Duarte José Marques Alemão

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

Dynamic Scheduling for Maintenance Tasks Allocation supported by Genetic Algorithms

Dissertação para obtenção do Grau de Mestre em Engenharia Eletrotécnica e de Computadores

Orientador: José Barata de Oliveira, Professor Doutor, FCT-UNL
Coorientador: Mafalda Parreira-Rocha, MSc, UNINOVA

Júri:

Presidente: Prof. Doutor Fernando José Almeida Vieira do Coito
Arguente(s): Prof. Doutor João Paulo Branquinho Pimentão
Vogal(ais): Prof. Doutor José António Barata de Oliveira

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro 2017

Dynamic Scheduling for Maintenance Tasks Allocation supported by Genetic Algorithms

To Catarina, my family and all those who helped me to get here

Acknowledgments

In this section I want to express my gratefulness to all those who supported me and contributed for the conclusion of this thesis.

First and foremost, I would like to thank my supervisor, PhD Professor José Barata, for giving me the opportunity of developing this thesis for which I became interested since the first moment, as well as the opportunity to participate in the PERFoRM H2020 project.

To my co-supervisor, MSc Mafalda Parreira-Rocha. Her support, dedication and patience were essential. It was an honor to learn so much with her and I can only be very grateful. Most of the success of this work was due to her.

I can't forget André Rocha for the great help and support during the last year, since I start working on this thesis. His teachings were very valuable.

I would also like to thank Pedro Monteiro and Ricardo Peres for helping me integrating within the workgroup and for their availability to help every time I have needed.

Furthermore, I want to thank my friends Afonso, Gonçalo and Tiago for being present for so long and always support me.

I also want to express my gratitude to all my family, mainly my grandparents and my uncles Celízia and Marianela.

A huge thank you to Catarina. For always being present, in my best and worst moments. For always encourage and try to motivate me. For all the support. For everything you mean to me. Thank you.

To my amazing brother, who deserves all the best. I cannot describe my gratitude for always supporting and helping me. Thank you.

To my mom. For everything what she means to me. It was not possible without her support and dedication through these years.

Finally, to my dad who always supported, encouraged and inspired me. Thanks for let me get where I am today. I will always be grateful to him.

To all of you,

Thank You

Resumo

Desde que surgiram as primeiras fábricas, o homem sempre tentou maximizar a sua produção de forma a aumentar os seus lucros. Contudo, as exigências do mercado têm-se alterado e, hoje em dia, não é tão fácil obter o máximo rendimento dos sistemas de produção. As linhas de produção estão a tornar-se cada vez mais flexíveis e dinâmicas e a quantidade de informação disponível e processada nas fábricas tem aumentado a um grande ritmo. Isto leva a um cenário onde os erros no escalonamento da produção ocorrem com mais frequência.

Diferentes abordagens têm sido usadas ao longo do tempo para fazer o planeamento e alocação das tarefas na linha de produção da fábrica. Porém, muitas delas não consideram alguns fatores que estão presentes em ambientes reais, como é o caso de as máquinas de produção não estarem sempre disponíveis e necessitarem de intervenções regulares ou não. Isto leva a que os sistemas se tornem mais complexos e difíceis de planear. Deste modo, devem ser usadas abordagens mais dinâmicas que consigam explorar mais eficientemente os grandes espaços de busca que este tipo de problemas proporciona.

Neste trabalho é proposta uma arquitetura para obter um escalonador que inclui tarefas de produção e manutenção, que são frequentemente ignoradas na literatura. São, também, tidos em consideração os turnos de manutenção disponíveis para alocar essas tarefas.

A arquitetura proposta foi implementada recorrendo a algoritmos genéticos, que já demonstraram que conseguem resolver problemas de análise combinatória, como é o caso do problema de Job-Shop Scheduling. A arquitetura considera a ordem de precedência das tarefas do mesmo produto, assim como os turnos de manutenção disponíveis na fábrica.

A arquitetura foi testada num ambiente simulado de modo a analisar o comportamento do algoritmo. Contudo, foram usados conjuntos de dados de tarefas de produção e estações de trabalho reais.

Palavras-Chave: Job-Shop Scheduling, Alocação de Tarefas, Algoritmos Genéticos, Sistemas de Manufatura

Abstract

Since the first factories were created, man has always tried to maximize its production and, consequently, his profits. However, the market demands have changed and nowadays is not so easy to get the maximum yield of it. The production lines are becoming more flexible and dynamic and the amount of information going through the factory is growing more and more. This leads to a scenario where errors in the production scheduling may occur often.

Several approaches have been used over the time to plan and schedule the shop-floor's production. However, some of them do not consider some factors present in real environments, such as the fact that the machines are not available all the time and need maintenance sometimes. This increases the complexity of the system and makes it harder to allocate the tasks competently. So, more dynamic approaches should be used to explore the large search spaces more efficiently.

In this work is proposed an architecture and respective implementation to get a schedule including both production and maintenance tasks, which are often ignored on the related works. It considers the maintenance shifts available.

The proposed architecture was implemented using genetic algorithms, which already proved to be good solving combinatorial problems such as the Job-Shop Scheduling problem. The architecture considers the precedence order between the tasks of a same product and the maintenance shifts available on the factory.

The architecture was tested on a simulated environment to check the algorithm behavior. However, it was used a real data set of production tasks and working stations.

Keywords: Job-Shop Scheduling, Task Allocation, Genetic Algorithms, Manufacturing Systems

Table of Contents

Table of Figures

Table of Tables

Acronyms

AMS	Agile Manufacturing Systems
ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
AI	Artificial Intelligence
CM	Corrective Maintenance
DJSSP	Dynamic Job-Shop Scheduling Problem
DRCJSP	Dual Resource Constrained Job-Shop Scheduling Problem
EA	Evolutionary Algorithm
FJSSP	Flexible Job-Shop Scheduling Problem
FMS	Flexible Manufacturing System
GA	Genetic Algorithm
GUI	Graphical User Interface
JSSP	Job-Shop Scheduling Problem
LS	Local Search
NP	Non-deterministic Polynomial-time
PML	PERFoRM Markup Language
PERFoRM	Production harmonizEd Reconfiguration of Flexible Robots and Machinery
PM	Preventive Maintenance
PSO	Particle Swarm Optimization
SA	Simulated Annealing
SI	Swarm Intelligence
TBM	Time-Based Maintenance
TS	Tabu Search

1

Introduction

1.1. Scope & Motivation

One of the greatest challenges for man in the most distinct areas has always been to maximize his work efficiently and effectively. To do so, it is important to plan a well-structured schedule with detailed description of the tasks to execute, mainly when and where they should be performed. This is applicable to areas like transportation services, staff distribution and production systems, among others.

As the demand for more diversified and customized products grew a lot, the factories needed to change their production systems, since the old ones were not capable to deal with this huge demand. Thus, in medium or large shops it is not easy to schedule this large number of products. This problem is further complicated when it is not possible to know exactly how much time a product will take to get finished by some machine. Even more, when new products arrive, to the factory during a day of work or if a machine unexpectedly breaks down and it becomes necessary to route products to another one.

However, the stations present in the shop-floor are not available all the time. They are frequently subject to maintenance operations. Sometimes, those maintenance interventions are previously defined and can be included in the task allocation process to get more realistic schedules.

This work presents a suggestion to solve this problem, which involves scheduling and task allocation in a job-shop. It deals with production and maintenance tasks allocation simultaneously, in order to minimize stations' total execution time.

In the next section, a research question and a possible hypothesis to solve this problem are presented.

1.2. Research Question and Hypothesis

Considering the problem addressed in section 1.1, it should be cleared that the maintenance tasks are an important element that need to be considered in the shop-floor's scheduling process. So, they should be considered in the task allocation process. Therefore, it comes the next research question:

How is it possible to develop a dynamic job-shop scheduling integrated solution, including both production and maintenance tasks?

By developing a tool based on artificial intelligence, namely, genetic algorithms, the maintenance and production tasks can be merged into only one schedule in order to optimize the production allocation in the shop-floor. Genetic algorithms provide a good solution in a reasonable amount of time which is essential to plan the production in the shop-floor.

A summary of the work carried out in this thesis is presented in the next section.

1.3. Accomplished Work

In this study case is proposed a job-shop scheduling architecture capable to generate a schedule that efficiently incorporates maintenance and production tasks. A task is a process where some working station performs an operation in a product. To do so, it is considered a part of a shop floor with a non-defined number of non-identical stations, yet the rest of the factory is not known in this work.

Those tasks should be allocated only to a station capable to execute the required operation. At the end, the start and end times of each task will be set, being thus possible to know which operations should be performed at any time.

The communication between the algorithm and the shop-floor is done through a middleware that get and send information to the factory. This middleware is a layer which guarantees the interaction between the hardware devices and the software applications present in the system.

Since this is a dynamic environment, new products could arrive to the factory and new maintenance tasks could be necessary to perform. Which means that the algorithm needs to be fast enough to generate new schedules when required.

The proposed architecture gets a set of tasks to perform as well as the stations available in the shop-floor from the middleware. Then, with all the needed information acquired, the algorithm is executed. When the algorithm stops, a legible schedule is sent back to the middleware.

A Genetic Algorithm (GA) was implemented to solve this problem. GAs already proved to solve scheduling problems efficiently, even in large scale (Balin, 2011; Werner, 2011). Although they do not always get an optimal solution, a near optimal solution is reached quickly most of the times, due to their search capability based on natural evolution.

After design the architecture, this one was implemented using the Java language and the Jenetics library, which allows to implement a GA using the concept of stream for executing the evolution steps. This implementation was validated in a simulated environment.

With this work, the author expects to contribute with an efficient algorithm to solve a flexible and dynamic job-shop scheduling problem, based on algorithms found in literature.

The major contributions of this work are presented next.

1.4. Main Contributions

The accomplished work proposes a solution which allows the system to plan and schedule the production and maintenance tasks on the shop-floor.

This job-shop scheduling problem deals with production tasks as well as maintenance tasks which need to be scheduled in specific time slots according to the factory requirements. The presented architecture does not ignore the maintenance operations which are a constant in manufacturing environments. Thus, it portrays a more reliable scenario from the real factories, providing more realistic schedules.

Furthermore, the implementation of the proposed architecture was integrated in the Production harmonizEd Reconfiguration of Flexible Robots and Machinery (PERFoRM) H2020 project. In the PERFoRM project the tools are capable to communicate with each other through a central

middleware, where they can exchange information in real-time. So, the scheduling tool can provide the obtained schedules for the other tools.

Finally, in the next section, is presented and defined the structure of this thesis.

1.5. Dissertation Structure

In the chapter 1 is introduced an overview of the presented document. Mainly, it is identified the problem to solve and how it will be solved. It is also presented the important contributions provided by this work.

Chapter 2 is dedicated to the state of the art. It is an introduction to the problem addressed in this thesis, how it works and how the author wants to solve it. It includes some examples in the literature about similar problems and several ways to solve them. This is a walk-through by scheduling, JSSP, GAs and how they are involved.

In chapter 3 it is presented the scheduling architecture, with the intention to clarify how this scheduling problem will be approached and solved.

The implementation of the algorithm used in this thesis will be presented in chapter 4. Here, the architecture will be put into practice. There will be a deep explanation about how the algorithm works and how it is integrated with the middleware.

Chapter 5 will be dedicated to the tests of the algorithm, presentation and discussion of the obtained results.

In chapter 6 are presented the conclusions got after analyzing the obtained results as well as some suggestions about the further work needed to improve this work.

2

State of the Art

Industrial systems have, drastically, evolved over the past decades. First, they were constructed to carry out mass production, where the products' variety was minimal and the systems were relatively simple. But, the market changes imposed an adaptation of the manufacturing systems in order to produce a wide variety of products and their variants to satisfy a large number of consumers and their unique demands (Rocha et al., 2014).

Due to those changes, factories now need to quickly adapt their systems to give a fast response to the market demand. So, these changes in the production industry led to an increased demand for more agile systems as well as it brings more complexity to those systems themselves. However, this agility entails a new problem: now the products do not all follow a single path in the shop-floor but they can be allocated to different stations, which implies developing a plan to achieve the best solution possible to improve the production process.

That plan consists in allocate a set of operations, to perform in some products, to the stations which are capable to accomplish that operation. This process is known as job-shop scheduling problem (JSSP) (Buzatu & Bancila, 2008). However, it is important to consider that the stations

are not available all the time. Sometimes they need to be repaired or maintained. So, the maintenance operations should be considered when a production process is scheduled.

2.1. Agile Manufacturing Systems

Once technology has evolved, and so has manufacturing systems, factories have started to adapt their shop-floors in order to become more agile, reconfigurable and flexible to face new market requirements that are constantly changing. Agile Manufacturing Systems (AMSs) were developed to give firms those flexibility and agility (Rocha et al., 2014).

There is no consensual definition for agile manufacturing. Nevertheless, many authors define it as the ability of a factory to keep up and respond quickly and efficiently to the unpredictable market changes, both in terms of product models and product lines (Chalfoun, Kouiss, Huyet, Bouton, & Ray, 2013; Dionisio Rocha, Peres, & Barata, 2015; Gunasekaran, 1999; Kretschmer, Pfouga, Rulhoff, & Stjepandić, 2017). For (Yusuf, Sarhadi, & Gunasekaran, 1999), “the main driving force behind agility is change”. Thus, an agile factory needs to change and adapt very quickly to the necessities.

In AMSs there a large range of different stations used to perform specific tasks, so it is important to keep those stations working and avoid unpredictable breakdowns. Therefore, it is necessary to perform some regular maintenance operations on the shop-floor to keep the work flowing well. That is why the maintenance tasks need to be considered in scheduling problems. However, there are different maintenance processes that can be used, mainly to prevent breakdowns or to repair them.

2.1.1. Flexible Manufacturing Systems

A Flexible Manufacturing System (FMS) is defined as the possibility of the system to share tools between different machines, so the ability to produce multiple types of products is improved (Lei, Xing, Han, & Gao, 2017). In this way, the number of combinations performed between tools and machines is increased and, also, increases the number of performed services provided by the system. All the machines are linked through a material handling system that moves parts to the next machine. Both are controlled by a central system (Elkins, Huang, & Alden, 2004; ElMaraghy, 2005; Lei et al., 2017).

2.1.2. Reconfigurable Manufacturing Systems

The Reconfigurable Manufacturing Systems (RMS) paradigm focus on the machine-tool reconfiguration (Galan, Racero, Eguia, & Garcia, 2007). In the literature, a system is classified as reconfigurable if its physical structure can be easily changed and if it was designed for a part family (Y. Koren et al., 1999). Besides the similarities with the FMS paradigm, RMS have the capacity to facilitate the systems' reconfiguration by adding, removing or updating new components to the system (Galan et al., 2007; Yoram Koren & Shpitalni, 2010).

2.2. The Process of Maintenance to Agile Manufacturing Systems

The maintenance process is essential in the AMSs, where there are a large variety of stations with specific requisites which need to keep working the maximum time possible to maximize the production. So, sometimes they need to be subject to maintenance operations.

There are many different types of maintenance such as preventive maintenance, also known as Time-Based Maintenance (TBM), Predictive Maintenance (PdM), Corrective Maintenance (CM), among others, which work in different ways (Aissani, Beldjilali, & Trentesaux, 2009; Raza & Ulansky, 2017).

CM is only applicable after some failure occur, for instance a breakdown, trying to recover a resource to a functional state (Aissani et al., 2009; Froger, Gendreau, Mendoza, Pinson, & Rousseau, 2015). This type of maintenance implies that occurs repair and/or replacement of resources (Y. Chen, Cowling, Polack, Remde, & Mourdjis, 2017). To reduce the risk of unexpected breakdowns and resource's downtime due to failures, another maintenance types should be applied. (Kenné & Gharbi, 2004) proposed a corrective maintenance strategy to improve the machines' availability. The production and the machine repair rates are decisive to execute the algorithm successfully, which was model using a Markov process.

TBM is done to prevent, in advance, potential faults resulting in malfunctions (Aissani et al., 2009; Ruiz, Carlos García-Díaz, & Maroto, 2007; Yoo & Lee, 2016). It is taken while the system is still working and consists in perform the operations in resources before the failure happens and minimize the failure's probability. The main advantage is that the system is in good conditions the most of the time and maintains the resource's performance over the time (Aissani et al., 2009; Ruiz et al., 2007). This type of maintenance is normally done at predefined time intervals, based on experience (Y. Chen et al., 2017; Froger et al., 2015), but it can also be done by finding the

optimal time to apply the maintenance (Ruiz et al., 2007). TBM reduces failures on the machines, which leads to an increase of the machines' availability and a reduction in costs, improves the production planning and management and improves safety (Ruiz et al., 2007). Include TBM into the production schedule has attracted the researcher's attention in last years although it continues being a quite unexplored subject (Ángel-Bello, Álvarez, Pacheco, & Martínez, 2011). (Ruiz et al., 2007) proposed a preventive maintenance solution for a flow shop problem with different policies. They considered preventive maintenance at predefined intervals aiming to maximize machines' availability or keeping a minimum level of reliability after the production period. Evolutionary algorithms were used and they found that those algorithms provide good solutions to their study case. (Gao, Gen, & Sun, 2006) studied a flexible job-shop scheduling problem where each machine was subject to an arbitrary number of preventive tasks and observed that after certain planning horizons one or more maintenance tasks may have to be scheduled, although it is possible to have no maintenance processes in some time intervals.

PdM, on the other hand, can be applied to prevent future failures "if there is a deteriorating physical parameter like vibration, pressure, voltage, or current that can be measured" (Raza & Ulansky, 2017), instead of doing it in predefined time intervals. It not only prevents from failures but also improve the production performance (Selcuk, 2016). PdM is performed based on the Remaining Maintenance Life metric, which was proposed to set a security threshold before the machines reach the remaining useful life. That remaining time is based on a prognostic. It infers the current state and predicts the future progression to estimate the time before a failure occur (Ladj, Varnier, & Tayeb, 2016). (Aissani et al., 2009) proposed multi-agent model to solve dynamic scheduling of maintenance tasks in a petroleum production system. Their solution can generate schedules for both predictive and corrective maintenance and improve the quality of the system. (Ladj et al., 2016) proposed a genetic algorithm with the objective of minimize the total interventions cost on a single machine subjected to predictive maintenance.

2.3. Scheduling of Agile Manufacturing Systems

Scheduling is a process of time optimization, where a set of jobs (products with one or more operations that will be processed by the stations) is assigned, reasonably, to a group of resources (Chung & Kim, 2016; Jose & Pratihar, 2015). It is defined in which sequence they are executed, by each resource, during each day of work in a predefined time horizon, trying to avoid conflicts and to optimize the objectives (Cardon, Galinho, & Vacher, 2000). Scheduling determines what is going to be made, when, where and with what resources (Cardon et al., 2000; W. Zhang, Gen, & Jo, 2014).

(Csaji & Monostori, 2006) have classified task allocation techniques in three categories: predictive, proactive and reactive. Predictive solutions assume a deterministic situation, which means that all the information is known in advance. If there are uncertainties and some data will be available only during the scheduling execution, like tasks' duration, this is called a proactive solution. This solution assigns the jobs to resources with a certain order, but doesn't determine starting times of each operation. Finally, when a solution makes decisions while more information becomes available, this is a scenario of reactive task allocation.

Most of the scheduling problems are Non-deterministic Polynomial-time Hard (NP-Hard) problems (Lenstra & Rinnooy Kan, 1978; Ullman, 1975), which are quite difficult to reach an optimal solution with traditional optimization techniques, because a solution is searched in a large search space (Petrović, Vuković, Mitić, & Miljković, 2016). For a deeper explanation of NP problems the reader is invited to see (Ullman, 1975). Although Mathematical optimization methods can achieve optimal solutions for relatively small problems, in a larger scale they are very limited (Balin, 2011).

Even though most researchers assume some constraints like all resources are always available (W. Zhang et al., 2014) or that the processing time of a job is known in advance and remains constant during the whole process, in real situations this is not always true. For example, jobs' duration is not always constant but increases over time, depending on the sequence of jobs or their starting times. Such case is generally known as the *deterioration of resources* in scheduling problems (Chung & Kim, 2016). On the other hand, resources are not always available during the scheduling process. In real environments, sometimes, they need maintenance or can broke and stay unavailable for some periods of time (Gao et al., 2006; Yoo & Lee, 2016). Maintenance scheduling is used to try to minimize the effect of machine's breakdowns and to maximize the capacity's availability at a minimum cost. Also, improve maintenance tasks can provide financial gains and safety enhancements (Aissani et al., 2009). For those reasons, and once maintenance charges cover a large percentage of the total operation costs (Ángel-Bello et al., 2011), maintenance tasks should be considered in the production plan to obtain a more representative schedule. However, most of the scheduling problems considered in literature are deterministic, this means that all data is known and given in advance and remains constant during the whole process, like the processing times of a job (Chung & Kim, 2016; Werner, 2011).

Scheduling has been largely applied to many different areas like energy consumption (Lu, Gao, Li, Pan, & Wang, 2017; J. Wang, Tang, Xue, & Yang, 2016), transportation , staff distribution (Ernst, Jiang, Krishnamoorthy, & Sier, 2004) and production (Balin, 2011; X. Li & Gao, 2016),

to help the industries to plan their activities. Specific algorithms and mathematical models should be developed for scheduling solutions in different areas of application (Ernst et al., 2004).

In agile manufacturing environments, products can have several different feasible process plans and most of the times is very hard to find a good one for all the products. Production scheduling is a very important decision making in a factory and it can be a difficult problem depending on the number of calculations necessary to obtain a schedule that gives an optimal solution for the given objectives (Balin, 2011). In this way, it is necessary to have an efficient schedule generator accordingly to the type of the factories and their constraints.

There are several job scheduling environments depending on the stations' layout and the flow of the products. A flow shop (Lu et al., 2017) is a shop in which the machines are disposed in series, each job has exactly *n* operations and the route travelled by each one through the machines is the same to every job. They all start processes in the same initial machine and conclude in the same final machine (Balin, 2011; Werner, 2011). In a job-shop (Buzatu & Bancila, 2008) scenario, products can be processed on machines in any order. A specific route is given to each job and they can start the process in different machines, accordingly to operations that need to be performed in each one (Werner, 2011). However, the operation sequence of each task should be respected. Jobs can visit the same machine several times and there is only one machine capable to perform a given operation. In fact, there are often several replicas of the same machine (parallel machine), so, there are numerous machines that can process the jobs. In the end, the schedule provides the order in which operations are to be done (Balin, 2011). When products do not have precedence constrains, the processing routes have to be chosen, this is known as open shop. It is assumed that each job has to be processed on any machine and it can visit the same machine several times (Strusevich, 1999; Werner, 2011).

2.4. Job-Shop Scheduling Problem

The JSSP is one of the most famous scheduling problems and is a part of production scheduling. The traditional JSSP can be described as a set of *m* stations $\{M_1, \ldots, M_m\}$ that should process a set of *n* different jobs $\{J_1, \ldots, J_n\}$. Each job has a set of operations to be processed in a given order in the different stations. An operation O_{JM} of a job *J* requires the use of the station *M* for an uninterrupted duration, dominated processing time, and each station can only operate a job at a time. In Figure 2-1 is represented a schedule for a JSSP with three jobs, each one with three different operations, and three different stations.

A full list of restrictions of the basic JSSP is presented in (Mattfeld, 1996):

- No two operations of one job may be processed simultaneously.
- No preemption (i.e. process interruption) of operations is allowed.
- No job is processed twice on the same machine.
- Jobs may be started at any time.
- Jobs may be finished at any time.
- Jobs must wait for the next machine to be available.
- No machine may process more than one operation at a time.
- Machine setup times are negligible.
- There is only one of each type of machine.
- Machines may be idle within the schedule period.
- Machines are available at any time.
- The technological constrains are known in advance and are immutable.

A schedule is a set of completion time of each operation that satisfies the necessary constraints, mainly the precedence between operations. The aim of JSSP is to find a job sequence on each station considering a given objective function. Frequently, the objective function used in JSSP is to minimize the maximum completion time of all jobs (from here referred as makespan) (Karimi, Ardalan, Naderi, & Mohammadi, 2016; Noor, Ikramullah Lali, & Saqib Nawaz, 2015; Yamada & Nakano, 1997).

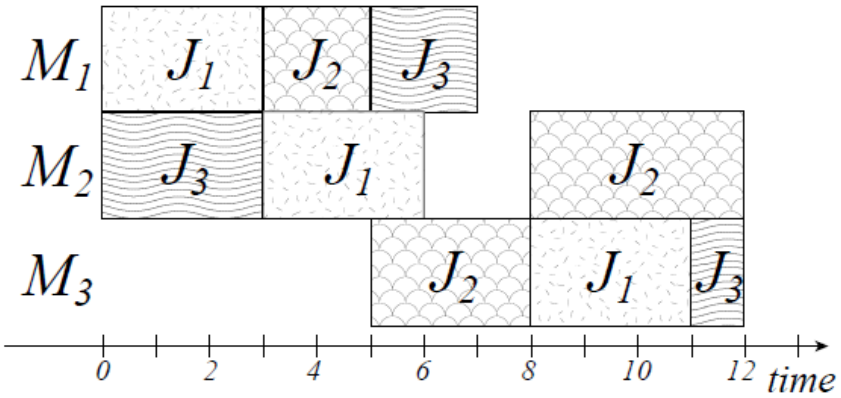


Figure 2-1 - Gantt Chart representation for a 3x3 problem. Image taken from (Yamada & Nakano, 1997)

The JSSP is one of the most important combinatorial optimization problems. It is not only NP-Hard but is one of the worst members in the class. This means that it is maybe impossible to find an optimal solution in the whole search space (Kundakcı & Kulak, 2016; J. Li, Huang, & Niu, 2016; Yamada & Nakano, 1997). Such complexity is due to larger product variety and constraints and because there is a factorial number of solutions that can result in a feasible schedule. This problem is even trickier when operations can be processed by more than one

machine or if machines can execute more than one type of operation (Balin, 2011; Y. Liu, Dong, Lohse, & Petrovic, 2014).

