268	242281	ta102	200	20	11203	1245271
ta023	20	20	2326	33920	ta063	100	5	5175	237832	ta103	200	20	11281	1269673
ta024	20	20	2223	31661	ta064	100	5	5014	227738	ta104	200	20	11275	1238349
ta025	20	20	2291	34557	ta065	100	5	5250	240301	ta105	200	20	11259	1227214
ta026	20	20	2226	32564	ta066	100	5	5135	232342	ta106	200	20	11176	1227604
ta027	20	20	2273	32922	ta067	100	5	5246	240366	ta107	200	20	11337	1243707
ta028	20	20	2200	32412	ta068	100	5	5094	230945	ta108	200	20	11301	1246123
ta029	20	20	2237	33600	ta069	100	5	5448	247921	ta109	200	20	11145	1234936
ta030	20	20	2178	32262	ta070	100	5	5322	242933	ta110	200	20	11284	1250596
ta031	50	5	2724	64802	ta071	100	10	5770	298385	ta111	500	20	26040	6698656
ta032	50	5	2834	68051	ta072	100	10	5349	274384	ta112	500	20	26520	6770735
ta033	50	5	2621	63162	ta073	100	10	5676	288114	ta113	500	20	26371	6739645
ta034	50	5	2751	68226	ta074	100	10	5781	301044	ta114	500	20	26456	6785991
ta035	50	5	2863	69351	ta075	100	10	5467	284681	ta115	500	20	26334	6729468
ta036	50	5	2829	66841	ta076	100	10	5303	269686	ta116	500	20	26469	6724085
ta037	50	5	2725	66253	ta077	100	10	5595	279463	ta117	500	20	26389	6691468
ta038	50	5	2683	64332	ta078	100	10	5617	290908	ta118	500	20	26560	6783916
ta039	50	5	2552	62981	ta079	100	10	5871	301970	ta119	500	20	26005	6711305
ta040	50	5	2782	68770	ta080	100	10	5845	291283	ta120	500	20	26457	6755722

Table A.2: Upper bounds for the VRF-large benchmark.

Inst.	m	$C_{\max}$	C_{sum}	Inst.	m	$C_{\max}$	C_{sum}	Inst.	m	$C_{\max}$	C_{sum}	Inst.	m	$C_{\max}$	C_{sum}
$n = 100$				$n = 300$				$n = 500$				$n = 700$			
1	20	6198	370505	1	20	16149	2523289	1	20	26411	6627488	1	20	36394	12462020
2	20	6306	376870	2	20	16512	2587590	2	20	26681	6651569	2	20	36337	12493179
3	20	6238	371265	3	20	16173	2527732	3	20	26409	6591256	3	20	36568	12546348
4	20	6245	371611	4	20	16181	2535577	4	20	26124	6569889	4	20	36452	12505160
5	20	6296	372351	5	20	16342	2553971	5	20	26781	6723570	5	20	36584	12615626
6	20	6321	371304	6	20	16137	2543509	6	20	26443	6610095	6	20	36671	12613222
7	20	6434	380903	7	20	16266	2534488	7	20	26433	6623457	7	20	36624	12633435
8	20	6104	360718	8	20	16416	2566575	8	20	26318	6605760	8	20	36522	12582915
9	20	6354	380394	9	20	16376	2572219	9	20	26442	6660460	9	20	36329	12472593
10	20	6145	365528	10	20	16899	2614287	10	20	26072	6523495	10	20	36417	12435677
1	40	7881	521213	1	40	18298	3132230	1	40	28548	7704603	1	40	38964	14369819
2	40	8007	528423	2	40	18454	3156461	2	40	28793	7758178	2	40	38775	14331534
3	40	7935	523271	3	40	18457	3166587	3	40	28607	7763377	3	40	38621	14225505
4	40	7932	522267	4	40	18351	3131237	4	40	28828	7770027	4	40	38785	14322418
5	40	8011	526676	5	40	18484	3163630	5	40	28683	7764420	5	40	38671	14247857
6	40	8023	528140	6	40	18449	3154768	6	40	28524	7673231	6	40	38710	14234796
7	40	8006	525428	7	40	18419	3150785	7	40	28760	7790028	7	40	38585	14197707
8	40	7979	525889	8	40	18392	3129925	8	40	28698	7787592	8	40	39059	14370105
9	40	7931	520608	9	40	18394	3160251	9	40	28870	7818312	9	40	38814	14287743
10	40	7952	525072	10	40	18401	3139338	10	40	28758	7788876	10	40	38850	14318043
1	60	9395	648645	1	60	20522	3707175	1	60	30861	8714335	1	60	41436	15798728
2	60	9596	671835	2	60	20399	3643848	2	60	30828	8722191	2	60	41375	15748792
3	60	9349	649076	3	60	20434	3691014	3	60	31125	8788261	3	60	41317	15761936
4	60	9426	663003	4	60	20395	3664961	4	60	30928	8751520	4	60	41401	15818918
5	60	9465	657055	5	60	20341	3672274	5	60	30935	8746090	5	60	41262	15792995
6	60	9667	676821	6	60	20388	3658510	6	60	31027	8762638	6	60	41340	15824919
7	60	9391	648846	7	60	20457	3692011	7	60	30928	8802172	7	60	40876	15699749
8	60	9534	671043	8	60	20410	3667736	8	60	30988	8745641	8	60	41474	15885806
9	60	9527	655747	9	60	20549	3702836	9	60	30978	8754250	9	60	41291	15824197
10	60	9598	668132	10	60	20472	3683425	10	60	31050	8787967	10	60	41377	15890062
$n = 200$				$n = 400$				$n = 600$				$n = 800$			
1	20	11305	1218846	1	20	21120	4314350	1	20	31433	9387438	1	20	41558	16210004
2	20	11265	1218865	2	20	21457	4377793	2	20	31418	9331959	2	20	41407	16175223
3	20	11327	1238395	3	20	21441	4385256	3	20	31429	9332251	3	20	41425	16232883
4	20	11208	1220315	4	20	21247	4342427	4	20	31547	9347021	4	20	41426	16189964
5	20	11208	1222215	5	20	21553	4359068	5	20	31448	9309212	5	20	41710	16279305
6	20	11367	1245221	6	20	21214	4333730	6	20	31717	9442067	6	20	42010	16393581
7	20	11380	1242387	7	20	21625	4407540	7	20	31527	9309328	7	20	41425	16046562
8	20	11141	1208818	8	20	21277	4310045	8	20	31564	9341869	8	20	41492	16141953
9	20	11123	1202829	9	20	21346	4312657	9	20	31577	9343846	9	20	41796	16162928
10	20	11310	1231792	10	20	21538	4382884	10	20	31130	9280685	10	20	41574	16167724
1	40	13132	1587197	1	40	23578	5200164	1	40	33839	10823949	1	40	43671	18233265
2	40	13102	1580719	2	40	23456	5189295	2	40	33467	10678033	2	40	43746	18295624
3	40	13264	1594558	3	40	23575	5222161	3	40	33866	10799825	3	40	43749	18317278
4	40	13232	1592167	4	40	23409	5205544	4	40	33693	10741208	4	40	43892	18337563
5	40	13043	1574224	5	40	23339	5153042	5	40	33553	10699427	5	40	43905	18400263
6	40	13124	1588395	6	40	23444	5194071	6	40	33809	10832704	6	40	43811	18311339
7	40	13299	1577987	7	40	23556	5229211	7	40	33686	10788797	7	40	43766	18184174
8	40	13238	1570803	8	40	23411	5195887	8	40	33482	10680459	8	40	43839	18292850
9	40	13166	1576462	9	40	23637	5224918	9	40	33697	10809443	9	40	43879	18276197
10	40	13228	1576366	10	40	23720	5236043	10	40	33642	10791755	10	40	43861	18326210
1	60	14990	1905428	1	60	25607	5970180	1	60	36198	12041857	1	60	46470	20155848
2	60	14954	1909316	2	60	25656	5953027	2	60	36184	12045538	2	60	46493	20165942
3	60	15200	1918779	3	60	25821	5977399	3	60	36201	12025222	3	60	46389	20079863
4	60	15044	1890335	4	60	25837	6013511	4	60	36136	12026209	4	60	46457	20079888
5	60	15130	1911805	5	60	25877	5941405	5	60	36153	12076881	5	60	46401	20126125
6	60	15035	1898598	6	60	25536	5905353	6	60	36116	11996401	6	60	46421	20121428
7	60	15040	1901833	7	60	25600	5948560	7	60	36179	12038658	7	60	46319	20097752
8	60	14968	1905974	8	60	25800	5955653	8	60	36185	12067964	8	60	46474	20130414
9	60	15022	1894955	9	60	25882	5999431	9	60	36195	12109094	9	60	46538	20140038
10	60	15000	1891278	10	60	25767	5964812	10	60	36163	12052077	10	60	46244	20024895

