UCLouvain

École polytechnique de Louvain

Autonomous shuttles for a logistical warehouse

Automatic design of the warehouse

Author: **Cyril Pletinckx**
Supervisor: **Pierre Schaus**
Readers: **Vianney Coppé, Frédéric Gérard, Pierre Schaus**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Abstract

Automation has been growing in every part of the humans' life in the last decade. We are at the beginning of a new era where efficiency is pushed to its limits by making all the repetitive tasks automated. Robots, or more generally machines, are able to achieve a much higher throughput of operations per hour than humans and are not subject to tiredness or emotions which makes them also more reliable. A field that is definitely prone to be automated concerns the logistical warehouses. This comes with scheduling and tasks assignment algorithms but the choice to automate a warehouse also induces some constraints.

Good management algorithms are needed for an automated warehouse to work as intended but it is not enough to use these algorithms in an environment that is not suited for the task at hand. This thesis aims at analyzing how the design choices of the warehouse could be automated as well. The goal of this work is to study existing algorithms that would allow to find the best number of shuttles to use or the dimensions of the warehouse to have under a certain workload for the warehouse to work as intended and as efficiently as possible. The design term is used here to refer to all the choices that must be made to build a particular warehouse.

Acknowledgments

I would like to thank my supervisor, Prof. Pierre Schaus, and Vianney Coppé, a Ph.D. student that also worked on this project. They both took the time to guide me through the development of this thesis when it was needed and led me to the point where I ended. I especially thank Vianney Coppé who always answered my numerous questions and accepted to read this thesis and its accompanying code several times in order to give me advice to improve its form and its content.

I also thank Mr. Frédéric Gérard who accepted to be one of my jury members and hence to read this thesis and Catherine Dewandelaer who helped me to check the English syntax and vocabulary of this report.

Finally I would like to warmly thank all the people that accepted to review my thesis or that motivated me during this year such as my parents, Claire Delplanque and my closest friends.

Contents

Chapter 1

Introduction

The goal of this master thesis is to investigate intelligent algorithms to automatically design the warehouse that would fit a certain client's demand. This design does not only include the dimensions of the warehouse - length, width and height - but all the design choices that has to be made before even optimizing the way the warehouse will be managed by the shuttles. Intuitively this could be considered as a hyper-optimization problem which aims at finding which are the best parameters to use for the main optimization problem which itself consists in assigning loading, unloading or reorganization missions to the different shuttles in order to minimize the service time of each truck arriving at the warehouse. The objective here will then be to try different values of the parameters - for example the number of shuttles to use in the warehouse - until the hyper-algorithm finds the optimal values for which the task assignment algorithm is also optimal - what optimal means remains to be defined. In the literature this problem is referred as the automatic algorithm configuration problem or the parameter tuning problem or even the hyper-optimization problem.

This field has been of a growing importance in the last few decades as highly parameterized algorithms have emerged. For example numerous of them have been developed to solve NP-complete or NP-hard problems in a heuristic way. These algorithms generally need to be run with certain values of their parameters to work well. This is why finding these optimal - or at least very good - parameters is mandatory. In the past, this was done manually by making experiments for several weeks or even months. Today, state-of-the-art algorithms allow to do it automatically. These are for example SMAC, FocusedILS, IFRace or GGA. They all allow to handle all kinds of parameters - continuous, discrete and categorical - except for FocusedILS which needs a discretization of the continuous parameters. They also permit to use multiple instances while, previously to these algorithms, only one instance could be used to achieve the task of finding the best parameters of

the target algorithm. Further characteristics of these will be described in chapter 3.

While they were not designed especially for our kind of problem in the beginning, these algorithms are well suited for finding the best configuration for the warehouse under a certain distribution of arriving trucks. To do this we first need a parameterized target algorithm for which the parameters have to be optimized. In this thesis it will consist in a discrete time simulation of the warehouse that will be described in section 2.5. Numerous parameters can be considered to find the best design of a warehouse - for example the number of shuttles to use or the dimensions of the building - and only a subsample of these will be studied. It is also essential to define what will be measured in our optimization process. The execution time of the target algorithm is generally an objective of interest in the literature but in our case we will rather minimize a cost function composed of the service time of the trucks and the monetary costs of the chosen design for the warehouse. Finally it is necessary to define instances on which the simulation will be run to assess the performances of each tested configuration. These scenarios will be composed of series of trucks arriving at the warehouse at specific times given by a certain distribution - for example a normal or a uniform one. These adaptations of the initial way these automatic configuration algorithms have been designed will allow us to use the state-of-the-art in the domain for our application at hand. The goal will then be to evaluate at which point they can be helpful in our case.

Chapter 2 will detail the problem that will be tackled in this thesis, chapter 3 will review the algorithms that will be studied in the experimental phase in chapter 4. Finally chapter 5 will be devoted to examining what could have been improved or studied further in this thesis if it was not limited in time.

Chapter 2

Problem definition

2.1 Introduction

This chapter aims at iteratively defining the problem at hand from the formal problem definition in section 2.2 to the concrete definition of the algorithm that will be implemented to simulate the warehouse during the automatic configuration process in section 2.5. For this purpose the problem will be defined in the context of a logistical warehouse in section 2.3 before being simplified to fit in this master thesis time constraints in section 2.4.

2.2 Formal problem definition

A generic definition of the problem at hand is described in [1] : given

- an algorithm A with parameters $p_1, ..., p_k$ that affect its behaviour,
- a space C of configurations (i.e., parameter settings), where each configuration $c \in C$ specifies values for A's parameters such that A's behaviour on a given problem instance is completely specified (up to possible randomisations of A),
- a set of problem instances I,
- a performance metric m that measures the performance of A, on an instance set I for a given configuration c,

a configuration $c^* \in C$ that results in optimal performance of A on I according to metric m must be found.

Before covering, in the next chapter, well-known methods to solve this algorithm configuration problem, it is essential to define the different components of the problem in the specific application of the design of a warehouse, namely the algorithm A, the parameters $p_1, ..., p_k$ and their domains - which will then define the space of configurations C -, the set of problem instances I and the performance metric m.

2.3 Problem definition in the context of the design of warehouses

In the context of this master thesis the different components described above will be defined as follows.

The algorithm A for which we have to find the best configuration c^* is an algorithm (or a set of algorithms) that will assign loading, unloading or reorganization tasks to the shuttles in order to minimize the service time of the trucks. In our specific case it will consist in a single discrete time simulation algorithm that will be further described in section 2.5.

The set of problem instances I could be defined through several components. We could imagine different distributions of the trucks arrival time, different distributions of the number of pallets to be loaded/unloaded from the trucks and different numbers of types of pallets - a pallet of type one would not be the same as a pallet of type two. These instances could be used to simulate a specific demand on a daily, weekly or even monthly basis.

The performance metric m has to be defined with two components : we need to know what will be measured and how. The first component is related to the target algorithm A and defines which characteristics of A we want to optimize. One possibility is to minimize the needed time to complete the execution of the algorithm. Another is to minimize or maximize the objective value returned by A - for example, the service time of the trucks in our case - and a third option could be to optimize a mix of these two objectives. Other objectives could be imagined such as maximizing the percentage of the shuttles that are used or the part of the stock that is effectively used on average. The second component of the metric m is typically a statistical metric. For example one could optimize the mean execution time of the algorithm or the median objective value, this is a design choice to make.

The choice of the parameters $p_1, ..., p_k$ and their domains is generally rather easy

to make as, in general, the target algorithm A is already well defined - for example the parameters of a particular SAT solver are defined beforehand by its authors. In the case of this master thesis though the target algorithm is not yet defined and could also be improved through time - i.e. new parameters could be added to the simulation algorithm in later versions - thus this choice is harder to make. Before digging into the details let's first distinguish the different types of parameters.

The categorical parameters are defined such that they can only take discrete values from a predefined set of values and among which no order relation exists. For example the brand of the shuttles would be a categorical variable. We will also include binary parameters in this category. Discrete parameters have a discrete numerical domain as opposed to continuous parameters which have a continuous numerical domain but both of them include an order relation among their values. Finally integer parameters are a special case of discrete parameters for which each discrete value is an integer.

Knowing these five categories of parameters we can now imagine which parameters could be investigated for the design of a warehouse. In table 2.1 a non-exhaustive list of potential candidates is presented along with their categories and units.

It is important at this point to clarify certain definitions. To keep more flexibility the number of parallel tracks and the number of spots per track are defined along the width and the length of the warehouse while these four parameters are obviously related. For a fixed width and length of the tracks and the spots it should not be possible to exceed the defined width and length of the warehouse.

Concerning the two proposed categorical parameters, the arrangement of the warehouse should be used to allow various relative positions between the different parts of the warehouse such as the truck loading/unloading area, the storage area and the battery charging area. The brand of the shuttles on the other hand could lead to the definition of impacting characteristics of the shuttles such as their monetary cost in a finite and well-defined case.

As every part of the algorithm configuration problem is now well defined in the context of this master thesis it is possible to state a more precise goal. The goal of this thesis is to investigate specific implementations of the parameters tuning problem in order to find algorithms that could automatically discover optimal warehouse configurations - concrete sets of values for the parameters - according to a user-defined metric m - for example the mean service time of the trucks or the

Parameter	Category	Unit
Width of the warehouse	Continuous	Meters
Length of the warehouse	Continuous	Meter
Number of floors	Integer	#
Number of lifts	Integer	#
Number of parallel tracks in one floor	Integer	#
Number of pallet spots per track	Integer	#
Number of shuttles	Integer	#
Battery life of the shuttles	Continuous	Minutes
Charging time of the batteries of the shuttles	Continuous	Minutes
Average speed of the shuttles	Continuous	Meters/second
Supported loading weight of the shuttles	Continuous	Kilograms
Loading/unloading average time of the shuttles	Continuous	Seconds
Arrangement of the warehouse	Categorical	/
Brand of the shuttles	Categorical	/
Number of docks	Integer	#
Number of charging sites	Integer	#

Table 2.1: Table regrouping the different parameters that could be evaluated through the configuration process of the set of algorithms A

total monetary cost of the design of the warehouse - on a set of instances which simulate typical scenarios in the context of a logistical warehouse. However, as this goal remains difficult to attain in the time available to elaborate this thesis, this work will focus on a subset of what was described in this section to build a strong prototype in a simplified world that could be further improved later.

2.4 Description of the simplified warehouse world

In order to have a deeper insight of what will exactly be investigated in this report the definition given in the last section must be refined and limited to a simpler

version of the world of logistical warehouses which is complicated.

In this simplified world six parameters will be studied : the number of shuttles, the number of parallel tracks per floor, the number of pallet spots per track, the battery life of the shuttles, the charging time of the batteries and the average speed of the shuttles. As this prototype has to be as generic as possible all the parameter types will be represented. The charging time of the batteries will have a continuous domain, the average speed of the shuttles will be hidden behind three categories - namely slow, normal and fast - and becomes thus a categorical parameter, the battery life of the shuttles will have a discrete domain and the remaining parameters will be integer parameters as shown in table 2.2.

This limitation in the number of parameters that will be investigated induces some hypotheses about the warehouse. In this simplified version all the floors are supposed to be identical (same height, same number of lifts, same number of parallel tracks and same number of pallet spots per track) even though in this case we will only consider warehouses with one floor - as the number of floors was not selected as a parameter. This single floor will still simulate a lift mechanism - the only way for the shuttles to go from one floor to another - by allowing the shuttles to only enter the stock area through four entries. Further extensions to this thesis could overcome these limitations.

Another area that is not discussed in this work concerns all the other parts that compose a warehouse outside of the stock area. Parameters like the arrangement of the warehouse, the number of docks or the number of charging sites will not be studied. In this work it is then supposed that the number of docks and the number of charging sites are infinite but also that they are all at two distinct locations in space - one for the charging sites and one for the docks.

To keep this world relatively simple we will also make the assumption that the warehouse is represented by a grid of positions. This simplified version of the world can be represented as in figure 2.1.

Alongside this grid hypothesis it is now more convenient to define the battery life of the shuttles as an integer parameter describing the number of steps each shuttle can make before they have to charge. This simplification removes the only discrete parameter that had to be investigated. As a reminder it was decided to consider the battery life of the shuttle as a discrete parameter rather than a continuous one in order to have all the parameter types represented in the experimentation phase. A discrete parameter was defined to have a finite and ordered domain composed of

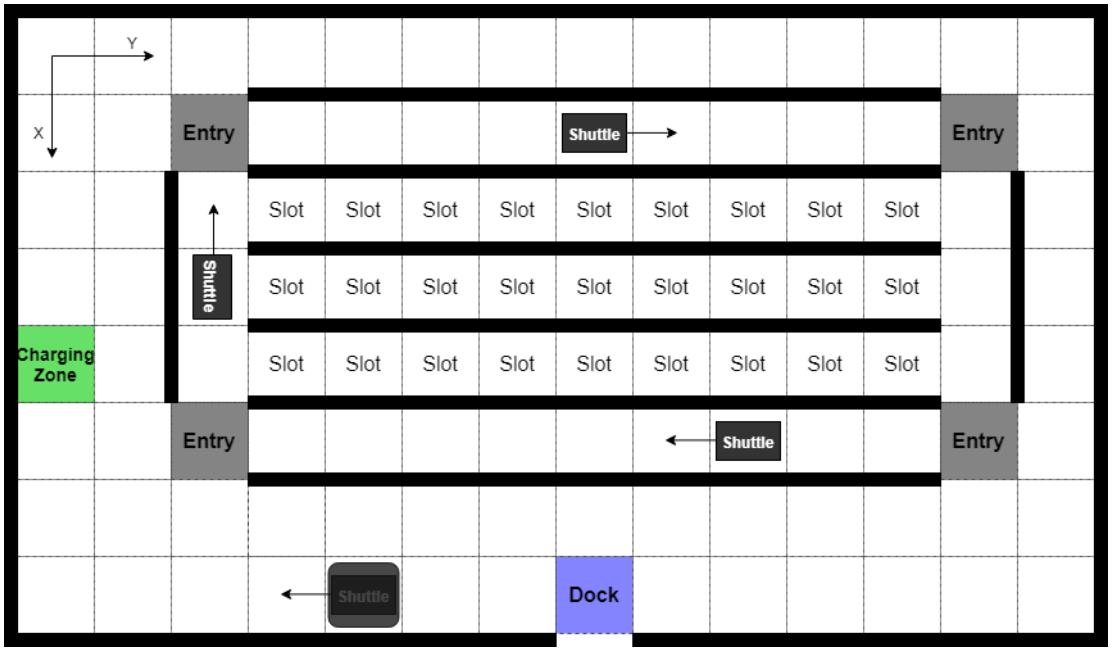


Figure 2.1: Simplified warehouse world schema

floating-point values. However as integer parameters are a special case of discrete parameters, this small change doesn't induce a loss of generality. Table 2.2 gathers a final view of which parameters will be investigated in this thesis.

Parameter	**Category**	**Unit**
Number of parallel tracks in one floor	Integer	#
Number of pallet spots per track	Integer	#
Number of shuttles	Integer	#
Battery life of the shuttles	Integer	#
Charging time of batteries of the shuttles	Continuous	Minutes
Average speed of the shuttles	Categorical	/

Table 2.2: Table regrouping the different parameters that will be evaluated through the configuration process of the target algorithm A in this thesis

To further simplify the problem at hand restrictions for the instances and the performance metric must also be done. Through this work we will consider that

only one type of pallet exists and that the distribution of the number of pallets inside each truck will be uniformly distributed with a minimum of one pallet and a maximum of max pallets where max will be a parameter in the experimentation phase. Two different distributions of the arrival time of the trucks will be considered : the normal distribution to simulate workload peaks for the shuttles and the uniform distribution to estimate the "average" performances of the configurations of the warehouse.

Concerning the performance metric, this thesis will only focus on the average statistical metric but further extensions could consider the median, quartiles or other methods. What is more important to define is what will be measured. Two primordial aspects must be taken into account, the monetary cost and the performances of the configurations. The monetary cost is rather easy to define : each additional shuttle, track or slot will increase it and better batteries - i.e. with lower charging time and higher battery life - and faster shuttles will be considered more expensive. The performances of the configurations however can include several parameters such as the service time of the trucks, the filling rate of the stock and the usage rate of the shuttles. This work will focus on the average service time of the trucks. While this metric could induce an inter-instance bias due to the fact that two different instances might involve two different numbers of missions for the shuttles, as the configurations will be compared on the same instances it should be enough to consider the average service time over the number of trucks rather than over the number of missions. Moreover, the two other aspects are not completely neglected as they will influence the monetary cost. A bigger amount of shuttles would lower the usage rate, which is not directly measured, but increase the monetary cost, which is part of the objective function to optimize, thus reducing the number of shuttles while keeping the same average service time of the trucks would induce a better objective function. The same reasoning can be applied to the filling rate of the stock.

This definition of the metric m implies that our automatic configuration algorithms should be able to handle multi-objectives objective functions and yet such algorithms are quite rare in the literature. This is why this work will rely on an aggregation of the different objectives into a single one through a weighted objective system. This aggregation will be further described in the experimentation section.