Once technology has evolved over past decades, factories have started to use flexible machines, able to perform more than one type of operation, this is known as Flexible Job-Shop Scheduling Problem (FJSSP). This is very important in actual factory systems. However, there is an additional problem, since it is not only needed to get the sequence of operations on machines but it is also needed to assign operations to those machines that are able to change their operation mode (Karimi et al., 2016; X. Li & Gao, 2016).

Another case, very common in factories nowadays, is when both stations and workers are necessary to execute the jobs and those cannot be processed if stations, workers or both are not available. This scenario is known as Dual Resource Constrained Job-Shop Scheduling Problem (DRCJSP). This situation presents additional challenges that must be considered on scheduling, such as the interaction between workers and resources (J. Li et al., 2016; Xu, Xu, & Xie, 2011).

The basic JSSP and most scheduling problems considered are deterministic (static) ones, which means that all information, like processing time of each operation and some constraints, is fixed and known in advance and there is no machine breakdown (Bierwirth & Mattfeld, 1999; Mattfeld, 1996; Werner, 2011). In (Y. Liu, Dong, Lohse, Petrovic, & Gindy, 2014) a deterministic job-shop model is used, where the number of jobs is a fixed value and all of them are available at the beginning of the process.

However, in real world, jobs arrive unexpectedly and duration times are not known in advance and can change during the process, these are non-deterministic problems (Bierwirth & Mattfeld, 1999; Kundakcı & Kulak, 2016).

To solve this kind of scheduling problems, traditional approaches are not enough. It is necessary to have more dynamic approaches where the changes are performed during the process. It is needed to consider a lot of constraints, like happens in real cases, that sometimes are not considered in classic JSSP approaches, for instance the arrival of new jobs constantly during the time horizon defined in the schedule, station breakdowns, problems with maintenance teams, changes of execution times, insertion or removal of stations among others. Such approach is called Dynamic Job-Shop Scheduling Problem (DJSSP) (Kundakcı & Kulak, 2016; Sharma & Jain, 2015; Xiong, Fan, Jiang, & Li, 2017). In dynamic scheduling changes are made while the system is running and are included in the current schedule (Scrimieri, Antzoulatos, Castro, & Ratchev, 2015).

It is possible to find a lot of approaches in literature that can be applied in order to solve JSSP and its extensions. They are mainly divided into exact method and approximation method (heuristic and meta-heuristic techniques). Exact methods contain mathematical programming models, while the second one is based in dispatching rules and Artificial Intelligence (AI), which includes Swarm Intelligence (SI), Local Search (LS) and Evolutionary Algorithm (EA) (X. Li & Gao, 2016).

In (Choi & Choi, 2002) a mixed integer program model is presented, along with a local search scheme, to solve a JSSP considering alternative operation sequences and sequence-dependent setups. (Lee, Wang, & Lin, 2016) developed a branch-and-bound algorithm to solve a parallel-machine scheduling problem optimally with fewer jobs. However, exact algorithms are not effective for solving large scale FJSP instances (X. Li & Gao, 2016; Pezzella, Morganti, & Ciaschetti, 2008).

(Tay & Ho, 2008) solved a multi-objective flexible job-shop problem using dispatching rules discovered through genetic programming. (Baykasoğlu & Özbakir, 2010) used dispatching rules to evaluate the effect on the scheduling performance of job-shops with different levels of flexibility. Dispatching rules are simple and can solve large scale problems, however the quality of the results is not very good (X. Li & Gao, 2016).

(Sobeyko & Mönch, 2016) deal with flexible job-shops with parallel machines, where the objective function is given by total weighted tardiness. A shifting bottleneck heuristic is proposed, hybridized with a local search and a variable neighborhood search approach.

SI is a meta-heuristic method that include, among others, Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO) and Artificial Bee Colony (ABC) (Karimi et al., 2016; X. Li & Gao, 2016). (Singh & Mahapatra, 2015) developed a hybrid Particle Swarm Optimization (PSO) algorithm where mutation, a used operator in GAs, has been introduced, to solve a FJSSP. (Zhao, Tang, Wang, & Jonrinaldi, 2014) also have used a based PSO for solving a multi-objective JSSP. An ABC approach is presented to solve a JSSP in (Madureira, Pereira, & Abraham, 2013). In (Saidi-Mehrabad, Dehnavi-Arani, Evazabadian, & Mahmoodian, 2015) an ACO algorithm is proposed to minimize the makespan of a JSSP. (J. Q. Li & Pan, 2012) proposed a chemical-reaction optimization for solving the FJSSP with three minimization objectives. Each solution is represented by a molecule and a LS method was embedded in algorithm to perform exploitation process. SI techniques are global search methods and should be improved with local search algorithm to get better results. SI methods have proved to be efficient in solving scheduling problems (Madureira et al., 2013).

EA is an effective type of meta-heuristic method based on the principals of natural evolution. EA includes GA, genetic programming, evolution strategies and evolution programming (Werner, 2011). (Kundakcı & Kulak, 2016) developed a GA for minimizing makespan in a DJSSP where a new heuristic and well-known dispatching rules where integrated in the algorithm. GAs are one of the most preferred algorithms to solve hard combinatorial optimization problems, because of its adaptability, near optimization and they are not difficult to implement (Balin, 2011). EAs are effective for the scheduling problems because of their powerful global search ability. Even though, they do not have good local search ability, EAs can be combined with other local search algorithms to improve their performance (X. Li & Gao, 2016).

LS is often used to improve the performance of other algorithms, searching for a locally optimal solution (Gonçalves, De Magalhães Mendes, & Resende, 2005). It includes Tabu Search (TS), Simulated Annealing (SA) and so on (X. Li & Gao, 2016). LS is a common technique of hybridization to improve the performance of another algorithms like PSO, ACO or GA algorithms. (Asadzadeh, 2015) presented a local search GA to solve a JSSP with intelligent agents and showed that the algorithm improved the efficiency. A TS method to solve a dynamic shop scheduling problem was developed successfully by (S. Q. Liu, Ong, & Ng, 2005). Once LS are based in the neighborhood, they lack in global search capacity, so they should be combined with other global search algorithms (X. Li & Gao, 2016).

Since GAs proved to be very efficient to solve JSSP, due to its capability of adaptation (Kundakcı & Kulak, 2016) and the good ability to search for a global optimum (T. Wang, Liu, Chen, Xu, & Dai, 2014), and they also proved to be good solving preventive maintenance problems (Ladj et al., 2016) and once they are easy to implement (Balin, 2011), a GA approach will be used to solve the DJSSP in this study.

2.5. Genetic Algorithms

First proposed by Holland (Holland, 1975) in the 1970s, GAs are stochastic searching adaptive approaches to solve optimization problems based on the natural selection mechanisms where only the best individuals can survive (Chung & Kim, 2016; Gonçalves et al., 2005). They work by evolving a population of chromosomes of possible solutions using genetic operators, such as selection (survival of the fittest), crossover and mutation, that are inspired by the natural evolution, first settled by Darwin in *The Origin of Species* (Balin, 2011; Gonçalves et al., 2005; J. Li et al., 2016).

Once GAs are heuristic methods, they have powerful search ability. They are known to be good solving optimization problems since they don’t need to evaluate all the search space to extract a

good solution (Joo & Kim, 2015; T. Wang et al., 2014). In each generation, the fitness of each individual is evaluated. GAs instead of focus in a specific solution and try to improve it, they use a population where all individuals are evolving in different directions. They not always can find an optimal solution but most of the times they can find a near-optimal solution and can easily handle the occurrence of new events (Joo & Kim, 2015; J. Li et al., 2016). Each individual has its own fitness value, that is the value given by an objective function to calculate how good the individual is in the population. In each generation, the fitness of each individual is evaluated and the best individuals are randomly selected to perform crossover and create a new generation (Jose & Pratihar, 2015). The crossover and mutation's probabilities affect significantly how good the solution is and the convergence speed in GA. Usually, values of crossover probability between 0.5 and 1 and mutation probability between 0.001 and 0.05, are used in GA problems (T. Wang et al., 2014). In dynamic environments, convergence becomes a big problem. Yang (Yang, 2008) affirms that this is the main reason why traditional GAs do not perform well in dynamic environments, once convergence deprives the population of diversity. So, when a change in the system occurs, it is hard for GA to adapt to the new environment.

GAs are global search methods using to perform exploration, so they may not be able to find a locally optimal solution. But they can be hybridized with another algorithm, like TS algorithm that has good local search ability and can perform exploitation, to have a better performance (Gonçalves et al., 2005; X. Li & Gao, 2016; D. Wang, Xiong, & Fang, 2016).

They can be applied to real world problems, if the solution in the population is properly encoded (represented), by mimicking the process of natural evolution (Gonçalves et al., 2005; Noor et al., 2015).

GAs are also one of the most attractive techniques to the researchers. They have solved successfully a variety of combinatorial and numerical optimization problems, in the past decades, in many different areas (Ting, Su, & Lee, 2010; Werner, 2011). (Y.-H. Zhang, Gong, Gu, Li, & Zhang, 2017) developed a GA in order to solve node placement problems, such as the deployment of radio-frequency identification or wireless sensor networks. (Ordóñez Galán, Sánchez Lasheras, de Cos Juez, & Bernardo Sánchez, 2017) used a method based on GA for completion of missing data in knowledge and skills tests. (Apolinar & Rodríguez, 2017) proposed a microscope vision system based on micro laser line scanning and a GA to retrieve metallic surface. The GA calibrates the microscope vision parameters with precision to avoid errors. (J. Li et al., 2016) developed a meta-heuristic algorithm named Branch Population Genetic Algorithm, based on GAs, to solve a dual resource constrained job-shop scheduling problem. GA was used with

success in a wafer factory (J. C. Chen, Chen, & Liang, 2016) to solve the capacity allocation problem in order to minimize the difference in loading between machines.

GAs have been applied, in the past decades, to many types of job-shops and a lot of considerations were made. (Çakar, Köker, & Demir, 2008) used a GA to minimize the mean tardiness of a schedule with jobs that have precedence constraints, in an identical parallel robots' system. They found that GA was very successful in large-scale problems. (Balin, 2011) proposed a new algorithm in order to adapt GA to non-identical parallel machine scheduling problem, implementing a new crossover operator and a new optimality criterion. The algorithm proposed proved to be good for non-identical parallel machine scheduling problem to minimize the makespan, however (like in the most of the cases) setup times and due dates are not considered. (Costa, Cappadonna, & Fichera, 2016) developed a hybrid meta-heuristic method based on GA to deal with large-scale identical parallel machine environment with periodic maintenance management under the makespan minimization objective. (Jia, Fuh, Nee, & Zhang, 2007) proposed a GA integrated with Gantt chart to get good factory combinations to produce the jobs. A schedule is obtained with several sub-schedules for all the factories available. There is the possibility to get multiple objectives, such as minimizing makespan, job tardiness and manufacturing cost. (Gao et al., 2006) proposed a hybrid GA to solve the FJSSP that includes maintenances tasks, with the purpose of minimize the makespan, the maximal machine workload and the total working time over all machines. The completion time of maintenance tasks is not fixed and need to be determined during the scheduling process. Complementarily, they used a local search method in order to improve the algorithm. (L. Zhang, Gao, & Li, 2013) developed a hybrid GA and TS for DJSSP. They considered random job arrivals and machine breakdowns in a multi-objective problem, namely efficiency and stability of the schedule. The proposed method demonstrated to have good performance solving DJSSP on both sparse and dense job arrival conditions. Kundakcı (Kundakcı & Kulak, 2016) introduced a hybrid GA to minimizing makespan in DJSSP. In their work, they considered arbitrary job arrivals, machine breakdowns and changes in processing times. They demonstrate that hybrid GA techniques are better than conventional GAs and TS algorithms. Although most of the literature concentrates on static FJSSP or DJSSP independently, (Fattahi & Fallahi, 2010) developed a GA to solve dynamic scheduling on a FJSSP. Their algorithm achieved optimal solutions for small problems and near optimal solutions for medium size problems.

Since GAs are inspired on the natural selection, they follow the natural selection techniques to evolve. After establish the chromosome representation and the fitness function, the GA initializes a population and improves it through the selection, crossover, mutation and elitism operators.

2.5.1. Encoding

For JSSP, the situation is a bit different from other scheduling problems. Here, several encoding strategies exist, and it is not clear in advance which is the best. Many authors distinguish between a direct and an indirect representation. In the first case, a solution is directly encoded into the genotype, but in the second case not (this means that an indirect representation encodes the instructions to a schedule builder) (Werner, 2011). Normally, each chromosome represents a solution (schedule) in JSSP. Each gene contains some information about a task, such as the job that it belongs to, the resource that it is assigned to, duration of the operation and so on. Often, the order of the operation in the chromosome represents the operations sequence (Kundakcı & Kulak, 2016).

Werner (Werner, 2011) observed in his study (about GAs applied to shop scheduling problems) that different solution's representations exist for job-shops problems and the best one may not be clear a priori, unlike flow shop problems where a solution is often represented as a permutation.

In this case, each chromosome is composed by a set of genes where each one represents a different task to execute. Each gene holds the information relative to that task, such as the identification, processing time, associated station and product and start and end times.

2.5.2. Selection

Some individuals, designated as parents, of the population are selected randomly to perform crossover between each other and create a new individual, designated by child, which will be present in the next generation (Kundakcı & Kulak, 2016). There are several techniques to choose the parents. Two of the most used selection methods are *roulette wheel selection* and *tournament selection*. For the first approach, to each individual is assigned a probability according to its fitness value, where higher fitness means higher probability to be selected, since it is attributed a larger percentage to be selected, as it is demonstrated in Figure 2-2. In the second one, *n* individuals are chosen from the population and the one with higher fitness (the strongest) is selected. In this case, several tournaments need to be run to get several individuals (X. Li & Gao, 2016; Page, Keane, & Naughton, 2010; Werner, 2011).

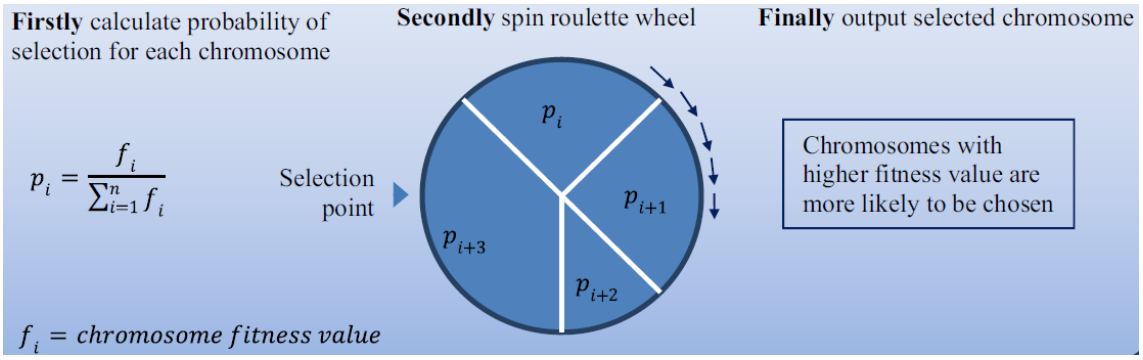


Figure 2-2 - Roulette wheel selection. Image taken from (Casas, Taheri, Ranjan, Wang, & Zomaya, 2016)

2.5.3. Crossover

Crossover is the main factor to create new improved generations since the characteristics of the parents are preserved to the children. In this JSSP, the crossover will be used to create a chromosome with a new operations sequence trying to minimize the total makespan. Crossover is applied over two individuals with a given probability creating one or two new individuals (Noor et al., 2015). Some crossover techniques are single-point crossover, which swap all data beyond that point of parent one with parent two, and two-point crossover that selects two points and everything between that two points is swapped between the parents. Each one produces two children individuals. In task allocation problems, the crossover should preserve the chromosome information and not duplicate it. To do that, some different crossover techniques should be used. Some examples are order crossover (Noor et al., 2015), position based crossover (Kundakcı & Kulak, 2016) or partially-matched crossover (Ting et al., 2010).

In Figure 2-3, partially-matched crossover is performed by cut chromosome in two random points and the segment between them is taken from one parent to another. To repair an infeasible chromosome, the parents should be mapped. From parent 1 to parent 2, 4 is mapped to 7 and 7 is mapped to 2. So, in second gene, the final value is 2 (Werner, 2011).

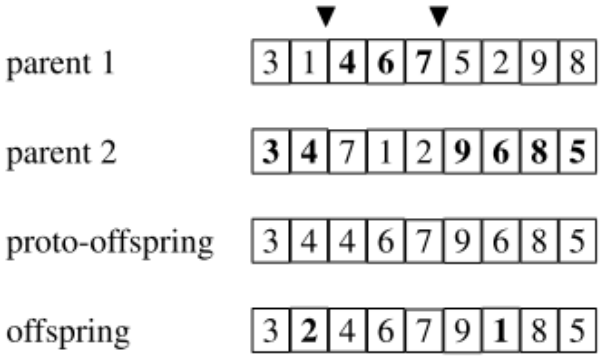


Figure 2-3 - Partially-Matched Crossover. After perform crossover, normally an invalid chromosome is obtained (proto-offspring). To repair chromosome, it is needed to apply an exchange outside the crossing region. At the end a valid chromosome is obtained (offspring). Image taken from (Werner, 2011)

2.5.4. Mutation

After crossover, mutation is applied to a small number of individuals with the purpose to replace genetic material lost during reproduction and crossover (Gonçalves et al., 2005; Jia et al., 2007). A mutation technique could be assign a random gene with a random value or swap two random genes in a chromosome. Mutation is also used to prevent population to get stuck (to converge too fast) due to super chromosomes (Jia et al., 2007; Kundakcı & Kulak, 2016).

In the case of JSSP the mutation operator needs to be a swap mutator or another one that changes the genes' positions and does not duplicate them, since the chromosome is a sequence and each gene represents a task (Noor et al., 2015; Page et al., 2010).

In Figure 2-4 is represented a swap mutation where two genes of the chromosome are randomly selected and their position are exchanged with one another.

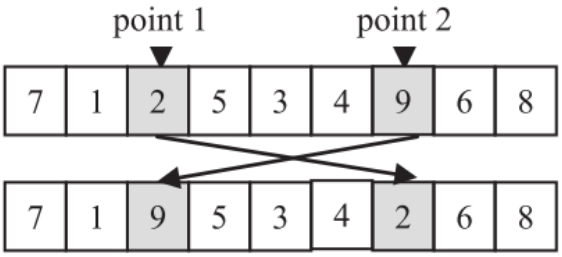


Figure 2-4 - Swap mutation. Image taken from (Chung & Kim, 2016)

2.5.5. Elitism

Even though, to improve the results obtained by the fitness function, the elitism component could be used. It allows to use a fraction of the population (the best individuals of the population) which survive to the next generation. This means that, those individuals are copied to the next generation without being subjected to the reproduction techniques (Ordóñez Galán et al., 2017).

2.5.6. Fitness Function

The fitness function is where chromosomes are evaluated. Each individual has its own fitness value that evaluates how strong it is in population and indicates the survival probability (Casas et al., 2016; El-Abbasy, Elazouni, & Zayed, 2016; Page et al., 2010). They have better values of fitness accordingly to how good they are in population. In the end of the algorithm the best solution(s) found will be available. To obtain good solutions it is necessary to try the algorithm with different probability values on both crossover and mutation and find the better ones.

Normally, this function evaluates the maximum completion time of the schedule and the one with smallest value is considered the fittest one (Balin, 2011; El-Abbasy et al., 2016; Kundakcı & Kulak, 2016). However, other evaluations can be used. In this study case, fitness function will focus in get the lowest total makespan, with the constraint that the maintenance tasks should be allocated to the begin or the end of the time window defined before initialize the schedule. Also, is available a third method that allocates the maintenance tasks the same way as the production tasks, not considering if they are allocated to the begin or the end of the schedule time window.

2.6. Summary

GAs are a great solution to solve combinatorial optimization problems, such as scheduling problems, due to their capacity to achieve good solutions in large search spaces. Besides, generally, GAs require more process power and consume more time than other algorithms, such as dispatching rules, they are more flexible and achieve better solutions most of the times, like it is possible to verify by references above. GAs are very efficient and easy to implement, so they will effortlessly solve a dynamic flexible job-shop scheduling problem. How was possible to verify above, there are some solutions in literature that try to solve dynamic task allocation in a flexible job-shop. The author will try to get the best solution possible for this exact problem, based on the presented literature.

3

Scheduling Architecture

As described in Chapter 2, in the last years the market changes are leading to a rearrangement of the manufacturing industry. To face the new trends, manufacturing systems need to quickly adapt in order to give fast responses to the market changes. Consequently, the shop-floor needs to be more agile and flexible.

In that way, the PERFoRM H2020 project is looking to a smarter and pluggable system, where resources are able to communicate with each other more efficiently, in a distributed system.

The proposed architecture was integrated in the PERFoRM project and it aims to develop a scheduling tool capable to plan schedules with both production and maintenance tasks in different production stations. However, to better understand how the system works and the interaction between tools, the architecture of PERFoRM is presented in this chapter.

The scheduling architecture developed allows to generate new schedules and add or remove operations to the system. It can interact with other tools through a common middleware, this way it can get the information about the operations to allocate and the stations available in the shop-

floor. This information needs to be PERFoRM-compliant, i.e. it should respect the PERFoRM data model, so all the tools can communicate by a common data format.

Along with the scheduling tool itself, it was developed a Graphical User Interface (GUI), where a human operator can interact with the tool to generate schedules or add new tasks to the list.

3.1. PERFoRM Architecture

The architecture presented here was developed as part of the PERFoRM project. The system architecture, illustrated in Figure 3-1, for the production system is based on a network of hardware devices and software applications which are connected by an industrial middleware. The work carried out in this thesis focused on one of the tools of the PERFoRM, namely the scheduling tool.

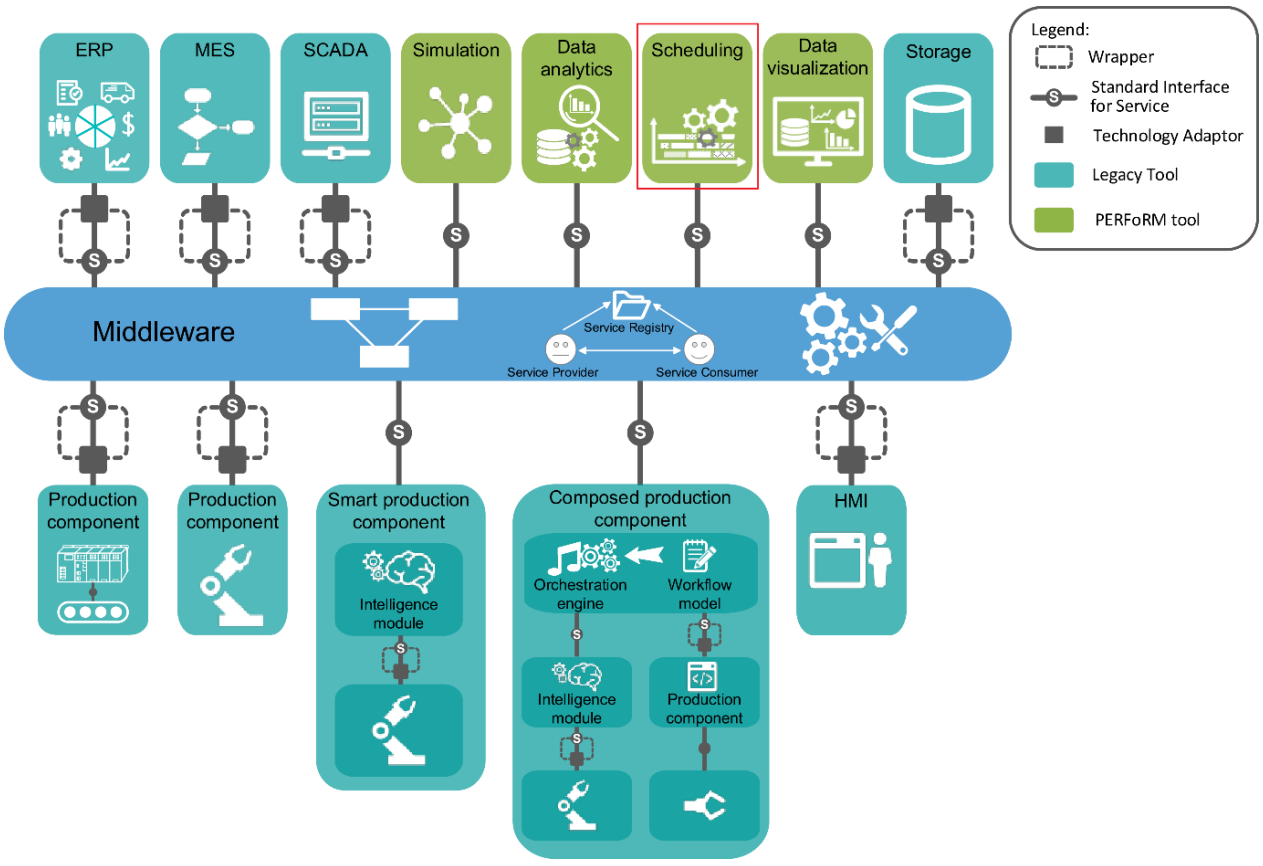


Figure 3-1 - Overview of the PERFoRM system architecture (PERFoRM, 2016)

The middleware is a distributed service-based integration layer and its main goal is to guarantee a transparent, secure and reliable interconnection of the different hardware devices (such as robotic cells and Programable Logic Controllers) and software applications (such as monitoring tools) presented at the PERFoRM ecosystem. The middleware allows each application to interact

with each other, without the need to know about their inner structure, from low level sensors and actuators to management systems. Legacy production systems need to use adapters, which translate their own data model into the standard interfaces defined in PERFoRM, to be compliant to this approach. The middleware provides a service registry which stores the different services available in the system. Furthermore, it takes care of connecting components which need to communicate and translating the data in a transparent manner (PERFoRM, 2016).