The upper bounds for $C_{\max}$ are due to Vallada, Ruiz and Framiñan (2015), while the upper bounds for C_{sum} are the best solutions obtained by A_7 or MRSILS(BSCH) in the experiments described in Section 3.1.2.

APPENDIX B — ALGORITHMS

In this chapter, we present algorithms in pseudocode for each one of the implemented local search procedures for minimizing total completion time in permutation flow shops. We consider that the best solution visited so far π^* is implicitly mantained, and that n and m are the number of jobs and machines, respectively.

Algorithm 8 `insertLS`

Input: Solution π, Tiebreaker T
Output: Best solution found π^*
function INSERTLS(π, T)
repeat
$\pi' = \pi$
for $j = 1$ to n **do**
k = j^{th} job in π'
π' = REMOVE(k,π')
π' = INSERTBESTPOSITION(k,π',T)
if $f(\pi') \leq f(\pi)$ **then**
$\pi = \pi'$
end if
end for
until no improvement is found
return π^*
end function

Algorithm 9 `lsTasgetiren`

Input: Solution π
Output: Best solution found π^*
function LSTASGETIREN(π)
do
$\pi' = \pi$
π'' = INSERTTASGETIREN(π')
π'' = SWAPTASGETIREN(π'')
if $f(\pi'') \leq f(\pi)$ **then**
$\pi = \pi''$
end if
while $\pi'' \neq \pi'$
return π^*
end function

Algorithm 10 `swapTasgetiren`

Input: Solution π
Output: Best solution found π^*
function SWAPTASGETIREN(π)
$i = 1$
while $i < n$ **do**
$j = i + 1$
while $j \leq n$ **do**
$\pi' = \pi$
π' = SWAP(i,j,π')
if $f(\pi') \leq f(\pi)$ **then**
$\pi = \pi'$
$i = 1$
$j = i + 1$
else
$j = j + 1$
end if
end while
$i = i + 1$
end while
return π^*
end function

Algorithm 11 `insertTasgetiren`

Input: Solution π
Output: Best solution found π^*
function INSERTTASGETIREN(π)
$i = 1$
while $i < n$ **do**
$j = i + 1$
while $j \leq n$ **do**
$\pi' = \pi$
k = i^{th} job in π'
π' = REMOVE(k,π')
π' = INSERTATPOSITION(k,j,π')
if $f(\pi') \leq f(\pi)$ **then**
$\pi = \pi'$
$i = 1$
$j = i + 1$
else
$j = j + 1$
end if
end while
$i = i + 1$
end while
return π^*
end function

Algorithm 12 `insertFPR`

Input: Solution π
Output: Best solution found π^*

```
function INSERTFPR(π)
    π' = π
    i = 1
    t = 0
    while t < n do
        i = (i + 1)%n
        j = i^th job in π'
        π'' = REMOVE(j,π')
        π'' = INSERTBESTPOSITION(j,π'')
        if f(π'') ≤ f(π') then
            π' = π''
            t = 0
        else
            t = t + 1
        end if
    end while
    return π*
end function
```

Algorithm 13 `fpe`

Input: Solution π, Integer x
Output: Best solution found π^*

```
function FPE(π, x)
    π' = π
    do
        π'' = π'
        for i = 1 to n do
            k = i^th job in π''
            for j = 1 to x do
                π''' = π'
                p = position of job k in π'''
                if p + j > n then
                    break
                end if
                π''' = SWAP(p,p + j,π''')
                if f(π''') ≤ f(π') then
                    π' = π'''
                end if
            end for
        end for
    while f(π') < f(π'')
    return π*
end function
```