2.5 Description of the target algorithm A

This section will detail what will be used as the target algorithm A to achieve the goal of finding the best possible configuration for the warehouse under the hypotheses described before and subject to a particular set of instances I and a metric m. It is defined relatively to figure 2.1.

This algorithm should simulate the simplified warehouse world when it is facing what we shall call $TruckArriveEvents$. A $TruckArriveEvent$ is simply the shuttles AI triggering event - this AI is denoted by the $Controller$ in this report - and consists in one truck entering the simulation process. To implement this, a discrete event simulation algorithm was designed.

2.5.1 The $Warehouse$

A $Warehouse$ object gathers the information that is needed by the simulation process to compute paths in our simplified world. These computations are done through a simple graph which size is defined by two of our six parameters, the number of parallel tracks in one floor - which will now be called the number of aisles - and the number of pallet spots per aisle. An example of a $Warehouse$ with three aisles and nine slots per aisle is depicted in figure 2.1. Each position in this figure will be represented by a node and will be connected to each adjacent position except where these positions are separated by a thick line. As explained before there are three main areas in a $Warehouse$ namely the charging zone, the dock and the $Stock$. The charging zone and the dock are represented by a unique node to simplify the problem. The $Stock$ area can be entered through four different entries and will be further detailed in the next section.

As the two parameters define the size of the $Stock$ rather than the size of the $Warehouse$ it was decided that the $Warehouse$ will always be composed of five plus the number of slots per aisle columns and five plus the number of aisles rows. Two additional rows and columns will be used to complete the $Stock$ with a rectangular contour to allow the shuttles to move from one aisle to another and the rest will be used to simulate the fact that a warehouse is not only composed of the $Stock$ but is larger than that. However as the arrangement of the warehouse is not a parameter of interest in this first version of the automatic configuration procedure the dock and the charging zone will always be positioned at the same place relatively to the number of aisles and the number of slots per aisle. Denoting the top left position as the origin - the position having the (0,0) coordinates - the charging zone and the dock will always be placed at (nAisles+1, 0) and (nRows-1, nCols/2) positions, respectively. The top left entry of the $Stock$ will always be at

the (1,2) position and the top left slot at the (2,3) position. These restrictions fix the layout of the warehouse no matter what the number of aisles and the number of slots per aisle.

2.5.2 The *Stock*

A *Stock* object is composed of two *PriorityQueue*, one containing the filled slots and one containing the empty slots. These slots are sorted relatively to the distance between them and the dock where each $Mission$ has to start or to end - see next section. This allows to quickly pick one filled slot - for a loading $Mission$ - or one free slot - for an unloading $Mission$ - when a new $Mission$ is created. In other words the simulation will always assign the closest available slots to the new $Missions$ first. A reserved slot, i.e. a slot that is already assigned to a $Mission$, will not be present in any of the two *PriorityQueue* and thus a slot can't be assigned to two different $Missions$ at the same time.

In order to simulate a real environment, approximately half of the slots are filled randomly at the start. This will allow to design scenarios where the first truck to come needs to load pallets rather than to unload some. If the *Stock* was empty at the beginning of the simulation, a $Truck$ arriving with loading $Missions$ would have to wait for another $Truck$ with unloading $Missions$ to arrive. Moreover this takes into account the fact that the real $Warehouse$ will probably not be empty when the first customer will knock at its door.

2.5.3 The $Missions$

A $Mission$ object is composed of three main attributes, a starting time, a starting position and an ending position. The starting time is the time when the truck which led to the creation of this $Mission$ arrived. This will allow the $Controller$ to assign the oldest $Missions$ to the $Mobiles$ first. If the $Mission$ is to unload one pallet then the starting position is the position of the dock and the ending position is one of the available slots at the time the $Mission$ was created. On the other hand if the $Mission$ is to load one pallet then the starting position is one of the filled slots at the time the $Mission$ was created and the ending position is the position of the dock.

In the algorithm A once a $Mission$ object is created - i.e. a truck arrived at the dock - it is linked to one slot - either an empty one for an unloading $Mission$ or a filled one for a loading $Mission$ - that remains reserved until the $Mission$ is completed by one of the $Mobiles$. The pool of slots corresponding to the type of the $Mission$ might be empty when the $Mission$ object is created either because all the slots are filled - or empty - or because they are all locked by other $Missions$.

If this happens the *Mission* is partly initialized - no slot is linked to this *Mission* as none of them is available - and added to an *IdleMissions* pool. Each time a regular *Mission* is completed the algorithm will check the *IdleMissions* pool and fill one partly initialized *Mission* by assigning the new available slot to it whenever possible. If the freed slot could not be assigned - for example because it is an empty slot but all the *IdleMissions* are loading *Missions* - it remains available until a new *Mission* that needs this type of slot is created.

2.5.4 The *Trucks*

All the simulation process is based on the initial *TruckArriveEvent* events which allow to report that a new *Truck* has to be managed by the *Controller*. A *Truck* is a simple object with two attributes, a list of pallets to load and a list of pallets to unload. These lists will be needed to create the corresponding *Missions* when a *TruckDockEvent* is triggered. In our case though all the pallets are of the same type but the simulation process doesn't allow to unload and immediately reload the same pallet into the trucks. A pallet will always need to be placed inside the *Stock* before being picked up again.

So then, to declare a new truck a *TruckArriveEvent* which is composed of one *Truck* and its arrival time has to be created. This event will be handled by the *Controller* that will create a new *TruckDockEvent* - happening at the same time in our case but we could imagine to extend this to take into account a travel time between an outer road and the docks of the warehouse thus inducing a different time for the *TruckDockEvent*. A *TruckDockEvent* is an event that will simply initialize all the *Missions* depending on the *toLoad* and *toUnload* lists of the corresponding *Truck* and on the designated dock - in our case we only have one position for the dock but this could be extended once again.

The last event concerning the *Trucks* is the *TruckDoneEvent*. This event is triggered when all the *Missions* created by the corresponding *TruckDockEvent* - both events concern the same truck - have been satisfied. The *Truck* can thus leave the warehouse and we can compute the service time for this truck and add it to the objective value.

2.5.5 The *Mobiles*

The *Mobiles* objects are the most complicated ones as they are linked to several events. Initially all the mobiles are located at the charging zone position waiting for the *Missions* to be assigned to them. The properties of the *Mobiles* are

dependent on two parameters, the average speed of the mobiles and their battery life.

Once a *Mission* is assigned to a *Mobile* it triggers a *MobileMissionStartEvent* which is used for telling the *Mobile* to go to the starting position of the assigned *Mission*. Once this is done a *MobileMissionPickUpEvent* is created to tell the *Controller* that this *Mobile* picked up a pallet and is now on its way to the ending position of its assigned *Mission*. If the pallet is coming from a filled slot it is now marked as free and if the pallet is coming from a *Truck* the *Truck* is informed about the fact that this pallet left its shipment - this is used to know when a *TruckDoneEvent* can be triggered. When the *Mobile* arrives at the ending position of its *Mission* it triggers a *MobileMissionEndEvent*. This event informs the *Controller* that this *Mobile* is now available to complete another mission but also that the slot can be marked as filled - if the *Mission* was an unloading one - or that one pallet was added to the corresponding *Truck* shipment - if the *Mission* was a loading one.

As the *Mobiles* have a finite battery life they sometimes have to charge. When the *Controller* detects a *Mobile* that can't be assigned to any available *Missions* it sends it to the charging site through a *MobileChargingEvent*. The detected *Mobile* then goes to the charging zone and charges its battery up for a certain amount of time that depends on its battery level and on the charging time parameter. During this time the *Mobile* is made unavailable. Once it is fully charged a *MobileChargedEvent* is triggered to get the corresponding *Mobile* available again.

One final management procedure concerning the *Mobiles* is the management of the collisions that might happen during the simulation, i.e. when two different *Mobiles* encounter each other at the same position at the same time. This is quite a difficult problem that has to be tackled by our algorithm. To keep it simple - in terms of difficulty but also in terms of computations - collisions can only happen in the *Stock* area delimited by the rectangular contour around the slots and including it. The hypothesis behind this decision is that outside the *Stock* it is fairly easy for the *Mobiles* to move aside without wasting much time. This is not the case inside the *Stock* and the collisions are then managed as follows.

Each *Mobile* that is currently fulfilling a *Mission* is following a certain assigned path through the *Warehouse*. This path is composed of timed positions, i.e. positions that are crossed by the *Mobile* at specific times depending on the speed of the *Mobile* and the starting time of its *Mission*. Each time a new *Mission* is assigned to one *Mobile* its timed path in the *Stock* area is computed and memorized. If the timed path of this *Mobile* crosses the timed path of another *Mobile* then a collision

occurs. When a collision happens the timed path of the $Mobile$ that would induce this collision is modified in order to account for a penalty : all the positions in the path that are beyond the colliding one are paired with a new time being the previous time plus k times the time needed to travel one square. This simulates the fact that this $Mobile$ will let the others pass until the colliding square is free. At this time - which is k time steps later - the mobile can reenter the track at the same position and continue its journey which has been delayed. Once the new timed path has been computed this process is redone until the timed path reaches its destination. At this moment the timed path is added to the memorized paths pool. When a $Mobile$ arrives at a target position its previous timed path is removed from the pool and a new one might be computed and added to it. For example when a $Mobile$ arrives at the dock to pick up its pallet its previous path going from its previous position to the dock is removed from the pool and its new path going from the dock to the ending position of its $Mission$ is added to it.

This penalty mechanism is an approximation of what could happen in the real warehouse : one of the colliding mobiles has to let the others pass before going further and thus it is delaying its own $Mission$. The main goal of it is to avoid the bias of having as most $Mobiles$ as possible no matter what the size of the warehouse is in order to reduce the service time of the trucks. If too many shuttles are moving inside the $Stock$ area a big amount of collisions might happen and thus it might penalize the service time more than it reduced it. Other more advanced collision management mechanisms could be used in a further version of this algorithm such as a cooperative pathfinding method described in [2].

2.5.6 The <$Mobile$, $Mission$> $Selector$

Defining all the objects that are part of the discrete event simulation algorithm is not enough to fully describe it. The decisions that are made inside the algorithm are of an equal importance.

Most of the decisions that are happening during a simulation have been explained in the previous sections and are summarized in algorithm 1. The last decision-making part - and probably the most important one - concerns the assignment of the $Missions$ to the available $Mobiles$. As for the other definitions the goal is to keep this part quite simple but still representative enough of what could be done in a real application.

The idea behind the <$Mobile$, $Mission$> $Selector$ is to always assign the $Missions$ - in time order - to the closest available $Mobiles$. For this purpose the $Missions$ are ordered by their creation time. For each $Mission$ in this ordered

list the closest available *Mobile* is selected to be assigned to it. If none of the *Mobiles* can satisfy the *Mission* - due to the battery life constraint - it is kept in the *Missions* pool for the next time the *Controller* is called - it is not moved in the *IdleMissions* pool as this pool is reserved for the *Missions* not having an ending or starting position due to a missing available slot in the *Stock*. The *Selector* iterates through the list until no *Mobiles* are available anymore - either because a *Mission* has been assigned to them or because they need to be sent to the charging site. A *Mission* can be assigned to a *Mobile* only if this *Mobile* can complete the whole *Mission* from its current position and go back to the charging site without running out of power. The *Mobiles* that remain available at the end of the selection phase are sent to the charging site - as it seems that they can't satisfy any of the *Missions* because their battery level is too low.

2.5.7 Main framework of A : The *Controller*

The main framework of the simulation is depicted in algorithm 1.

Algorithm 1: The simulation framework

Input: A list of TruckArriveEvent events sorted by time
Output: The value of the objective function
Side effect: *Adds events to the list*

```
Simulate (events)
    Missions ← ∅;
    IdleMissions ← ∅;
    AvailableMobiles ← all mobiles;
    foreach event in events do
        if IdleMissions ≠ ∅ then
            moveIdleMissionsToMissions();
        Pairs ← matchMobilesMissions(Missions, AvailableMobiles);
        foreach pair in Pairs do
            startMission(pair);
        if event is TruckArriveEvent then
            sendTruckToDock();
        foreach AvailableMobile not in pairs do
            sendToChargingSite(AvailableMobile);
    return ServiceTime + MonetaryCost
```

The simulation begins with a list of $TruckArriveEvents$ sorted by time which are used to report the arriving times of the trucks to the $Controller$ in a particular scenario - i.e. instance. The $Controller$ will then create other events accordingly to what was explained in the previous sections and add them to the list of events (which is kept in order of time). Two limit cases still have to be handled by the target algorithm A in our domain of interest.

The simulation might not end because at least one truck is still waiting for some of its $Missions$ to be assigned to the $Mobiles$. This situation can occur either when the $Mobiles$ don't have suited batteries for the longest $Missions$ or because no free slots are available and the remaining $Missions$ are unloading $Missions$ or because no filled slots are available while the remaining $Missions$ are loading $Missions$.

The first case is due to the fact that the evaluated configuration could be composed of a big warehouse with shuttles having a small battery life. In this case the furthest slots - and thus the longest corresponding $Missions$ - could not be accessible even by shuttles being fully charged. This is a situation that has to be avoided and thus the objective value returned in these case is positive infinity in order for this configuration to never become the best one.

The second case happens because the scenarios - i.e. instances - are limited in time. In the real world the waiting trucks would have to wait for other trucks to fill their requirements or for a production line to produce the desired pallets. When this case occurs a penalty is added to the service time as the average time needed to complete a $Mission$ times the number of remaining $Missions$ to fulfill for each waiting truck.

Chapter 3

Automatic configuration algorithms

3.1 Introduction

The problem of automatically defining the parameters of an algorithm is not new and the interest in this problem has greatly grown in the last few decades. Numerous scientific papers have been published in this period and in the second part of this report I will try to give a broad overview of the different techniques and algorithms that exist to achieve the goal at hand. As many different methods exist this overview will focus on what are considered the state-of-the-art algorithms in the domain either because they have been shown to perform well on several applications or because they are cited in well-known reference books such as [1]. Small sections will also be dedicated to an overview of other methods that are not discussed in details. This chapter will be divided into five parts. The first section will detail three state-of-the-art model-free algorithms that will be implemented in the library linked to this report. The second section will have the same goal for two model-based algorithms and the third section will give an insight of other existing methods that could be used for automatic configuration of algorithms. Finally the fourth part will concern a small discussion about the different algorithms and the fifth one will cover the concrete library implementations.

3.2 Model-free methods

In the algorithm configuration problem framework two big classes of methods exist. One is based on statistical or machine learning models that are iteratively updated to sample the configurations to be evaluated while the other one relies on usually well-known exploration techniques to examine the configuration space. In

this section I will present three frameworks that use different exploration techniques.

ParamILS [3] is an algorithm configuration framework that is based on an iterated local search method. It was implemented through four different algorithms namely BasicILS, FocusedILS, MO-BasicILS and MO-FocusedILS. The two last ones are adaptations of the two former ones to tackle multi-objectives objective functions but this thesis will focus on single-objective objective functions in order to be able to compare the methods more easily. In the next section the FocusedILS approach will be presented in more details as it is the most promising version [3] and the implemented one inside the Java library associated to this thesis.

GGA [4] is a completely different framework that uses the power of genetic algorithms on behalf of the parameter tuning problem. As in other more classical problems for which genetic algorithms are used it has shown to be competitive with the other state-of-the art methods such as SMAC or FocusedILS [1, 4].

The racing framework [1, 5, 6, 7] could be considered partly as a model-based method as the IF-race implementation uses statistical models to sample new configurations but as it is not the most important part of the algorithm it is rather described in this section. Its first underlying algorithm named F-race [6] is a completely model-free method that uses statistical tests to decide between the configurations while its most advanced underlying algorithm - IF-race for iterated F-race - uses probabilistic distributions to sample new configurations based on the result of the previous iterations.

3.2.1 The iterated local search approach : paramILS

Iterated local search methods have been used for a wide variety of optimization problems when the goal is to search an optimum in such a large space that it is intractable to design an exact procedure. The broad idea of paramILS is inspired by the manual parameter optimization process which consists in an iterative first improvement procedure. From an initial parameter configuration the algorithm will search among its neighbors for a potential better configuration. Once it finds one, this new configuration - which we call the incumbent - will become the working base of the next iteration. This process is repeated until none of the neighbors of the current incumbent is detected as a better configuration.

The authors of paramILS used this simple procedure as the backbone of their framework from which two algorithms will be built, BasicILS and FocusedILS. The general framework is depicted in algorithm 2. The iterated local search algorithm

that was designed for paramILS is composed of five different components :

- An initialization procedure that compares a default configuration with r random ones - where r is a parameter of paramILS - to find the starting incumbent which will be used as the starting point of the algorithm,
- a subsidiary local search procedure - in this case an iterative first improvement procedure - to find the local optima associated to the investigated configurations,
- a randomization phase to escape local optima - in this case s subsequent one-exchange perturbations where s is a parameter -,
- an acceptance criterion to decide if a certain local optima should be used as the initial configuration of the next iteration,
- a restart mechanism that sets the initial configuration of the next iteration to a random configuration with probability $p_{restart}$ - the third parameter of paramILS.