The PERFoRM architecture adopts the standard interfaces as the main drivers for pluggability and interoperability, to allow the connection between hardware devices, such as robots and controllers, and software applications, such as databases and management or analytics tools, in a transparent way. Such interfaces should support the devices, tools and applications that fully expose and describes their services in a unique, standardized and transparent way to improve the seamless interoperability and pluggability, fully specifying the semantics and data flow involved in terms of inputs and outputs required to interact with these elements. To expose the system functionalities as services, a common data model is adopted, serving as the data exchange format shared between the PERFoRM-compliant architectural elements. This data model covers the semantic needs associated to each entity. In this context, two data abstraction levels are taken into account, more specifically the machinery level, covering mainly the automation control and supervisory control layers, and the data backbone level, which covers manufacturing operations management and business planning and logistics layers (PERFoRM, 2016).

Interoperability and harmonization of data at a system of systems level is achieved by connecting the standard interfaces with the data model for a common representation of data and system semantics. Even though, considering the integration of legacy devices and their own individual data models and semantic requirements it is necessary to add technology adapters, in order to enable the translation and mapping of legacy data into the common PERFoRM representation (PERFoRM, 2016).

The technology adapters are the crucial elements to connect legacy systems to the PERFoRM middleware and to transform the legacy data model into the standard interface data model described in section 3.2. Thus, the technological adapters are only necessary when there is the need to connect a legacy component (e.g., an existing DB or robot) to the PERFoRM system (PERFoRM, 2016).

The human integration is a crucial aspect in PERFoRM project to improve flexibility. Human-machine and human-human interfaces support decision-makers to take strategic decisions and operators or maintenance engineers, at operational level, to perform their tasks. Those interfaces allow to share the view, the screen, the information, among other options, between colleagues

even if they are not physically present on the shop-floor. It is also possible to alert the operator if some abnormal event occurs or provide a mobile turn-by-turn guidance to navigate through the factory to retrieve tools or equipment (PERFoRM, 2016).

Once machines are becoming to be smarter and the number of data being generated in shop-floor is increasing at a very high rate, robots and automation machinery need to be empowered with intelligence and higher processing capabilities to run more complex algorithms. Thus, machines are being equipped with a large number of sensors which generate huge streams of raw data. By analyzing this data locally instead of overloading the network with non-meaningful data, communication latency times can be reduced, enabling faster reaction to events or trend deviations (PERFoRM, 2016).

Tools designed with advanced algorithms and technologies to support the production planning, scheduling and simulation may improve the system performance and reconfigurability. These tools should be PERFoRM compliant. Thus, they should follow or be translated to the PERFoRM data model.

3.2. System Data Model

Despite being outside of the scope of the work developed, it is important to give a brief explanation of the data model system, to understand which ones and how the classes of the system are used by the scheduling algorithm.

In the next sections are described the classes from the data model proposed in PERFoRM project, presented in (PERFoRM Project, 2017), which have some interaction with the developed scheduling algorithm. This data model was developed to allow the PERFoRM-compliant elements to share the same data exchange format, using the PERFoRM Markup Language (PML). This way, it is possible to interact between the scheduling tool and the middleware, allowing to access classes such as *PMLEntity*, *PMLSkill*, *PMLConfiguration*, *PMLProduct*, *PMLOperation* and *PMLSchedule*.

Using this PERFoRM-compliant data model, the proposed scheduling architecture can access the information provided by other tools through the middleware, with only one data exchange format used by all PERFoRM elements.

3.2.1. PMLEntity

The PMLEntity class is the generic representation of shop-floor entities, which encapsulates all the necessary information associated to both components and subsystems. PMLComponent and PMLSubsystem classes extend the characteristics of a PMLEntity, Figure 3-2.

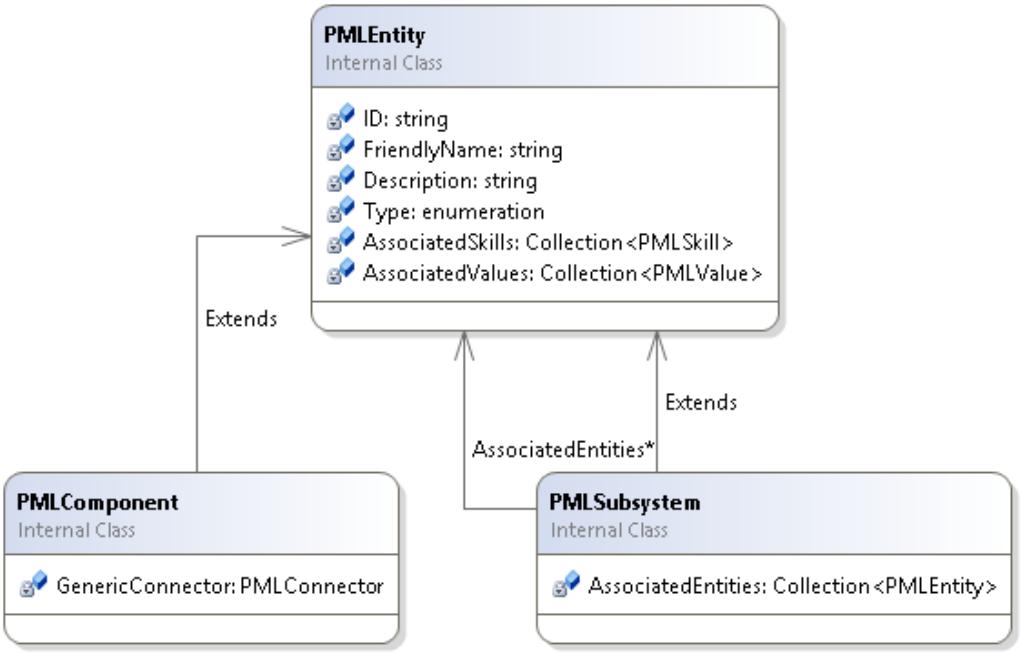


Figure 3-2 - PMLEntity, PMLComponent and PMLSubsystem classes

Thus, PMLEntity class allows the existence of generic collections of elements that can be either components or subsystems. The PMLComponent is basically a single entity that can perform skills and present relevant data regarding its state and operation. On the other hand, the PMLSubsystem is a recursive element once it can contain other PMLEntities within.

This generic design allows that for example a robot could be the lowest entity of a system, being treated as a simple component, and the same robot can be seen as a subsystem inside the system, involving different sensors as its components, using the same data model.

3.2.2. PMLSkill

The PMLSkill class represents the tasks that a PMLComponent or PMLSubsystem can perform. A certain skill is characterized by a unique identifier, as well as by series of associated configurations and parameters. Like PMLEntity, the PMLSkill is not directly used, enabling

instead the creation of generic collections of PMLAtomicSkill and PMLComplexSkill, Figure 3-3.

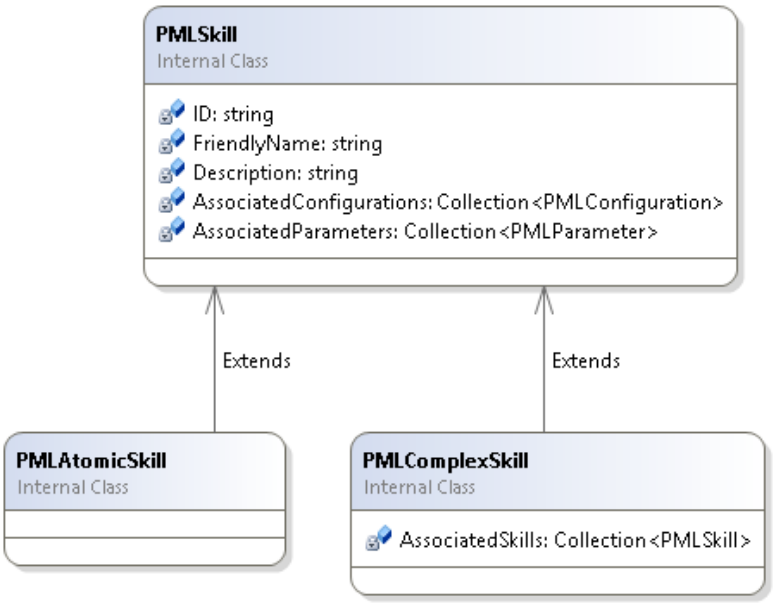


Figure 3-3 - PMLSkill, PMLAtomicSkill and PMLComplexSkill

The PMLAtomicSkill represents the simplest form of a skill, a single action performed by a given entity. On the other hand, the PMLComplexSkill specifies a skill which consists in the combination of multiple skills, which can be atomic or complex.

3.2.3. PMLConfiguration

The PMLConfiguration class provides a high-level description of a possible configuration to execute a given skill, according to a set of specified parameters. Is this class which provides the duration time of a task. Its composition is represented in Figure 3-4.

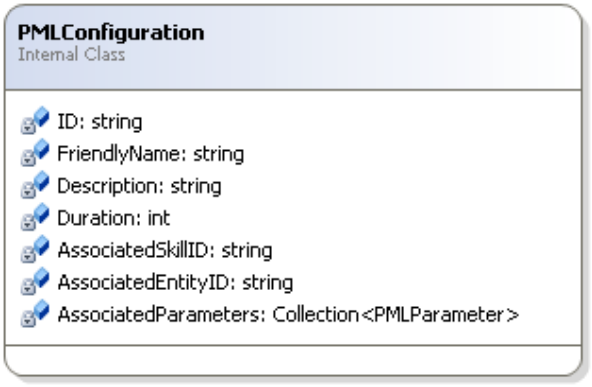


Figure 3-4 - PMLConfiguration

3.2.4. PMLProduct

The PMLProduct class provides an abstraction of a given product variant, along with its core-defining characteristics to enable a process-oriented description of the product, as represented in Figure 3-5. The product is defined by a unique identifier and contains a collection of skills, representing the skills by which it can be produced. It can contain a link to an external description of its geometric characteristics, using for this purpose the COLLAborative Design Activity (COLLADA) interchange format, an opened standard XML schema for the exchange of 3D assets among software applications.

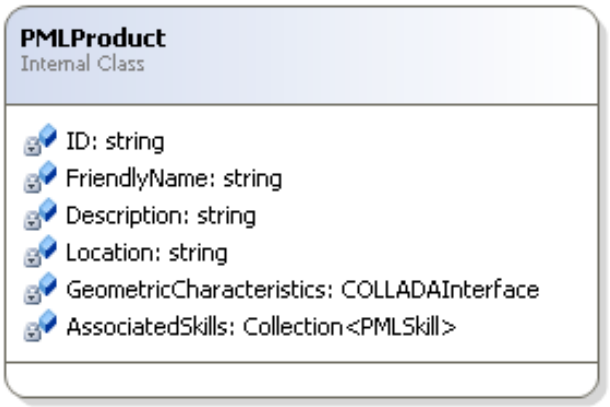


Figure 3-5 - PMLProduct

3.2.5. PMLSchedule and PMLOperation

The PMLSchedule class represents a unique schedule, associated to a certain PMLEntity. It contains the start and end times and a collection of the corresponding associated PMLOperations, as demonstrated in Figure 3-6.

The PMLOperation class is characterized by a unique identifier and contains all the information related to a task. Namely, the product to which that task is associated, the start and end times, the associated skill used to perform the task and the operation type, indicating if it is a production or maintenance task.

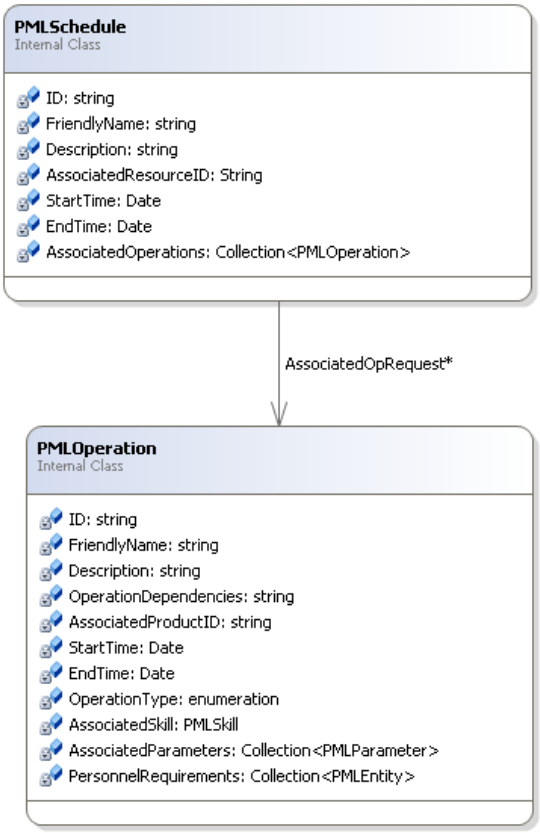


Figure 3-6 - PMLSchedule and PMLOperation

In the next section is presented the dynamic scheduling tool developed in this work.

3.3. Dynamic Scheduling Architecture

The scheduling architecture proposed here is integrated in a more complex environment as demonstrated in section 3.1. The tool can interact directly with the middleware to send and receive information from the resources operating on the shop-floor. Thus, it is possible to know which working stations are available to operate, the operations that need to be allocated and the maintenance shifts available. The scheduling tool is composed by two modules. The first one is the scheduling itself, which processes the information and generates the schedules. The second module is the GUI, which is an interface where is possible to add and remove tasks to the tool and to request new schedules. The interaction with the tool is done through a human operator.

This operator does not need to be present in the factory, since the tool can be controlled remotely, through the GUI. The human operator can add or remove operations to the tool using his experience or using the information provided by the other tools. Those interactions are represented in Figure 3-7.

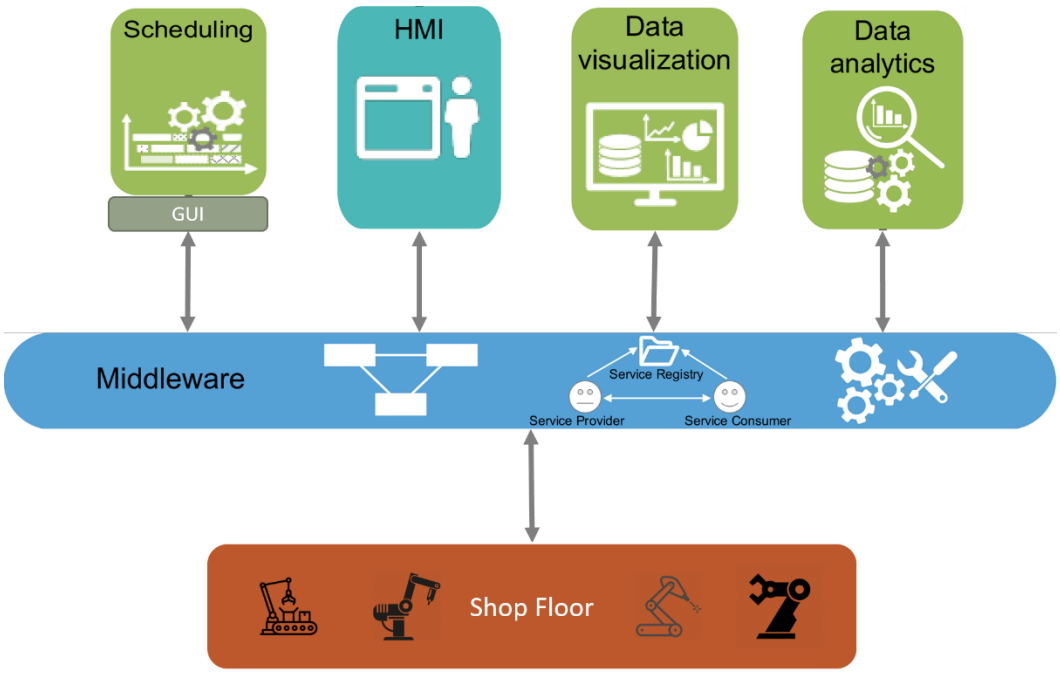


Figure 3-7 - Overview of architecture interactions

More specifically, the interaction between the scheduling tool and the human operator is done through a graphical interface. The interfaces communicates with the scheduling tool through a web service which allows computers to communicate through the Internet. Through the interface, it is possible to insert new maintenance tasks and remove them if necessary. Each time the operator chooses to add a task the information about that task, such as the ID, associated station and task duration, is sent to the tool and the task is inserted. To remove a task is sent the task ID to the tool so that task is removed from the set of tasks to allocate. Then, if it has been successful, the information is sent back to the graphical interface which is, at that time, updated, as shown in Figure 3-8.

When the operator chooses to generate a new schedule, the interface accesses the middleware in order to get the information about the production tasks to allocate and the stations available to operate those tasks. Then, it sends all that information to the scheduling tool. The scheduling tool receives the production tasks, the available stations and accesses the stored maintenance tasks to generate a schedule for each station. After executing the GA and obtaining the schedules, the scheduling tool sends it to the GUI which in turn sends it to the middleware. So, any other tool can access the available generated schedules. This process is represented in Figure 3-8.

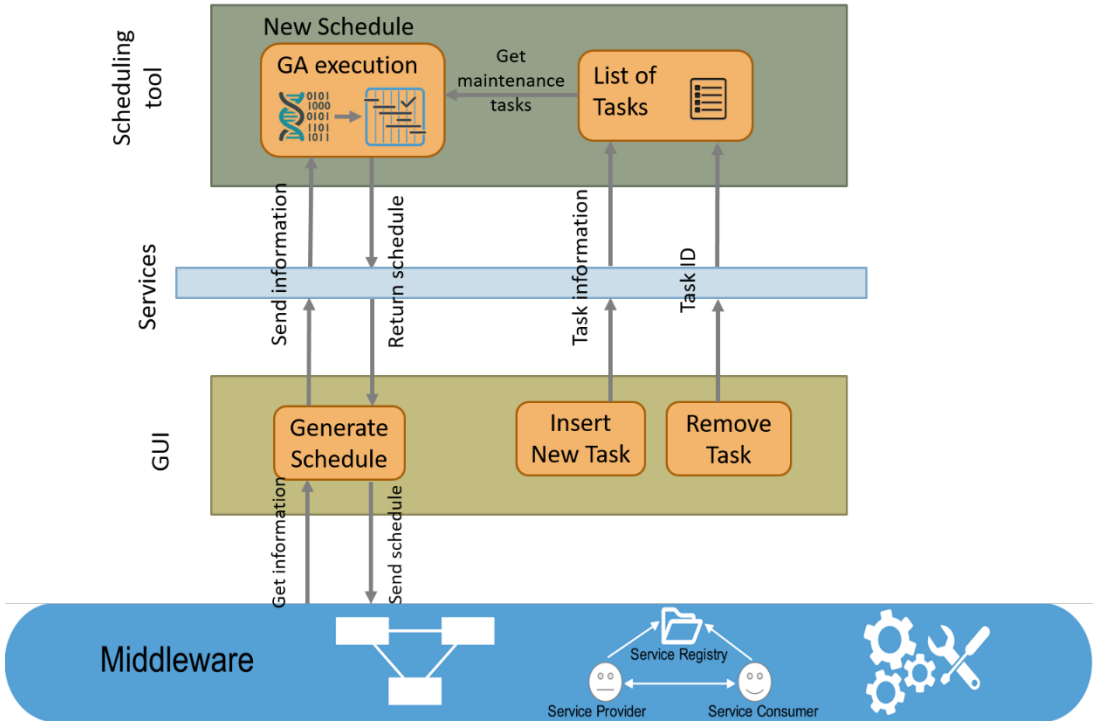


Figure 3-8 – Scheduling tool architecture

The scheduling tool architecture proposed here consists, mainly, in acquire the information needed to execute the GA, execute it, which generates a collection of schedules, and, finally, send that collection to the middleware. The workflow of the developed tool is represented in Figure 3-9.

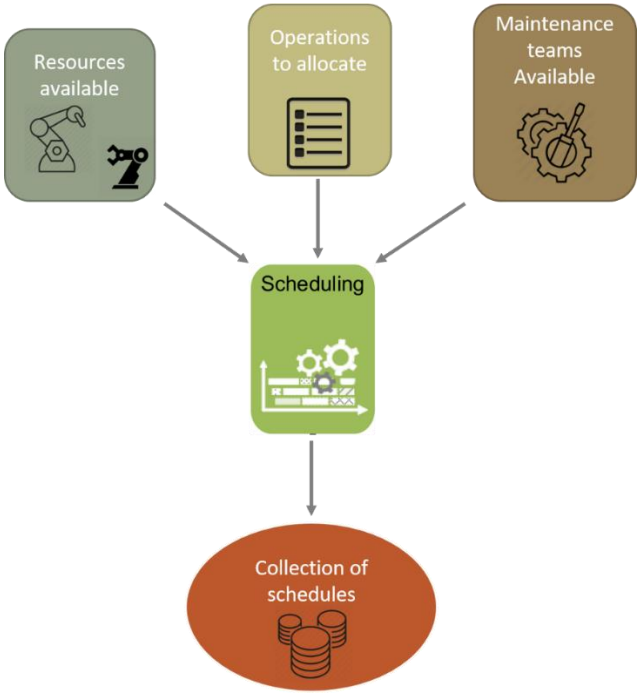


Figure 3-9 - Scheduling tool workflow

First, it is necessary to know which resources are available in the shop-floor before executing the algorithm. That data is obtained from the middleware, which contains the information of each station. That information includes the station ID, the skills it can perform, the configurations used for each skill (which contains the task duration) and if it is available or not.

Another initial requisite that should be known is the information about the maintenance execution, i.e., the number and type of maintenance teams that are available, to thereby the maintenance operations could be properly allocated.

Finally, but not least, a crucial part is to obtain the list of operations to allocate. The production operations are available in a file stored in the middleware that contain all the information about them, for example the ID, expected duration of the operation and in which station it will be performed. The list of maintenance operations can be available in the same file as production operations or it can be added a single operation through the GUI. There are some mandatory fields which each operation should contain, such as the operation ID, the associated skill ID, the associated product ID and the operation's number and type. Without them the schedule to the corresponding operation is not performed. Operation ID represents the identification (unique) of each operation. The associated skill ID refers to the skill identification associated to that operation, whereby is possible to obtain the configuration to that skill and finally the duration of each one. The associated product ID is the identification of the product to which that operation corresponds. With the operation number, it is possible to know the production order of the operation of each product, lowest is the first and highest the last. Finally, the operation type specifies if that operation is a production or maintenance operation. In the case of maintenance tasks, Associated Product ID and Operation Number are not necessary, once they do not have a product associated and is considered that each one only corresponds to one operation. It is important to note that, in this case, maintenance tasks should be both predictive and time-based. If a corrective intervention is needed then a reschedule should be done, which can include or not a CM operation.

Then, with all necessary data collected, the algorithm should be executed and a collection of schedules obtained. In this collection is present a schedule for each work station present (and available to perform operations) in the shop-floor. Each one contains all the operations to be executed in that station. To each of those operations, start and end times are assigned, so it is possible to know when a given operation will be performed. As a final note, the schedules need to be translated to be PERFoRM-compliant before the collection of schedules be sent to the middleware, where is submitted to simulation tests, to verify its feasibility.

To generate a new schedule, it is necessary to allocate the tasks efficiently to obtain a feasible and valid schedule, i.e. avoiding that a station is operating more than one task at the same time or preventing a product from being present in different stations at the same time. Since this is a combinatorial problem, it is presented a GA architecture to solve this JSSP. The description of GA's architecture proposed is described in section 3.4.

3.4. Genetic Algorithm

In this chapter is presented the GA's approach used to perform the tasks' allocation in the scheduling tool.

As mentioned in section 2.5, to create new individuals for the new generations, GAs use three genetic operators: selection, crossover and mutation. However, first, the chromosome needs to be encoded, which means that the structure of the chromosome needs to be defined.

Then, the fitness function needs to be defined. The fitness function will evaluate each individual in the population and attribute it a value, the better the value the stronger it is. The fitness value of a chromosome is represented by the higher makespan value between all the stations present in the chromosome multiplied by a value which represent the number of errors on that schedule. There is an error when some task is allocated before its precedent task. At the end, the chromosome with the lowest fitness value represent the best solution found in each generation.

Before starting the GA's engine, a random population with a predefined size, based on the encoded chromosome, was generated. To do so, it was set the population size. After generating the initial population, the fitness value of each individual is calculated. Then, the genetic operators are applied until the termination criteria is reached, as it is demonstrated in Figure 3-10. The termination criteria can be reached, for example, by a predefined number of iterations or if the best fitness value doesn't change for a certain number of successive iterations.

In scheduling problems, each individual of a population represents a schedule. From generation to generation, some percentage of the fittest individuals (or only the fittest one) are maintained (survive) and new ones are created. This way, it is ensured that the best schedule found will be in the final solution. Each individual has its own fitness value that evaluates how strong it is in the population and indicates the survival probability.