Algorithm 14 `bpe`

Input: Solution π, Integer x
Output: Best solution found π^*

```
function BPE(π, x)
    π' = π
    do
        π'' = π'
        for i = n to 1 do
            k = i^th job in π''
            for j = 1 to x do
                π''' = π'
                p = position of job k in π'''
                if p − j < 1 then
                    break
                end if
                π''' = SWAP(p,p − j,π''')
                if f(π''') ≤ f(π') then
                    π' = π'''
                end if
            end for
        end for
    while f(π') < f(π'')
    return π*
end function
```

Algorithm 15 `iRZ`

Input: Solution π
Output: Best solution found π^*

```
function IRZ(π)
    π' = π'' = π
    do
        π' = π''
        for i = 1 to n do
            j = i^th job in π'
            π''' = REMOVE(j,π'')
            π''' = INSERTBESTPOSITION(j,π''')
            if f(π''') ≤ f(π'') then
                π'' = π'''
            end if
        end for
    while f(π'') < f(π')
    return π*
end function
```

Algorithm 16 `riRZ`

Input: Solution π
Output: Best solution found π^*
function RIRZ(π)
$\pi' = \pi'' = \pi$
do
$\pi' = \pi''$
for $i = n$ to 1 **do**
j = i^{th} job in π'
π''' = REMOVE(j,π'')
π''' = INSERTBESTPOSITION(j,π''')
if $f(\pi''') \leq f(\pi'')$ **then**
$\pi'' = \pi'''$
end if
end for
while $f(\pi'') < f(\pi')$
return π^*
end function

Algorithm 17 `raiRZ`

Input: Solution π
Output: Best solution found π^*
function RAIRZ(π)
$\pi' = \pi'' = \pi$
do
$\pi' = \pi''$
π^R = random permutation of jobs
for $i = 1$ to n **do**
j = i^{th} job in π^R
π''' = REMOVE(j,π'')
π''' = INSERTBESTPOSITION(j,π''')
if $f(\pi''') \leq f(\pi'')$ **then**
$\pi'' = \pi'''$
end if
end for
while $f(\pi'') < f(\pi')$
return π^*
end function

Algorithm 18 `swapInc`

Input: Solution π, Integers q_{min}, q_{max} and s_{max}
Output: Best solution found π^*

```
function SWAPINC(π)
    q = q_min
    do
        π' = π
        i = 1
        do
            j = i + q
            π'' = π
            π'' = SWAP(i, j, π'')
            if f(π'') < f(π) then
                π = π''
            end if
            i = i + 1
        while i + q ≤ n
        if f(π) < f(π') then
            q = q_min
        else
            q = q + 1
        end if
    while q < q_max
    return π*
end function
```

Algorithm 19 `swapFirst`

Input: Solution π, Integer n_{ls}
Output: Best solution found π^*

```
function SWAPFIRST(π, x)
    π' = π
    p = 1
    r = n_ls × (n − 1)
    while r > 0 do
        π'' = SWAP(p,p + 1,π')
        if f(π'') ≤ f(π') then
            π' = π''
        end if
        if p + 1 < n then
            p = p + 1
        else
            p = 1
        end if
        r = r − 1
    end while
    return π*
end function
```

Algorithm 20 `swapBest`

Input: Solution π, Integer n_{ls}
Output: Best solution found π^*

```
function SWAPBEST(π, x)
    π' = π
    r = n_ls × (n − 1)
    while r > 0 do
        Evaluate all possible adjacent swaps in π' and choose the best pair of jobs i, i+1
        π'' = SWAP(i,i + 1,π')
        if f(π'') ≤ f(π') then
            π' = π''
        else
            break
        end if
        r = r − 1
    end while
    return π*
end function
```

Algorithm 21 `swapR`

Input: Solution π
Output: Best solution found π^*

```
function SWAPR(π)
    π' = π'' = π
    for i = 1 to n do
        for j = 1 to n do
            if i ≠ j then
                π'' = SWAP(i,j,π')
                k = j^th job in π'
                π'' = REMOVE(k,π'')
                π'' = INSERTBESTPOSITION(k,π'')
                if f(π'') ≤ f(π') then
                    π' = π''
                end if
            end if
        end for
    end for
    return π*
end function
```

APPENDIX C — ADDITIONAL EXPERIMENTS WITH TOTAL COMPLETION TIME MINIMIZATION

Our work on the NPFSSP involved generating a high-quality initial permutation solution. Therefore, we updated our grammar for the PFSSP presented in Figure 3.1 as part of our work on the NPFSSP. We repeated our computational experiments for the PFSSP with the updated grammar and presented the updated results in this appendix. In comparison to the grammar in Figure 3.1, there are six new perturbation strategies, a new constructive heuristic (*RC*), three new acceptance criteria, and the possibility of defining the frequency to apply the second local search procedure through parameter f_{ls}. We removed the option to use a pool of solutions, as it was never selected in our previous experiments. Instead, we focus on simpler ILS methods without the pool. The new grammar is shown in Figure C.1, and the set of parameters to be configured with irace is presented in Table C.1.

We ran irace ten times and selected the best candidate of each run. Each irace run had a budget of 50000 candidate runs with a time limit of $10nm$ milliseconds. The training set contained 120 randomly generated instances with the same dimensions as those in the Taillard benchmark, with processing times in the interval $[1, 99]$. We used version 3.0 of irace, with all parameters set to default values. The algorithms are presented in Table C.2.

Many similarities with the algorithms from Section 3.1.2 can be seen, such as the initial solution being built with *BSCH*, the variation in the tiebreaker for the perturbation, and the alternation between insertion and swap neighborhoods for the methods that use two local search procedures. On the other hand, record-to-record travel is the acceptance criterion of all algorithms (as opposed to Metropolis in Section 3.1.2), and seven methods use newly introduced perturbation strategies.

We evaluated the algorithms on Taillard's and VRF-large benchmarks. Each algorithm ran for $30nm$ milliseconds with 10 replications per instance on a computer with two Intel Xeon E5-2697 v2 processors (12 physical cores each) at 2.7 GHz, running Ubuntu 18.04.3. The algorithms were implemented in C++ and compiled with the GNU C++ compiler version 7.4 with an optimization level of 3.