The procedures $IterativeFirstImprovement$ and *better* can both update the incumbent when an encountered configuration is detected as better than the current incumbent. The $Nbh(\theta)$ procedure returns the one-exchange neighbors of θ. BasicILS and FocusedILS will be different by considering two distinct instantiations of the *better* procedure. As FocusedILS is more polyvalent and more robust than BasicILS, I only present the former in this thesis. The reader might learn more about BasicILS in [3].

FocusedILS

The advantage of FocusedILS in comparison with BasicILS is that it uses more wisely the time spent at evaluating the different configurations. The idea of FocusedILS is to adaptively select the number of instances on which each configuration is run. This allows to focus the search on the most promising configurations and to avoid wasting time on poor ones. To be able to declare that θ_1 is better than θ_2 FocusedILS must use the concept of domination. It will detect that θ_1 is better than θ_2 if and only if the number of instances on which θ_1 was run - $\mathrm{N}(\theta_1)$ - is greater than the number of instances on which θ_2 was run - $\mathrm{N}(\theta_2)$ - and that the estimated objective value of θ_1 on the first $\mathrm{N}(\theta_2)$ instances is lower than the one of θ_2 on the same instances.

This concept allows to describe the *better* procedure of FocusedILS. It first begins by doing one additional algorithm run for the configuration that has the

Algorithm 2: The ParamILS framework

Input: A default configuration θ_0, algorithm parameters $r, p_{restart}$ and s.
Output: The best parameter configuration θ_{inc} found.

```
for i = 1,...,r do
    θ ← random θ ∈ Θ;
    if better(θ, θ_0) then
        θ_0 ← θ;
θ_ils ← IterativeFirstImprovement(θ_0);
θ_inc ← θ_ils;
while not TerminationCriterion() do
    θ ← θ_ils;
    // Perturbation phase
    for i = 1,...,s do
        θ ← random θ' ∈ Nbh(θ);
    // Basic local search phase
    θ ← IterativeFirstImprovement(θ);
    // Acceptance criterion
    if better(θ, θ_ils) then
        θ_ils ← θ;
    with probability p_restart do
        θ_ils ← random θ ∈ Θ;
return θ_inc
```

least amount of runs - or for both if $N(\theta_1) = N(\theta_2)$. Then it continues doing this until one of the two configurations dominates the other. Whenever the first argument of *better* is the best configuration - i.e. the function returns true - this configuration is subject to B bonus algorithms runs where B is the total number of configurations evaluated since the last time θ_{ils} changed - i.e. the last time *better* returned true. This mechanism ensures that the good configurations are assessed on many different instances and that the error made in every comparison decreases on expectation. One last important comment to do about the *better* procedure is that it compares both configurations on the same instances - a mechanism that is called *blocking* in the literature.

Adaptive Capping

The authors of ParamILS comes with another idea, the capping mechanism. The idea behind capping is to stop evaluating a configuration once we are sure that it will perform worse than either the currently evaluated configuration θ_{ils} - trajectory-preserving capping - or the current overall best configuration θ_{inc} - agressive capping. This allows once again to limit the time spent at evaluating poor configurations.

The trajectory-preserving capping saves time but sometimes takes more time than needed to be active as it compares the challenging configurations to θ_{ils} - the best configuration encountered in one specific iteration - which might be a bad configuration. Aggressive capping on the other hand compares them with the best configuration found so far and thus is always active. However if this mechanism is overly aggressive it could throw away good configurations that encountered unlucky instances - with respect to their parameters - on their first runs while they could have been detected better than θ_{inc} if they were executed on more instances. This is why the implementation of FocusedILS with agressive capping uses one additional parameter that is called the bound multiplier bm. The upper bound on the objective value of any configuration is then set to the currently estimated cost of θ_{inc} times bm. This allows to avoid or at least limit, in a user-defined way, the issue explained above.

3.2.2 The genetic algorithm approach : GGA

The genetic algorithm approach has also been widely used in the context of optimization. As for ParamILS with respect to iterated local search procedures, the authors of GGA adapted this approach to parameter tuning based on two observations : genetic algorithms are known to be robust to undesirable objective landscapes - which is the type of landscapes encountered in algorithm configuration problems - and are inherently parallel - this allows to design fast configuration

algorithms.

The necessary condition to implement a genetic algorithm for the parameter tuning problem is to find a way to represent an individual. The authors of GGA are using variable trees - which are inspired from And/Or-tress [8] - for this purpose. A variable tree is represented as in figure 3.1. This structure does not only allow to represent a configuration but it also permits the user to specify relationships between the variables thanks to the special *And* nodes. Variable trees are built as follows :

- Each node is labelled with a variable or with & - the special character to denote an *And* node. Each variable of the target algorithm is associated with at least one node.
- Nodes that correspond to continuous or integer variables have at most one child and *And* nodes have at least two children.
- The children of categorical nodes are all linked with their parent through a labelled connection where each label is one of the value of the categorical variable. Categorical nodes partition the tree into several subtrees where this particular variable was assigned to a particular value denoted by the label on its parent branch.

The special & label is used to separate variables that are independent and can thus be optimized independently. In the general case where it is hard or impossible to define the dependencies at first glance an *And* node must be used as the root of the tree and each subtree will consist in one of the variables of the target algorithm. Figure 3.1 is an example of such a tree taken from [4] and based on the following function to minimize.

$$(1 - x_1)\Big(\frac{x_2 \sin(\pi(x_2 - x_3))}{x_3} + (x_4 - 2)^2\Big) + 2x_1\Big(|\frac{x_5}{x_3} - 7| + (x_2 x_3 - 1)^2\Big) \quad (3.1)$$

In this tree X_1 is presented as a categorical variable that can take values 0 or 1. If X_1 is assigned to 0, looking at figure 3.1, X_5 doesn't have to be optimized anymore while if it is assigned to 1 X_4 is excluded from the optimization process. This is directly correlated with function 3.1. If X_1 is equal to one the whole left part of the equation is equal to 0 and, as it is the only place where X_4 is present, this explains why the right branch of the tree doesn't include this variable in its nodes : the value of X_4 doesn't matter anymore for the optimization process in this part of the tree. Going further in the tree X_3 is the first encountered node. We can

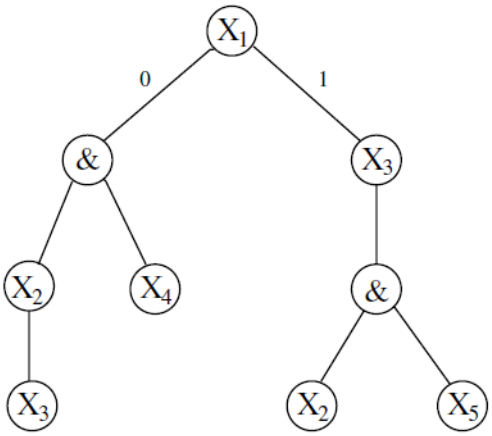


Figure 3.1: An example of a variable tree

see that X_3 is directly linked to X_5 and X_2 in function 3.1. Once X_3 is assigned we can separately optimize X_5 and X_2 in order to minimize the value of the function. This is why an $And-node$ follows the X_3 node to separate X_2 and X_5 once X_3 has been assigned a value. The same reasoning can be applied to the left part of the tree.

This way of representing the individuals of GGA allows to take into account the possible dependencies between the variables of the optimized function. Thereby conditional parameters could be included in the process. These trees also define the basics of the mating procedure that will be described later.

To go further the authors of GGA also imagined a genetic algorithm where two genders are co-existing - what is called a gender-based genetic algorithm. The two genders that are used in GGA are the competitive gender and the non-competitive gender. What they proposed in their paper is to have different selection pressure for each gender, i.e. that the mating rules for one gender are not the same as for the other one. In the context of GGA the individuals that are of the competitive gender must compete to gain the right to mate while the ones from the non-competitive part of the population must not. This idea comes from two observations. First, the runtime of GGA is dependent on the time that it takes to evaluate the fitness of individuals - which is done by running the target algorithm with one specific configuration on at least one particular instance. By dividing the population into two groups this reduces drastically the number of fitness evaluations that must be done as the non-competitive part of the population is not subsect to this fitness evaluation. Second, because it takes time to run the target algorithm with one configuration, GGA is not able to handle a large population size. The idea is then

to use the non-competitive part of the population as a "variety store" where good genes - i.e. good specific parameter values - might exist and could thus integrate the competitive part of the population through the mating process. This allows to have one part that focuses on intensification while the second one serves the purpose of diversification.

Before presenting the framework of GGA the mating process deserves a bit more explanations. By reusing the example shown in figure 3.1 the crossover operation can be described as in figure 3.2. As said before a crossover always happens between an individual from the competitive part of the population - denoted by C - and an one from the non-competitive part - denoted by N. Each individual can be seen as an instantiation of the model given is figure 3.1 : each parameter is replaced by a particular value coming from the domain of the corresponding parameter.

The new individual issued from the crossover is constructed in the following way. The new individual's gender - whether it is a C individual or an N individual - is determined at random. Its genome is then determined from traversing the genomes - i.e. trees - of its parents top-down. A node can be labelled O, C or N. First, if the root is an *And* node or if both parents agree on its value the new individual's root is labelled O - for open -, otherwise it is labelled C or N at random. Then the crossover algorithm looks at the child nodes of the root. If the root has the label C or N then with probability P - which is a parameter of GGA - they inherit from the same label as the root node, otherwise they inherit from the opposite label. If the root node has the label O then its children are processed in the same way as it was processed. This crossover algorithm iterates on each subtree considering each child as the new root until the leaves. Finally, once each node has been labelled, a value is assigned to them. The O-nodes inherit the value from the trees of the two mating partners (as this value is identical for both). The C-nodes inherit the value of the mating individual of gender C while the N-nodes inherit it from the mating individual of gender N.

The framework of GGA can be found in algorithm 3. GGA is based on five parameters :

- X is the percentage of the competitive population that is selected as the best individuals and thus that gain the right to mate,
- P is the probability for a node to be assigned the same label as its parent node in the crossover algorithm that was explained earlier,
- M is the probability for a child to be mutated - an individual that is selected to mutate will be assigned a different value for each of its parameters where

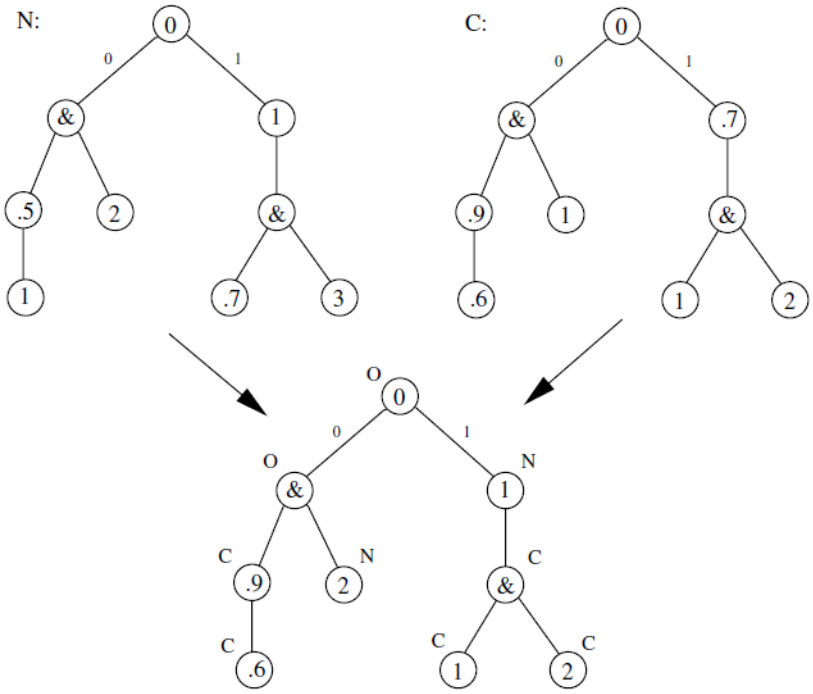


Figure 3.2: The crossover operation when two individuals mate

continuous and integer parameters mutations will be based on Gaussian distributions with the expected value set to the initial value -,

- S is the fraction of the domain of a continuous or integer parameter that is used as the standard deviation of the Gaussian distribution in the mutation process,
- A is the age at which an individual dies - all children issued from a crossover have an age of 0.

Algorithm 3: The GGA framework

```
Input: Algorithm parameters X, P, M, A, S
Output: The best parameter configuration θ_inc found.
population ← initializeRandomly(maxAge=A);
while not TerminationCriterion() do
    bestIndividuals ← getBestCompetitiveIndividuals(perc=X);
    partners ← getRandomNonCompetitiveIndividuals(perc=200/A);
    foreach partner in partners do
        individual ← getRandomIndividualFrom(bestIndividuals);
        child ← mate(individual, partner, inheritanceProb=P);
        with probability M do
            child ← mutate(child, std=S);
        population ← population ∪ {child};
    increaseAge(population);
    foreach individual in population do
        if individual.age > A then
            population ← population \ {individual};
return θ_inc = getBestIndividual(population)
```

Before exploring IF-race, three small additional details about GGA must be clarified. First, the initialization procedure sets the gender, the age - from 1 to A - and the genome - i.e. the values of each parameter - of the individuals uniformly at random. Second, selecting a number of mating partners as $200/A\%$ of the non-competitive part of the population for all the competitive individuals allow to keep the population size quite stable during the process - which is desirable for the execution time of the algorithm to not explode. Finally the way to compare individuals will be by using racing. Racing is the process of running each candidate configuration on the same instances. As the number of instances might be quite large the authors of GGA adopted a technique similar to the FocusedILS mechanism

that evaluates a good configuration more than a bad one. At each mating season - i.e. iteration - a random subset of the whole set of instances is selected for the racing and the size of this subset is increasing through time.

3.2.3 The racing approach : IF-race

The concept of racing is quite simple to describe. All the candidates are competing for the top of the rankings and configurations that are detected to be statistically worse than the incumbent - the best configuration seen so far - are discarded from the race. Before digging into iterated F-race the reader must understand how the F-race algorithm works.

F-race : the building block of IF-race

The framework of F-race is depicted in algorithm 4. The algorithm begins by sampling $nSamples$ configurations from the configuration space and will then build a sequence of nested sets of candidates where each subsequent set contains only the configurations that are considered statistically as good as the incumbent at this point in the procedure. This mechanism is repeated until less than N candidates remain in the race, until a given number of iterations I has been completed or until a predefined computational time budget B has been exhausted.

This algorithm uses the concept of blocking which is the process to evaluate the configurations on the same set of instances. Let's denote the sequence of instances on which the challengers will be evaluated by $i_1, i_2, ..., i_k, ...$ At iteration k all the remaining configurations will be assessed on instance i_k and the obtained objective value for each configuration θ will be appended to its array of observed costs through the race until iteration k-1, $\mathbf{c^{k-1}}(\theta) = (c_1^\theta, c_2^\theta, ..., c_{k-1}^\theta)$, to obtain $\mathbf{c^k}(\theta) = (c_1^\theta, c_2^\theta, ..., c_{k-1}^\theta, c_k^\theta)$. Statistical tests are then conducted to separate the candidates that remain promising from the one that should be dropped based on their vector of costs.

For this purpose F-race uses the Friedman two-way analysis of variance by ranks [9]. This method is a family-wise statistical test which is designed to test the hypothesis H_0 that all the candidates are equivalent. As stated in its name this test is based on ranks. Its usage inside F-race is twofold accordingly to [5]. The first one is that it allows to use a non-parametric test as rankings can be deduced from any list of elements among which an order exists. The second benefit is that it serves as a way of normalizing the costs observed on different instances. No matter the actual costs - which can vary significantly from one instance to another - the configurations will always be compared based on their ranks rather than on the

raw costs.

At step k, if we consider that m candidates remain in the race, the Friedman test assumes that the observed costs are k mutually independent m-variate random variables that are called blocks. One such block exists for each iteration until and including step k : $b_1, b_2, ..., b_k$. Each block b_j contains the costs of the m remaining configurations on instance i_j : $b_1 = (c_1^{\theta_1}, c_1^{\theta_2}, ..., c_1^{\theta_m})$, $b_2 = (c_2^{\theta_1}, c_2^{\theta_2}, ..., c_2^{\theta_m})$, ..., $b_k = (c_k^{\theta_1}, c_k^{\theta_2}, ..., c_k^{\theta_m})$. Withing each block the configurations are ranked based on their cost - in increasing order when minimizing - and ties are broken by setting the rank of the tied configurations to the average of the ranks that would have been assigned to them without the tie. For example two configurations having cost 1.37 and being randomly assigned to rank 3 and 4 by the ordering algorithm will rather be assigned rank 3,5.

Based on these ranks the Friedman statistic is computed and compared to the $1 - \alpha$ quantile of a χ^2 distribution with $m - 1$ degrees of freedom - where α is a parameter that is usually set to 0.05. If it is greater the null hypothesis is rejected, i.e. at least one of the configuration performs better than at least one of the others. This leads to a bunch of pair-wise comparisons between the estimated best configuration - the one having the least sum of ranks - and all the other ones. The candidates that are considered statistically worse than the best configuration are discarded and iteration $k+1$ can begin. Several ways to conduct these pair-wise comparisons exist and the author decided to pick one of the various tests presented in [9]. The details can be found in [5].