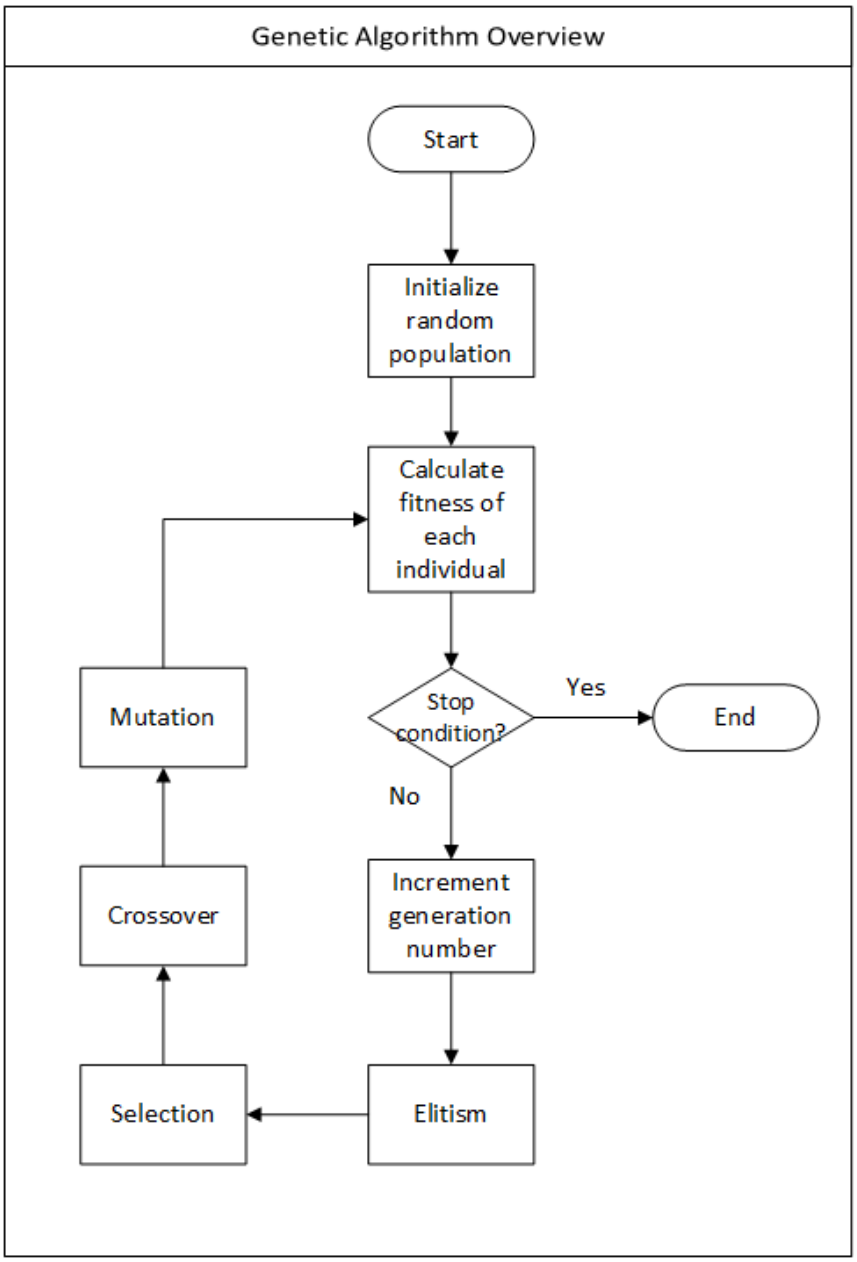


Figure 3-10 - Genetic algorithms flowchart

The GA's architecture proposed in this work consists mainly in four steps:

- The most important but also most difficult step is to transform the problem into an appropriate genotype representation. In this case, the goal is to obtain a chromosome representing a valid schedule. To initialize the chromosome a sequence of elements was created. Each of those elements represents a gene and each task to allocate is assigned to a different gene. Then was created a genotype with the previous chromosome, where a fitness function can be applied to evaluate the individuals. Since it is needed to change the genes' position keeping the exactly same genes in the

chromosome, it was used a genotype containing a permutation chromosome. This chromosome generates a random sequence of those genes and allows to model permutations between them;

- The next step was to define the fitness function. To obtain the solution with the shortest execution time, this function was minimized. This means that the schedule with the lowest fitness value will be selected. The fitness function is described in detail in section 3.4.6.
- The third step was to create the evolution engine. This engine is responsible for evolving the given population. Here, it is possible to control the parameters of the evolutionary environment. To alter the behavior of the environment, different alterers and selectors can be used, as well as the population size, the offspring fraction and many other parameters.
- Finally, the evolution process from the previous engine was created. It allows to limit the process evolution or to peek the statistics of the procedure. The stream needs to be limited in order to do not run forever. Then, the final solution, in this case the best phenotype, which is composed by the genotype and the corresponding fitness value, is collected.

So, the first step is to translate the problem from the real world to a genotype, denoting what means a chromosome and its genes in the real problem. This way, is possible to improve the solution to a feasible one.

3.4.1. Encoding

The first step of the GA was to define what is a chromosome and how it is composed, so that it portrays the real problem.

A chromosome is a schedule solution for the entire shop-floor, where each gene represents an operation to perform. So, each different chromosome represents a different schedule. Assuming that Station j, M_j only performs one type of tasks on a Product P_i. In Figure 3-11 is an example of the chromosome representation. Each gene results from the merge of a product associated to a station, represented as P_iM_j. The first operation of product 1 will be performed in station 2, then comes the first operation of product 2 to be performed on station 3, etc.

P_1M_2	P_2M_3	P_1M_1	P_3M_1	…	P_2M_2

Figure 3-11 - Chromosome encoding

Each gene is composed by a task which contains the following information:

- Task ID;
- Task duration;
- Associated station;
- Task type;
- Associated product;
- Task priority.

When the best solution is obtained, that chromosome is split into several schedules. So, the schedule for the entire shop-floor is divided into a schedule for each station present on it. Then, it is translated to be PERFoRM-compliant and is ready to be sent to the middleware.

After defining the chromosome, it is necessary to choose the number of generations which the process should evolve before stops and the size of the population.

3.4.2. Generations and Population

The number of generations and the population size are both crucial parameters in GA and should be chosen cautiously to have enough flexibility to let the algorithm evolve and do not waste too many time and processing power.

The number of generations in the algorithm is the simplest way for terminating the evolution process. It should be tested several times with different values to find a good one to each situation. A small value may not be enough for the algorithm to evolve into a good solution but on the other hand, a high value could be consuming processing time unnecessarily because a good solution could be found too soon.

In the proposed architecture, it was also defined another limit for the number of generations. This limit truncates the evolution process by a steady fitness value, this means that, if the fitness value remains constant for a given number of generations, the evolution process stops. It could help to stop the process earlier, since, if it does not evolve for a while it may be stagnant.

At the same time, the population size should be large enough to have diversity among the individuals, but not too large that processing power and time are being consumed pointlessly. A small population could be very difficult to evolve into a feasible solution since the mating between individuals may not lead to a good offspring when the search space is too large.

So, both the number of generations and the population size should be defined depending on the problem size and complexity. Different values for these parameters used in different problems are present in section 5.2.

After choosing the right parametrizations for the number of generations and the population size, the selection process needs to be defined. This decides how the population for the next generation is selected.

3.4.3. Selection

The selection process could be chosen independently for both the survivor and the offspring populations. Several selector types are available to choose the individuals which will be selected to the reproduction process. In the proposed GA, the selector used for both cases was the tournament selector with size three.

As shown in section 2.5.5, first it is necessary to calculate the probability, *p*, of selection for each chromosome, using the equation

$$p_i = \frac{f_i}{\sum_{i=1}^{3} f_i}$$

where *f* is its fitness value and *i* is the individual.

After spin the roulette wheel, a chromosome is chosen according to its probability. The one with a best fitness value has a higher chance to be chosen. In Figure 3-12 is presented an example, merely illustrative, of this method. Three individuals from the population are chosen randomly to dispute the tournament. Assuming that individual A has a selection probability of 64%, individual B has 25% and individual C has 11%. When the wheel is turn, the individual with higher probability, i.e. with the best fitness value, has higher chances to be subject to the genetic operators.

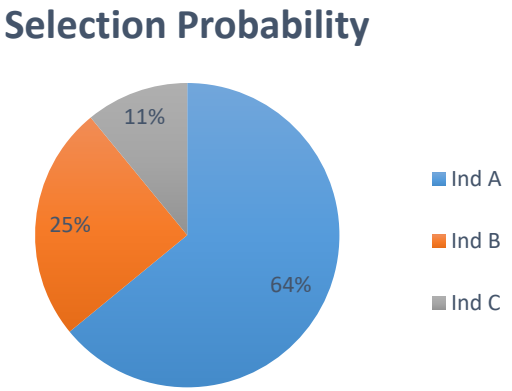


Figure 3-12 - Roulette wheel selection with size three

The next step is to define the elitism, the fraction of the population which will be directly present in the next generation, without being submitted to the genetic processes.

3.4.4. Elitism

To choose which portion of the population survives to the next generation and which part is created (offspring) it is needed to select the elite population. Different values of the elite fraction could be used in different situations. In this work was used an elitism value of 4%, left that way a value of 96% to the offspring population. Tests with different values were executed and the probability of 4% proved to get better fitness values with small processing times than others. Using a small value to select the elite population, it is possible to ensure that only the best individuals are preserved to next generations and the rest of the population is created, allowing to explore a larger search space.

After selecting the elitism fraction, it is necessary to choose the genetic operators to use, namely crossover and mutation.

3.4.5. Crossover and Mutation

Crossover and mutation determine how the search space can be crossed and are responsible for the genetic diversity of the evolution process.

Mutation allows the population to explore the search space by making small moves. This is normally lower than crossover but can be a great help in problems where crossover is disruptive. Mutation can also be crucial to provide the diversity which crossover needs.

Crossover creates a new chromosome by recombining parts of two individuals, exploring most of the search space. Crossover probability should also be optimized accordingly to the problem characteristics.

In section 5.1 are presented experiments with different genetic values applied to JSSP with different sizes.

Finally, it is necessary to define the fitness function, which evaluates how good is each individual present in the population.

3.4.6. Fitness Function

The fitness function is, also, a very important part in the GA's modelling. It evaluates each chromosome and sets the fitness value of each one. This value allows the evolution engine to select the offspring and survivor population.

To face this scheduling problem, the fitness function was implemented taking into account some important rules. The best fitness value is calculated by finding the minimum total makespan in all generations, where the total makespan is given by the station with the maximum execution time in each generation. The fitness value also considers the order by each operation is performed, i.e. if operation 2 is performed before the operation 1 it leads to an invalid solution which is represented by a fitness value which is multiplied by 10, this way is possible to identify invalid solutions.

Scheduling rules:

- Only one operation of each job may be processed at a time;
- No pre-emption is allowed, which means that is not possible to interrupt a task and resume it later;
- Each job must be processed to completion;
- Jobs may be started at any time, no release times exist;
- Jobs may be finished at any time, no due dates exist (within the time window defined);
- No station may process more than one operation at a time;
- Station setup times are not considered;
- Stations may be idle within the schedule period;
- Stations are available at any time, since they are considered available;
- An operation can only be performed once;
- Operations precedence within a product should be respected;

- Maintenance tasks must be executed during the maintenance shift.

To obtain the fitness value of each individual, it is necessary to go through the genes of each chromosome. The start and end times are set to the first task present in the chromosome, then to the second one and so on. The way those times are calculated is defined below, in equations (4) to (7).

A mathematical explanation is presented here to clarify how the fitness value of each individual is reached.

The equation (1) describes the fitness function. The fitness value *f* is given by the minimum of the maximum completion time (makespan) of each generation times a weight which is increased if some task is not allocated in the right order.

$$f = Min[\, C_{max} * weight \,] \tag{1}$$

The total makespan of the job-shop, C_{max}, denotes the execution end time of the last station and is represented in equation (2). It represents the total time to process all the operations in the shop-floor, including idle time between operations.

$$C_{max} = Max[fs_j] \;\; , \;\; j = 1, \dots, n \tag{2}$$

The total execution time of each station *j*, fs_j, is set by the later final time of the operations allocated to that station, f_{ih}, calculated in (3), since the operation *h* is allocated to station *j*.

$$fs_j = \begin{cases} f_{ih}, & fs_j < f_{ih} \,;\, h \in j \\ fs_j, & otherwise \end{cases} \;\; , \qquad i = 1, \dots, n; \;\; h = 1, \dots, m; \;\; j = 1, \dots, k \tag{3}$$

The ending time of each operation is calculated in (4), where the start time of operation *h* of product *i*, s_{ih}, is replaced by the start time of maintenance task *h*, sm_h, and the process time of operation *h* of product *i*, p_{ih}, is replaced by the process time of maintenance task *h*, pm_h, in maintenance tasks case.

$$f_{ih} = s_{ih} + p_{ih} \;\; , \;\; i = 1, \dots, n; \;\; h = 1, \dots, m \tag{4}$$

To calculate the start time of the operation it depends if it is a production operation or a maintenance operation. In first case, constraint (5), it is calculated based on the previous operation of the corresponding product if any, otherwise starts at time zero. Though, if there is already another operation, in the same station, allocated to that time slot, this one is altered to a new time, starting at the end of that task, f_{jh}.

$$s_{ih} = \begin{cases} s_{ih-1} + p_{ih-1}, no\ overlapping\ h > 1 \\ 0, \quad no\ overlapping; h = 1 \\ f_{jh-1}, \quad overlapping \end{cases}, \quad i = 1, \dots, n; \quad h = 1, \dots, m \tag{5}$$

If it is a maintenance operation the start time of the operation is calculated based on the user intent. It could be desirable to have the maintenance tasks allocated as soon as possible, as late as possible or a third option where there are no pretensions about maintenance tasks allocation. However, there is an important aspect to respect, the maintenance tasks should be executed during the maintenance shift, which in this case is between 6a.m. and 2p.m.

The maintenance task start time to "as soon as possible" case is calculated in (6). The Maintenance Shift Start time is represented by MSS, the Maintenance Shift End time is represented by MSE, DT represents the Day the Task is performed and 1440 minutes are the total minutes of one day. It is assigned a start time and if there is a conflict with other operation in the same station that time is changed to the next slot within the maintenance shift time. If that maintenance shift is already full, it is allocated to the beginning of the maintenance shift of the day after and so on.

$$sm_h = \begin{cases} MSS + 1440 * DT, & there\ is\ overlapping;\ f_{ih} > MSE \\ s_{ih-1} + p_{ih-1}, & there\ is\ overlapping;\ f_{ih} \leq MSE \\ MSS, & there\ is\ no\ overlapping\ between\ tasks \end{cases} \tag{6}$$

To calculate the start time of a maintenance task "as late as possible", the constraint (7) is used. First, the task is allocated to the last shift maintenance slot of the time window. If there is overlapping with another task, it is allocated to the next slot in that shift. In the case that shift is already full, it is allocated to the previous day and so on. The *timeWindow* represents the amount of days reserved to the schedule.

$$sm_h = \begin{cases} MSS + 1440 * (DT - 2), & there\ is\ overlapping;\ f_{ih} > MSE \\ s_{ih-1} + p_{ih-1}, & there\ is\ overlapping;\ f_{ih} \leq MSE \\ MSS + 1440 * (timeWindow - 1), & there\ is\ no\ overlapping\ between\ tasks \end{cases} \tag{7}$$

To check the operations order, each product contains a collection with all operations belonging to that product. If an operation is executed after another one with higher priority, then a parameter representing the weight of the errors occurred is incremented (multiplied by 10), leading to a very large fitness value when the solution is invalid. If the total makespan is, for example, 125, but there is one error, so the fitness value will be 1250. This way is possible to know the number of errors present in each generation.

Finally, it is obtained a solution containing a schedule for each station in the shop-floor, with the respective start and end times of each operation.

Based on this definition of the architecture an implementation is suggested in chapter 4.

4

Implementation

In this chapter is described the implementation of the proposed scheduling architecture, discussed in sections 3.3 and 3.4. The scheduling tool was developed using Java language, the Jenetics library and the Restlet API to provide REST services.

Restlet is a REST framework for Java platform. This allows the computer systems to communicate between them through the Internet.

Jenetics is a library written in java which allows an easy, but very complete, implementation of GA in different problem domains. The only runtime dependency to other libraries is to Java 8 runtime. The library has a clear definition of the different concepts of GA, such as Gene, Chromosome, Genotype, Phenotype, Population and Fitness Function. It allows to easily change between minimize and maximize the fitness function without the need to tweak it (Wilhelmstotter, 2016). For a deeper knowledge about Jenetics the reader is encouraged to visit its webpage, where is all the information about this library, at http://jenetics.io/.

In the next section are presented the implementation of the services provided by this scheduling tool.

4.1. Services

The scheduling tool provides several different services. One of them is to generate a new schedule, based on the information available about the operations to perform and the stations available. The other one allows an operator to insert new maintenance tasks in the system, which will be included in the schedule when a new one is requested. The third one, allows to remove a task inserted by the operator.

To create the RESTful web services, an object of *CorsService* from *restlet.service* class was instantiated. This allows cross-domain communication from the browser and is possible to specify the allowed origins and the allowed credentials, using the *setAllowedOrigins* and *setAllowedCredentials* methods, respectively. The parameter passed to the allowed origins method was a new instance of *HashSet(Arrays.asList("*"))*, which means that the resource can be accessed by any domain . The allowed credentials were set to *true*, which means that the browser will expose the response to the application. This way, it was possible to run both the server and the client sides on the same machine.

In order to set up new services an instance of the *Component* object from the *restlet* class was created. Through the *getServices().add* method from the *Component* class, was defined that the service would run under HTTP protocol on port 8182. By the *getServices.add* method, the *CorsService* instantiated before was added to the list of services.

From here, it was only needed to execute the *getDefaultHost.attach* method from the component class, to provide a new service, passing the name of the service and its class as parameters, such as the *newTask*, *removeTask* or *newSchedule* services, as represented in Figure 4-1.

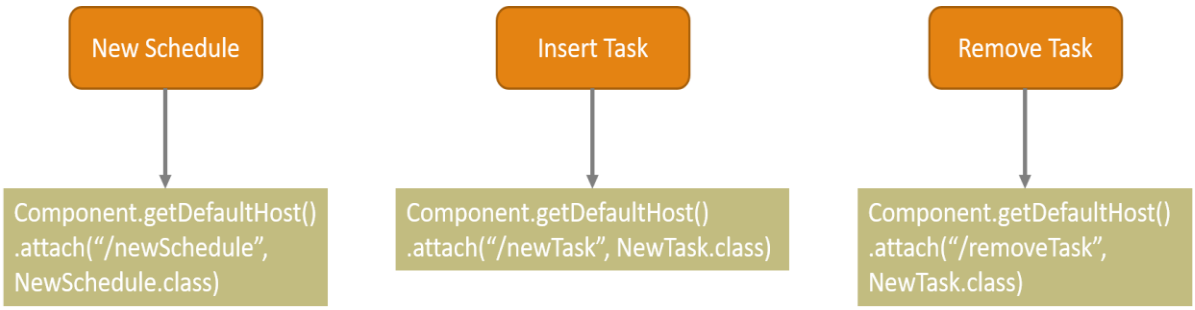


Figure 4-1 - Service methods to execute a new schedule, insert a task and remove a task, respectively

Finally, the services were set up using the *start* method from the *Component* class.

The implementation of the scheduling tool itself is defined in the following section.

4.2. Scheduling

To implement the scheduling tool, it was developed an algorithm which processes all the information received from the PERFoRM middleware, generates a collection of schedules and sends it back to the middleware. The class diagram developed is presented in Figure 4-2 and specifies the interactions between the created classes and the main methods used in the algorithm.

The *SchedulingInterface* class stores the information about stations, products and operations to allocate, obtained from the respective files present in the middleware. It also stores an *HashMap* of the generated schedules.

The S*tation* class defines a work station and includes the associated skills it is able to execute, the operations list allocated to that station and its makespan, among others.

In the same way, the *Task* class contains all the information about an operation to execute. Such as its ID, the station where it will be executed and the duration, in minutes, of the task.

On the other hand, the *Product* class abstracts a product, containing its ID as well as the operations needed to be performed.

The *GeneticProcessing* class translates the information obtained by the interface to be able to be executed by the GA in the *Scheduling* class, using the *convertOperations* method. Once the GA algorithm is performed and the best solution is found, i.e. the best schedule for the shop-floor, this one is divived in several schedules, one for each station. Finally, the GeneticProcessing class translates the schedules obtained to be PERFoRM-complaint, using the *toPMLSchedule* method, and sends this collection of schedules to the middleware.

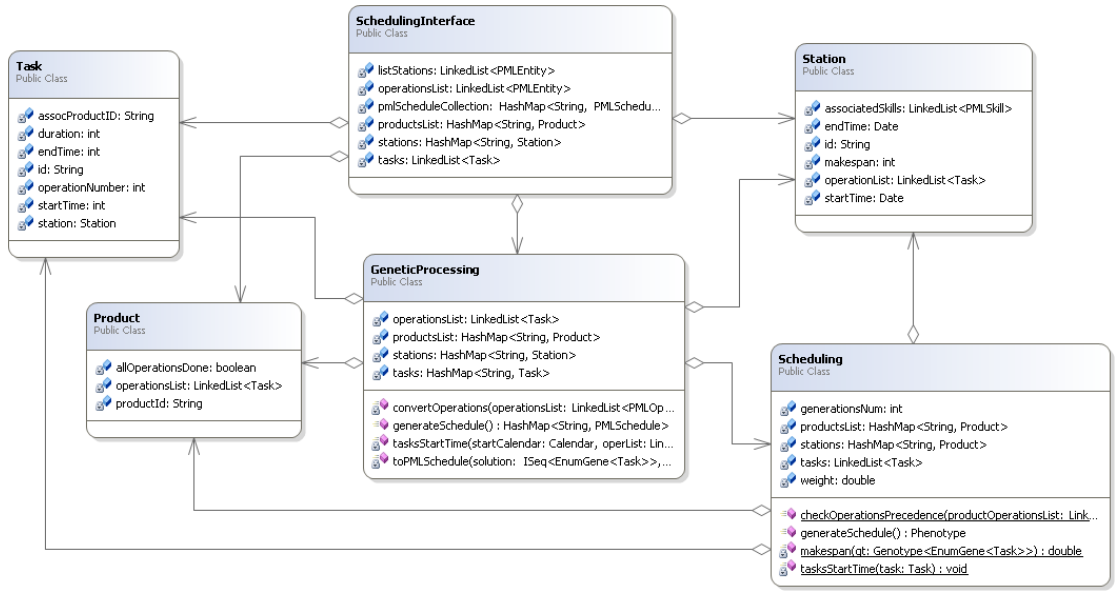


Figure 4-2 - Scheduling class diagram

The *Scheduling* class executes the GA process by call the *generateSchedule* method. It uses the *checkOperationsPrecedence* method the check if the operations respect their order and the *tasksStartTime* method to attribute the starting times to each task. The *makespan* method corresponds to the fitness function and determines the fitness value of each individual.

In Figure 4-3, is represented the sequential diagram of the scheduling tool indicating the interactions between the different classes. The scheduling tool first gets all the available stations from the middleware and instantiate each one of them as a Station object. Then, it gets the collection of PMLOperations, also from the middleware, and creates a Product object each time there is no product instantiated for that operations. After that, a GeneticProcessing object is created with the information received and the PMLOperation collection is converted into a collection of Task objects ready to be used by the GA. In the next step, the SchedulingInterface starts the generation of the PMLSchedule. The GeneticProcessing class creates a new instance of the Scheduling class and invokes the method to generate a new schedule. After a solution is reached, it is sent back to the GeneticProcessing class and is converted to a PMLSchedule by each station, with all the times (start and end times) set. Lastly, the PMLSchedule collection is sent to the SchedulingInterface where it can be sent to the middleware.

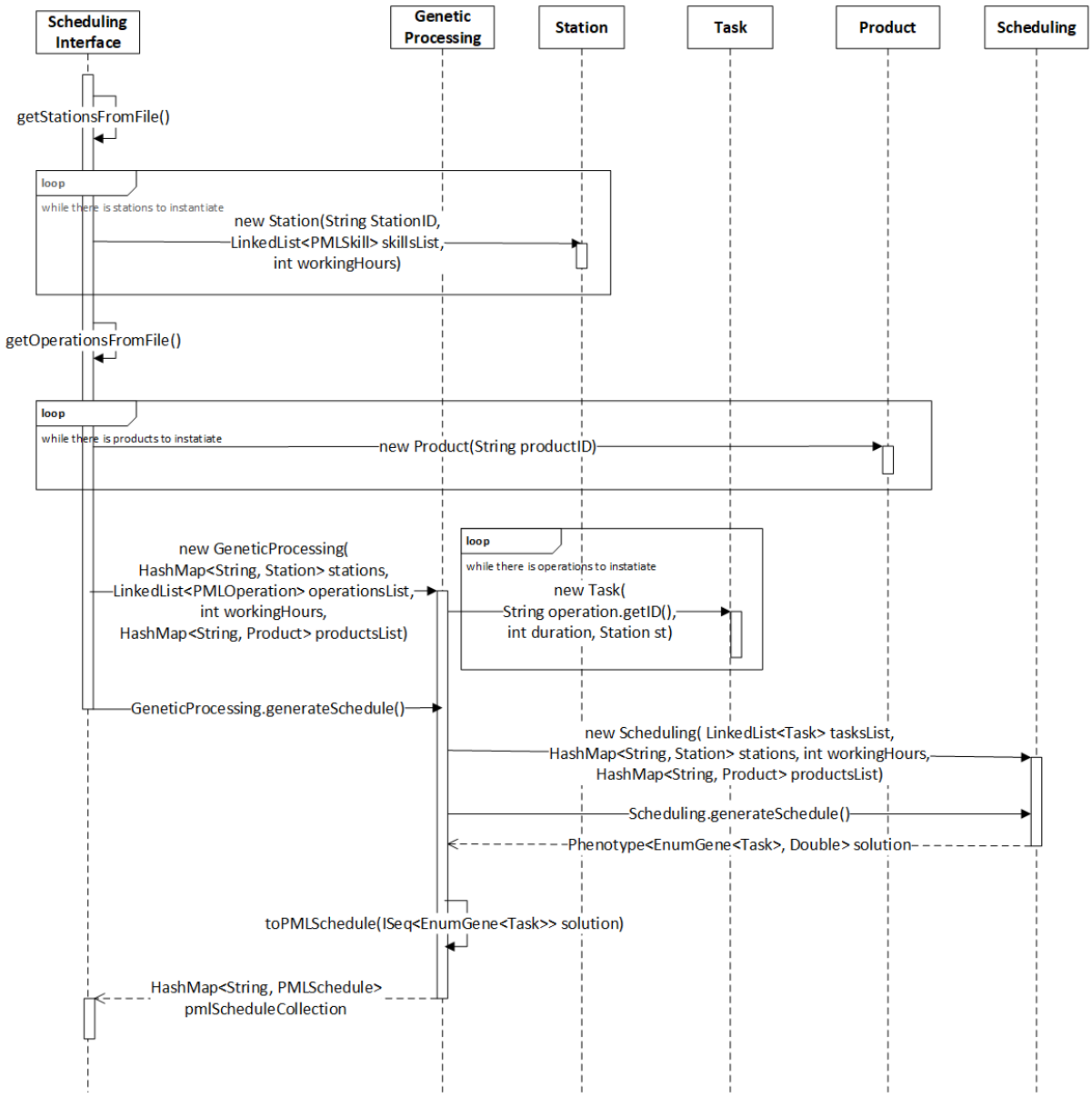


Figure 4-3 - Scheduling sequential diagram

4.2.1. SchedulingInterface

SchedulingInterface class implements the interface where it is possible to generate new schedules and add maintenance tasks manually to the list of tasks to allocate.