We present the results for the Taillard benchmark in Table C.3 as the ARD in percent from the values in Appendix A. The results are mostly similar to those of Section 3.1.2, although the best algorithm in that section had lower ARD than the best one in Table 4.4.

Figure C.1: A grammar of iterated local search algorithms for the PFSSP. For simplicity, the numerical parameters have been omitted.

```
 1  <START>       ::=  ILS(<INI_SOL>,<PERTURB>,<LS>,<ACCEPT>)
 2  <INI_SOL>     ::=  NEHCsum(<TIEBREAK>) | LR |
 3                     FRB5(<ORDER>,<TIEBREAK>) | RC | BSCH
 4  <ORDER>       ::=  noninc | nondec | KK1 | KK2
 5  <TIEBREAK>    ::=  KK1 | KK2 | first | last | random
 6  <PERTURB>     ::=  ri | gi(<TIEBREAK>) | rs | gs |
 7                     ras | gi_asls(<TIEBREAK>) |
 8                     ils_gi(<TIEBREAK>) |
 9                     gi_ils(<TIEBREAK>)
10  <LS>          ::=  ϵ | <LS_PROC> |
11                     alternate(<LS_PROC>,<LS_PROC>)
12  <LS_PROC>     ::=  insertion(<TIEBREAK>) | swapTasgetiren |
13                     swapInc | insertTasgetiren |
14                     lsTasgetiren | fpe | bpe | iRZ |
15                     riRZ | raiRZ | insertFPR | swapFirst |
16                     swapBest | swapR
17  <ACCEPT>      ::=  met | lac | rrt | thr
```

The results for the VRF-large benchmark are presented in Table C.4 as the ARD from the upper bounds of Vallada, Ruiz and Framiñan (2015) (see Appendix A). We cannot compare these results to the previous ones because we only evaluated the best algorithm of Section 3.1.2 on this benchmark.

Table C.1: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
ini_sol	Categorical	*LR*, *NEHCsum*, *FRB5*, *BSCH*, *RC*	
order	Categorical	*noninc*, *nondec*, *KK1*, *KK2*	*ini_sol* $\in \{$*NEHCsum*, *FRB5*$\}$
tb_ini_sol	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in \{$*NEHCsum*, *FRB5*$\}$
tb_perturb	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*perturb* $\in \{$*gi*, *gi_asls*, *ils_gi*, *gi_ils*$\}$
tb_ls_proc_1	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $\in \{$*insertLS*$\}$
tb_ls_proc_2	Categorical	*KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $\in \{$*insertLS*$\}$
perturb	Categorical	*ri*, *gi*, *rs*, *gs*, *ras*, *gi_asls*, *ils_gi*, *gi_ils*	
ls	Categorical	$0, 1, 2$	
ls_proc_1	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	*ls* $\in \{1, 2\}$
ls_proc_2	Categorical	*insertLS*, *fpe*, *bpe*, *swapTasgetiren*, *insertTasgetiren*, *lsTasgetiren*, *swapInc*, *iRZ*, *riRZ*, *raiRZ*, *viRZ*, *swapFirst*, *swapBest*, *swapR*	*ls* $= 2$
accept	Categorical	*met*, *lac*, *rrt*, *thr*	
r	Ordinal	$1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 1000$	*ini_sol* $=$ *RC*
d	Integer	$[1, 20]$	
n_{ls}	Ordinal	$1, 2, 3, 4, \infty$	*ini_sol* $=$ *FRB5* or *ls* $\in \{1, 2\}$ or *perturb* $=$ *ils_gi*
α	Real	$[0.01, 1.0]$	*accept* $=$ *met*
l	Ordinal	$1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000$	*accept* $=$ *lac*
$rrtd$	Ordinal	$0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500$	*accept* $=$ *rrt*
$thres$	Ordinal	$0, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5$	*accept* $=$ *thr*
f_{ls}	Ordinal	$2, 3, 4, 5, 10$	*ls* $= 2$

The columns contain the name of the parameter, the type specified in irace, the domain, and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

Table C.2: Automatically designed algorithms for minimizing the total completion time on the PFSSP.

		Perturbation			Local search				Acceptance	
Alg.	CH	Method	Tie	d	LS Proc. 1	LS Proc. 2	f_{ls}	n_{ls}	Crit.	*rrtd*
A_0	*BSCH*	*gi_asls*	*first*	8	*raiRZ*	*bpe*	4	4	*rrt*	100
A_1	*BSCH*	*gi_asls*	*KK1*	7	*swapR*	*riRZ*	3	2	*rrt*	200
A_2	*BSCH*	*gi_ils*	*KK2*	4	*fpe*	*riRZ*	5	4	*rrt*	200
A_3	*BSCH*	*gi_asls*	*first*	6	*fpe*	*iRZ*	4	4	*rrt*	200
A_4	*BSCH*	*gi_asls*	*KK2*	7	*raiRZ*	*swapR*	3	∞	*rrt*	80
A_5	*BSCH*	*gi_asls*	*first*	5	*bpe*	*iRZ*	5	2	*rrt*	100
A_6	*BSCH*	*gi*	*first*	8	*fpe*	*insertFPR*	3	-	*rrt*	100
A_7	*BSCH*	*gi*	*last*	7	*bpe*	*raiRZ*	3	2	*rrt*	300
A_8	*BSCH*	*gi*	*last*	8	*swapR*	*iRZ*	2	3	*rrt*	90
A_9	*BSCH*	*gi_asls*	*rand*	5	*insertFPR*	-	-	-	*rrt*	90

Each row contains the components and parameter values for one of the algorithms obtained with irace, named A_0 to A_9 in the first column. The second column contains the constructive heuristic. The next three columns contain the perturbation strategy, its tiebreaker, and the perturbation intensity d. The next four columns refer to the local search and contain the first procedure, the second procedure, if any, the parameter f_{ls} that defines the frequency to perform the second local search procedure, and the maximum number of full neighborhood scans n_{ls}. The last two columns contain the acceptance criterion and its respective parameter.

Table C.3: ARD for the Taillard benchmark.