Once only two configurations remain in the race the Friedman test reduces to the binomial sign test for two dependent samples [10]. However the author considered that the Wilcoxon matched-pairs signed-ranks test [9] was more powerful and data efficient than the former one based on [11]. They thus adopted this test instead of the Friedman test when only two candidates remain in the race.

The remaining grey part of this algorithm is the initialization. Several methods are proposed in [5] such as the full factorial design or a random initialization. However the iterated F-race method is the initialization method that leads to the IF-race version of the racing algorithm for parameter configuration.

IF-race : an iterated version of F-race

The basic idea of iterated F-race is to use F-race as a subroutine that will be run over the sampled configurations for each iteration of IF-race. Its framework is depicted in algorithm 5. Each iteration of IF-race will call F-race and the result of

Algorithm 4: The F-race framework

```
Input: Algorithm parameters B, I, N
Output: The best parameter configuration θ_inc found.
population ← initialize();
time ← 0.0;
iteration ← 1;
while time < B and iteration < I and population.size() > N do
    if population.size() > 2 then
        H_0 ← Friedman(population);
    else
        H_0 ← Wilcoxon(population);
    if not H_0 then
        bestCandidate ← getBestConfiguration();
        foreach candidate in (population \ {bestCandidate}) do
            worse ← pairwiseTest(candidate, bestCanditate);
            if worse then
                population ← population \ {candidate};
    iteration++;
    time ← updateTime();
return θ_inc = getBestIndividual(population)
```

this call - i.e. the candidates that survived this race - will influence the sampling of the next iterations. The author of IF-race based their work on [7] and modified it slightly to obtain their version of the iterated F-race algorithm. This new version was designed based on several questions that must be asked when designing such an algorithm.

The number of iterations : the authors of IF-race proposed to adapt the number of iterations to perform based on the difficulty of the target algorithm A. In their paper they evaluated the difficulty of an algorithm as the number of parameters that have to be optimized.

The computational budget to allocate to each iteration : As exploration is very important in the first iterations the authors proposed to allocate more time to the first iterations and to decrease it along the process.

The number of configurations to sample at each iteration : The idea is that in the first iterations the configurations will be quite easily separated by the statistical tests and thus fewer executions of the target algorithm A per configuration will be needed in F-race. For the same computation budget B it is then possible to allow more configurations in the first iterations of IF-race than in the last ones where several executions of A will be needed to be able to distinguish good configurations from bad ones.

The termination criterion of F-race : Algorithm 4 generally used $N = 1$ which is not always the best choice to make. In IF-race the authors adapted F-race to allow to use a parameter named N_{min}. They also discarded the I parameter as the goal is to always either obtain a number of elite configurations lower than or equal to N_{min} or use the whole computational budget for this iteration. To set N_{min} the authors propose to adapt it based on the number of parameters of the target algorithm A : the more parameters it has the more important a big enough exploration of the configuration space is.

The sampling of the candidates at each iteration : In the paper all the candidate configurations are sampled randomly according to a probability distribution over the parameter space. The initial probability distribution is a uniform one and it will be modified through the process to bias the distribution towards the survivors of each iteration. Once an iteration of F-race terminates the elite configurations that survived the race are weighted according to their ranks. These configurations will be used to sample $N_{l+1} - N_{elite}$ new candidates for the

next iteration that must have N_{l+1} candidates but will already contain the N_{elite} survivors of the last iteration. The sampling process is done as follows : first one of the elite configuration is selected at random with a probability proportional to its weight. Then a value is sampled for each parameter based on the current distribution and assigned to the parameters of the new candidate.

For numerical parameters the sampling distribution follows a normal distribution $N(x_i, \sigma_i^l)$ where x_i is the value of the selected elite configuration for this parameter and σ_i^l is defined as $\sigma_i^l = v_i * (\frac{1}{N_l})^{\frac{l}{d}}$. v_i is the size of the domain of the parameter, N_l is the number of configurations sampled at iteration l of IF-race and d is the number of parameters of the target algorithm A. This allows to sample a new value centered at the value of the selected elite configuration and which will be closer to this value when the iteration counter increases. This is motivated by the fact that in the last iterations the elite configurations are supposed to be already quite good and thus that only small variations should be observed to obtain better candidates.

For a categorical parameter X_i with n_i levels its distribution will be adapted as follows. Let's suppose that the selected configuration E^z takes value f_i for this parameter. Then the probability distribution used for this parameter at the next iteration $l + 1$ is defined as

$$p_{l+1}(f_j) = p_l(f_j) * (1 - \frac{l}{L}) + I_{j=i} * \frac{l}{L}$$

for $j = 1, ..., n_i$ and where L is the maximum number of iterations of F-race to perform in IF-race and I is an indicator function. In other words the probability to sample value f_i increases and the probability to sample the other values decreases. As these changes depend on l the bias towards the elite configurations become stronger in the last iterations of IF-race.

Once all these decisions have been made, i.e. concrete values or functions to decide the values have been assigned to each parameter of IF-race, the algorithm is fully defined and can be run with the target algorithm A on parameter space X. For example the authors assigned these values as follows.

- Number of iterations : $L = 2 + round(log_2(d))$ where d is the number of parameters of A,
- Time budget at iteration l : $B_l = (B - B_{used})/(L - l + 1)$,
- Number of candidate configurations to sample at iteration l : $N_l = floor(B_l/\mu_l)$ where $\mu_l = 5 + l$,

Algorithm 5: The IF-race framework

Input: Parameter space X, target algorithm A, computational budget B
Output: The best parameter configuration θ_{inc} found.
model ← initialModel(X);
iteration ← 1;
while not *TerminationCriterion()* **do**
    population ← sample(model);
    B-iter ← partOfBudget(B);
    elite ← F-race(population, A, B-iter);
    model ← updateModel(elite);
    iteration++;
return θ_{inc} = *getBestIndividual()*

- Termination of F-race : stop it if at most $N_{min} = 2+round(log_2(d))$ candidates remain.

These values have to be adapted to each target algorithm A and they will be revisited in the experimentation section.

3.3 Model-based methods

As stated before, model-based methods rely on a statistical or machine learning model to select promising configurations on which the target algorithm should be executed. These executions are then used to refine the model iteratively in order to concentrate the search on small parts of the configuration space.

Several models could be used for this purpose. In this thesis two of them will be briefly presented namely the Gaussian process regression model and the random forest regression model. They will be investigated in section 3.3.1.

The SMBO framework [1, 12, 13, 14, 15, 16] and its most advanced algorithm SMAC [1, 15, 16] is a model-based approach that uses a random forest model to predict promising configurations to evaluate.

GGA++ [17] is the cousin of GGA [4] but it brings modifications to it such that GGA++ is considered as a model-based algorithm. This last framework is also using a random forest model to select promising configurations on which to run the target algorithm.

A slightly different version of the SMAC algorithm - actually its predecessor - presented in [16] and called *ActiveConfigurator* was designed to accept one of two possible models : either a random forest model - the algorithm was named AC(RF) - or a Gaussian process model - AC(GP). AC(RF) led to the emergence of SMAC but AC(GP) was slightly abandoned. This is why it will not be presented in details in this thesis. However the Gaussian process regression model will be described in the next section as this powerful model could be included in a future version of the configuration library designed alongside this thesis.

3.3.1 Models overview

Random Forest regression

As the reader might already know a random forest is a machine learning tool initially designed for classification that is based on random decision trees. A decision tree is built to split the labeled training data into several nodes through a series of simple tests on the features that compose the data. In each node of the tree one feature is first selected to split the data upon. Then a particular value of this feature is set as either the threshold - for numerical variables - or the value - for categorical variables - that will split the data into two nodes. If a numerical feature has been selected then the left node might contain all the samples that have a value for this feature greater than the threshold and the right node might contain the remaining samples. The same reasoning can be done for categorical variables except that the samples will be split on a particular value rather than on a threshold.

The selection of the feature to split the data upon and the value of this feature in each node is done in order for the resulting nodes to be as less impure as possible where, roughly speaking, impurity is measured as the diversity that exists in a certain node about the values of the target variable of the data - i.e. the class in a classification problem. So, for example, if a node only contains training samples that are of the class "dog" its impurity is equal to zero. The goal will then be to build a tree that allows, for a new unseen data sample, to classify it by going through the nodes from the root of the tree following the result of each test. The sample will be classified with the same class as the data samples that compose the leaf - i.e. ending node - in which the new sample fell into.

For regression problems - which is the case here where the target variable is the objective value obtained when running the target algorithm A on a particular configuration, instance pair - the impurity is computed differently but the idea behind the tests that allow to split the training data into several categories remains the same. A new unseen sample data will be assigned a value being, for example,

the mean of the values of the target variable of the data samples present in the leaf it fell into.

Not only one method to build regression or classification trees exists and each of them are different in terms of parameters and features they use to achieve the task. This description aims at giving only a brief overview of what a regression tree is but the interested reader might look at [18] for further information.

In the case of automatic algorithm configuration the training data will be composed of the configuration on which the target algorithm A was run and the objective value obtained by this execution - which will be our target variable. Additionally, certain methods - like SMAC - also allow to use the instance on which the configuration was tested or more precisely features of the instance - for example the number of trucks that will arrive at the warehouse in our case. Then, after the regression tree has been built, new configurations - or new configuration, instance pairs - can be evaluated through the model without executing the target algorithm A. This then gives an idea of the performance of an unseen configuration - according to the model which depends on the data it was fed with - at least cost.

One very-well known problem with decision or regression trees is overfitting. Roughly speaking, overfitting is the fact that the model will try to fit the training data so well that it doesn't generalize to unseen data. This then gives poor estimations of the target variable especially in the case of regression trees. Plenty of methods to overcome this problem exist but most of them go beyond the topic of this thesis - see [18] for more information. One solution to solve this issue is to build several trees instead of a single one to obtain an estimation of the target variable : a random forest.

A random forest is thus a collection of random decision or regression trees. Random is used in the sense that each tree will be built with a random subset of the whole training data and thus each tree will tend to be different from the other ones in the random forest. This trick allows to gather multiple predictions - one for each tree in the forest - and to aggregate them into a more generalized prediction hence avoiding overfitting to be highly pronounced. This aggregation will typically consist in the mean of the predictions - for regression - or of a majority vote among the obtained classes - for classification.

Random forest is thus a powerful method for automatic algorithm configuration as it combines both categorical and numerical variables into a model that allows to predict the performance of unseen configurations without executing the target

algorithm and hence gaining a big amount of time through the configuration process. It is the model that is implemented in SMAC and GGA++ - though slightly modified in the latter - that will be presented in the next sections.

Gaussian process regression

Gaussian process regression models aims at approximating the probability to observe a certain value for the target variable of an unseen data sample based on its predictive variables - i.e. all the other features that form a data sample - and all the training samples. As its name suggests GP models are based on Gaussian distributions but also on parameterized kernel functions, i.e. a function that computes the similarity between two data points - in our case between two configurations.

This kernel can be of various forms but as stated in [16] the most common ones have the following form :

$$K(\theta_i, \theta_j) = exp\Big[\sum_{l=1}^{d}(-\lambda_j(\theta_{il} - \theta_{jl})^2)\Big],$$

where $\lambda_1, ..., \lambda_d$ are the kernel parameters. These kernel parameters along with σ^2, the observation noise variance, constitute the hyper-parameters of the Gaussian process model and are typically set by maximizing the marginal likelihood - i.e. the probability to observe the set of target variables of the training data set - with a gradient-based optimizer. For target algorithm A which doesn't require an observation noise variance parameter, σ^2 can be set to 0, as done in the famous DACE model used in EGO, one of the predecessors of SMAC [12].

This parameterized kernel allows to build a model on any kind of data and is perfectly suited for predictions of the results of the target algorithm on a specific instance in the algorithm configuration problem. Once its parameters are set it is possible to obtain such predictions by using the following Gaussian distribution :

$$p(o_{n+1}|\theta_{n+1}, \theta_{1:n}, o_{1:n}) = \mathcal{N}(o_{n+1}|\mathbf{k_*}^T[\mathbf{K} + \sigma^2\mathbf{I}]^{-1}o_{1:n}, k_{**} - \mathbf{k}_*^T[\mathbf{K} + \sigma^2\mathbf{I}]^{-1}),$$

where o_{n+1} is the value to predict, θ_{n+1} are the predictive variables of the new unseen data sample, $\theta_{1:n}$ and $o_{1:n}$ are the training samples and the corresponding values of the target variable, $\mathbf{I}$ is is the n-dimensional identity matrix, $\mathbf{K}$ is the matrix that gathers the results of applying the kernel between each pair of samples in the training data set, $\mathbf{k}_*$ is the vector that gathers the same results between each of the training data sample and the new unseen data sample and $k_{**} = k(\theta_{n+1}, \theta_{n+1}) + \sigma^2$. Finally the notation $p(a|b) = \mathcal{N}(a|\mu, \sigma)$ might be a bit

confusing but is simply there to denote that the conditional distribution of a given b is a Gaussian with mean μ and covariance matrix σ. The reader might look at [19] for a full derivation.

The issue with Gaussian process regression models inside parameter tuning algorithms is that it is computationally expensive to inverse a matrix and to fit the hyper-parameters of the model each time it has to be learnt - i.e. at each iteration of the algorithm. This is why approximations of this model exist. One of them is described in [16] at section 11.3.2. As this model will not be part of any algorithms presented in this thesis the details of this approximation will not be given here.

3.3.2 The SMBO framework : SMAC

SMAC is one of the most - or even the most - widespread model-based method in the automatic algorithm configuration community. It has been tested/adopted in a large amount of papers - see for example [15, 17] or numerous papers in [20] - and has proven to work pretty well in various different contexts.

This is not a coincidence as SMAC is the most advanced algorithm of the SMBO framework which has quite a long history in the domain. Starting from the EGO algorithm [12] which used the well-known DACE model - a noise-free model - and going through SPO, SKO, SPO+, TB-SPO, SPO* and much more - see for example [13, 14, 16] - this framework ends its journey - at least to my knowledge - in SMAC, an algorithm that uses the best features of all its predecessors to build a general automatic configuration procedure that is capable of managing multiple instances, numerical and categorical variables, conditional parameters and a rather complex model - random forest - to guide the search through any parameter space.

The SMBO framework is depicted in algorithm 6. In this framework the cost metric $\hat{c}$ might be the mean or the median for example, $\mathbf{R}$ is the run history, i.e. the training data of the random forest model which is the pairs of configuration and objective value obtained when running the target algorithm A with this configuration. In the main loop $\mathcal{M}$ is the learnt model for this iteration, $\vec{\mathbf{\Theta}}_{new}$ is a list of promising configurations that might be evaluated in $Intensify$ and t_{fit} and t_{select} are the times that were required to fit the model and to select the promising configurations respectively.

SMAC is a direct implementation of this framework where each of the functions - $Initialize$, $FitModel$, $SelectConfigurations$ and $Intensify$ - are implemented in a specific way. The full implementations or descriptions of these procedures can be found in [15] but only a short description of each of them will be given below.

Algorithm 6: The SMBO framework

Input: Parameter space $\mathbf{\Theta}$, target algorithm A, instance set Π, cost metric $\hat{c}$
Output: The best parameter configuration θ_{inc} found.
$[\mathbf{R}, \theta_{inc}] \leftarrow$ Initialiaze($\mathbf{\Theta}$, Π);
while *total time budget for configuration not exhausted* **do**
  $[\mathcal{M}, t_{fit}] \leftarrow$ FitModel($\mathbf{R}$);
  $[\vec{\mathbf{\Theta}}_{new}, t_{select}] \leftarrow$ SelectConfigurations($\mathcal{M}, \theta_{inc}, \mathbf{\Theta}$);
  $[\mathbf{R}, \theta_{inc}] \leftarrow$ Intensify($\vec{\mathbf{\Theta}}_{new}, \theta_{inc}, \mathcal{M}, \mathbf{R}, t_{fit} + t_{select}, \Pi, \hat{c}$);
return θ_{inc}

The *Initialize* procedure is the simplest one to describe as it just runs the target algorithm A with a default parameter configuration given by the user - or a random one if none is provided - on a randomly selected instance among the instance set Π. $FitModel$ will build a random forest model based on the run history $\mathbf{R}$. To build a random forest regression model a numerous amount of parameters have to be set - or arbitrarily chosen. The author of SMAC suggests the following ones :

- The number of trees to build in the forest, B, is set to 10 to keep the computational overhead small,
- The number of data points to build each tree upon are sampled with repetition from the whole data set,
- A random subset of the parameters are considered eligible to split a node upon at each split point. This value is set to 5/6 of the number of variables by default,
- The minimum number of data points required to be in a leaf to not continue splitting the tree is set to $n_{min} = 10$.

Of course the values of these parameters can be adapted for the application at hand or depending on the library the user might use to build the forest.