When initialized it gets the information of the stations available in the shop-floor and then the operations to be performed by those stations. This information is obtained from files stored in the middleware and are, posteriorly, converted to the respective classes.

The stations are converted to a Station object and stored in an *Hashmap* collection identified by its ID. On the other hand, the operations are stored in a list of PMLOperation and each time an

operation of a non-existing product arrives that product is created and stored in an *Hashmap* to, later, be possible to identify the precedence of operations in each product.

4.2.2. Station

This class is used to instantiate each station available in the shop-floor. Thus, it is possible to access any of them. Each instance is characterized by a unique identifier, a collection of the skills which the station can perform, a start and an end time, a collection of operations to be performed by that station and the total time the station is planned to be operating.

4.2.3. Task

The Task class is the generic representation of the operations to be executed in the shop-floor. Converting each PMLOperation into a Task object makes it more generic, once it can be adapted to a different situation, and easier to handle the GA once the order of the operation (*operationNumber*) and the station associated are available in this class, but not in the PMLOperation class. Otherwise, would be harder to access them each time a new task needs to be allocated. Similarly, the duration of the operation is not available in PMLOperation class, it is needed to access the *associatedSkill* parameter and then the configurations collection to access the D*uration* field. The start and end times of the operation are represented as an *int* which makes it easier to process the times once the duration is an *int* too. Each task contains an identifier and the associated product identifier that let it know to which product it belongs to.

4.2.4. Product

The Product class is the representation of the existing products in the shop-floor ready to be performed. Each one has an identifier and contains a collection of the operations needed to be executed. There is also a *Boolean* field to indicate if all the operations are done or not and, consequently, know if a product can already be scheduled, since some products depend on other to be executed. However, the precedence of products was not implemented in this work.

4.2.5. Station

The Station class provides an abstraction of the working stations available in the shop-floor. It contains all the necessary information about the stations to perform the scheduling, including a collection of the associated skills it can execute, the total working time since the first operation starts till the last one ends, a collection containing all the operations which will be performed in the station and the start and end times of each station.

4.2.6. GeneticProcessing

This class basically treats the data before and after performing the schedule. It converts the collection of PMLOperations into an *HashMap* of Tasks, this way it is easy to access each one of them to alter the start and end times. The *generateSchedule* method instantiates the Scheduling class to obtain a scheduling solution and returns a collection of PMLSchedules. The *toPMLSchedule* method converts the solution obtained in the Scheduling class to a collection of PMLSchedule, assigning the times to each PMLOperation, ready to be sent to the middleware. Finally, the *tasksStartTime* method translates the times of each operation coming from the Scheduling class into Date type values.

4.2.7. Scheduling

The Scheduling class implements the GA. It stores the collections of operations, products and stations needed to execute the algorithm. The *generateSchedule* method sets the parameters of the GA and starts the algorithm. The *eval* method is used as the fitness function and evaluates the algorithm. For that, *tasksStartTime* method assigns the start and end times to the operations, accordingly to the sequence of operations received and the *checkOperationPrecedence* method verifies if the precedence between operations is respected. There is a variable, *weight*, which is used to increase the value of the fitness function if some task is in a wrong order.

Next is defined the implementation of the data acquisition and processing, necessary to then execute the GA.

4.3. Data Acquisition

The data acquisition is done through the middleware and the GUI, as demonstrated in Figure 4-4. Each time a new schedule is requested, the algorithm gets the following data, necessary to perform the task allocation:

- Production operations collection;
- Maintenance operations collection;
- Stations available in the shop-floor.

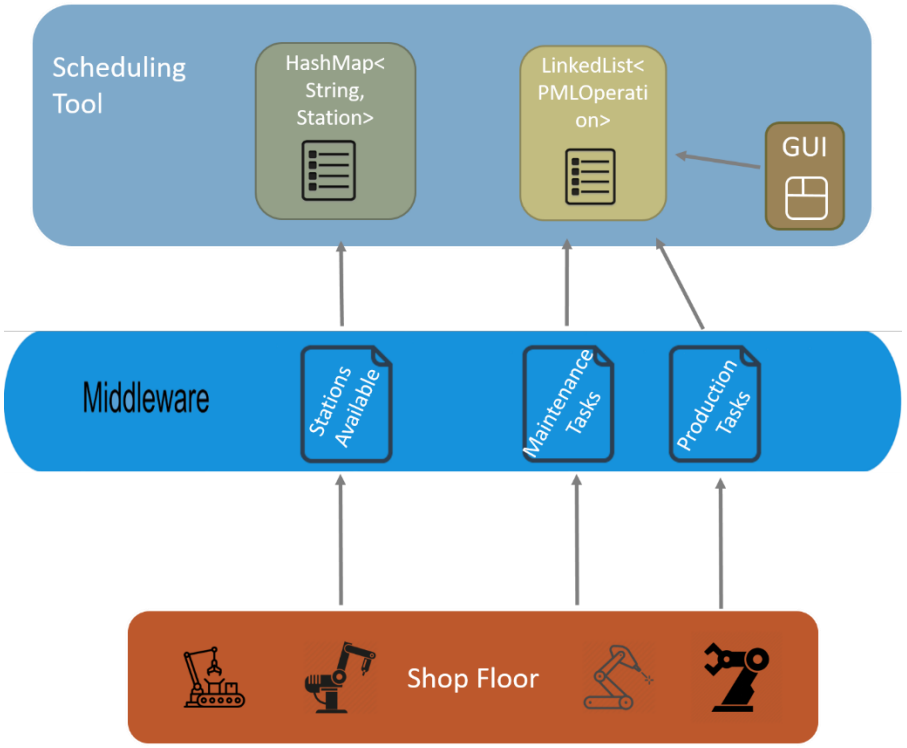


Figure 4-4 - Data acquisition

Both production and maintenance operations are loaded from the middleware and stored in a common *LinkedList* of operations. Each new entry of the list is stored on the form of a *PMLOperation*, as demonstrated in Figure 4-5. Also, the maintenance tasks could be inserted from the GUI interface. In this case, they are stored in a *LinkedList* with the same format, where they can be removed, until a new schedule is requested. At this point, the maintenance tasks are added to the common list of operations.

Each operation present in the file is converted to a PMLOperation object and contains necessarily an operation ID, an associated skill, with the respective ID and configuration list and the operation type, production or maintenance. Each configuration should have the associated entity ID and the duration of that operation.

In the production operations case, it should, additionally, contain the operation number and the associated product ID.

Also, it is necessary to get the information about all the stations in the shop-floor available to execute operations. That information is stored in an *HashMap*. The key of each entry in the *HashMap* is the station ID and each station is stored in the *HashMap* over the format of a *Station* object. So, the information is loaded from the middleware and for each different station present on it a new *Station* object is instantiated, containing the ID and the skills of each station. Each

skill is a PMLSkill instance and contains an ID, the corresponding station ID, task's duration, and a PMLConfiguration collection, since each skill could have different configurations, depending on which station it is executed.

Additionaly, was implemented an *HashMap* to store the different products whose operations where loaded from the middleware. The key of each entry is the product ID and each product is stored in the format of a *Product* object. It contains an ID and the number of operations to be executed in that product. In this way, is possible to check the precedence between products. All this process is represented in Figure 4-5.

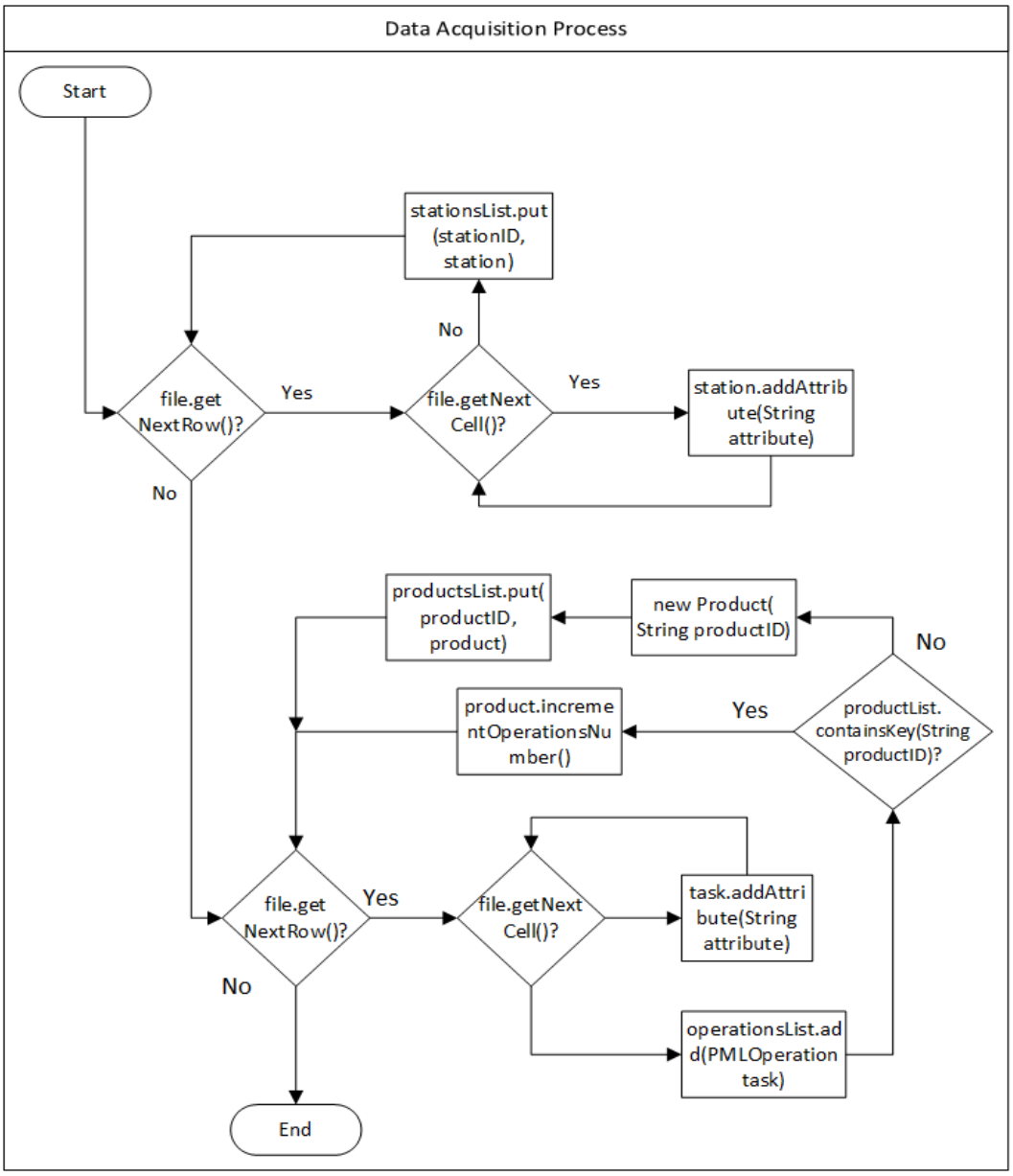


Figure 4-5 - Data acquisition flowchart

It was assumed that the maintenance teams are always available and the maintenance shift is done between 6am and 2pm. It was also assumed that the factory is always operating and there are no holiday times.

The GA can only be executed after all the previous data is known and stored. The GA's implementation is described below.

4.4. Genetic Algorithm

In order to implement the genetic processing, the previous information was used together with the *Jenetics* library.

To easily access the tasks stored by the algorithm, the list of tasks was converted into an *HashMap*. The key of each entry is the ID of the operation, which is unique and each one is stored over the format of a *Task* object. This allows to access the operation number and duration directly, unlike on the list of PLMOperation.

Also, it was created a PMLSchedule variable, where the generated schedule (for all stations yet) is stored. To generate a new schedule, the method *generateSchedule* of the *Scheduling* class was called and this is where the *Jenetics* library comes in.

In Figure 4-6 are presented the steps performed by Jenetics to execute the GA. First, the initial population is created. After that, is calculated the fitness value of each individual in the population. Then, the generation number is increased and in lines 5 and 6, respectively, the survivor and the offspring population are selected, through the *offspringFraction* property of the *Engine.builder*. In line 7 the offspring population is altered. Finally, the survivor and the altered offspring populations are combined into a new population. The fitness value of each individual of the new population is calculated again, in line 9. While the termination criteria is not reached, steps from line 4 to 9 are repeated (Wilhelmstotter, 2016).

```
1  P_0 ← P_initial
2  F(P_0)
3  while !finished do
4      g ← g + 1
5      S_g ← select_S(P_{g-1})
6      O_g ← select_O(P_{g-1})
7      O_g ← alter(O_g)
8      P_g ← filter[g_i ≥ g_max](S_g) + filter[g_i ≥ g_max](O_g)
9      F(P_g)
```

Figure 4-6 - Pseudo-code of the Jenetics genetic algorithm steps. Image taken from (Wilhelmstotter, 2016)

The process describing the instantiation of the GA using the Jenetics library is described next and presented in Figure 4-7.

It was created an *ISeq* sequence from the values of the *HashMap* of tasks to allocate, containing objects of *Task* type. From this sequence was created a permuted chromosome using the method *of* of the *PermutationChromosome* class. Then, a genotype *Factory* was instantiated containing objects of type *Genotype<EnumGene<Task>>*, using the method *of* of the *Genotype* class, which allows to create new individuals during the evolution process. This process is represented in the first steps of Figure 4-7.

To evaluate the individuals, a method, called *eval*, was created, which represents the fitness function. The main goal of this method is to attribute the start and end times to each task and get the fitness value of each generation. Once the values of each station are changed in each generation, it was necessary to create a loop to reset the values of each instance of the Station class present in the *HashMap* containing all the stations. Then, the same was did to the products *HashMap*.

Then, was created the evolution engine, in step 5 of the Figure 4-7, responsible for evolving a given population, through the *builder* method of the *Engine* class. The fitness function and the genotype factory were passed as parameters to the *builder* method, which allows to evaluate each individual present in the population. Several methods of the *Engine.Builder* class were called in order to set the needed parameters to perform the GA.

- *populationSize*: the number of individuals which form the population;
- *offspringFraction*: the offspring population fraction;
- *survivorsSelector*: the selector used for selecting the survivor population;
- *offspringSelector*: the selector used for selecting the offspring population;
- *optimize*: the optimization strategy used by the engine;
- *alterers*: the alterers used for alter the offspring population. New instances of *PartiallyMatchedCrossover* and *SwapMutator* (from *Jenetics* library) were created. It changes the order of genes in the chromosome, where no duplicated genes within the chromosome are allowed.;
- *build*: builds a new *Engine* instance from the set properties.

All the previous parameters are adaptable, depending on the problems' size and complexity, except *optimize*, which was set to minimum, making it possible to obtain the minimum value from the fitness function.

After that, the phenotype was generated, via the *stream* method of the *Engine* class. This created a new infinite evolution stream with a newly created population. To prevent the process to evolve infinitely, a limit needed to be set. The *limit* method from the *EvolutionStream* class, was set twice. One of them, indicating the maximum generations allowed by the algorithm. The other one truncating it by steady fitness, i.e. if truncating the evolution stream if no better phenotype is found after a given number of generations. It was done using the *bySteadyFitness* method of the *limit* class. Finally, after the process evaluate all the individuals, the evolution result was collected from the evolution stream, using the *collect* method, which is inherited from interface *java.util.stream.Stream*.

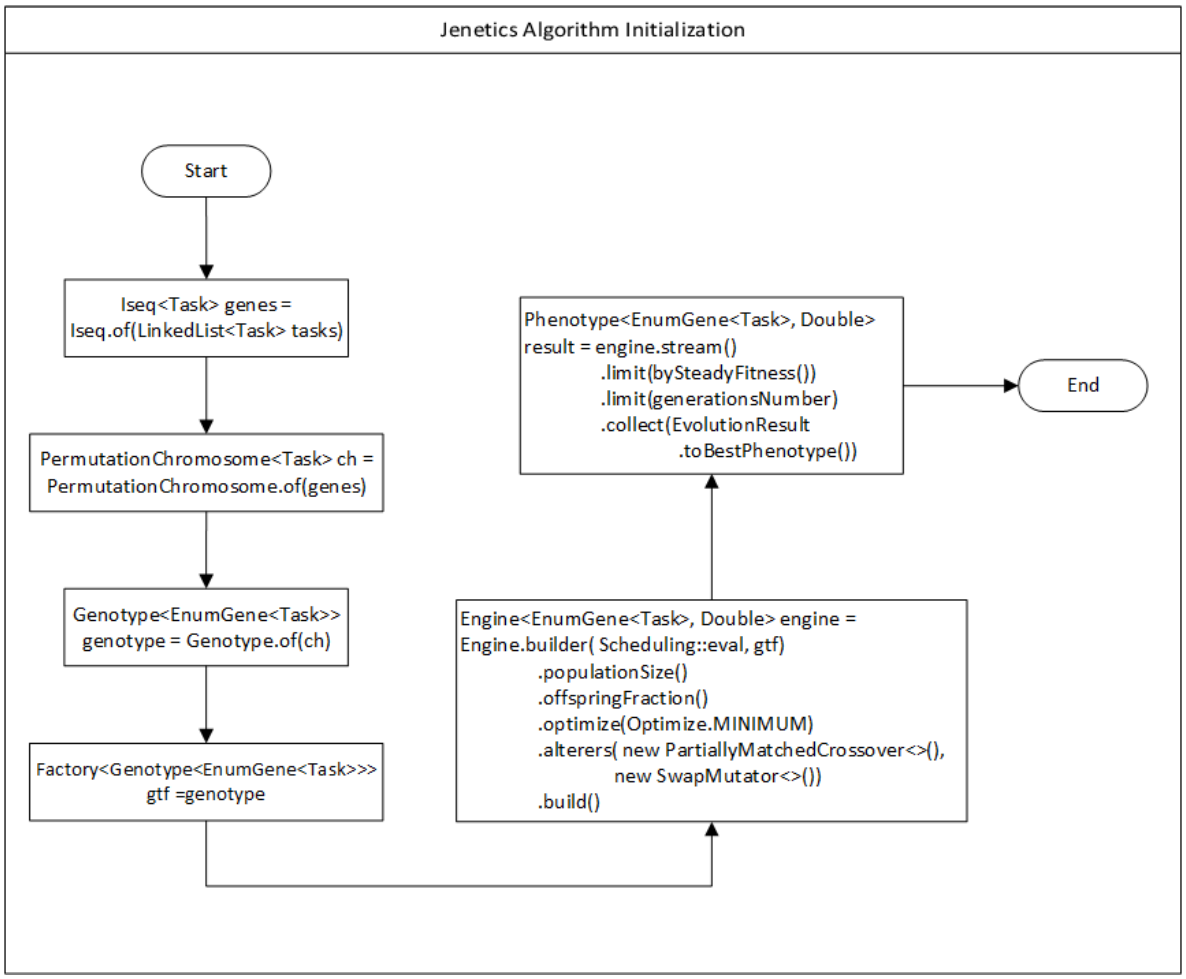


Figure 4-7 - Flowchart representing the process of Jenetics library

The fitness function is represented in Figure 4-8. To implement it was created a loop going through all the chromosomes present in the genotype. For each chromosome, another loop was created, going through all the genes present in that chromosome. Then, an instance of the *Task* object was obtained from each gene and the task's duration, the associated station ID and the operation's type were stored in variables. If the task is a production operation, so another variable

is created to store the associated product ID and the tasks' precedence of that product is verified. If it is a maintenance operation, there is no product ID associated. In the case that the station ID is not present in the stations *HashMap,* the current iteration of the loop is ignored and moves on to the next.

At that time, the times of the task were defined. How it was explained in the architecture chapter, in the production operations case, the start time was calculated based on the previous operation of the corresponding product if there was any, otherwise the start time is zero. The end time was obtained by add the task's duration to the start time. However, if there was already another operation allocated to that station at the same time, causing a conflict, a new time is set to the current task, starting at the end of the other, with which there was a conflict.

In the maintenance tasks case, the start time depends on the user intent. It could be "as soon as possible" or "as late as possible", defined in equations (6) and (7) of the section 3.4.6, respectively. To check the conflicts in the case of maintenance tasks, the principle is the same, however it is necessary to check if the maintenance is performed during the maintenance shift. So, if it is not, it should be allocated to the next day, in the case of the "as soon as possible" scenario, or the previous day, in the case of the "as late as possible" scenario, using the *day* variable present in the *Task* class.

After getting the times of the task, the makespan of the associated station is updated, if the task end time is greater than the current station's makespan. Finally, that task is added to the station's list. And finally, that list is sorted from the lower start time to the highest, thus the maintenance tasks could be inserted without having to go through all the tasks.

Here, if the task was a production task and the corresponding product ID exists in the products *HashMap*, that task is added to the end of operations' list of that product and the precedence between operation is checked. So, for each other operation which the operation number is not lower than the current operation number, the *weight* variable is multiplied by 10. This way, the fitness value will be much higher in the case where the precedence between operations was not respected.

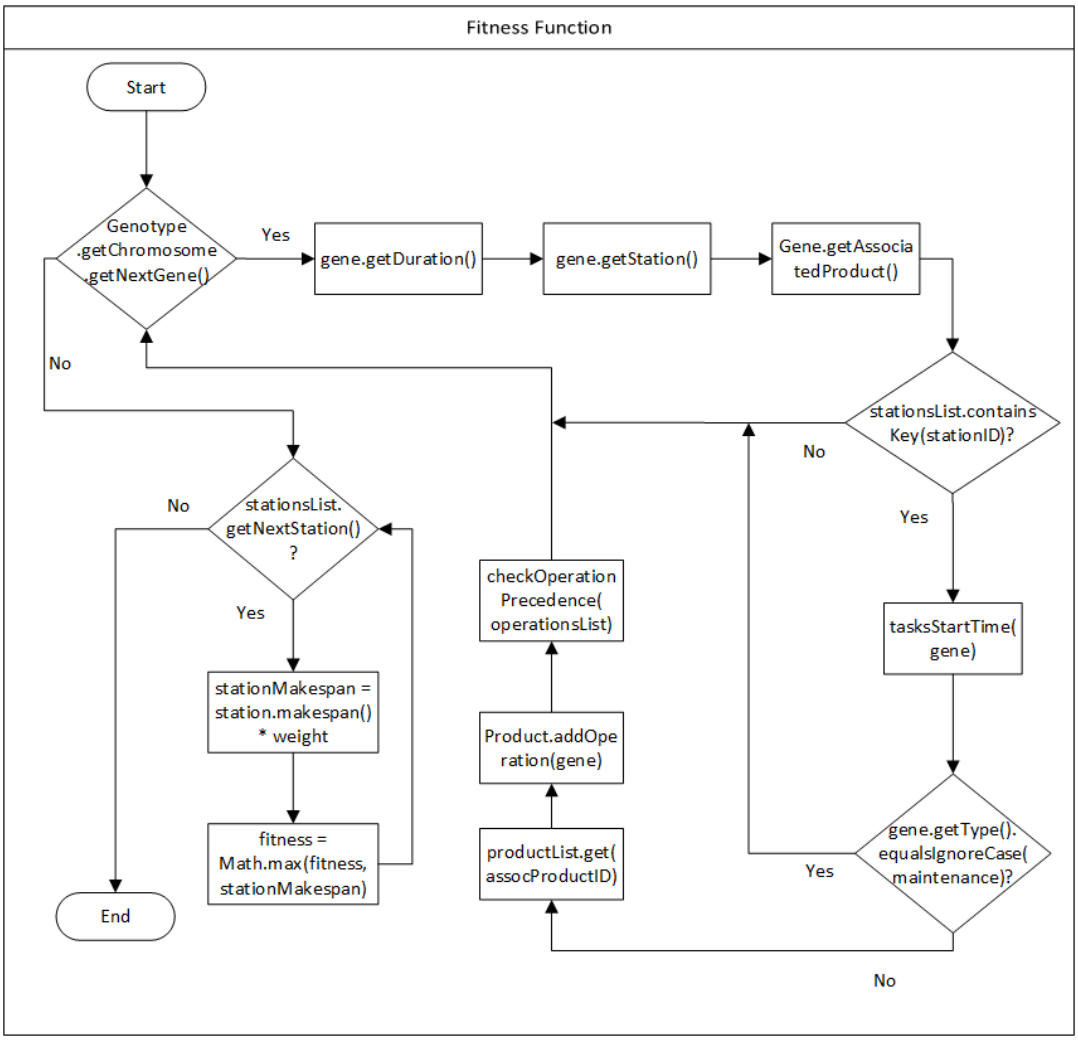


Figure 4-8 - Flowchart of the implemented fitness function

To complete the fitness function, it was necessary to get the minimum value of the chromosomes present in the genotype. Hence, it was necessary to create a loop to iterate over the stations' collection to find the station with the higher makespan value, which represents the total amount of time the set of tasks of that chromosome will need to be executed. After get the value of all chromosomes from the genotype, the minimum value, representing the best solution found in that generation, can be obtained.