Inst.	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
20×5	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.000
20×10	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.000
20×20	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.000
50×5	0.165	0.178	0.177	0.172	0.150	0.157	0.154	0.237	**0.145**	0.160	0.145
50×10	0.535	0.515	0.492	0.500	0.528	0.634	0.517	**0.416**	0.561	0.532	0.416
50×20	0.439	0.450	0.419	0.454	0.478	0.559	0.470	**0.377**	0.477	0.489	0.377
100×5	0.084	0.082	0.078	0.076	0.075	**0.071**	0.072	0.097	0.073	0.086	0.071
100×10	0.236	**0.230**	0.239	0.232	0.236	0.245	0.236	0.238	0.241	0.238	0.230
100×20	0.464	0.457	0.452	0.459	0.470	0.510	0.468	**0.442**	0.477	0.468	0.442
200×10	-0.675	**-0.681**	-0.677	-0.676	-0.674	-0.672	-0.672	-0.677	-0.671	-0.672	-0.681
200×20	-0.865	-0.862	-0.861	-0.861	-0.862	-0.848	**-0.867**	**-0.867**	-0.858	-0.864	-0.867
500×20	-1.891	-1.890	-1.890	**-1.892**	-1.891	-1.891	-1.891	**-1.892**	-1.890	-1.891	-1.892
Avg.	-0.126	-0.127	-0.131	-0.128	-0.124	-0.103	-0.126	**-0.136**	-0.121	-0.121	-0.146

The results presented in this table are the ARD over the upper bounds in Table A.1. A_0 to A_9 are the algorithms found with irace. The best results per instance group are highlighted in bold, and column "Best" contains a combination of the best results.

Table C.4: ARD for the VRF-large benchmark.

Inst.	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
100×20	0.114	0.126	0.121	0.117	0.122	0.148	0.125	**0.087**	0.129	0.117	0.087
100×40	0.146	0.157	0.129	0.138	0.147	0.173	0.142	**0.118**	0.165	0.148	0.118
100×60	0.129	0.139	0.125	0.132	0.131	0.178	0.140	**0.102**	0.147	0.140	0.102
200×20	**0.012**	0.013	0.013	0.013	0.014	0.016	0.015	0.013	0.016	0.014	0.012
200×40	0.036	0.037	0.035	0.034	0.037	0.048	0.042	**0.032**	0.046	0.036	0.032
200×60	0.033	0.039	0.030	0.033	**0.029**	0.047	0.038	0.034	0.043	0.030	0.029
300×20	0.009	0.012	0.009	0.012	**0.008**	0.013	0.010	0.009	0.012	0.011	0.008
300×40	0.013	0.013	**0.012**	0.014	0.015	0.020	0.018	0.015	0.020	0.015	0.012
300×60	0.009	0.011	**0.006**	0.010	0.008	0.021	0.016	0.010	0.021	0.010	0.006
400×20	**0.004**	**0.004**	0.005	0.005	**0.004**	**0.004**	0.005	0.005	**0.004**	0.005	0.004
400×40	0.008	0.008	**0.007**	**0.007**	0.011	0.011	0.012	0.010	0.013	0.009	0.007
400×60	0.018	0.020	**0.014**	0.017	0.016	0.020	0.021	0.017	0.024	0.019	0.014
500×20	0.003	**0.002**	0.003	**0.002**	0.003	0.003	0.003	0.003	0.003	0.003	0.002
500×40	0.010	0.011	**0.008**	0.010	0.010	0.012	0.012	0.011	0.013	0.010	0.008
500×60	0.007	0.005	**0.004**	0.007	0.005	0.008	0.009	0.007	0.010	0.007	0.004
600×20	**0.002**	**0.002**	**0.002**	**0.002**	0.003	**0.002**	**0.002**	0.003	0.003	0.003	0.002
600×40	**0.005**	**0.005**	**0.005**	**0.005**	**0.005**	0.006	0.006	**0.005**	0.008	0.006	0.005
600×60	0.009	**0.008**	**0.008**	**0.008**	0.009	0.011	0.010	0.010	0.013	**0.008**	0.008
700×20	0.001	**0.000**	0.002	0.003	0.002	0.002	0.003	**0.000**	0.003	0.003	0.000
700×40	0.003	**0.002**	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.002
700×60	0.009	0.009	**0.008**	0.009	0.010	0.009	0.012	0.009	0.011	0.010	0.008
800×20	**0.001**	0.002	0.002	0.003	0.004	0.003	0.004	0.003	0.005	0.004	0.001
800×40	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**	0.004	0.004	**0.003**	0.003
800×60	0.008	0.008	0.007	0.008	**0.006**	0.009	0.010	0.008	0.012	0.010	0.006
Avg.	0.025	0.027	0.023	0.025	0.025	0.032	0.028	**0.022**	0.030	0.026	0.020

The results presented in this table are the ARD over the upper bounds in Table A.2. A_0 to A_9 are the algorithms found with irace. The best results per instance group are highlighted in bold, and column "Best" contains a combination of the best results.

APPENDIX D — ADDITIONAL EXPERIMENTS WITH MAKESPAN MINIMIZATION

In this appendix we address the PFSSP with makespan minimization. As in Appendix C, we repeated the experiments described in Section 3.2.2 with the updated grammar for the PFSSP that was obtained during our work on the NPFSSP. In a comparison to the grammar in Figure 3.2, there is a new tiebreaker (*FF*), a new acceptance criterion (*thr*), and the possibility of defining the frequency to apply the second local search procedure through parameter f_{ls}. The new grammar is shown in Figure D.1 and the set of parameters to be configured with irace is presented in Table D.1.

We ran irace ten times and selected the best candidate of each run. Each irace run had a budget of 50000 candidate runs with a time limit of $10nm$ milliseconds. The training set contained 120 randomly generated instances with the same dimensions as those in the Taillard benchmark, with processing times in the interval $[1, 99]$. We used version 3.0 of irace, with all parameters set to default values. The algorithms are presented in Table D.2.