Once the model is built a new configuration that was not run with the target algorithm A can be evaluated through it. In SMAC not only the mean of the prediction of each tree will be returned but also the variance σ^2 obtained in the predictions across the trees. These values will then be used in $SelectConfigurations$ to select promising configurations. It is also important to note that the SMAC authors used

a logarithmic transformation of the target variable to build the successive random forest models. This was done because this paper - and several of its predecessors - studied the running time of the target algorithm A rather than the solution returned by the algorithm, and it was shown that a logarithmic transformation was improving the quality of the model in this particular case [21]. However in our case we will not use this transformation as we study the solution returned by our simulation rather than its execution time.

A mechanism introduced by SMAC to improve the SMBO framework is the ability to evaluate the target algorithm on sets of instances while its predecessors could only optimize the configuration based on a single instance. However this implies that the data used to feed the predictive model must include information about the instance(s) on which each configuration was run in the run history or the model might be biased towards configurations that were run on the simplest instances. To achieve this, SMAC relies on instance features which are either already built or must be built by the user depending on which target algorithm A has to be optimized.

For known configuration problems - for example optimizing the parameters of SAT solvers - plenty of examples of features that can be used to describe an instance exist. However it is not always the case and the authors of SMAC gave some general features that could be used in any situation. For example the instance size or the solution given by running the algorithm with the default configuration on this instance could be used to describe it. The random forest regression models will then be built based on two sets of predictive variables, one concerning the parameters of the target algorithm A and one concerning the instance features chosen by the user. To evaluate a new unseen configuration the model will predict its mean performance across all instances present in the instance set Π for each tree and then these results will be aggregated to output the general mean performance μ and variance σ^2 of the evaluated configuration.

As explained before these two values will be used in $SelectConfigurations$ to select promising configurations that will then be assessed by running the target algorithm A to refine the model in the next iterations. This procedure will use the Expected Improvement Criterion - EIC - to achieve this task.

Several variants of this criterion exist but each of them tries to estimate to which extent a certain configuration would improve the current best solution and thus would possibly become the new incumbent. This criterion will be large for configurations with a low predicted target algorithm solution μ - in the case

where we minimize it - and for those with a high predicted uncertainty σ^2. The SMAC authors used the $\mathrm{E}[I_{exp}]$ criterion described in [14] because they used log-transformed running times of the target algorithm A but $\mathrm{E}[I^2]$ is a good choice when optimizing the raw target variable as it accounts for the variance among the predictions which can be useful especially when the target algorithm A is randomized. This criterion is computed as follows :

$$E[I^2] = \sigma^2 * [(u^2 + 1) * \mathbf{\Phi}(u) + u * \phi(u)],$$

where $u = \frac{f_{min} - \mu}{\sigma}$ (f_{min} is the current best solution) and ϕ and $\mathbf{\Phi}$ denote the probability density function and cumulative distribution function of a standard normal distribution, respectively. Thanks to this criterion and to the random forest regression model the performance of a new unseen configuration can be efficiently evaluated.

What $SelectConfigurations$ does is first to select the ten best configurations among all the ones that are present in the run history based on their EICs. It then starts a local search procedure at each of them to find local optima - always based on their EICs - among their neighbors by using a best improvement search. In order to be able to increase the exploration part of the algorithm they join the evaluations of 10 000 random configurations to the 10 previously computed EICs and then sort them in decreasing order. The corresponding ordered list of configurations is then returned as $\vec{\mathbf{\Theta}}_{new}$ and passed to $Intensify$.

These two preceding steps - $FitModel$ and $SelectConfigurations$ - take time and to avoid being stuck in an iteration evaluating the 10 010 configurations output by $SelectConfigurations$ in $Intensify$, the latter procedure uses a maximum time of $t_{fit} + t_{select}$ to evaluate as many configurations as possible in this list starting from the best ones. The problem is that in general the best configurations will always be the same ones - typically the ones on which the local search procedures were launched. In order to include some exploration in SMAC the authors interleave the evaluation of these best configurations with random ones. This will allow to induce a certain diversity of configurations in the run history and hence it will increase the chance to pick the good ones to initialize the local search in $SelectConfigurations$. This will thus allow to escape local optima. That's why $Intensify$ will always evaluate at least two configurations on the target algorithm A even if the time needed for this exceeds $t_{fit} + t_{select}$. This permits to always consider one good configuration and one random configuration at each iteration.

It is important to note that SMAC uses a blocking mechanism on instances, i.e. that configurations will always be compared based on the same instances - although

these are picked at random. Moreover the number of instances on which the incumbent has been evaluated will always grow at each iteration - until a specified threshold $maxR$ - by executing it on a new instance each time a new configuration is assessed inside $Intensify$. Once this is done the challenger configuration will be evaluated on an increasing number of instances - starting from one - on which the current incumbent was run until either the challenger performs worse than the current incumbent or until it has been assessed on all the instances on which the current incumbent has been evaluated itself and it achieves better or equal performances, in which case the challenger becomes the new incumbent.

The rejection mechanism can be considered as very aggressive as the challenger is rejected as soon as it is detected to perform worse than the current incumbent. However this is mitigated by the fact that once a challenger enters $Intensify$ it adds its run(s) to the run history and hence gains the chance to be selected again later by $SelectConfigurations$ in the ten best configurations seen so far. If this is the case the challenger will be evaluated again on a - or several - new instance(s) and the new resulting estimated value of the solution for this configuration could be lower - if we consider that we are minimizing the objective - than the one of the incumbent. If this is not the case then it means the algorithm potentially found even better configurations that must be assessed to refine the model and thus, given a sufficient amount of time, SMAC would eventually find the global optimum.

All these procedures follow an iterative process in which the model will always become better to evaluate potentially good configurations and these configurations will always be evaluated on more instances than in the previous iterations, reducing the uncertainty about their cost estimates. Once the main loop of SMAC terminates the current incumbent is returned along with its estimated objective value.

3.3.3 Extension of the genetic algorithm approach : GGA++

GGA++ is a model-based algorithm which is an extension of GGA described in section 3.2.2. As for SMAC, GGA++ uses random forest regression models to predict which configurations would perform well on the target algorithm A. But in GGA++ it is not the only usage of the model. Actually the authors of GGA++ [17] brings three new ideas compared to GGA.

The first idea is to use genetic engineering to find, once two parents have been chosen to produce offspring together, the combination of the parents' genes that results in the fittest offspring as predicted by the random forest model. For this purpose the authors suggest to use a limited focused search in the restricted parameter space spanned by the two parents' genomes. It means that for any

parameter only a maximum of two values will be tried - and only one if both parents agree on the value of a certain parameter. However, even in this limited parameter space, this is computationally expensive as explained in the paper. This is why the authors proposed a targeted sampling method to reduce the computational overhead induced by genetic engineering.

For each tree in the forest the authors of the paper gather the leaves in which any potential offspring of the two selected parents could fall into. For each of such leaf they then randomly sample m of full assignments of the parameters. These assignments are subject to some constraints depending on the parents' genomes as the sampled configurations must respect the path that led to this leaf in the tree hence respecting the different constraints that each node adds to the parameter space - for example one node in the path might need $X_1 \geq 25.6$. The non-constrained parameters can be chosen at will. Once a genome has been fully sampled it is evaluated on the entire random forest to obtain its estimated objective value. This is repeated for all leaves of all the trees and the overall best genome is returned as the fittest offspring.

One issue with this mechanism is that it intensifies the search. This might lead to a loss of diversity in the genetic algorithm which is very important to not get lost in local optima. To alleviate this problem the authors of GGA++ proposed to remove a slightly larger percentage of the population in each generation - compared to GGA - and to replace these individuals with randomly sampled ones.

The second extension that is added to GGA to obtain GGA++ is to slightly modify the way the random forest is learnt in order to focus the forecasting on the areas of the parameter space where it is expected to have high-performance configurations. The idea behind this mechanism is that it is not needed to learn the whole function giving the performance of a configuration based on the values of its parameters but rather that this function should be learnt only where it is interesting to learn it.

To achieve this the authors of GGA++ introduce an alternative splitting procedure in the random forest model that aims at having higher "resolution" of the trees in areas where performance is high. For this purpose the splitting procedure will aim at separating the top-performing configurations from the remaining ones as best as possible. The percentage of the training data that will be considered as the top-performers is set to $q = 10\%$. The goal will then be to maximize, in each partition, the ratio of high performances over lower performances. This splitting procedure might be seen as a special version of the one used in classical random

forest regression where the impurity would be computed as a quantification of the mixture of low-performances configurations with high-performances ones. The full details of this new splitting procedure can be found in [17].

The last contribution that GGA++ brings to GGA is to use sexual selection before the mating process begins. In GGA, after the top-performing X% of the competitive population has been selected to mate, 200/A% of the non-competitive part of the population was selected at random as the mating partners. Then each of these randomly selected individuals was randomly assigned to one of the competitive individual to obtain a mating couple. The idea behind sexual selection is to allow the competitive part of the population to choose their partners based on their attractiveness which is computed as their predicted performances based on the learnt random forest model.

The procedure is organized as follows. First the X% best individuals of the competitive part of the population are selected. Then each of them are assigned a number k of mating partners it can have for the current mating season where the sum of all the k's must be equal to 200/A% of the non-competitive population's size - let's denote this number by n. This assignment is done uniformly at random by assigning each number from 1 to n to one of the competitive survivors. Each of these survivors that have k numbers assigned to them must choose k mating partners among the non-competitive part of the population. To simulate sexual selection these individuals are assigned a probability to be selected proportional to their estimated performance given by the random forest. Once each survivor that has $k > 0$ has been assigned k partners the mating season can begin following the same procedure as in GGA but with genetic engineering as an additional process.

3.4 Other methods

This section aims at giving a broad overview of the automatic configuration algorithms that were not presented in the last sections.

3.4.1 Similar methods

Plenty of methods that exist in the literature were not presented in this thesis either because they were not adapted for the problem at hand or because they seem to be slightly outclassed by the state-of-the-art.

Other methods based on the SMBO framework Although they were already aforementioned the vast diversity of methods that led to SMAC were not

presented in this thesis as, unfortunately, most of them are not well suited for the problem at hand. Starting from EGO [12] which was designed to only support numerical parameters optimization for noise-free function, the succeeding methods all added a new feature to finally end up in SMAC.

EGO laid the foundations for model-based automatic configuration methods. It uses the noise-free Gaussian DACE model to build its response surface model exactly as SMAC uses random forest regression for this purpose. It also uses a simple initialization mechanism based on the Latin Hypercube Design which aims at finding which parts of the configuration space are the most promising on a very low granularity (typically two or three values per parameter). For example it can be used to detect that, in general, a configuration having a high value for parameter X_1 would tend to perform well. The aim is to restrict the parameter space that the search will focus on.

SPO [13] introduced a new regression model based on a polynomial of order two coupled with a noise-free Gaussian correlation function. It also introduced the $\mathrm{E}[I^2]$ expected improvement criterion in the automatic algorithm configuration problem and a new intensification mechanism that increases the number of runs on which each configuration is tested as the algorithm progresses towards the optimum. SKO [22] is also an extension of EGO but that brings the idea that a better expected improvement criterion can lead to the optimization of stochastic algorithms.

TB-SPO [23] added the management of a time budget inside the SMBO framework in order to better spread this time between the different parts of the framework. SPO+ [14] brings the idea that the target variable that is predicted by the surface response model can be modified before fitting the model to obtain a better version of it - for example the log-transformation for positive functions. It also improved the intensification mechanism once again to never choose a new incumbent if it was not tested on at least as many runs as the current incumbent.

SPO* [16] is an improved version of SPO+ that better handles the time spent in each function of the SMBO framework and that introduces random configurations in the $SelectConfigurations$ function to improve the exploration of the parameter space. It was also one of the first methods to use the $\mathrm{E}[I_{exp}]$ criterion.

Finally first ROAR [16] and then SMAC [15] added the ability to block on instances, seed pairs in order for the algorithms to handle sets of instances. SMAC, as seen previously, also introduced the random forest regression model to handle categorical parameters, instance features to be able to fit the model on configuration,

instance pairs and a local search procedure to apply to the best configurations seen so far in order to improve the intensification part of the algorithm. These features were also inspired from paramILS [3] another framework that was presented in section 3.2.1.

CALIBRA CALIBRA [24] is a parameter configuration software based on the Taguchi fractional factorial experimental design coupled with a local search procedure. The idea is typically to reduce the domain of each parameter until a single value remains. This method can only handle 5 parameters but the software can manage 15 parameters among which 10 of them will be assigned a fixed value before starting the main algorithm on the five remaining parameters - typically the ones that are the most difficult to set.

3.4.2 Per-instance methods

A category of automatic parameter configuration algorithms that was not discussed in this thesis concerns the methods that adapt the parameters to use for the target algorithm A depending on the instance that has to be solved. Such methods generally uses the same idea as presented earlier in SMAC which is to describe the instances with features that are supposed to provide an insight of the difficulty of an instance - or more generally of its characteristics.

ISAC [25] is an algorithm based on this method. It combines GGA with the stochastic offline programming paradigm [26] and an algorithm to compute instance features. It can handle all types of parameters and sets of training instances. Then, when the model has been learnt, it will return for any given unseen instance a configuration that is supposed to work well on this instance but not necessarily on other instances - this is the main difference with more classical methods. According to its authors their method is capable of obtaining better results on several target algorithms than other methods presented in this thesis. However one could question the generality of this method as it is highly dependent on the features that are computed on the instances which are themselves target algorithm dependent.

3.4.3 Multi-objective methods

This thesis could have tackled multi-objective methods alongside with the other methods that were presented in the previous sections. However the only accomplished methods - to my knowledge at least - that exist in the literature are MO-basicILS and MO-focusedILS [27] - adaptations of BasicILS and FocusedILS for multi-objective automatic algorithm configuration - and MAC [28].

While our application would have needed such a method to be able to optimize both the service time of the trucks and the monetary costs of the warehouse more easily, it was decided to focus only on single-objective methods as it allows to compare them on an equal footing. However in future adaptations of the aforementioned algorithms we will certainly see mechanisms that will allow to handle multiple objectives.

3.4.4 Helper methods

Besides the big classes of algorithms that exist to tackle the algorithm configuration problem, helper methods have been developed along the way to improve the frameworks presented in the previous sections. These methods are generally inspired by the manual configuration strategies that were used when automatic algorithm configuration did not exist yet. They are especially used to initialize the different search strategies and has shown to be so much helpful in some cases [29] that they could even be - and have been - used for automatic parameter tuning in its whole process as well.

Any methods or heuristics used in classic optimization applications could also be included in further algorithms that aim at automatically finding parameters for a specific target algorithm as local search was included into several existing ones - paramILS and SMAC for example. Statistical or machine learning tools also seem to be promising allies for parameter tuning methods as we saw with the random forest regression model used in GGA++ and SMAC and with some statistical tools used in IF-race.

3.5 Discussion

Every automatic configuration algorithm has strengths and weaknesses. This section aims at comparing those that were presented in this thesis and giving some insights about what could be improved in these methods to build even better procedures. Three aspects will be studied in this section : the actual performances of the methods that were presented, their capabilities and the tools that might be missing in the field.

3.5.1 How do they perform in practice ?

The first aspect about which we have to be critical is the actual performances of these algorithms as presented by the authors in their papers. This subsection will

give a broad overview of what was tested by the authors and how their method competes with others.

ParamILS ParamILS has two variants, BasicILS(N) and FocusedILS, that can both be extended with the trajectory-preserving - TP - or agressive capping - aggr - techniques presented earlier. The relatively broad tests achieved in [3] - both about relatively simple examples and real-life applications - tend to show that both of these implementations are useful depending on the application. For some scenarios BasicILS(N) performs better than FocusedILS - with N being sufficiently large as it is the number of instances on which each configuration might be evaluated - and inversely for others. The authors also showed that both of these techniques are performing better than random search techniques and that the configurations obtained at the end of these procedures perform better than the default configurations of the target algorithms they studied. Moreover their capping techniques are effective as they allow to perform, with the same amount of time, more iterations than without using them hence leading to better configurations as well. These two algorithms have also demonstrated their strengths in a various number of application-specific papers where their authors used paramILS to configure their own algorithms. The only missing part is a comparison of paramILS with other similar algorithms in the literature but as it was one of the first algorithm to define such a complete procedure this might be understandable.

GGA GGA is an improved genetic algorithms designed for automatic algorithm configuration purposes. Their authors showed in [4] that it first substantially outperforms classic genetic algorithms for this task - both in terms of results and execution time - where there is no gender separation and a less performing mating procedure. Then they showed through four different applications that GGA generally outperforms paramILS - although they did not mention which version of paramILS they used and if it was used with or without capping techniques thus it must be taken with carefulness. Finally they showed that GGA tends to find good configurations faster than paramILS at least in the first hour of the search. In a later paper that presented the SMAC algorithm [15], a broad comparison of different state-of-the-art procedures that existed at that time allows to detect that GGA actually doesn't seem to achieve substantially better performances than paramILS as stated before but rather achieve similar or worse performances. This might be due to the fact that the comparison done in SMAC uses the most advanced version of FocusedILS at that time - version 2.3 - which could be an improved version over the one used in the paper of GGA. However this confirms the fact that GGA might actually be outclassed by paramILS.