Here, it is important to note that if the value of the *weight* variable is the same as the beginning, so the solution found is a valid solution, representing a feasible schedule to all the operations. Otherwise it is an invalid solution.

Over the generations, the *Jenetics* library saves the solution with the minimum value found till the moment, once the *optimize* method from the *Engine.Builder* class was set to minimum.

The way which the collection of schedules, provided by the solution found in the GA, is sent to the middleware, is described in the next section.

4.5. Send Generated Schedules

Once the valid solution may not have been the last solution found and the current values present on *Station*, *Task* and *Product* classes may be from other solution, it was necessary to create a method which replicates the process described above to obtain the start and end times for each operation, but only for the best solution.

Then, in order to be possible to send the collection of schedules to the middleware, that solution was converted to a *PMLSchedule*. It was created an *HashMap* to store a schedule of each station. The key of each entry is the station ID and the information is stored in the format of a *PMLSchedule*. It was also created an *HashMap* to store the set of operations which each station will perform, where the key of each entry is the station ID and it stores information in the format of a *LinkedList<PMLOperation>*. Then, for each station was created a new instance of the *PMLSchedule* object and each one was stored in the first *HashMap*. After that, each operation was added to the list of operations of the respective schedule (station). At that point, for each *PMLSchedule* the associated operations were added, using the *setAssociatedOperations* method from the *PMSchedule* class, passing as argument the value of the second *HashMap*.

At this point, the collection of *PMLSchedules* for each station was obtained. Being thus possible to manage it as intended. So, to send the collection to the middleware, it was used the method *writeValueAsString* from the *fasterxml.jackson.databind.ObjectMapper* class, which allows to send any object as a string.

5

Tests and Results

This chapter describes the tests performed to validate the proposed implementation of the scheduling algorithm.

The tests were executed on a computer running Microsoft Windows 7 Professional operating system with a 64 bits' architecture, with Java version 8, an Intel Core i7-4770 processor at 3.40GHz and an 12GB RAM drive.

All the information about operations to perform the schedule algorithm was loaded from a .xlsx file, as well as the stations available and correspondent skills which each one can execute. Those files were used to validate the proposed algorithm, trying to mimic the middleware function, yet they were stored locally. They contained all the information necessary about both operations and stations.

The data files were provided by a PERFoRM project partner and contain real information about production tasks to execute and the stations used to perform those tasks. It includes all the necessary information, such as task's ID, duration, associated product and associated station. As well as station's ID and associated skills.

A GUI was created to test and validate the developed algorithm, yet it will not be implemented in the PERFoRM project. In this interface is possible to insert new tasks to the list of tasks to execute, indicating its ID, type, duration and the station where it will be executed. It is also possible to generate a new schedule based on the available tasks and stations, as presented in Figure 5-1.

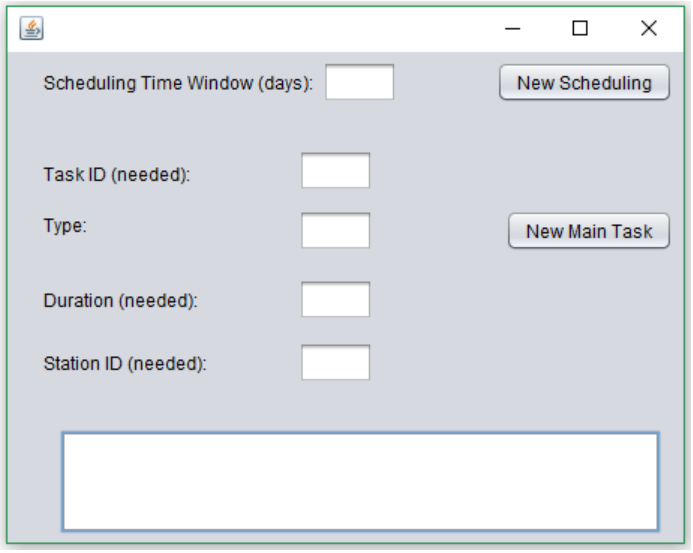


Figure 5-1 - Graphical user interface

In order to reach the parameters presented in this section, exhaustive tests in the 3x3 and 5x5 problems were performed, since in this case it was faster to get results. A 3x3 problem means that there are three different products containing three operations each, and so on. In this case, there can be till three different stations operating.

The different problems used were:

- 3x3 problem;
- 5x5 problem;
- 7x7 problem;
- 9x9 problem.

Taking into account the 3x3 scenario, it was loaded a file with three different products, each one with three different operations to perform on three different stations. Completing a total of 9 different operations to perform over three different stations. For the 5x5 scenario, was loaded a file with 5 different products, each one with 5 different operations to perform on 5 different stations and so on. To the other cases, the parameters were chosen taking into account the best values of the previous cases. All the tasks used had execution times between 1 and 720 minutes, i.e. the shortest task had an execution duration of 1 minute and the longest one of 720 minutes.

The file containing the stations' information was the same for all scenarios, containing data about 12 stations, so the number of stations available and use in the schedule doesn't compromises the algorithm performance.

The tests were performed in a set of non-identical stations, where each one of them perform a different type of operation.

So, initially, high values to the number of generations and population were chosen to ensure that a solution could be found no matter the processing time. Then, different values of crossover probability, mutation probability, selectors and elitism were set. Always considering that small values of crossover and mutation probabilities could not lead the process to evolve and high values could not reach good solutions since the individuals can be damaged if it is too changed. Thus, the values of crossover were tested between 10% and 100% and the range for mutation was between 1% and 30%.

The parameters used may not be optimal but perform efficiently for the presented cases. Those parameters were reached after performing exhaustive tests in the 3x3 and 5x5 problems. To the other cases, the parameters were chosen based on the best of the previous problems' parameters. Several tests were performed where the crossover and mutation probability were generated manually or randomly within a certain range and the parameters of the best results were taken.

Here is presented a brief explanation and results on how this was performed for the 5x5 problem, since 3x3 problem is too small and the difference of results is mainly observable when parameters change too much.

The crossover probability values were tested within the range between 10% and 100% and mutation probability values between 1% and 30%. However, not all the cases are demonstrated in the following tests, only the necessary to verify the difference when an attribute is changed.

5.1. Crossover and Mutation

As mentioned before, the following tests were performed in a 5x5 problem, which contains five different products with five operations each and five stations are available to perform those operations. So, a total of 25 tasks were allocated. The values listed below were fixed and do not changed during the following tests:

- Number of generations: 60;
- Population size: 200;
- Steady fitness: 20;

- Elitism: 4%;
- Survivor selector: Tournament selector with 3 individuals;
- Offspring selector: Tournament selector with 3 individuals.

The number of generations represents the maximum iterations the GA will perform before finishing the process. The population size parameter defines the number of individuals present in each generation of the GA. The steady fitness parameter limits the evolution of the GA, i.e. the number of generations that the GA should perform without improvements in the evolution process. The elitism reflects the percentage of the best individuals of the previous generation which will be present in the next generation. Finally, the selectors are used to define how survivor and offspring populations will be selected for the next generation, in this case is used a Tournament Selector, where the strongest of the *n* individuals is chosen.

For each value of the crossover probability, different values of mutation probability are tested and the results are presented in the following tables.

- Best Fitness (%): number of times which the best fitness found was reach, among all the cases
- Best Fitness: the value of the best fitness found
- Worst Fitness: the value of the best fitness found
- Average Generations: the number of generations needed to found the best solution
- Successful Cases: quantity of valid solutions found

Crossover 90%

In Table 5-1 it is possible to observe that 90% crossover probability performs well for lower mutation probability values, once the successful cases decrease when mutation probability is increased.

Table 5-1 – Results with crossover probability at 90%

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	75%	1976	2277	41,5	100%
0,08	60%	1976	2566000	46,15	85%
0,11	35%	1976	18300000	42,3	55%
0,14	5%	1976	20470000000	39,05	15%

Crossover 80%

Results of Table 5-2 show that the number of successful solutions increased compared with the 90% probability. Also, the number of generations to obtain a solution decreased. Thus, it means that a crossover probability of 80% should perform better than 90% most of the times.

Table 5-2 – Results with crossover probability at 80%

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	50%	1976	2317	31,9	100%
0,08	65%	1976	2294	37,2	100%
0,11	40%	1976	26050000	37,55	85%
0,14	20%	1976	180900000	38,25	65%

Crossover 70%

With a crossover probability of 70%, Table 5-3, were obtained more successful cases than any other and the best fitness reached was found around half of the times.

Table 5-3 – Results with crossover probability at 70%

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	55%	1976	2136	31,95	100%
0,08	45%	1976	179500	33,3	95%
0,11	40%	1976	2277	36,5	100%
0,14	60%	1976	179500	41,45	95%

Crossover 60%

However, with a crossover of 60%, Table 5-4, the results are identical but it took less generation to found a solution, despite the best fitness not be found so many times.

Table 5-4 – Results with crossover probability at 60%

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	45%	1976	2294	26,3	100%
0,08	45%	1976	213600	28,1	95%
0,11	40%	1976	2367	32,85	100%
0,14	40%	1976	222400	29,9	80%

Crossover 50%

For a crossover of 50%, present in Table 5-5, it performs better than 90% and is similar to 80%, though, despite it found the best fitness less times, it took less generations to do so. Comparatively to 60% and 70% values, it found slightly less successful cases and the best fitness is reached less times, which keeps it a little below from the others.

Table 5-5 – Results with crossover probability at 50%

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	30%	1976	230900	21,75	90%
0,08	40%	1976	2294000	24,55	90%
0,11	15%	1976	1795000	28,9	80%
0,14	50%	1976	2525	30,1	100%

Maintenance Tasks

Since the crossover probability value of 70% presented more successful cases and found more times the best solution than the other values, two maintenance tasks of 100 minutes were added on two different stations, using a crossover value of 70%. For each day of work there is only one maintenance shift where is possible to allocate the maintenance tasks. Which means that for each 1440 minutes (one day), 480 minutes are available to allocate maintenance tasks. The results are shown in Table 5-6.

Table 5-6 - Results with crossover value of 70% and a total of 27 tasks to allocate, including two tasks with 100 minutes each

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	55%	1976	2136	28,35	100%
0,08	45%	1976	2294	36,65	100%
0,11	60%	1976	2277	39,5	100%
0,14	50%	1976	2136000	39,45	95%

Then, another two maintenance tasks were added to the same stations, with the same duration, performing a total of two maintenance tasks on each station. The results are presented in Table 5-7.

Table 5-7 - Results with crossover value of 70% and a total of 29 tasks to allocate, including four tasks with 100 minutes each

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	60%	1976	2248	32,15	100%
0,08	60%	1976	179500	35,15	95%
0,11	75%	1976	2253	41,65	100%
0,14	30%	1976	2136000	40,3	90%

Then, the execution time of the maintenance tasks used in Table 5-7 were increased from 100 minutes to 250 minutes. The results are shown in Table 5-8.

Table 5-8 - Results with crossover value of 70% and a total of 29 tasks to allocate, including four tasks with 250 minutes each

Mutator	Best Fitness (%)	Best Fitness	Worst Fitness	Average Generations	Successful Cases
0,05	55%	2159	2294	31,25	100%
0,08	50%	2159	2294	35,3	100%
0,11	45%	2159	205000	38,5	95%
0,14	30%	2159	247700	43,45	90%

In the next section, is presented how both population size and the number of generation affect the different types of problems.

5.2. Population and Steady Fitness

In the following tests, the parameters of crossover and mutation were set to 60% and 100%, respectively. The goal is to demonstrate how different population sizes and steady fitness values affect the results in terms of algorithm runtime and successful solutions found.

For that, four different size JSSPs were taken, namely 3x3, 5x5, 7x7 and 9x9, and their performance were compared for the same parameters. The red dots in the runtime charts represent the invalid schedules obtained.

Population of 500 individuals

In a case with a population of 500 individuals and a steady fitness of 100 generations, successful solutions, without conflicts, were always found in the smallest problems, as demonstrated in the first chart of Figure 5-2. However, these parameters had difficulty to find successful solutions in

larger problems, which is possible to verify by the invalid solutions found (red dots). The chart of the execution runtime shows that successful solutions were found relatively quickly.

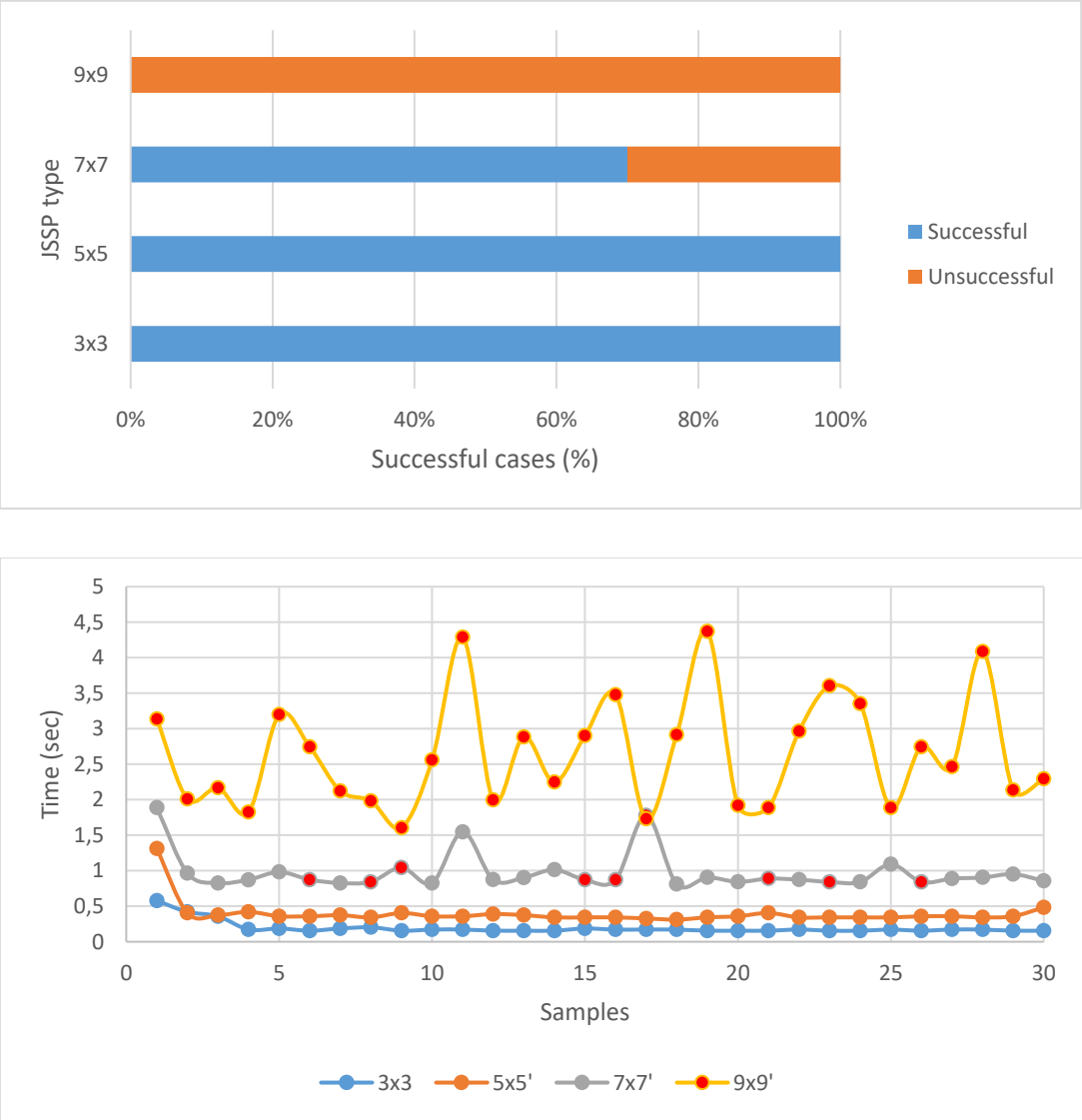


Figure 5-2 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 100 generations

When the steady fitness was incremented to 300 generations, more successful cases were found in the 7x7 and 9x9 problems, as demonstrated in the first chart of Figure 5-3. However, comparing with the previous case, the runtime execution increased.

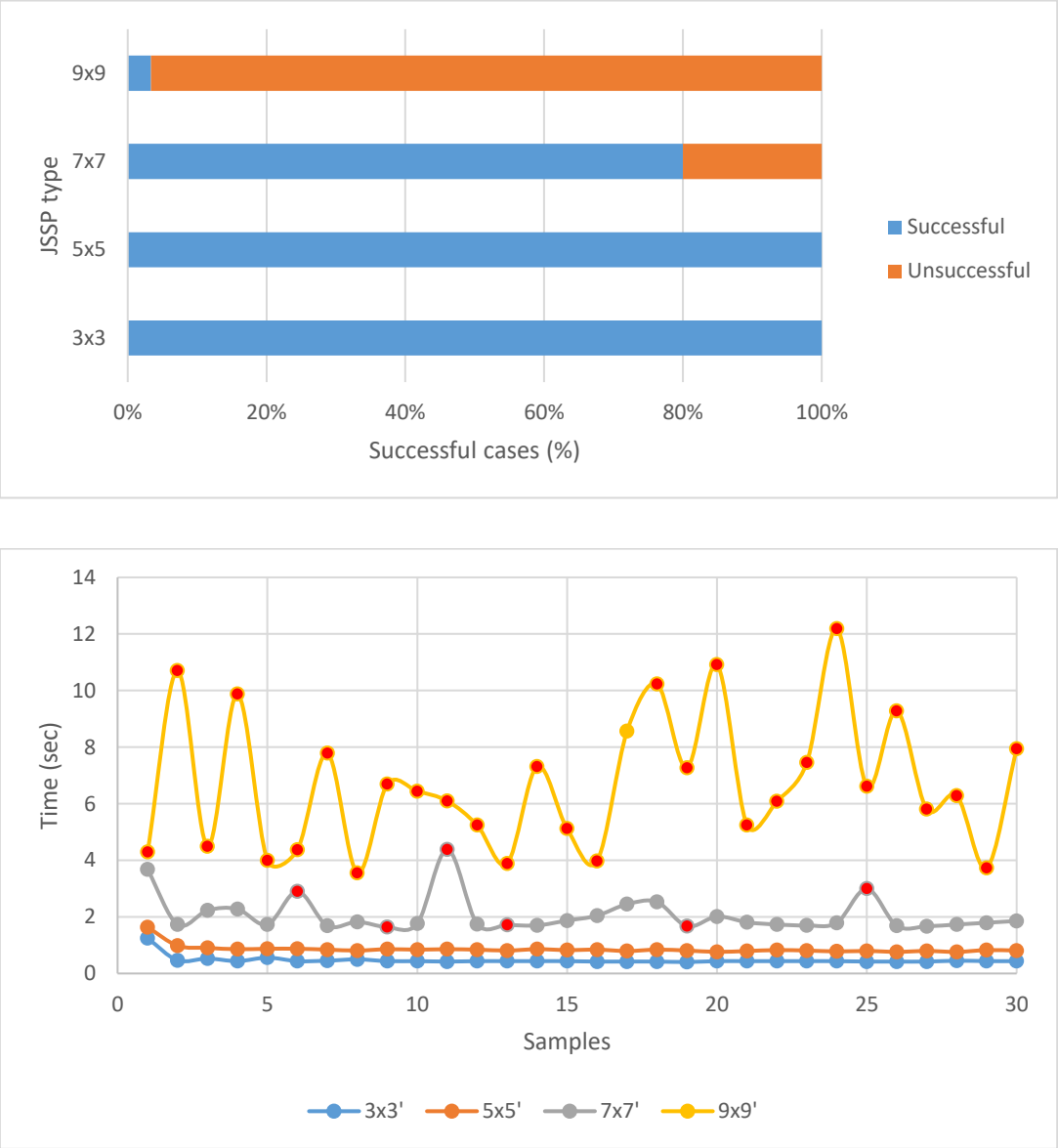


Figure 5-3 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 300 generations

As represented in Figure 5-4, after augmented the steady fitness for 500 generation, more successful solutions were found in the largest problems. However, the runtime increased as in the previous case.

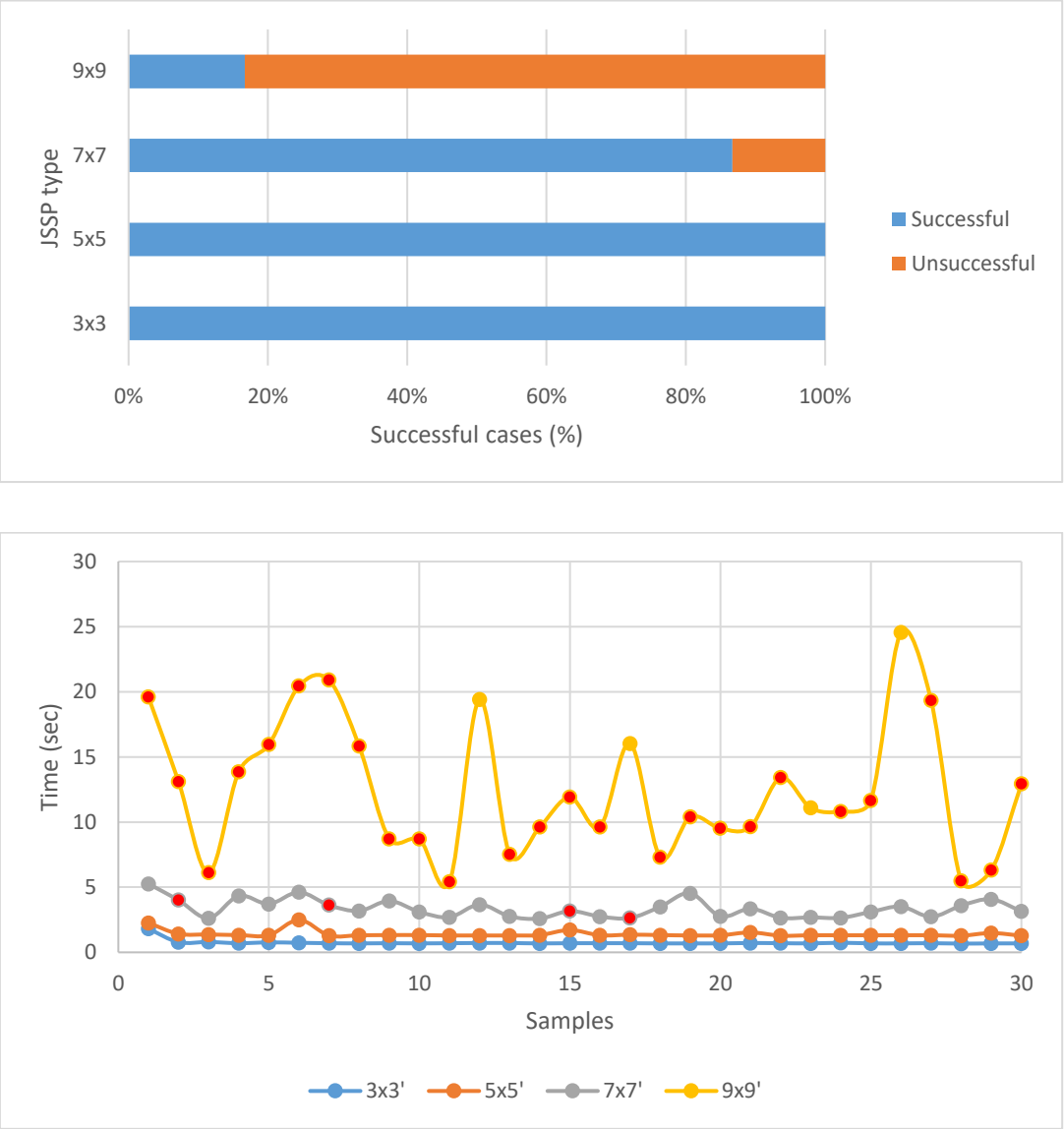


Figure 5-4 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 500 generations

Population of 1000 individuals

Once the number of individuals present in the population was increased to 1000, the number of unsuccessful solutions found decreased (less red dots in the chart), even with a steady fitness of 100 generations, as it is possible to observe in Figure 5-5. Also, the runtime decreased when compared with the case of Figure 5-4.