Figure D.1: A grammar of iterated local search algorithms for the PFSSP. For simplicity, the numerical parameters have been omitted.

```
1  <START>      ::=  ILS(<INI_SOL>,<PERTURB>,<LS>,<ACCEPT>)
2  <INI_SOL>    ::=  NEH(<TIEBREAK>) | RC |
3                     FRB5(<ORDER>,<TIEBREAK>)
4  <ORDER>      ::=  noninc | nondec | KK1 | KK2
5  <TIEBREAK>   ::=  FF | KK1 | KK2 | first | last | random
6  <PERTURB>    ::=  ri | gi(<TIEBREAK>) | rs | gs |
7                     ras | gi_asls(<TIEBREAK>) |
8                     ils_gi(<TIEBREAK>) |
9                     gi_ils(<TIEBREAK>)
10 <LS>         ::=  ϵ | <LS_PROC> |
11                    alternate(<LS_PROC>,<LS_PROC>)
12 <LS_PROC>    ::=  insertion(<TIEBREAK>) | NS |
13                    Pc(<TIEBREAK>)
14 <ACCEPT>     ::=  met | lac | rrt | thr
```

Table D.1: Tunable parameters of the algorithm construction.

Parameter	Type	Values	Conditions
ini_sol	Categorical	*NEH*, *FRB5*, *RC*	
order	Categorical	*noninc*, *nondec*, *KK1*, *KK2*	*ini_sol* $\in$ {*NEH*, *FRB5*}
tb_ini_sol	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ini_sol* $\in$ {*NEH*, *FRB5*}
tb_perturb	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*perturb* $\in$ {*gi*, *gi_asls*, *ils_gi*, *gi_ils*}
tb_ls_proc_1	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_1* $\in$ {*insertLS*, *Pc*}
tb_ls_proc_2	Categorical	*FF*, *KK1*, *KK2*, *first*, *last*, *random*	*ls_proc_2* $\in$ {*insertLS*, *Pc*}
perturb	Categorical	*ri*, *gi*, *rs*, *gs*, *ras*, *gi_asls*, *ils_gi*, *gi_ils*	
ls	Categorical	$0, 1, 2$	
ls_proc_1	Categorical	*insertLS*, *NS*, *Pc*	*ls* $\in \{1, 2\}$
ls_proc_2	Categorical	*insertLS*, *NS*, *Pc*	*ls* $= 2$
accept	Categorical	*met*, *lac*, *rrt*, *thr*	
r	Ordinal	$1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,$ $300, 400, 500, 1000$	*ini_sol* $=$ *RC*
d	Integer	$[1, 20]$	
n_{ls}	Ordinal	$1, 2, 3, 4, \infty$	*ini_sol* $=$ *FRB5* or *ls* $\in \{1, 2\}$ or *perturb* $=$ *ils_gi*
α	Real	$[0.01, 1.0]$	*accept* $=$ *met*
l	Ordinal	$1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600,$ $700, 800, 900, 1000$	*accept* $=$ *lac*
$rrtd$	Ordinal	$0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,$ $300, 400, 500$	*accept* $=$ *rrt*
$thres$	Ordinal	$0, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.02,$ $0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,$ $0.2, 0.3, 0.4, 0.5$	*accept* $=$ *thr*
f_{ls}	Ordinal	$2, 3, 4, 5, 10$	*ls* $= 2$

The columns contain the name of the parameter, the type specified in irace, the domain, and the conditions for the parameter to be active. If no condition is shown, the parameter is always active.

Table D.2: Automatically designed algorithms for minimizing the makespan on the PFSSP.

	Initial solution					Perturbation			Local search						Acceptance	
Alg.	CH	Order	Tie	r	*rev*	Method	Tie	d	LS Proc. 1	Tie 1	LS Proc. 2	Tie 2	f_{ls}	n_{ls}	Crit.	α, l
A_0	*FRB5*	*noninc*	*last*	-	no	*ils_gi*	*FF*	1	*insertLS*	*FF*	-	-	-	∞	*met*	0.6
A_1	*FRB5*	*KK2*	*KK1*	-	no	*gi_asls*	*FF*	6	*insertLS*	*FF*	-	-	-	∞	*met*	0.5
A_2	*RC*	-	-	60	no	*gi_ils*	*KK2*	2	*insertLS*	*FF*	-	-	-	∞	*met*	0.7
A_3	*NEH*	*FF*	-	-	no	*gi_ils*	*KK1*	3	*insertLS*	*FF*	-	-	-	∞	*met*	0.8
A_4	*NEH*	*FF*	-	-	yes	*gs*	-	5	*insertLS*	*FF*	-	-	-	∞	*met*	0.7
A_5	*RC*	-	-	90	no	*gs*	-	2	*insertLS*	*FF*	-	-	-	∞	*lac*	300
A_6	*FRB5*	*FF*	-	-	no	*gi_ils*	*FF*	3	*insertLS*	*FF*	*insertLS*	*KK1*	5	4	*met*	0.6
A_7	*FRB5*	*KK1*	*rand*	-	yes	*gi_ils*	*FF*	2	*insertLS*	*FF*	*insertLS*	*KK2*	5	4	*met*	0.6
A_8	*FRB5*	*KK1*	*KK1*	-	no	*gs*	-	5	*insertLS*	*FF*	*insertLS*	*KK1*	5	∞	*met*	0.6
A_9	*RC*	-	-	200	no	*gi_ils*	*FF*	3	*insertLS*	*FF*	-	-	-	∞	*met*	0.6

Each row contains contains the components and parameter values for one of the algorithms obtained with irace, named A_0 to A_9 in the first column. The other columns contain, respectively, the constructive heuristic to generate the initial solution, the ordering criteria for the constructive heuristic, tiebreaker for the constructive heuristic, the value for parameter r when *RC* is selected, whether or not to solve the reversed instance with the constructive heuristic, perturbation strategy, the tiebreaker for the perturbation, perturbation intensity d, first local search procedure, its tiebreaker, second local search procedure, its respective tiebreaker, the frequency f_{ls} to perform the second local search procedure, the maximum number of full neighborhood scans n_{ls} for local search procedures, the acceptance criterion, and the value for its respective parameter.

In general, the algorithms have many similarities with those of Section 3.2.2, such as the variation regarding the initial solution construction, the usage of the insertion local search procedure, although now mostly with the *FF* tiebreaker and the *met* acceptance criterion. On the other hand, the selected perturbation strategies are mostly different from those of Section 3.2.2, and frequently use the newly added *FF* tiebreaker.

We evaluated the algorithms on Taillard's and VRF-large benchmarks. Each algorithm ran for $30nm$ milliseconds with 10 replications per instance on a computer with two Intel Xeon E5-2697 v2 processors (12 physical cores each) at 2.7 GHz, running Ubuntu 18.04.3. The algorithms were implemented in C++ and compiled with the GNU C++ compiler version 7.4 with an optimization level of 3.