IF-race IF-race is an extension of F-race that typically uses an iterative process where it calls F-race at each iteration hence allowing to start it at multiple different configuration points in the configuration space. The authors of [5] showed that their algorithm was performing substantially better than classical F-race algorithms which in turn achieves better performances than t-test-based racing techniques. However they neither showed that their procedure allows to find better parameter values than the default ones of a certain target algorithm A nor that they compete with other automatic configuration algorithms but as for paramILS, IF-race is one of the oldest complete method in the domain and thus this might be understandable.

SMAC SMAC seems to be the most advanced method that was presented in this thesis - it is also the newest one with GGA++. In the paper that introduces this new algorithm [15], the authors compared their method with GGA and focusedILS - version 2.3 - but also with ROAR and TB-SPO where ROAR is a version of SMAC that doesn't use a response-surface model but rather randomly select new configurations to evaluate and TB-SPO is an improved version of SPO - a predecessor of SMAC in the SMBO framework - with a time bounding mechanism - see section 3.4.1. In various experiments they showed that SMAC either outperforms or at least performs as well as all these other automatic configuration algorithms in both scenarios where only one instance is used or multiple ones are provided.

GGA++ This procedure was inspired by GGA and SMAC and combines the advantages of a genetic algorithms for parameter tuning problems and a model-based one to build a genetic model-based algorithm for automatic configuration. The authors introduced three new features in their paper [17] : genetically engineered offspring, sexual selection and modified random forests model to lead the search towards high-performing parts of the parameter space. In the paper they first showed that genetically engineering offspring is leading to a boost of performances in a rather simple scenario - this was done with a low time complexity example in order to allow this feature to be deeply used and thus tested. However when this cannot be fully applied because of time constraints one might ask if genetically engineering offspring would lead to this substantial amount of improvements as well. They also showed that introducing more randomness when using this technique is necessary to balance exploration and intensification. After that they showed that their new random forest model was greatly improving the performances on two highly sophisticated SAT solvers over not using it or using a more classical version of random forests. They also showed that using sexual selection on top of the two previously mentioned features was detrimental. As stated in the paper, this is probably due to the fact that this feature induces an indirect competition in the non-competitive part of the population and thus undermines its ability to serve

as a diversity store, disrupting the exploration/intensification balance. Finally they compared GGA++ - without sexual selection - with GGA and SMAC and they concluded that it is performing better than its pairs. However although it is true that a small improvement exists when looking at these results one would rather say that it achieves similar performances with both of them as the mentioned improvement is not that big and was observed on only two different scenarios.

3.5.2 What are their capabilities ?

Another aspect which is worth mentioning is the capabilities of these methods. This subsection studies mainly the restrictions that are induced by each of these techniques but also the strengths of each of them.

ParamILS BasicILS(N) and FocusedILS both have one main limitation which is the fact that they can't handle continuous parameters. This means that each such parameter has to be discretized leading to the problem of choosing which type of discretization has to be used. However these techniques are for now the only one that allow to handle multiple objectives - at least to my knowledge - with their variants MO-basicILS and MO-paramILS. This will probably be a feature of interest to integrate in the other algorithms presented in this thesis.

GGA The main drawback of GGA might be that it has several parameters that need to be tuned themselves such as the percentage of the population that must be selected for having the right to mate. Moreover it introduces parameter trees which must be built by the user introducing even more choices to make. Although this makes GGA adjustable to the application at hand it might be confusing for the unaware user. However this allows the algorithm to have a simple way to represent dependent parameters and to handle continuous ones.

IF-race IF-race as presented in [5] has the same drawback as GGA : it is highly adaptive. However it is not the number of parameters that is problematic but rather the number of ways that exist to define them. IF-race could rather be seen as a framework rather than a particular implementation as the way it should manage the time budget allocated to each iteration or the number of samples to draw in each of them is highly dependant on the problem at hand. Although its authors proposed generic guidelines for these choices this makes IF-race an algorithm for which it is difficult to design an overall good implementation that would be fitted for any target algorithm. Moreover, as it relies on statistical tests, one might find it difficult to set the significance level at a good value - although the classical 0.05 value seems to work well in a vast variety of applications.

SMAC SMAC has the interesting capability to rely on a response surface model which is definitely helpful to automatically find parts of the parameter space that are promising. The main drawback of this is that it makes the algorithm deeply configurable as it introduces a random forest regression model which itself is already hard to configure. On top of this it also adds the possibility to handle instance features hence adding even more tuning capabilities. While this could be considered as an essential aspect for experts it might makes SMAC a difficult tool to use for newcomers or for people that just wants to use an automatic parameter tuning procedure to optimize their brand new highly parameterized algorithm. This could give rise to the emergence of self-tuning automatic parameter configuration algorithms or at least to the incorporation of simpler ones into sophisticated ones in order for the user to not worry about too much design choices.

GGA++ GGA++ encounters the same weaknesses as GGA or SMAC as it combines both of them in a certain way. However it points out the fact that getting inspiration from other state-of-the-art method to design a new algorithm for the same purpose might be highly beneficial. As other examples the authors of paramILS also mentioned in [3] that they would like to integrate statistical tests into their framework to make it an even better software and the authors of SMAC [15] are looking to integrate adaptive capping techniques to their method and to develop a per-instance version of it.

3.5.3 What is missing ?

In this final subsection I wanted to conclude on what seems to be missing in the field of automatic configurators at the time of writing. The main aspect that should be included in further versions of the aforementioned algorithms is probably the capability to handle multiple objectives through the configuration process. Although the authors generally compared their methods based on the objective to reduce the execution time of the target algorithms the industry might be more interested about methods to also optimize solution outputs or solution costs. A parameter tuning algorithm that would allow to optimize both of them would be a bargain for such users.

Another aspect that would deserve to be improved and that was already mentioned earlier is the self-tuning capability of such procedures. This would allow less experienced people to use them more easily as for now the vast majority of the state-of-the-art methods are presented with a bunch of parameters that must be adapted to the problem at hand. The main problem that might slow down or even make this improvement impossible to develop is the computational overhead

that such a feature would induce. Configurators are already slow to execute on large parameter spaces thus configuring the configurator itself might simply take too much time for it to become a reality - at least for now.

Other aspects might certainly be improved in a near future but these two features are, in my opinion, of a particular interest and that's why they were mentioned in this subsection.

3.6 Library implementations

The ultimate goal of this literature review was to identify which algorithms should be studied in more details for the application at hand, namely the automatic design of a logistical warehouse. Looking back at table 2.2 six parameters are under study in this thesis : four integer ones, one categorical one and one continuous one. It means that only the state-of-the art algorithms can be used in our case as only these can handle mixed numerical and categorical parameter spaces.

Ideally we would have studied multi-objective procedures but unfortunately it will be needed to rather rely on an aggregation of the different objectives - the average service time of the trucks and the monetary costs of the warehouse - through a weighted sum as it seems better to compare several different algorithms on an equal footing than to only implement one or two of them that handle multi-objective objective functions.

These design restrictions coupled with the fact that different ways of doing automatic parameter tuning exist leads us to implementing four different algorithms, namely FocusedILS with capping, IF-race, GGA and SMAC. GGA++ is not implemented in the library for two reasons. First it needs a custom random forest regression model that doesn't exist in the available libraries in Java and second the improvements achieved by this technique was judged too small to justify a deeper study given the time restrictions that govern this thesis.

Before briefly describing the concrete implementations - as some of them are slightly adapted over the descriptions given in chapter 3 - the reader might want to know that the library only implemented a thread-free version of the aforementioned algorithms. In the future, parts of them could become multi-threaded as done in some of the papers that described these methods.

FocusedILS The FocusedILS algorithm was implemented as described in [3]. It comes with four parameters plus the ability to decide between aggressive capping -

the challengers are compared to the current incumbent - and trajectory-preserving capping - the challengers are compared to the best configuration seen in the current iteration. These five parameters are the number of initial configurations to assess - with a default to ten configurations -, the number of perturbations to apply to the configurations during the local search - with a default to three perturbations -, the restart probability - default to 0.01 - and the bound multiplier that is used when using aggressive capping - with a default to two. The default values of the parameters are taken from [3] but doesn't guarantee good results when used with our target algorithm. This is why each of the implemented algorithms have at least two constructors for the user to be able to change these parameters at will.

IF-race As stated before IF-race is not an algorithm in its essence but rather a framework for which several questions have to be studied in order to lead to a particular implementation - cfr section 3.2.3. While the mechanisms that determine the algorithm remained the same the default parameter values were changed to obtain better performances when using them. These are ten for the maximum number of iterations, one for the initial value of μ - with an increment of one at each iteration - and five for the maximum number of survivors in each iteration. Finally, as the Friedman statistical test required to hardcode some values of the quantiles of the distribution used for its hypothesis testing tests a default value of α was set to 0.05 and can't be changed from the API. This significance level is the most widely used in the statistics literature.

GGA GGA uses variable trees to represent the different configurations. If not given by the user the implementation will build a default tree that has only two levels : an and-node as the root of the tree leading to all the parameters at the first level - i.e. we consider that all the parameters are independent and hence can be optimized independently. Except for this the algorithm is exactly the same as given in section 3.2.2 and its parameters - X, P, M, S, A as well as the initial population size and the initial number of instances on which the configurations are run - have default value of 0.1, 0.9, 0.1, 0.1, 3, 30 and 1 respectively. The intensification mechanism increments the number of instances to evaluate the configurations on by two at each iteration but we could imagine other intensification schemes like a doubling scheme for example.

SMAC The model-based algorithms and in particular SMAC have a bunch of parameters. In the library implementation these are first the maximum number of instances on which the incumbent will be evaluated, the maximum number of challengers at each iteration, the standard deviation used in the normal distribution that samples the neighbors of a configuration in the intensification mechanism,

the number of neighbors for numerical parameters to consider in this same mechanism, the maximum number of configurations to select for the multi-start local search intensification mechanism and finally the number of random configurations to sample at each iteration to compute their EICs. Their default values are set to 2000, 2, 0.2, 4, 10 and 10 000 as in [15]. On top of that we have to add the parameters of the random forest which are the number of trees, the ratio of randomly selected parameters to use for splitting at each node, the maximum depth of the trees, the maximum number of leaves in each tree, the number of samples under which a node will not be split and the sampling rate of the training data for building each tree. These are the parameters used by the random forest regression algorithm implemented in Smile, a Java library that gathers various machine learning and statistical tools. Their default values follow the ones used in [15] as 10, 5/6, infinity, infinity 10 and 1.0 - meaning sample with replacement in the Smile version -, respectively. Finally SMAC also allows to use instance features in pairs with the instances used in the configuration process. As our problem doesn't have any precomputed instance features the library uses a default one which is the solution obtained by running the target algorithm A with the default configuration. As this might take a long time when the instances are plentiful and difficult, this can be computed previously to the configuration process through the $computeInstancesFeatures$ function of the implemented configurators.

Studying the effects that these parameters have on the results of the configuration process is computationally intensive as for each tested set of parameters the target algorithm A would be run hundreds of times. This more or less makes the "model selection" intractable given the available time to write this thesis. Moreover the default parameters given in the different scientific papers describing these algorithms were wisely chosen and should give sufficiently good results for the problem at hand. Without conducting a real hyper-parameters study, I will still evaluate the impact of choosing a better variable tree than the default one for GGA in chapter 4.

Chapter 4

Experiments and results

4.1 Introduction

This section presents the experiments that will be conducted on the different algorithms that are implemented in the library in different set ups. This is done firstly to detect which algorithms might be the most suited for the application at hand and secondly to study some features of these configurators - or more precisely how they can be used for our problem - and to evaluate the impact of the choices that have to be made relatively to the objective function or the parameter space. All the experiments have been conducted by using the same random seed and by running them on the same machines - the student servers of UCLouvain with 8 GB of RAM and 8 threads each - without any other processes running - except the mandatory ones - in order for the comparisons to be done on an equal footing. Section 4.2 will compare the different configurators on three different instance sets in order to notice their different strengths and weaknesses, section 4.3 will tackle the problem of the parameter space size, section 4.4 will study different aggregations of the objectives into a single objective function to measure its impact on the results and section 4.5 depicts a preliminary experiment to the study of the influence of the hyper-parameters of the configurators.

4.2 Comparison of the different algorithms

In this section we compare the four implemented algorithms, namely GGA, FocusedILS, SMAC and IFRace, on different types of instances. The goal is to identify which algorithm performs the best for the application at hand which is finding the best configuration for the warehouse.

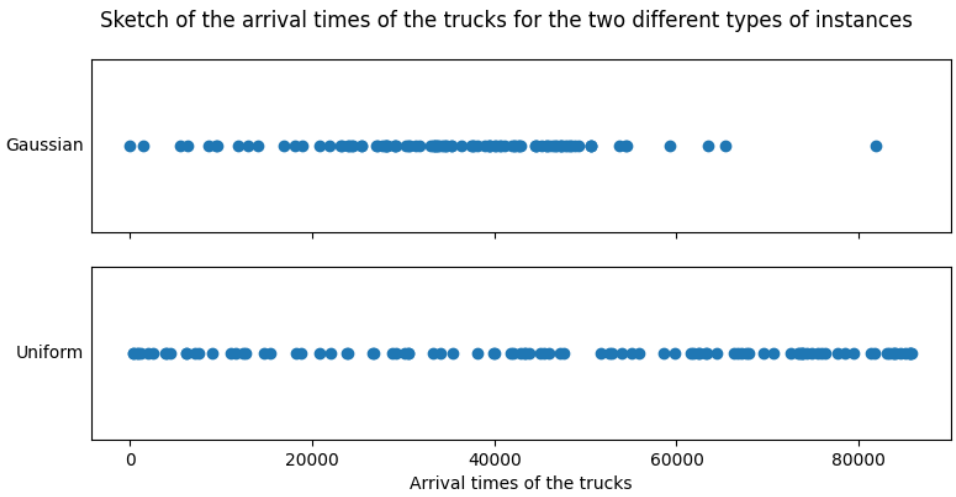


Figure 4.1: Example of the shapes of the two different types of instances used for the simulations

4.2.1 Experimental setup

This first experience is a preliminary one to identify the best-performing configurators for which longer runs will be conducted. The comparison will be done on two different types of instances. In order to keep the running time relatively small each algorithm will be run for approximately three hours and each execution must not exceed 15 seconds, the time after which a configuration will be considered as unsuccessful - i.e. being attributed an objective value of plus infinity. The first iteration that ends after the giving time stops the search.

Two sets of training instances were used to first identify which configuration is the best according to each configurator. Then the four identified configurations were run on two sets of 1000 unseen testing instances to assess their performances. These four sets were generated in order to simulate a warehouse on a per day basis. Each simulation - i.e. instance - is composed of 100 trucks arriving at specific times which are normally distributed in one case and uniformly in the other case from time 0 to time 86400 - i.e. 24 hours - at most - except maybe for some Gaussian instances. Each truck creates between two and 200 missions for the shuttles depending on their cargo - one to 100 pallets to load and one to 100 pallets to unload chosen uniformly at random. An example of the shapes of these instances can be found in figure 4.1.

These sets were used separately to evaluate the configurations in two different kinds of setup which are a day when a big part of the trucks are arriving

Parameter	Category	Domain (Step)	Default value
Nb tracks	Integer	[2-50] (2)	10 tiles
Nb spots per track	Integer	[2-200] (2)	50 tiles
Nb shuttles	Integer	[5-200] (5)	50 mobiles
Battery life	Integer	[100-10000] (100)	1000 tiles
Charging time	Continuous	[600-21600]	3600 seconds
Average speed of the shuttles	Categorical	[slow, normal, fast]	normal

Table 4.1: Table regrouping the different parameters that will be evaluated through the configuration process of the target algorithm A in this thesis

in a short amount of time - to test the ability of the configuration to support a high load of missions - and a day when the trucks are arriving at regular intervals - to test the efficiency of the configuration when dealing with regular arrival times typically to avoid having too high monetary costs that will not be used efficiently. A third set of training and testing instances combine these two sets to assess how the configurations perform when facing the two described types of day.

In table 2.2 we discussed the parameters that will be optimized in this thesis. The concrete domains along with the default values that were used in this first experiment are defined in table 4.1. The default values define a default configuration that will generally be used to initialize the different configurators depending on their implementation. These values were chosen accordingly to the design of the instances. As each of the hundred trucks will create one hundred missions on average the default stock is composed of 500 slots and there are 50 mobiles having a battery life of 1000 tiles and a charging time of one hour with a "normal" speed - defined as 5 tiles/second in the implementation. This leads to an average of two missions per mobile, each being able to complete approximately 40 missions before having to charge - if we consider that one mission is completed in approximately 25 tiles. If we discretize the continuous parameter with a step of 600 - which is what will actually be done to evaluate FocusedILS which can't handle continuous parameters - we obtain a parameter space composed of approximately 1 billion different configurations which seems to be a quite good parameter space size to handle in this thesis. The influence of the parameter space size on the results will be studied in section 4.3.