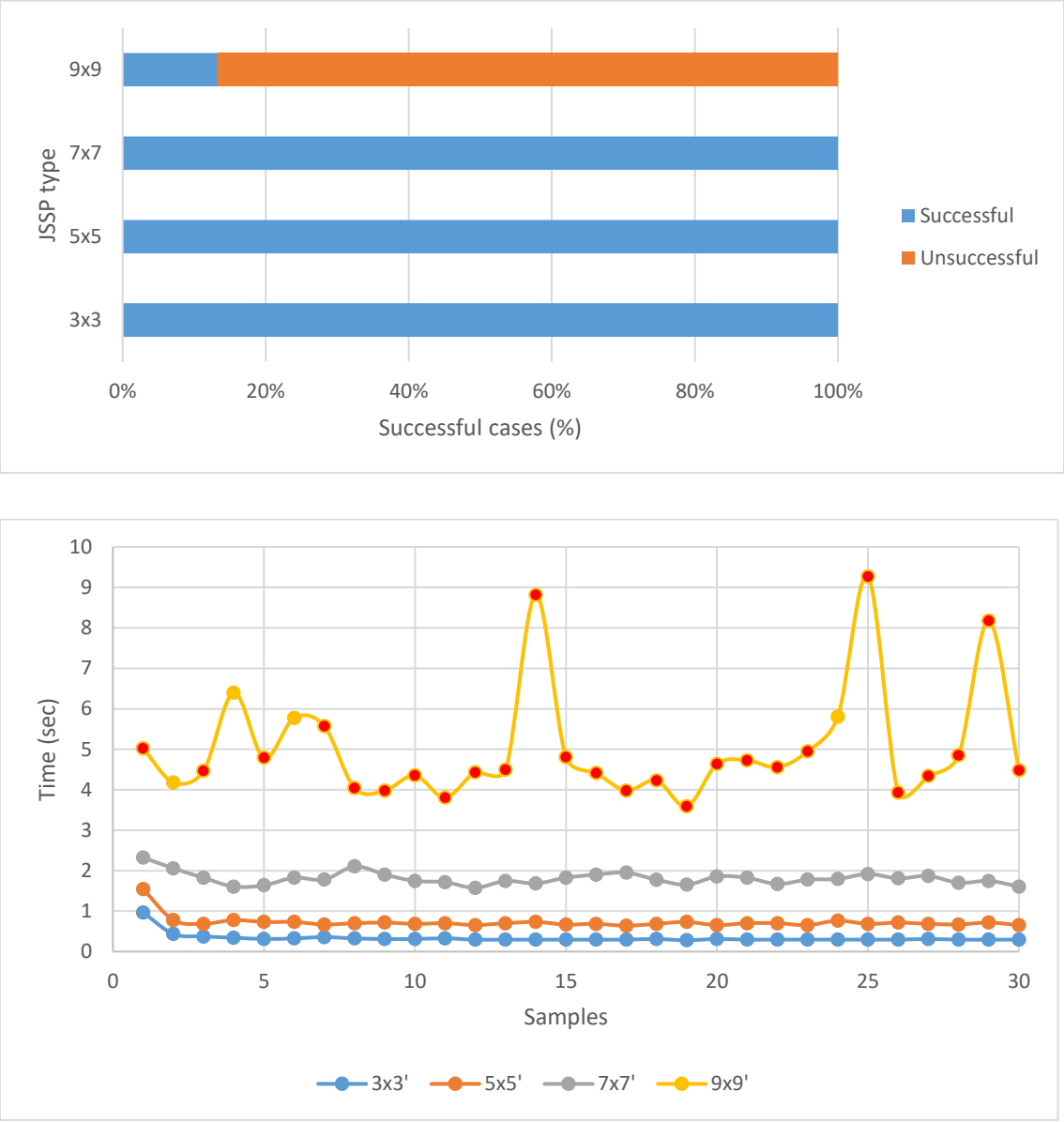
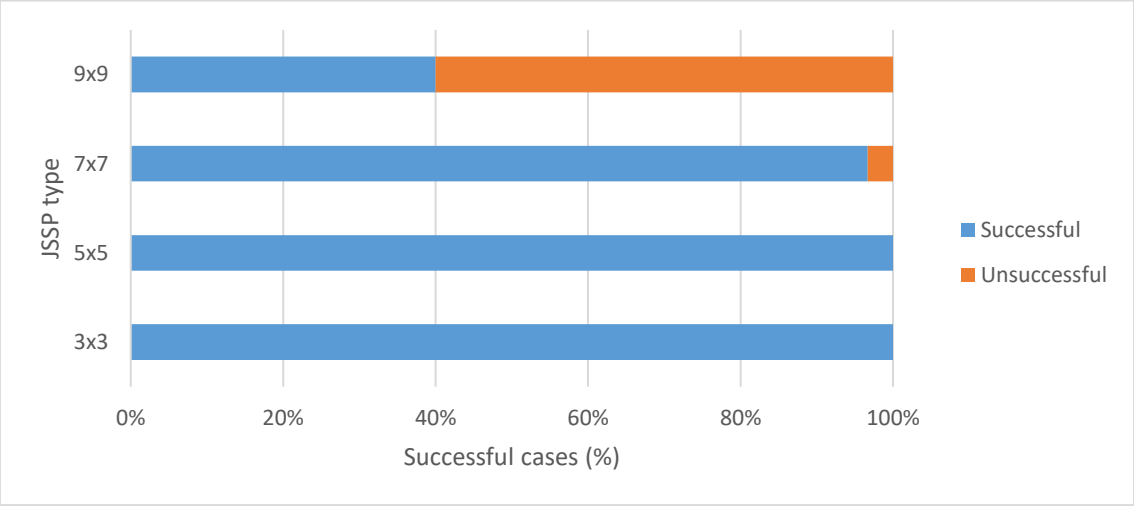


Figure 5-5 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 100 generations

When the number of steady fitness generations was increased to 300 generations, the biggest difference to the previous case was that in the largest size problem, the number of successful cases increased to more than double, as shown in Figure 5-6. Yet, once again, the runtime has increased.



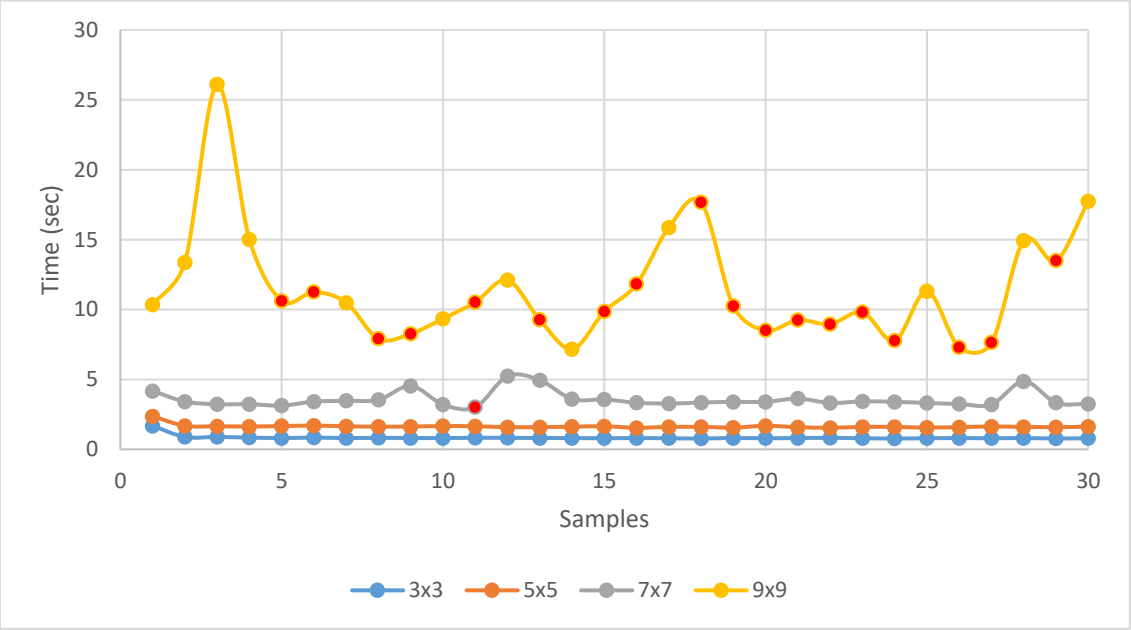
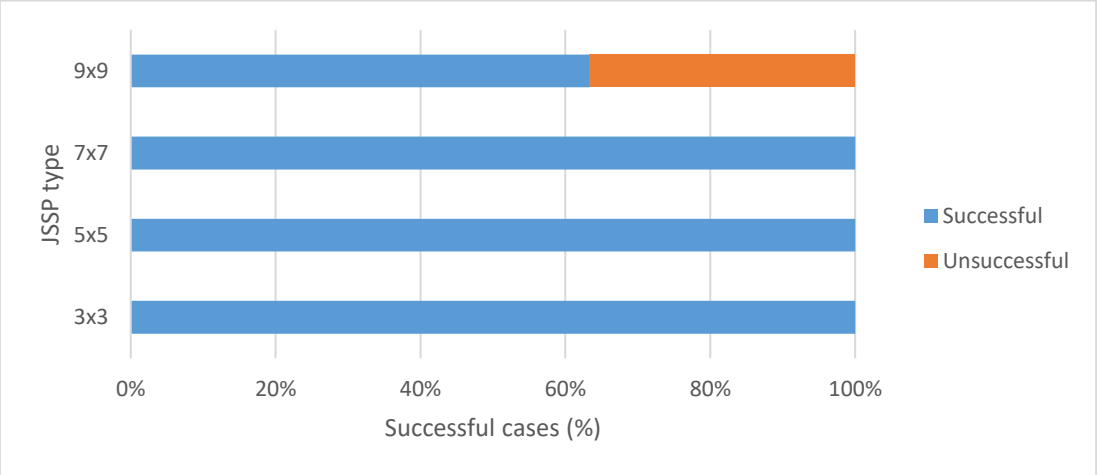


Figure 5-6 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 300 generations

As shown in Figure 5-7, the successful solutions found in the 9x9 problem increased for more than a half, while in the other size problems only successful solutions were found, when the steady fitness was incremented to 500 generations. Again, this has led to an increase in the runtime.



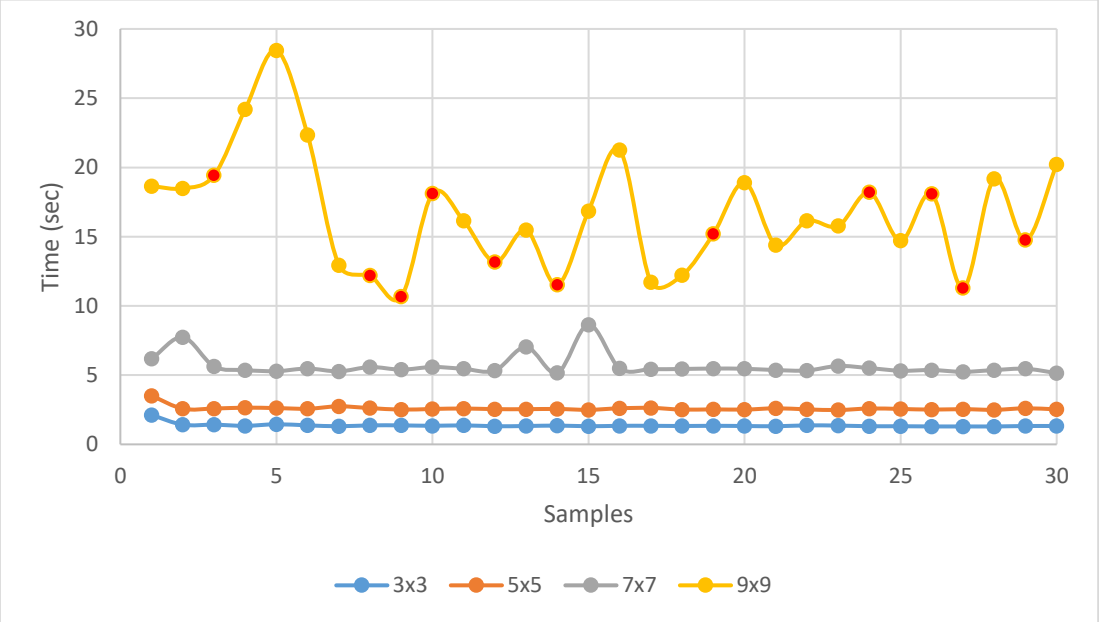
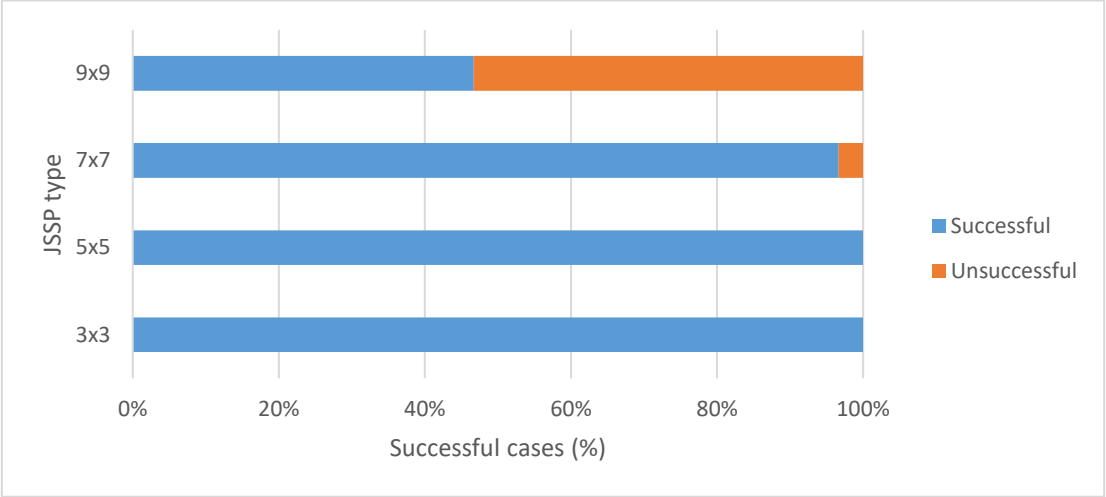


Figure 5-7 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 500 generations

Population of 1500 individuals

When the number of individuals present in the population was increased to 1500, with a steady fitness of 100 generations, as shown in Figure 5-8, the number of successful schedules found for 7x7 and 9x9 problems has decreased compared with the example of Figure 5-7. The runtime also decreased for all the scenarios.



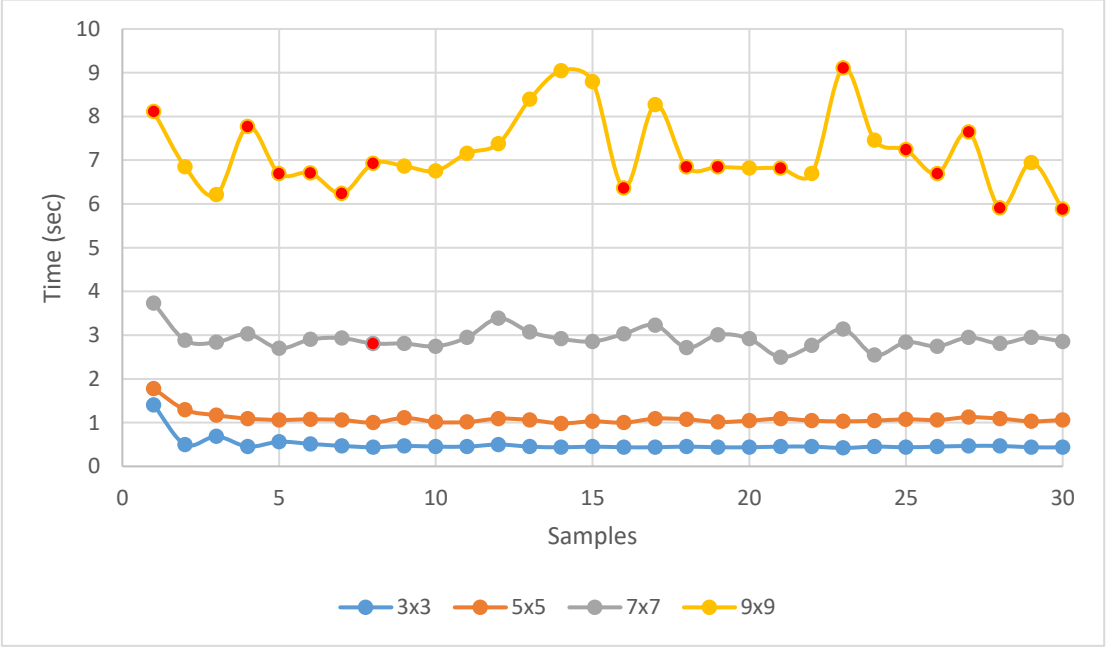


Figure 5-8 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 100 generations

In Figure 5-9, the steady fitness was increased to 300 generations. This has led to an improvement on the number of successful cases found in 9x9 problem, possible to observe by the decrease of the red dots, while in the other problems it remained the same. It also led to an increment of the runtime in all the scenarios.

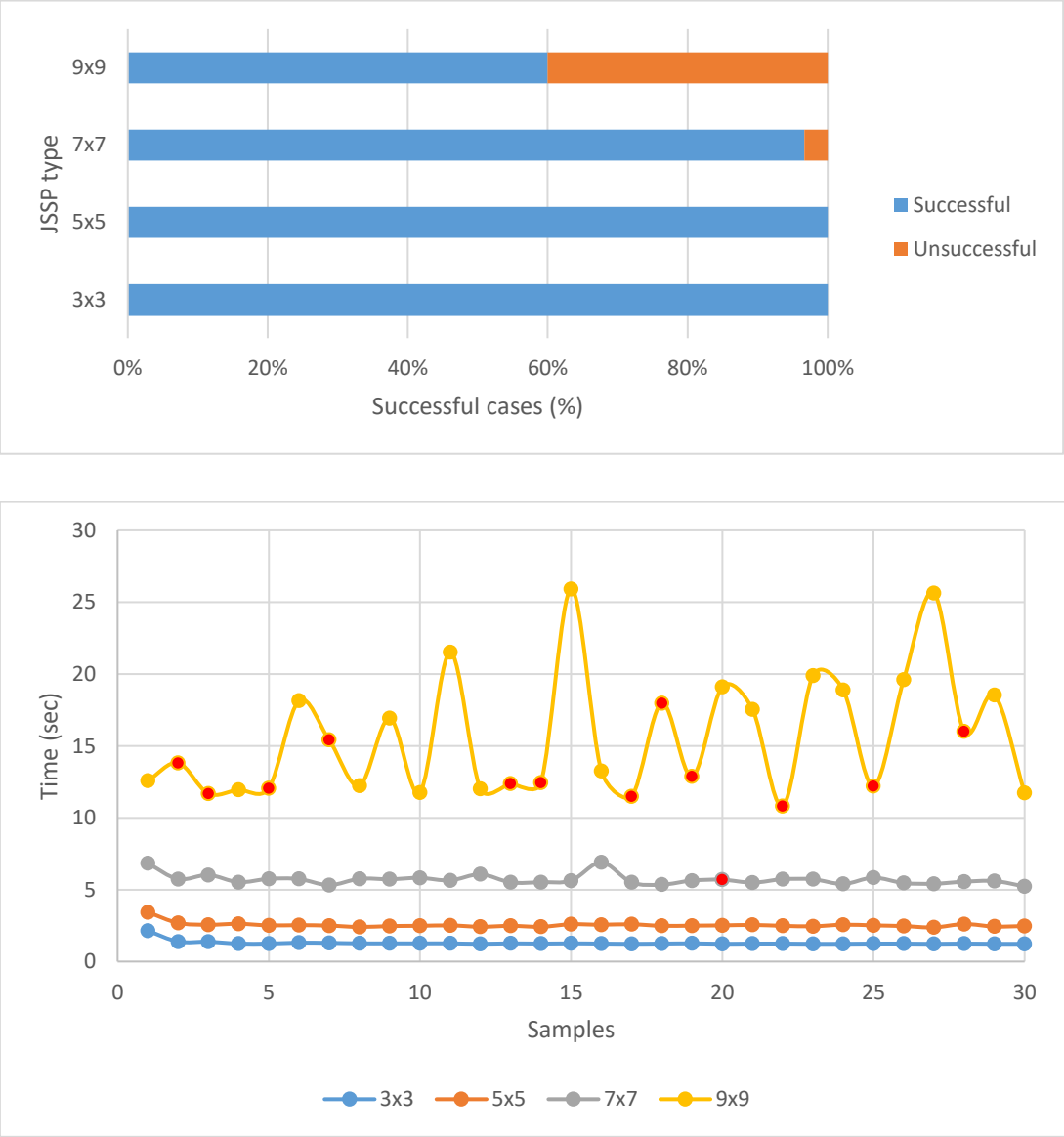


Figure 5-9 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 300 generations

By increased the number of generations of the steady fitness to 500, successful solutions were found in more than 75% of the cases, even in the largest problem, as demonstrated in the first chart of Figure 5-10. Once again, it led to an increase of the runtime, presented in the second chart of Figure 5-10.

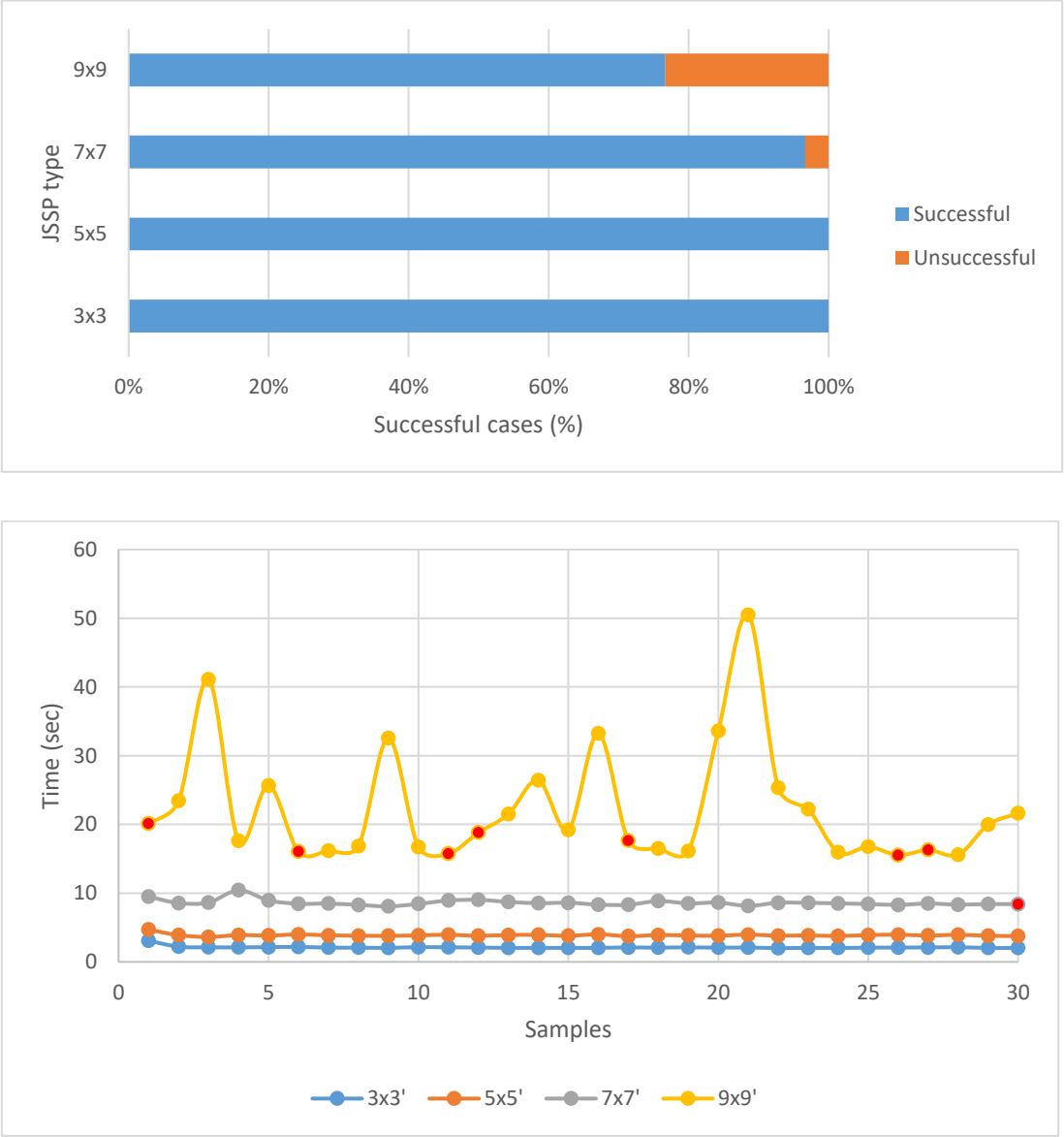


Figure 5-10 - Chart of successful solutions found and algorithm runtime, respectively. Steady fitness of 500 generations

In the next chapter are presented the conclusions about accomplished work, as well as the further work necessary to improve this scheduling tool.

6

Conclusion and Future Work

In this chapter, some conclusions about the developed work will be presented, as well as some attention to the subjects that should be approached in the future to keep developing the work presented in this document.

6.1. Conclusions

The main goal of this work was to develop a scheduling tool, able to efficiently allocate a set of production and maintenance tasks in a factory. The JSSP to be applied in a real case was a tough challenge, since there were a lot of considerations to take into account and the most research material is based on static and non-real environments.

That being said, after analyzing the results obtained in Chapter 5, some conclusions can be drawn from the presented work.

In what concerns to the case where the crossover and mutation were changed, section 5.1, different factors were tested. Such as the fitness value, the number of generations to get a solution and the percentage of successful solutions found in each case.