We present the results for the Taillard benchmark in Table D.3 as the ARD in percent from the values in Appendix A. The results showed improvements in a comparison to those of Section 3.2.2. The algorithm with the highest ARD in Table D.3 still has lower ARD than the best algorithm of Section 3.2.2. However, note that the experiments were performed on a different computer. Thus a direct comparison is not precise.

Table D.3: ARD for the Taillard benchmark.

Inst.	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
20×5	**0.000**	**0.000**	0.016	0.016	0.024	0.016	**0.000**	**0.000**	0.024	**0.000**	0.000
20×10	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.044	**0.000**	**0.000**	**0.000**	**0.000**	0.000
20×20	0.015	**0.000**	0.002	**0.000**	0.003	0.046	**0.000**	0.003	0.004	**0.000**	0.000
50×5	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.000
50×10	**0.277**	0.307	0.300	0.320	0.329	0.324	0.300	0.295	0.303	0.306	0.277
50×20	0.369	0.391	0.382	0.386	0.405	0.614	0.395	0.376	0.381	**0.345**	0.345
100×5	**0.000**	0.001	0.001	**0.000**	0.003	0.010	**0.000**	0.002	0.004	**0.000**	0.000
100×10	0.026	0.026	0.025	**0.023**	0.046	0.056	0.025	0.026	0.036	0.031	0.023
100×20	0.498	0.531	**0.462**	0.521	0.499	0.500	0.514	0.479	0.503	0.502	0.462
200×10	0.035	0.036	**0.033**	**0.033**	0.036	0.037	0.034	0.037	0.034	0.035	0.033
200×20	0.643	0.681	0.634	0.652	0.627	0.662	0.659	0.671	**0.608**	0.670	0.608
500×20	0.283	0.289	0.280	0.269	0.284	0.290	0.259	0.258	**0.255**	0.286	0.255
Avg.	0.179	0.188	**0.178**	0.185	0.188	0.216	0.182	0.179	0.179	0.181	0.167

The results presented in this table are the ARD over the upper bounds in Table A.1. A_0 to A_9 are the algorithms found with irace. The best results per instance group are highlighted in bold, and column "Best" contains a combination of the best results.

The results for the VRF-large benchmark are presented in Table D.4 as the ARD from the upper bounds of Vallada, Ruiz and Framiñan (2015) (see Appendix A).

Table D.4: ARD for the VRF-large benchmark.

Inst.	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	Best
100×20	0.224	0.234	**0.120**	0.215	0.231	0.259	0.209	0.165	0.192	0.182	0.120
100×40	0.222	0.385	**0.142**	0.201	0.337	0.420	0.246	0.229	0.356	0.175	0.142
100×60	0.352	0.486	**0.221**	0.308	0.460	0.515	0.301	0.307	0.492	0.278	0.221
200×20	0.174	0.219	0.151	0.154	0.165	0.157	0.200	0.178	**0.075**	0.174	0.075
200×40	0.140	0.283	0.111	**0.108**	0.325	0.160	0.384	0.294	0.238	0.261	0.108
200×60	0.109	0.293	**0.000**	0.010	0.201	0.191	0.180	0.158	0.182	0.143	0.000
300×20	0.164	0.188	0.193	0.184	0.154	0.137	0.216	0.165	**0.066**	0.207	0.066
300×40	0.217	0.123	0.294	0.289	0.564	0.144	0.412	0.355	**0.063**	0.391	0.063
300×60	0.128	0.214	0.130	0.106	0.324	**0.071**	0.368	0.247	0.190	0.235	0.071
400×20	0.103	0.127	0.113	0.123	0.116	0.112	0.106	0.092	**0.044**	0.133	0.044
400×40	0.202	0.012	0.262	0.268	0.567	0.103	0.382	0.270	**-0.100**	0.335	-0.100
400×60	0.101	0.206	0.287	0.249	0.485	**0.075**	0.284	0.331	0.112	0.359	0.075
500×20	0.138	0.145	0.140	0.128	0.136	0.130	0.150	0.114	**0.073**	0.154	0.073
500×40	0.169	-0.097	0.286	0.369	0.516	0.113	0.344	0.200	**-0.197**	0.346	-0.197
500×60	0.116	0.081	0.287	0.234	0.501	0.062	0.230	0.232	**0.005**	0.318	0.005
600×20	0.151	0.154	0.133	0.125	0.111	0.155	0.110	0.103	**0.069**	0.143	0.069
600×40	0.117	-0.188	0.209	0.254	0.445	0.028	0.168	0.109	**-0.215**	0.220	-0.215
600×60	0.145	**0.034**	0.397	0.328	0.525	0.156	0.265	0.295	0.083	0.441	0.034
700×20	0.109	0.113	0.099	0.088	0.085	0.107	0.078	0.072	**0.054**	0.098	0.054
700×40	0.037	**-0.365**	0.093	0.190	0.351	-0.066	0.087	-0.004	-0.310	0.139	-0.365
700×60	0.164	-0.109	0.282	0.283	0.488	0.063	0.144	0.142	**-0.131**	0.305	-0.131
800×20	0.048	0.063	0.063	0.048	0.078	0.074	0.050	0.051	**0.040**	0.058	0.040
800×40	0.091	**-0.337**	0.087	0.190	0.283	-0.040	0.069	0.007	-0.256	0.120	-0.337
800×60	0.153	**-0.180**	0.272	0.342	0.428	0.065	0.180	0.140	-0.161	0.281	-0.180
Avg.	0.149	0.087	0.182	0.200	0.328	0.133	0.215	0.177	**0.040**	0.229	-0.011

The results presented in this table are the ARD over the upper bounds in Table A.2. A_0 to A_9 are the algorithms found with irace. The best results per instance group are highlighted in bold, and column "Best" contains a combination of the best results.