As said previously in this thesis, the algorithms that have been implemented are not able to handle multi-objective objective functions. This restriction leads to an aggregation of the different objectives - which are the monetary cost and the average service time of the trucks in our case. The different parts of this objective have been defined as follows :

- The average service time of the trucks in seconds,
- the mobile cost as being the number of mobiles times their speed - 1 for "slow", 5 for "normal" and 10 for "fast",
- the warehouse size cost as being the number of slots available in the stock,
- the battery life cost as being the battery life of the shuttles,
- the charging time cost as being 86 400 divided by the charging time - a lower charging time leads to a higher monetary cost.

No data were available to approximately set up the weights that will be associated to each of these costs in order to build the aggregated objective - for example it is impossible to know in which proportion a mobile should be more costly than a slot in the stock. Arbitrary values were chosen in order for the different objectives to be more or less equally important but the user might want to modify these in order to account for some constraints. These weights were set manually by looking at the performance of some configurations on some instances and looking at the influence that each objective had on the aggregated objective value. As the aforementioned objectives lie in different orders of magnitude - for example the number of slots in the stock can be between 4 and 10 000 while the mobile cost could go from 5 to 2000 - the weights are drastically different. For all the experiments in this chapter - except in section 4.4 - these weights were set to 60, 0.3, 0.15, 0.25 and 3, respectively. Taking the different orders of magnitude into account this leads to a small preference for a lower average service time.

Additionally to the general setup, each algorithm has parameters to set before starting the simulations. For this first experiment, the default values - i.e. the ones that were given in the corresponding scientific papers - of these parameters will be used, except for IFRace for which they led to a bad behaviour of the configurator for the application at hand. These values were discussed in section 3.6. For IFRace, the parameters have to be adapted depending on the chosen maximum execution time as the time budget for an iteration and the number of samples to select at each of them directly depend on it. The default values sampled too much configurations at the first iteration which did not give enough time to the remaining ones to intensify the search on the survivors. In order to have a better equilibrium between exploration and exploitation the parameters of IFRace for this experiment were set to $nIterations = 40, \mu = 7$ and $N_{min} = 5$. These values led to the evaluation of 38 configurations at the first iteration and this number decreases with time while the number of evaluations for each configuration increases.

The cases of SMAC and GGA must also be discussed here. SMAC uses a random forest regression model as a surrogate model and this machine learning tools has plenty of parameters. In order for it to not induce a too high time complexity overhead the default parameters given in [15] were used for the model. It was also necessary to define the instance features that would be used to fit the model. To keep this first experience simple, only the value of the objective function obtained by executing the instance with the default configuration was chosen. For GGA, to keep this experience simple once again, the variable tree was chosen to be the default tree with an $And - node$ as a root and each parameter connected to it. In other words we consider that the parameters can be independently optimized.

4.2.2 Results

Table 4.2 gives the configurations that were found by each of the algorithm after searching for three hours for the Gaussian, uniform and mixed instance sets. This table is accompanied with table 4.3 that regroups the results obtained with these configurations on the different training and test sets. They represent the average objectives and the standard deviations obtained on the instances. For the training sets the number of instances on which it was run depends on the algorithm but for the testing sets it is equal to 1000. The most surprising observation is probably the fact that no matter the algorithm or the instance set, the standard deviation of the results on the test sets are very high compared to the mean. This strange behaviour is duo to the fact that some instances are really hard to solve and thus even if the best configurations found by the algorithms are quite good in general they perform poorly on these. This can be deduced from the fact that even if the configurations found by SMAC have been evaluated on far more instances than the ones found by the other algorithms during training, the standard deviations remain high on the test sets.

A second interesting observation is that IFRace seems to perform worse than the other configurators, especially when Gaussian instances are involved. Moreover, IFRace probably lacks a diversification mechanisms in selecting the instances on which the configuration are evaluated as the incumbents are assessed on a very low number of different instances while being evaluated through more runs than in GGA or ILS. This might be good when the target algorithm is randomized but in our case the simulation of the warehouse is deterministic hence evaluating the incumbent 330 times on only 12 distinct instances doesn't allow IFRace to find good configurations for the application at hand. For this reason IFRace will not be further investigated.

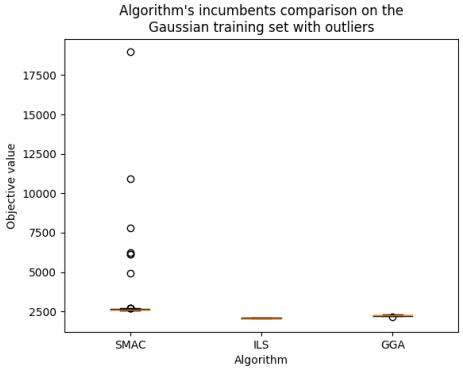
In figure 4.2 a comparison of the boxplots obtained with the different results of

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
			Gaussian			
SMAC	170	28	76	4800	21477.47	fast
GGA	195	38	50	3500	16516.01	fast
IFRace	165	32	26	1900	6551.66	fast
ILS	180	48	42	3000	18600.0	fast
			Uniform			
SMAC	95	48	42	2200	1362.50	fast
GGA	190	40	46	3500	5251.38	fast
IFRace	135	34	60	2000	1606.26	fast
ILS	95	42	52	1900	600.0	fast
			Mixed			
SMAC	185	40	74	4200	17488.18	normal
GGA	140	38	44	4100	5261.72	fast
IFRace	180	36	22	1700	16373.87	fast
ILS	165	44	46	3200	18000.0	fast

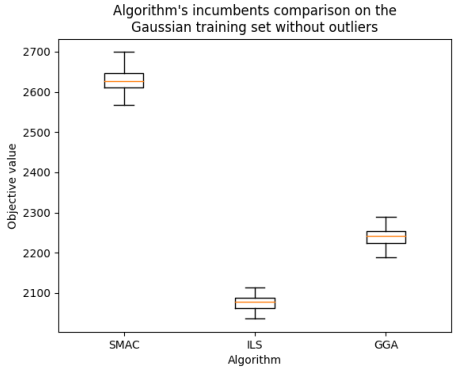
Table 4.2: Table regrouping the incumbents obtained by running the different algorithms on the Gaussian, uniform and mixed training sets for three hours each.

Train			
Instance set	Gaussian	Uniform	Mixed
SMAC	2681 ± 693 (838 - 838)	2128 ± 479 (718 - 718)	2921 ± 55 (648 - 648)
GGA	2240 ± 27 (31 - 31)	2250 ± 21 (27 - 27)	2264 ± 19 (63 - 63)
IFRace	33412 ± 46111 (12 - 330)	2091 ± 217 (9 - 209)	1552 ± 36 (2 - 2)
ILS	2077 ± 18 (38 - 38)	2188 ± 114 (90 - 90)	2108 ± 21 (53 - 53)
Test			
Instance set	Gaussian	Uniform	Mixed
SMAC	4091 ± 13451 (1000)	3400 ± 10170 (1000)	3053 ± 2174 (1000)
GGA	4613 ± 17654 (1000)	4200 ± 13864 (1000)	6001 ± 22902 (1000)
IFRace	33852 ± 64792 (1000)	3529 ± 10134 (1000)	42748 ± 70853 (1000)
ILS	3953 ± 15514 (1000)	3018 ± 7308 (1000)	4206 ± 17701 (1000)

Table 4.3: Results obtained by running the different algorithms on the training and test sets for three hours each. They are presented as "mean ± std (nInstances - nRuns)" where nInstances is the number of distinct instances on which the incumbent has been evaluated and nRuns is the number of runs achieved on these instances. For the test section these two values are identical.



(a) Performances of the incumbents with outliers on the Gaussian training set



(b) Performances of the incumbents without outliers on the Gaussian training set

Figure 4.2: Comparison of the boxplots with and without outliers

each algorithm on the Gaussian training set is depicted to show that the high standard deviations given in table 4.3 are indeed due to some more difficult instances - the same kind of graphs were obtained with the other instance sets.

Figure 4.3 presents the results obtained for the incumbents on the three different test sets without the outliers - as defined by the default parameters of the matplotlib python library - and on the mixed set with the outliers. These results seem to show that FocusedILS finds the best configuration in general - it might be debatable for the uniform test set. This is more or less consistent with what was obtained in table 4.3 which rather compares the means - while the boxplots show the median. However SMAC seems to be better when dealing with the mixed test set and comparing the algorithms with all the runs - i.e. including the most difficult instances. This can be explained by looking at the incumbents that were found by the different algorithms.

For the mixed instance set, SMAC found an incumbent with more mobiles and more stock slots than its pairs but with a "normal" mobile type. While the two first choices increase the monetary cost and the average service time in the uniform instances - as more mobiles have to go through the four entries thus including more collisions - the third choice allows to reduce it a bit. In the end the monetary cost is almost the same while the average service time in the uniform instances is probably higher with the SMAC incumbent. The difference happens in the Gaussian instances where a high load of missions must be handled by the warehouse. When facing such a situation it is important to have a larger stock and a larger fleet and this is exactly what the incumbent of SMAC looks like. This

explains why its results on the mixed test set as given in table 4.3 are the best while they are not if we look at figure 4.3 because the values of the outliers are way higher for GGA and ILS than for SMAC as shown in figure 4.3d.

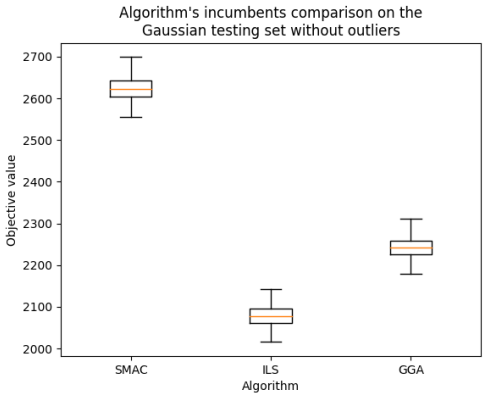
The same trend can be observed when looking at the differences between the incumbents found for the Gaussian case and the incumbents found for the uniform case. A normally distributed scenario creates peaks of missions to achieve, requiring more mobiles to reduce the service time - at the cost of having more collisions. Moreover, the battery life and the charging time of the batteries must be higher to be able to handle these peaks without having to go to the charging site too frequently and and without increasing the monetary cost too much - a higher charging time lowers it while a higher battery life increases it. For equally spaced arrival time a lower battery life and a lower charging time are affordable as, in this case, the shuttles can charge up between the arrival of two trucks. Also, a lower amount of mobiles is required as the shuttles have the time to fulfill the requirements of one truck before the next one arrives.

As a conclusion, when comparing the performances of the different algorithms, it seems that IFRace was performing way worse than the three other methods which obtained quite similar results depending on the instance set that was considered. SMAC seems to better handle instance sets with a stronger diversity in the instances while FocusedILS seems to slightly outperform its pairs when considering a unique kind of instances, probably due to its strong intensification and capping mechanisms.

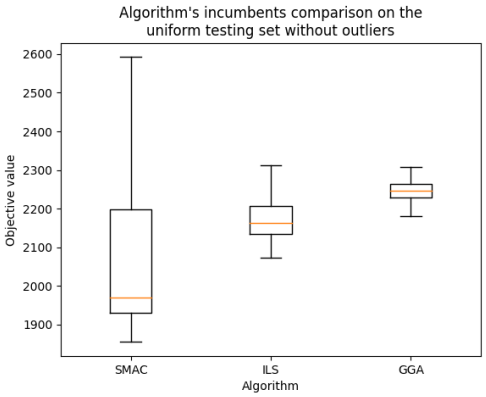
4.2.3 Longer runs on the Gaussian instances

We gather here the results obtained by running the different algorithms - except IFRace - for 12 hours on the Gaussian training set - making the assumption that we would observe the same kind of behaviors on the other sets. Table 4.5 shows the different numerical results while table 4.4 depicts the obtained incumbents. A quick observation leads to the conclusion that the incumbents themselves are quite similar than what was obtained with the three hours runs.

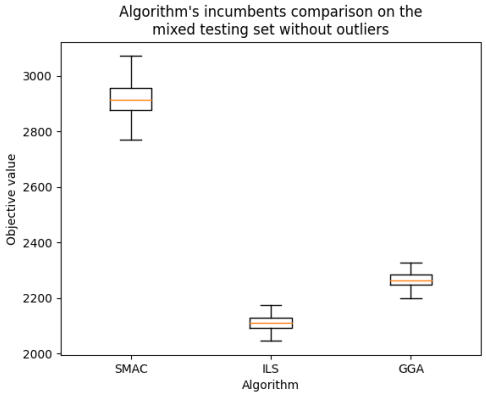
The number of instances on which each incumbent has been evaluated increased for SMAC and GGA but not that much for ILS. This is not of a great importance as ILS always compares the configurations only when they have the same number of runs. It means that ILS assessed way more different configurations (5989) than GGA (117). However it leads to poorer results on the test set as can be seen in table 4.5. The low number of instances on which the configuration was evaluated induces a bias towards the training set and thus a worse generality capability.



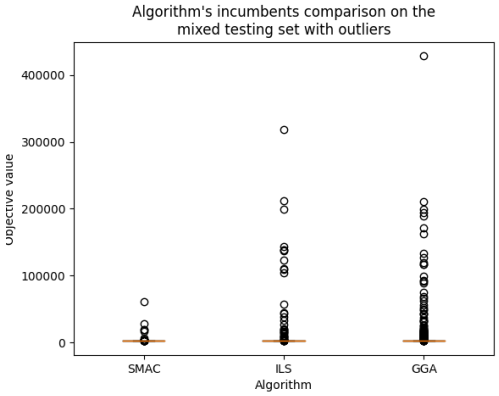
(a) Gaussian testing set



(b) Uniform testing set



(c) Mixed testing set



(d) Mixed testing set with outliers

Figure 4.3: Comparison of the incumbents obtained on the three testing sets with runs of 3 hours for each algorithm

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
			Gaussian			
SMAC	160	40	60	4100	20621.89	fast
GGA	185	38	54	3400	16471.31	fast
ILS	200	50	42	2400	19800.0	fast

Table 4.4: Table regrouping the incumbents obtained by running the different algorithms on the Gaussian, training set for twelve hours each.

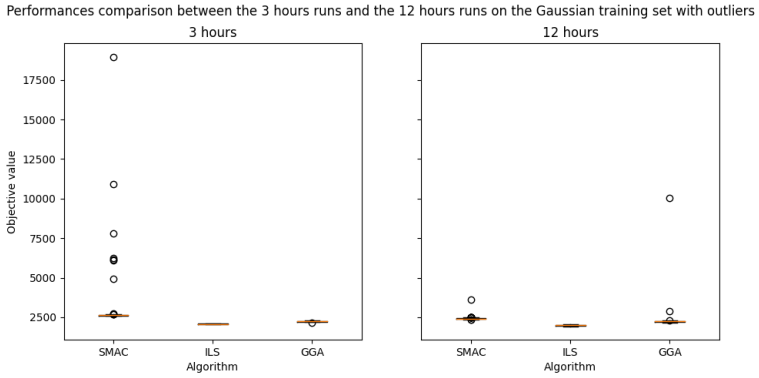
Train		Test	
Instance set	Gaussian	**Instance set**	Gaussian
SMAC	2250 ± 380 (2000 - 2000)	**SMAC**	3433 ± 10330 (1000)
GGA	1985 ± 23 (387 - 425)	**GGA**	3929 ± 14687 (1000)
ILS	2422 ± 37 (54 - 54)	**ILS**	4414 ± 18390 (1000)

Table 4.5: Results obtained by running the different algorithms on the Gaussian training and test sets for twelve hours each. They are presented as "mean ± std (nInstances - nRuns)" where nInstances is the number of distinct instances on which the incumbent has been evaluated and nRuns is the number of runs achieved on these instances. For the test section these two values are identical.

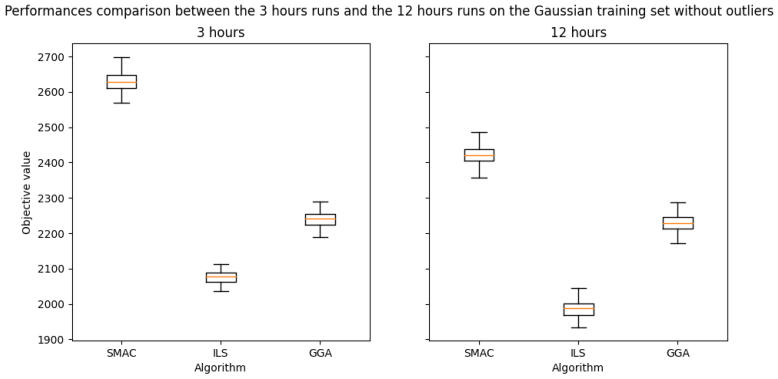
The results on the training set are encouraging as they are in general smaller than in table 4.3 although the incumbents have been evaluated on more instances - thus having more chance to include the difficult ones. It confirms the intuition that conducting longer experiments could give even better results as the parameter space would be much more explored and the incumbents would be assessed on more different configurations.