Analyzing the tables of the section 5.1, it is possible to observe that there are several possible crossover-mutation combinations to get results with near 100% of successful cases, to solve this scheduling problem. Some of them can reach the best fitness more times and other can reach the best solution in less generations. This can happen because partially-matched crossover and swap mutator act in an identical way, once both cross the genes of the chromosome and guarantee that there are no replicate genes. Even though, crossover probabilities between 0.5 and 0.7 and a small mutation probability, between 0.05 and 0.1, look to perform well in this JSSP.

After adding some maintenance tasks to the list of tasks to allocate, the tool was able to reallocate the production tasks in a way that the maintenance tasks could be executed during the maintenance shifts, without worsening the obtained fitness values. Since in cases where two and four maintenance tasks of 100 minutes were added, the best fitness value remained the same as the case when there were no maintenance tasks. However, when were added two tasks of 250 minutes to each station, the total execution time of the maintenance tasks exceeded the 480 minutes available per day to allocate the maintenance tasks and the best fitness found went from a value of 1976 to 2159, which means that some maintenance tasks were allocated to the next day. The average number of generations to found a solution remained almost the same as well as the successful solutions found, when the maintenance tasks were added.

In section 5.2 were tested the size of population and the steady fitness generations. The tests proved that when the population size and the number of generations were increased, the number of valid solutions found was bigger, nevertheless, it consumes more processing time which could be undesirable. Consequently, it is extremely important to get a good balance between the population size and the number of generations that limit the process evolution. If those values are too high, runtime time will also be, but if they are too low, valid solutions could not be found.

Next, are presented some conclusion about the different population sizes applied to problems with different sizes, i.e. different number of operations and stations available to perform those operations:

- **Population of 500 individuals:** A population of 500 individuals performed well for small size problems, however had some problems to find valid solutions for bigger problems, although it is possible to note some improvements were reached when the steady fitness

was increased. Also, the algorithm took only a few seconds to execute in smaller problems (less than 0.5 seconds), but once the complexity increased it took more time.

- **Population of 1000 individuals:** By increasing the population size to 1000 individuals, the successful solutions found for 7x7 increased almost to 100%. For 9x9 problem the valid solutions found increased largely when the number of generations were augmented. By the other hand, the processing time is bigger compared to a population of 500 individuals and, also, when the number of generations is increased.
- **Population of 1500 individuals:** By setting the population size to 1500 individuals, it got the most successful cases when the number of generations was 500, despite the time consumed was bigger than any other case. For smaller values of the number of generations, the results were similar to the ones with 1000 individuals' population, in terms of successful cases and the algorithm runtime. But, it always found more valid solutions than the 500 individuals' population.

The results obtained in this work suggest that GAs can be used to generate schedules in a JSSP. They can find feasible solutions in a reasonable amount of time, being that in a lot of cases they found an optimal solution, as shown in Chapter 5. Although they could take too long when there are many tasks to allocate, most of the times a good solution was found.

The main problem was when a product had a lot of tasks to be performed and they need to follow a predefined order, which requires the algorithm to consume a lot of processing power.

The proposed approach could be a good solution to solve the task allocation problem efficiently for small dimension problems. For small problems, till the 7x7 problem, it found feasible schedules most of the times. And those schedules were found always in less than 10 seconds, which was fast enough. Yet, it needs to be adjusted to solve more complex problems, such as the 9x9 problem, since good solutions were found just a few times.

Besides it can deal with a flexible shop-floor, where a working station can execute different actions, the algorithm was not designed to decide which one is better, so, the station configuration always need to be specified.

All the tests were performed using a real set of operation and stations, with complete information about each one, provided by one of PERFoRM's partner. It was an important help, in order to validate the results obtained, since the data was from a real environment.

6.2. Future Work

The future work should focus on get a better performance of the algorithm, by getting a better way to allocate the tasks corresponding to the same product. This can be done by set an initial chromosome where the tasks already respect the production order. Also, more efficient genetic operators could be adopted to avoid precedence errors.

In a more specific way, an initial chromosome could be instantiated based on the good chromosomes previously obtained. Or it can be initialized in a way which the order of the tasks was already well defined, instead of starting a random chromosome each time a new schedule was requested. Also, both crossover and mutation operators could be implemented in order to keep always the tasks' order of a product, changing the genes more efficiently, so the tasks of a specific product stay always in the same order. This way, the precedence errors between tasks would not occur and maybe some processing time could be saved.

Another important point is the precedence between products. In a real factory, some products can only be executed after others have been done. The further work should focus mainly at this point in order to have a more embracing and robust algorithm.

7

Bibliography

Aissani, N., Beldjilali, B., & Trentesaux, D. (2009). Dynamic scheduling of maintenance tasks in the petroleum industry: A reinforcement approach. *Engineering Applications of Artificial Intelligence*, *22*(7), 1089–1103. https://doi.org/10.1016/j.engappai.2009.01.014

Ángel-Bello, F., Álvarez, A., Pacheco, J., & Martínez, I. (2011). A single machine scheduling problem with availability constraints and sequence-dependent setup costs. *Applied Mathematical Modelling*, *35*(4), 2041–2050. https://doi.org/10.1016/j.apm.2010.11.017

Apolinar, J., & Rodríguez, M. (2017). Three-dimensional microscope vision system based on micro laser line scanning and adaptive genetic algorithms. *Optics Communications*, *385*(October 2016), 1–8. https://doi.org/10.1016/j.optcom.2016.09.025

Asadzadeh, L. (2015). A local search genetic algorithm for the job shop scheduling problem with intelligent agents. *Computers & Industrial Engineering*, *85*, 376–383. https://doi.org/10.1016/j.cie.2015.04.006

Balin, S. (2011). Non-identical parallel machine scheduling using genetic algorithm. *Expert Systems with Applications*, *38*(6), 6814–6821. https://doi.org/10.1109/TM

Baykasoğlu, A., & Özbakir, L. (2010). Analyzing the effect of dispatching rules on the scheduling performance through grammar based flexible scheduling system. *International Journal of Production Economics*, *124*(2), 369–381. https://doi.org/10.1016/j.ijpe.2009.11.032

Bierwirth, C., & Mattfeld, D. C. (1999). Production Scheduling and Rescheduling with Genetic

Algorithms. *Evolutionary Computation*, *7*(1), 1–17. https://doi.org/10.1162/evco.1999.7.1.1

Buzatu, C., & Bancila, D. (2008). A hybrid algorythm for job shop scheduling, (April). Retrieved from http://innomet.ttu.ee/daaam08/Online/Engineering Management/Buzatu.pdf

Çakar, T., Köker, R., & Demir, H. I. (2008). Parallel robot scheduling to minimize mean tardiness with precedence constraints using a genetic algorithm. *Advances in Engineering Software*, *39*(1), 47–54. https://doi.org/10.1016/j.advengsoft.2006.11.003

Cardon, A., Galinho, T., & Vacher, J. P. (2000). Genetic algorithms using multi-objectives in a multi-agent system. *Robotics and Autonomous Systems*, *33*(2), 179–190. https://doi.org/10.1016/S0921-8890(00)00088-9

Casas, I., Taheri, J., Ranjan, R., Wang, L., & Zomaya, A. Y. (2016). GA-ETI: An enhanced genetic algorithm for the scheduling of scientific workflows in cloud environments. *Journal of Computational Science*. https://doi.org/10.1016/j.jocs.2016.08.007

Chalfoun, I., Kouiss, K., Huyet, A. L., Bouton, N., & Ray, P. (2013). Proposal for a generic model dedicated to Reconfigurable and Agile Manufacturing Systems (RAMS). *Procedia CIRP*, *7*, 485–490. https://doi.org/10.1016/j.procir.2013.06.020

Chen, J. C., Chen, Y. Y., & Liang, Y. (2016). Application of a genetic algorithm in solving the capacity allocation problem with machine dedication in the photolithography area. *Journal of Manufacturing Systems*, *41*, 165–177. https://doi.org/10.1016/j.jmsy.2016.08.010

Chen, Y., Cowling, P., Polack, F., Remde, S., & Mourdjis, P. (2017). Dynamic optimisation of preventative and corrective maintenance schedules for a large scale urban drainage system. *European Journal of Operational Research*, *257*(2), 494–510. https://doi.org/10.1016/j.ejor.2016.07.027

Choi, I. C., & Choi, D. S. (2002). A local search algorithm for jobshop scheduling problems with alternative operations and sequence-dependent setups. *Computers and Industrial Engineering*, *42*(1), 43–58. https://doi.org/10.1016/S0360-8352(02)00002-5

Chung, B. Do, & Kim, B. S. (2016). A hybrid genetic algorithm with two-stage dispatching heuristic for a machine scheduling problem with step-deteriorating jobs and rate-modifying activities. *Computers and Industrial Engineering*, *98*, 113–124. https://doi.org/10.1016/j.cie.2016.05.028

Costa, A., Cappadonna, F. A., & Fichera, S. (2016). Minimizing the total completion time on a parallel machine system with tool changes. *Computers & Industrial Engineering*, *91*, 290–301. https://doi.org/http://dx.doi.org.ezproxy.javeriana.edu.co:2048/10.1016/j.cie.2015.11.015

Csaji, B. C., & Monostori, L. (2006). Adaptive Algorithms in Distributed Resource Allocation. *Proceedings of the 6th International Workshop on Emergent Synthesis (IWES)*, (June), 69–75. Retrieved from www.sztaki.hu/~csaji/csaji-iwes-2006.pdf

Dionisio Rocha, A., Peres, R., & Barata, J. (2015). An agent based monitoring architecture for plug and produce based manufacturing systems. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)* (pp. 1318–1323). IEEE. https://doi.org/10.1109/INDIN.2015.7281926

El-Abbasy, M. S., Elazouni, A., & Zayed, T. (2016). MOSCOPEA: Multi-objective construction scheduling optimization using elitist non-dominated sorting genetic algorithm. *Automation*

in Construction, *71*(Part 2), 153–170. https://doi.org/10.1016/j.autcon.2016.08.038

Elkins, D. A., Huang, N., & Alden, J. M. (2004). Agile manufacturing systems in the automotive industry. *International Journal of Production Economics*, *91*(3), 201–214. https://doi.org/10.1016/j.ijpe.2003.07.006

ElMaraghy, H. A. (2005). Flexible and reconfigurable manufacturing systems paradigms. *International Journal of Flexible Manufacturing Systems*, *17*(4), 261–276. https://doi.org/10.1007/s10696-006-9028-7

Ernst, A. T., Jiang, H., Krishnamoorthy, M., & Sier, D. (2004). Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, *153*(1), 3–27. https://doi.org/10.1016/S0377-2217(03)00095-X

Fattahi, P., & Fallahi, A. (2010). Dynamic scheduling in flexible job shop systems by considering simultaneously efficiency and stability. *CIRP Journal of Manufacturing Science and Technology*, *2*(2), 114–123. https://doi.org/10.1016/j.cirpj.2009.10.001

Froger, A., Gendreau, M., Mendoza, J. E., Pinson, É., & Rousseau, L.-M. (2015). Maintenance scheduling in the electricity industry: A literature review. *European Journal of Operational Research*, *251*(3), 695–706. https://doi.org/10.1016/j.ejor.2015.08.045

Galan, R., Racero, J., Eguia, I., & Garcia, J. M. (2007). A systematic approach for product families formation in Reconfigurable Manufacturing Systems. *Robotics and Computer-Integrated Manufacturing*, *23*(5), 489–502. https://doi.org/10.1016/j.rcim.2006.06.001

Gao, J., Gen, M., & Sun, L. (2006). Scheduling jobs and maintenances in flexible job shop with a hybrid genetic algorithm. *Journal of Intelligent Manufacturing*, *17*(4), 493–507. https://doi.org/10.1007/s10845-005-0021-x

Gonçalves, J. F., De Magalhães Mendes, J. J., & Resende, M. G. C. (2005). A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*. https://doi.org/10.1016/j.ejor.2004.03.012

Gunasekaran, A. (1999). Agile manufacturing: A framework for research and development. *International Journal of Production Economics*, *62*(1–2), 87–105. https://doi.org/10.1016/S0925-5273(98)00222-9

Holland, J. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press.

Jia, H. Z., Fuh, J. Y. H., Nee, A. Y. C., & Zhang, Y. F. (2007). Integration of genetic algorithm and Gantt chart for job shop scheduling in distributed manufacturing systems. *Computers & Industrial Engineering*, *53*(2), 313–320. https://doi.org/10.1016/j.cie.2007.06.024

Joo, C. M., & Kim, B. S. (2015). Hybrid genetic algorithms with dispatching rules for unrelated parallel machine scheduling with setup time and production availability. *Computers and Industrial Engineering*, *85*, 102–109. https://doi.org/10.1016/j.cie.2015.02.029

Jose, K., & Pratihar, D. K. (2015). Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. *Robotics and Autonomous Systems*, *80*, 34–42. https://doi.org/10.1016/j.robot.2016.02.003

Karimi, S., Ardalan, Z., Naderi, B., & Mohammadi, M. (2016). Scheduling flexible job-shops with transportation times: Mathematical models and a hybrid imperialist competitive

algorithm. *Applied Mathematical Modelling*, *41*, 667–682. https://doi.org/10.1016/j.apm.2016.09.022

Kenné, J. ., & Gharbi, A. (2004). Stohastic optimal production control problem with corrective maintenance. *Computers & Industrial Engineering*, *46*(4), 865–875. https://doi.org/10.1016/j.cie.2004.05.024

Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G., & Van Brussel, H. (1999). Reconfigurable Manufacturing Systems. *CIRP Annals*, *48*(2), 527–540. https://doi.org/10.1016/S0007-8506(07)63232-6

Koren, Y., & Shpitalni, M. (2010). Design of reconfigurable manufacturing systems. *Journal of Manufacturing Systems*, *29*(4), 130–141. https://doi.org/10.1016/j.jmsy.2011.01.001

Kretschmer, R., Pfouga, A., Rulhoff, S., & Stjepandić, J. (2017). Knowledge-based design for assembly in agile manufacturing by using Data Mining methods. *Advanced Engineering Informatics*. https://doi.org/10.1016/j.aei.2016.12.006

Kundakcı, N., & Kulak, O. (2016). Hybrid genetic algorithms for minimizing makespan in dynamic job shop scheduling problem. https://doi.org/10.1016/j.cie.2016.03.011

Ladj, A., Varnier, C., & Tayeb, F. B. S. (2016). IPro-GA: an integrated prognostic based GA for scheduling jobs and predictive maintenance in a single multifunctional machine. *IFAC-PapersOnLine*, *49*(12), 1821–1826. https://doi.org/10.1016/j.ifacol.2016.07.847

Lee, W. C., Wang, J. Y., & Lin, M. C. (2016). A branch-and-bound algorithm for minimizing the total weighted completion time on parallel identical machines with two competing agents. *Knowledge-Based Systems*, *105*, 68–82. https://doi.org/10.1016/j.knosys.2016.05.012

Lei, H., Xing, K., Han, L., & Gao, Z. (2017). Hybrid heuristic search approach for deadlock-free scheduling of flexible manufacturing systems using Petri nets. *Applied Soft Computing*, *18*(2), 240–245. https://doi.org/10.1016/j.asoc.2017.01.045

Lenstra, J. K., & Rinnooy Kan, a. H. G. (1978). Complexity of Scheduling under Precedence Constraints. *Operations Research*. https://doi.org/10.1287/opre.26.1.22

Li, J., Huang, Y., & Niu, X. (2016). A branch population genetic algorithm for dual-resource constrained job shop scheduling problem. *Computers & Industrial Engineering*, *103*, 330. https://doi.org/10.1016/j.cie.2016.12.016

Li, J. Q., & Pan, Q. K. (2012). Chemical-reaction optimization for flexible job-shop scheduling problems with maintenance activity. *Applied Soft Computing Journal*, *12*(9), 2896–2912. https://doi.org/10.1016/j.asoc.2012.04.012

Li, X., & Gao, L. (2016). An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. *International Journal of Production Economics*, *174*, 93–110. https://doi.org/10.1016/j.ijpe.2016.01.016

Liu, S. Q., Ong, H. L., & Ng, K. M. (2005). Metaheuristics for minimizing the makespan of the dynamic shop scheduling problem. *Advances in Engineering Software*, *36*(3), 199–205. https://doi.org/10.1016/j.advengsoft.2004.10.002

Liu, Y., Dong, H., Lohse, N., & Petrovic, S. (2014). Reducing environmental impact of production during a Rolling Blackout policy - A multi-objective schedule optimisation approach. *Journal of Cleaner Production*, *102*, 418–427.

https://doi.org/10.1016/j.jclepro.2015.04.038

Liu, Y., Dong, H., Lohse, N., Petrovic, S., & Gindy, N. (2014). An investigation into minimising total energy consumption and total weighted tardiness in job shops. *Journal of Cleaner Production*, *65*, 87–96. https://doi.org/10.1016/j.jclepro.2013.07.060

Lu, C., Gao, L., Li, X., Pan, Q., & Wang, Q. (2017). Energy-efficient permutation flow shop scheduling problem using a hybrid multi-objective backtracking search algorithm. *Journal of Cleaner Production*. https://doi.org/10.1016/j.jclepro.2017.01.011

Madureira, A., Pereira, I., & Abraham, A. (2013). Towards Scheduling Optimization through Artificial Bee Colony Approach. *2013 World Congress on Nature and Biologically Inspired Computing*, (August), 253–258. https://doi.org/10.1109/NaBIC.2013.6617872

Mattfeld, D. C. (1996). Evolutionary search and the job shop : investigations on genetic algorithms for production scheduling. *Production and Logistics*, (November), ix, 152 . https://doi.org/10.1007/978-3-662-11712-5

Noor, S., Ikramullah Lali, M., & Saqib Nawaz, M. (2015). Solving Job Shop Scheduling Problem With Genetic Algorithm. *Int . (Lahore)*, *27*(4), 3367–3371.

Ordóñez Galán, C., Sánchez Lasheras, F., de Cos Juez, F. J., & Bernardo Sánchez, A. (2017). Missing data imputation of questionnaires by means of genetic algorithms with different fitness functions. *Journal of Computational and Applied Mathematics*, *311*, 704–717. https://doi.org/10.1016/j.cam.2016.08.012

Page, A. J., Keane, T. M., & Naughton, T. J. (2010). Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system. *Journal of Parallel and Distributed Computing*, *70*(7), 758–766. https://doi.org/10.1016/j.jpdc.2010.03.011

PERFoRM, P. (2016). Deliverable D2.2 - Definition of the System Architecture, 1–82. Retrieved from http://www.horizon2020-perform.eu/files/documents/D2_2_public.pdf

PERFoRM Project. (2017). Deliverable 2.3 - Specification of the Generic Interfaces for Machinery, Control Systems and Data Backbone. Retrieved from http://www.horizon2020-perform.eu/files/documents/D2_3_public.pdf

Petrović, M., Vuković, N., Mitić, M., & Miljković, Z. (2016). Integration of process planning and scheduling using chaotic particle swarm optimization algorithm. *Expert Systems with Applications*, *64*, 569–588. https://doi.org/10.1016/j.eswa.2016.08.019

Pezzella, F., Morganti, G., & Ciaschetti, G. (2008). A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers and Operations Research*, *35*(10), 3202–3212. https://doi.org/10.1016/j.cor.2007.02.014

Raza, A., & Ulansky, V. (2017). Modelling of Predictive Maintenance for a Periodically Inspected System. *Procedia CIRP*, *59*(TESConf 2016), 95–101. https://doi.org/10.1016/j.procir.2016.09.032

Rocha, A., Di Orio, G., Barata, J., Antzoulatos, N., Castro, E., Scrimieri, D., … Ribeiro, L. (2014). An agent based framework to support plug and produce. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)* (pp. 504–510). IEEE. https://doi.org/10.1109/INDIN.2014.6945565

Ruiz, R., Carlos García-Díaz, J., & Maroto, C. (2007). Considering scheduling and preventive

maintenance in the flowshop sequencing problem. *Computers and Operations Research*, *34*(11), 3314–3330. https://doi.org/10.1016/j.cor.2005.12.007

Saidi-Mehrabad, M., Dehnavi-Arani, S., Evazabadian, F., & Mahmoodian, V. (2015). An Ant Colony Algorithm (ACA) for solving the new integrated model of job shop scheduling and conflict-free routing of AGVs. *Computers and Industrial Engineering*, *86*, 2–13. https://doi.org/10.1016/j.cie.2015.01.003

Scrimieri, D., Antzoulatos, N., Castro, E., & Ratchev, S. M. (2015). Automated Experience-Based Learning for Plug and Produce Assembly Systems. *IFAC-PapersOnLine*, *48*(3), 2083–2088. https://doi.org/10.1016/j.ifacol.2015.06.396

Selcuk, S. (2016). Predictive maintenance, its implementation and latest trends. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 0954405415601640-. https://doi.org/10.1177/0954405415601640

Sharma, P., & Jain, A. (2015). Performance analysis of dispatching rules in a stochastic dynamic job shop manufacturing system with sequence-dependent setup times: Simulation approach. *CIRP Journal of Manufacturing Science and Technology*, *10*, 110–119. https://doi.org/10.1016/j.cirpj.2015.03.003

Singh, M. R., & Mahapatra, S. S. (2015). A Quantum Behaved Particle Swarm Optimization for Flexible Job Shop Scheduling. *Computers & Industrial Engineering*, *93*, 36–44. https://doi.org/10.1016/j.cie.2015.12.004

Sobeyko, O., & Mönch, L. (2016). Heuristic approaches for scheduling jobs in large-scale flexible job shops. *Computers and Operations Research*, *68*, 97–109. https://doi.org/10.1016/j.cor.2015.11.004

Strusevich, V. A. (1999). A heuristic for the two-machine open-shop scheduling problem with transportation times. *Discrete Applied Mathematics*, *93*(2–3), 287–304. https://doi.org/10.1016/S0166-218X(99)00115-8

Tay, J. C., & Ho, N. B. (2008). Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers and Industrial Engineering*, *54*(3), 453–473. https://doi.org/10.1016/j.cie.2007.08.008

Ting, C. K., Su, C. H., & Lee, C. N. (2010). Multi-parent extension of partially mapped crossover for combinatorial optimization problems. *Expert Systems with Applications*, *37*(3), 1879–1886. https://doi.org/10.1016/j.eswa.2009.07.082

Ullman, J. D. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences*, *10*(3), 384–393. https://doi.org/10.1016/S0022-0000(75)80008-0

Wang, D., Xiong, H., & Fang, D. (2016). A Neighborhood Expansion Tabu Search Algorithm Based On Genetic Factors. *Open Journal of Social Sciences*, *4*(4), 303–308. https://doi.org/10.4236/jss.2016.43037

Wang, J., Tang, J., Xue, G., & Yang, D. (2016). Towards Energy-efficient Task Scheduling on Smartphones in Mobile Crowd Sensing Systems. *Computer Networks*. https://doi.org/10.1016/j.comnet.2016.11.020

Wang, T., Liu, Z., Chen, Y., Xu, Y., & Dai, X. (2014). Load Balancing Task Scheduling Based on Genetic Algorithm in Cloud Computing. *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 146–152.

https://doi.org/10.1109/DASC.2014.35

Werner, F. (2011). Genetic algorithms for shop scheduling problems: A survey. *Preprint*. Retrieved from http://www.math.uni-magdeburg.de/~werner/preprints/p11-31.pdf

Wilhelmstotter, F. (2016). Jenetics Library User's Manual. Retrieved from http://jenetics.io/

Xiong, H., Fan, H., Jiang, G., & Li, G. (2017). A simulation-based study of dispatching rules in a dynamic job shop scheduling problem with batch release and extended technical precedence constraints. *European Journal of Operational Research*, *257*(1), 13–24. https://doi.org/10.1016/j.ejor.2016.07.030

Xu, J., Xu, X., & Xie, S. Q. (2011). Recent developments in Dual Resource Constrained (DRC) system research. *European Journal of Operational Research*, *215*(2), 309–318. https://doi.org/10.1016/j.ejor.2011.03.004

Yamada, T., & Nakano, R. (1997). Job-shop scheduling. In *Genetic algorithms in engineering systems* (pp. 134–160).

Yang, S. (2008). Genetic algorithms with memory- and elitism-based immigrants in dynamic environments. *Evolutionary Computation*, *16*(3), 385–416. https://doi.org/10.1162/evco.2008.16.3.385

Yoo, J., & Lee, I. S. (2016). Parallel machine scheduling with maintenance activities. *Computers & Industrial Engineering*, *101*, 361–371. https://doi.org/10.1016/j.cie.2016.09.020

Yusuf, Y. ., Sarhadi, M., & Gunasekaran, A. (1999). Agile manufacturing: *International Journal of Production Economics*, *62*(1–2), 33–43. https://doi.org/10.1016/S0925-5273(98)00219-9

Zhang, L., Gao, L., & Li, X. (2013). A hybrid genetic algorithm and tabu search for a multi-objective dynamic job shop scheduling problem. *International Journal of Production Research*, *51*(12), 3516–3531. https://doi.org/10.1080/00207543.2012.751509

Zhang, W., Gen, M., & Jo, J. (2014). Hybrid sampling strategy-based multiobjective evolutionary algorithm for process planning and scheduling problem. *Journal of Intelligent Manufacturing*, *25*(5), 881–897. https://doi.org/10.1007/s10845-013-0814-2

Zhang, Y.-H., Gong, Y.-J., Gu, T.-L., Li, Y., & Zhang, J. (2017). Flexible genetic algorithm: A simple and generic approach to node placement problems. *Applied Soft Computing*, *52*, 457–470. https://doi.org/10.1016/j.asoc.2016.10.022

Zhao, F., Tang, J., Wang, J., & Jonrinaldi. (2014). An improved particle swarm optimization with decline disturbance index (DDPSO) for multi-objective job-shop scheduling problem. *Computers and Operations Research*, *45*, 38–50. https://doi.org/10.1016/j.cor.2013.11.019