APPENDIX E — CONFIGURAÇÃO AUTOMÁTICA DE ALGORITMOS PARA PROBLEMAS DE AGENDAMENTO EM FLOW SHOP

A resolução eficiente de problemas de otimização ajuda a melhorar nossas vidas, embora muitas vezes passe despercebida. Por exemplo, nos beneficiamos da redução dos prazos de entrega de produtos comprados online devido ao aprimoramento dos métodos de solução de problemas de logística, ou da redução de custos dos produtos devido aos ganhos de eficiência nos processos de fabricação. Problemas de agendamento, em particular, modelam muitos problemas de otimização enfrentados pelas indústrias de manufatura e serviços. Esses problemas geralmente consistem em alocar recursos de forma eficiente para tarefas cujo processamento requer um determinado tempo de tal forma que uma métrica de custo seja minimizada. Uma vez que os custos envolvidos na indústria costumam ser altos, a pesquisa por métodos de resolução mais eficientes é altamente desejável. No entanto, a otimização em ambientes de produção pode envolver problemas combinatórios desafiadores. A maioria deles são NP-difíceis (COOK, 1971; LEVIN, 1973), ou seja, nenhum algoritmo de tempo polinomial eficiente para resolvê-los é conhecido. Além disso, muitas vezes é impraticável resolver problemas de grande porte que surgem da indústria em um curto espaço de tempo por meio de abordagens exatas, como programação matemática. Uma solução comum nesses casos é a adoção de métodos heurísticos.

Heurísticas geralmente produzem resultados de alta qualidade, mas por outro lado podem ser complexas e difíceis de projetar. Além disso, os problemas de agendamento da indústria têm um grande número de variantes para representar as diferenças nos processos de produção na prática. As variantes podem incluir diferentes funções objetivo, restrições de sequência de tarefas e configurações de máquina. No entanto, a maioria das heurísticas é explicitamente projetada para uma única ou algumas variantes, gerando um aumento no custo de desenvolvimento quando os métodos precisam ser adaptados para resolver diferentes variantes do problema. Além disso, os métodos geralmente possuem um conjunto de parâmetros que devem ser calibrados para garantir bons resultados. A calibração geralmente é realizada manualmente, exigindo uma quantidade significativa de tempo e trabalho manual e sendo vulnerável a erro humano. A automação dos processos de projeto e calibração tem sido uma técnica útil que pode reduzir a carga de trabalho dos projetistas e aumentar a robustez em comparação com uma abordagem manual.

Neste trabalho abordamos problemas de agendamento em *flow shop* permutacional e não-permutacional. Nestes problemas cada tarefa é composta por uma sequência de

operações e cada operação deve ser processada sem interrupção por uma máquina específica por um determinado tempo. Todas as tarefas passam pela mesma sequência de máquinas, uma de cada vez. Nós abordamos a minimização do tempo máximo de conclusão, ou seja, o tempo de conclusão do último trabalho do cronograma e o tempo total de conclusão, ou seja, a soma dos tempos de conclusão de todas as tarefas. Ainda, neste trabalho vamos além do objetivo típico de propor um novo algoritmo. Em vez disso, construímos um *solver* que implementa componentes algorítmicos individuais e permite combiná-los para gerar métodos de busca local iterada ou algoritmos gulosos iterados. Nós utilizamos uma metodologia automatizada para encontrar combinações eficientes de componentes, gerando assim um conjunto de novos algoritmos, que são comparados com o estado da arte. Nosso objetivo é automatizar o projeto de heurísticas para problemas de agendamento em *flow shop*, reduzindo o trabalho manual necessário e gerando métodos competitivos com o estado da arte.

Nossa metodologia começa pela definição de uma gramática livre de contexto que estabelece como componentes algorítmicos individuais podem ser combinados para formar heurísticas. Esses componentes algorítmicos são, por exemplo, heurísticas construtivas, procedimentos de busca local e critérios de aceitação de soluções. A gramática e seus componentes são implementados em um *solver* parametrizado, de forma que uma heurística pode ser instanciada com um conjunto de valores de parâmetros. Para converter uma gramática em parâmetros, usamos uma abordagem semelhante à de SGE (LOURENÇO; PEREIRA; COSTA, 2016), em que parâmetros categóricos são vinculados aos símbolos não terminais. Mais precisamente, usamos um parâmetro para cada vez que cada não terminal pode ser expandido. O valor de cada parâmetro determina qual opção deve ser selecionada para a expansão. Por fim, o *solver* instancia o algoritmo correspondente e o usa para resolver uma dada instância do problema. O espaço de busca definido pela gramática é explorado com a ferramenta de configuração de parâmetros irace em busca de boas combinações de componentes.

No Capítulo 3 abordamos problemas de agendamento em *flow shops* permutacionais com minimização dos tempos de conclusão máximo e total. Nós usamos nossa metodologia para projetar dez algoritmos de maneira automatizada para cada função objetivo. Os métodos obtidos foram comparados aos métodos do estado da arte em dois conhecidos conjuntos de instâncias da literatura. Os resultados dos experimentos computacionais mostram que, embora diferentes do estado da arte, os algoritmos para minimização do tempo total de conclusão obtêm resultados equivalentes. Os algoritmos para

minimização do tempo máximo de conclusão são em geral similares ao estado da arte, mas algumas diferenças nos componentes selecionados e nos valores dos parâmetros desses componentes levam a resultados levemente melhores.

No Capítulo 4 abordamos problemas de agendamento em *flow shops* não permutacionais com minimização dos tempos de conclusão máximo e total. Nós usamos nossa metodologia para projetar dez algoritmos de maneira automatizada para cada função objetivo. Os métodos obtidos foram comparados aos métodos do estado da arte em dois conhecidos conjuntos de instâncias da literatura. Os resultados dos experimentos computacionais mostram que os algoritmos projetados de maneira automatizada geram uma substancial melhora em relação ao estado da arte quando minimizamos o tempo total de conclusão. Em contraste, quando minimizamos o tempo máximo de conclusão, os resultados obtidos são inferiores aos do método do estado da arte. Isso se deve a uma diferente representação para soluções não permutacionais que não é implementada no *solver*.

As principais contribuições deste trabalho são um *solver* que utiliza componentes algorítmicos da literatura implementados de forma eficiente, e um conjunto de algoritmos para cada variante do problema e função objetivo. Muitos componentes dependem de uma implementação eficiente de procedimentos de aceleração não triviais. Portanto, fornecer tal implementação é de grande interesse. Também estudamos problemas de agendamento em *flow shops* não permutacionais e produzimos mais evidências em favor de sua relevância. O código-fonte do *solver* está disponível publicamente em <https://github.com/arturfb/FSSolver>.