Figure 4.4 shows the improvements that have been made when running the algorithms for 12 hours instead of 3. We encounter the same relations between the three configurators : ILS seems to slightly outclass GGA which itself seems to outclass SMAC. We can also observe that SMAC and ILS are worth running for more time but not GGA - at least with the default hyper-parameters. Moreover the number of outliers decreased and their objective values are lower than before while the number of instances on which the incumbents have been evaluated increased which indicates that the found incumbents are more robust.

The same kind of conclusion can be drawn from the results on the Gaussian testing set. Besides for ILS, the mean and the standard deviation are lower for the twelve hours runs than for the three hours runs. Moreover when looking at figure 4.5 we can observe the same behaviour as in figure 4.4 : SMAC and ILS found better configurations while GGA did not. However these results must be taken with carefulness as the figure doesn't take the outliers into account. Although it is not shown here the comparison was quite similar to what was presented for the training set : the outliers are less numerous and with lower extreme objective values. For the robustness, SMAC seems to give the best results in general, relatively to what is observed in the two sets of runs.



(a) Performances on the Gaussian training set with outliers



(b) Performances on the Gaussian training set without outliers

Figure 4.4: Comparison of the results obtained with the 3 hours runs and the ones obtained with the 12 hours runs on the Gaussian training set

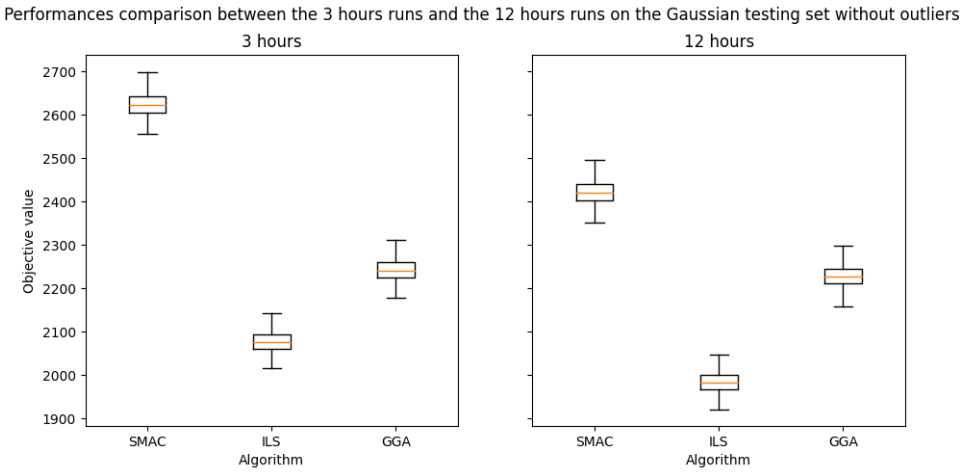


Figure 4.5: Performances on the Gaussian testing set without outliers

4.3 Influence of the parameter space size

This section will look in more details at the impact that the parameter space size has on the search. It will begin by looking at which part of the space size was searched in section 4.2 before conducting some experiments to study the influence of it on the search and on the optimality of the found incumbents.

4.3.1 Examined parameter space in the default runs

The parameter space is quite big for runs of 3 hours each. This can be seen in table 4.6 as the maximum number of configurations that were evaluated through the search is a bit more than 1500 - although it could be higher if we would look at the number of configurations that SMAC assessed through its random forest. It means that only 0,000154% of the parameter space has been examined. This might seem very low but a few remarks have to be taken into account.

The algorithms presented here are designed to identify the promising parts of the space on which the search will be intensified. It means that some parts of it are not even worth examining. For example it seems that having mobiles of type "slow" is very bad as none of the algorithms chose an incumbent with this value for the "mobileType" parameter. An other point is that some of the configurations can't solve the instances because their mobiles have a too low battery life for example. This prevents the mobile to be able to go to the furthest stock slots. Finally some configurations are deeply suboptimal : having only five shuttles to handle hundreds of missions or a stock of 100 000 slots to manage 1000 pallets is of course not desirable. The configurators will detect these parts of the space which should not be examined in depth and will focus on the interesting parts of it.

An other reason that would explain these low numbers is the running time that is rather low compared to what was done in the literature - although the algorithms to which they were confronted had far more parameters in general. Section 4.2.3 evaluated SMAC, FocusedILS and GGA on the Gaussian instance sets after twelve hours runs instead of three. We already showed in table 4.5 that the incumbents were assessed on more instances. Although only the Gaussian results were gathered due to the time constraints, for the twelve hours runs, SMAC analyzed 2639 different configurations, GGA analyzed 117 unique configurations and FocusedILS 5989 configurations. These numbers are already way higher than in table 4.6 and we can imagine that running the algorithms for days or weeks would end in exploring the whole relevant part of the parameter space.

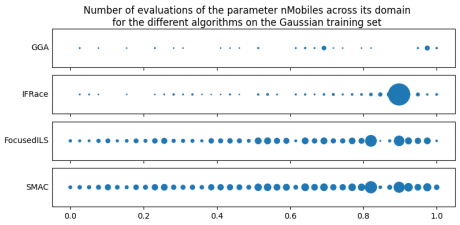
For the reader to better understand which parts of the parameter space were

Instance set	Gaussian	Uniform	Mixed
SMAC	786	673	601
GGA	86	90	84
IFRace	209	131	891
ILS	1186	1542	1451

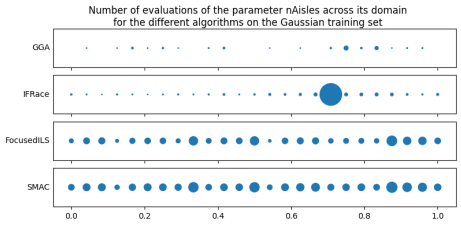
Table 4.6: Number of unique configurations evaluated with the target algorithm for each configurator on each training set

examined I depict in figure 4.6 the parts of the space explored for each parameter - independently - with each algorithm. While this comparison is far from being what really happens - because looking at only one dimension at a time doesn't give an overview of what happens in the whole parameter space - it already gives an insight of what was said previously. The parameters have all been normalized for the comparison but the values can be quickly mapped to the initial values by looking back at table 4.1.

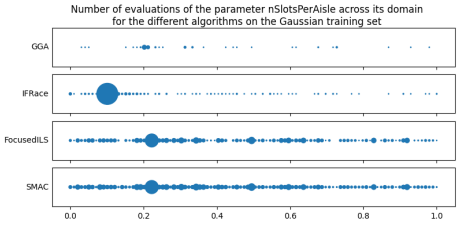
We can see that GGA is quite bad at exploring the space and it could be a reason why it generally performed worse than its pairs in section 4.2. IFRace on the other hand is highly focused on one specific part of the parameter space, typically the one that was found to perform statistically better than the rest. This also indicates that the exploration/intensification balance was not good and this might be a reason why IFRace was outperformed by the other configurators. Better hyper-parameters would probably be needed to solve this issue. For both FocusedILS and SMAC we can see that parts of the space are a bit more evaluated than others but that the whole space is more or less examined for each of them. Moreover these bigger points are shared among both configurators which indicates that they are probably good values for the parameters of the target algorithm. This behaviour is probably due to the fact that both of these configurators are using a kind of "restart" mechanism and a strong intensification phase : the "restart" mechanism is used to explore the space and once a promising candidate is found it is assessed at least as much as the current incumbent before being compared to it. The bigger points are thus indicating the values that compose the found incumbent as it is required to be the one that have been evaluated the most in these two algorithms. The same kind of graphs were observed for the twelve hours runs - although the points were all bigger and even more spread among the different domains.



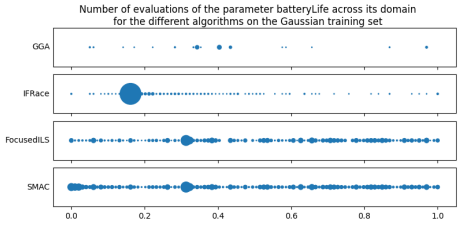
(a) nMobiles



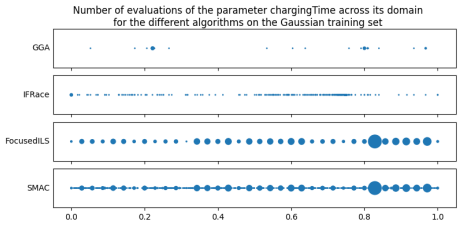
(b) nAisles



(c) nSlotsPerAisles



(d) batteryLife



(e) chargingTime

Figure 4.6: Comparison of the exploration of the parameter space for the different algorithms on the Gaussian training set

Parameter	Category	Domain (Step)
Nb tracks	Integer	[10-60] (10)
Nb spots per track	Integer	[20-100] (10)
Nb shuttles	Integer	[50-200] (10)
Battery life	Integer	[1000-6000] (1000)
Charging time	Continuous	[1200-21600]
Average speed of the shuttles	Categorical	[slow, normal, fast]

Table 4.7: Table regrouping the smaller domains of the different parameters that will be evaluated in section 4.3.2

4.3.2 Results on a smaller parameter space size

In this subsection we compare the results obtained by running SMAC, ILS and GGA on a smaller parameter space to the runs achieved on the initial space. For this purpose we revisit what was defined in table 4.1 to give the new domains in table 4.7. The default configuration remained the same in order for SMAC to be able to run on the same instance feature than before - which is the objective value obtained by running the default configuration on the instance.

For FocusedILS the continuous parameter - the charging time of the batteries - will be discretized with a step of 1200. Combining all the parameters, this leads to a space with a bit less than 300 000 unique configurations. As the number of unique configurations evaluated by each configurator was more or less similar to what was presented in table 4.6, we can deduce that approximately 0.5% of the new parameter space was explored. Again, this is a relatively small number but the configurators are designed to avoid exploring non-promising parts of the space.

Table 4.8 gives the different incumbents that were obtained by running the algorithms for 3 hours on each training set. Some results are very different compared to what was found in table 4.2 or table 4.4. For example we generally had that nAisles was greater than nSlotsPerAisle but it is the opposite here. The number of mobiles and the charging time are also higher on average.

Looking at table 4.9 we can deduce that reducing the parameter space size is not necessarily beneficial. The vast majority of the obtained results are higher in mean and in standard deviation indicating that the obtained incumbents are in general worse and less robust. As the granularity of the domains of the parameters have been reduced this might be caused by the fact that the configurators can't be as flexible as they want when choosing the parameters and thus they chose suboptimal values even in a local way - i.e. they were not even at the local optimum. Moreover

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
Gaussian						
SMAC	160	60	30	3000	20843.59	fast
GGA	120	50	30	4000	6638.65	fast
ILS	170	60	30	3000	20400.0	fast
Uniform						
SMAC	190	60	40	3000	7801.03	fast
GGA	120	50	30	4000	6638.65	fast
ILS	160	50	40	3000	13200.0	fast
Mixed						
SMAC	190	50	50	3000	20672.50	fast
GGA	120	50	40	4000	4715.77	fast
ILS	170	60	30	3000	19200.0	fast

Table 4.8: Table regrouping the incumbents obtained by running the different algorithms on the Gaussian, uniform and mixed training sets for three hours each on the smaller parameter space.

Instance set	Gaussian		Uniform		Mixed	
Space size	Big	Small	Big	Small	Big	Small
SMAC	4091 ± 13451	4931 ± 19864	3400 ± 10170	2648 ± 4963	3053 ± 2174	3065 ± 9500
GGA	4613 ± 17654	8432 ± 29118	4200 ± 13864	7319 ± 24281	6001 ± 22902	4148 ± 15903
ILS	3953 ± 15514	4951 ± 19865	3018 ± 7308	3895 ± 14922	4206 ± 17701	4647 ± 19241

Table 4.9: Results obtained by evaluating the different incumbents obtained with the smaller parameter space for three hours each on the three test sets . They are presented as "mean ± std". All the incumbents have been evaluated on the same 1000 test instances.

the new domains are more focused on the interesting parts of the initial ones meaning that exploring 0.5% of this parameter space might denote more or less the same part as the 0.000154% of the initial parameter space hence the configurators might actually not have explored a bigger part of it and the percentage increased rather only because the parameter space size decreased. A perfect balance would be to have a running time adapted to the size of the space and a granularity sufficient for the configurators to be as flexible as the time constraint permits it.

4.4 Influence of the weights in the aggregation

This section aims at observing the behaviors of the configurators when the aggregation of the different objectives is changed in order to bias the search towards one of the objective. For this purpose, simulations have been conducted on the same parameter space and set of instances and with the same hyper-parameters of

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
Gaussian						
GGA	200	48	56	4100	10417.59	fast
ILS	195	46	46	2600	16800.0	fast
Uniform						
GGA	200	40	46	3500	5251.38	fast
ILS	160	48	46	4000	9600.0	fast

Table 4.10: Table regrouping the incumbents obtained by running GGA and ILS on the Gaussian and uniform training sets for three hours each when the aggregation of the objectives was modified to bias the search towards a lower service time.

the configurators when the weight linked to the average service time is equal to 1000 (previously it was equal to 60) and then when the weights linked to the costs are increased tenfold - and the other weights are the same as in section 4.2. For simplicity only GGA and ILS were confronted to these simulations and only on the Gaussian and uniform training sets but always for three hours each. One can guess that the results for SMAC or IFRace would have been similar.

When comparing the results of table 4.10 to the results of table 4.2 it is clear that the service time has been favored compared to the monetary cost. The two configurators both gathered more resources in the warehouse in order for the trucks to be served as quickly as possible. It is especially the case for the chosen number of mobiles and the charging time of the batteries.

The results of the second experiment - when the weights linked to the cost have been increased tenfold - are depicted in table 4.11. It is surprising to see that the number of mobiles and the number of slots in the stock generally stayed identical or increased. This can be explained by the fact that both configurators chose to use normal mobiles instead of fast ones which, in our case, reduces the cost of each mobile by two - see objective definition in section 4.2.1. Moreover the battery life and the charging time both generally changed to reduce the monetary costs. These results seem to show that the number of slots in the stock can't be reduced too much or it would drastically increase the service time and that the number of mobiles is also important and can remain high as the mobiles type changed to "normal". The battery life and the charging time can also both be optimized a bit to lower the monetary cost without impacting the service time too much.

To grasp a better idea of what happens when modifying the weights of each cost metric, the costs obtained for each discussed case in this section on five random instances each are presented in table 4.12. When increasing the weights of the

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
Gaussian						
GGA	195	34	42	2900	17641.98	normal
ILS	165	24	76	1100	1200.0	normal
Uniform						
GGA	185	38	40	2900	5251.38	normal
ILS	155	40	52	1000	1200.0	normal

Table 4.11: Table regrouping the incumbents obtained by running GGA and ILS on the Gaussian and uniform training sets for three hours each when the aggregation of the objectives was modified to bias the search towards a lower monetary cost.

costs we observe that the service time is way higher than in the initial case while the monetary costs are way lower. This is not surprising as the configurators will focus on reducing the monetary costs, which have higher weights, to the detriment of the average service time. However when the weight linked to the service time is increased it doesn't modify much the results. In one case - GGA on the Gaussian instances - the service time is even higher than in the initial case. Nevertheless the monetary cost generally increased. These observations are probably due to the fact that the service time is limited by a lower bound which is the average service time of the trucks if they are all served optimally. The more the configuration is closer to this optimum the more difficult it is to lower the average service time. On the other hand, as the monetary costs are proportionally lower to the average service time than in the initial case, the configurators can increase it without impacting the total objective value too much, explaining why the monetary costs are generally higher.

4.5 Influence of the variable tree inside GGA

This final experiment aims at exploring the possibilities of the variable trees of GGA. Initially, the goal was to study the hyper-parameters of the configurators but unfortunately this will be relegated to chapter 5 where I will explain what I would have done with more time.

The purpose of this section was to identify if linking the parameters between them would have a positive impact on the results. For this purpose GGA was run on the Gaussian and on the uniform training sets for 3 hours with a different variable tree than in section 4.2 - assuming that the results on the mixed training set would be similar. This variable tree links the parameters in three ways. First, the type of the mobiles will be a child of the number of mobiles. Second, the

Algorithm	**GGA**						**ILS**					
Aggregation	**Initial**		**time**$_{1000}$		**cost**$_{10}$		**Initial**		**time**$_{1000}$		**cost**$_{10}$	
Metric	ST	MC	ST	MC	ST	MC	ST	MC	ST	MC	ST	MC
gaussian$_{14-191}$	8.33	7355	8.60	8796	15.75	5308	8.26	6821	8.23	6671	63.81	3821
gaussian$_{19-117}$	7.91	7355	8.16	8796	15.08	5308	7.75	6821	7.71	6671	25.87	3821
gaussian$_{19-190}$	7.70	7355	7.89	8796	14.91	5308	7.66	6821	7.62	6671	24.31	3821
gaussian$_{15-198}$	7.90	7355	8.14	8796	15.07	5308	7.82	6821	7.79	6671	49.08	3821
gaussian$_{10-111}$	7.60	7355	7.86	8796	14.53	5308	7.36	6821	7.36	6671	25.24	3821
average	**7.888**	**7355**	**8.13**	**8796**	**15.068**	**5308**	**7.77**	**6821**	**7.742**	**6671**	**37.662**	**3821**
uniform$_{10-121}$	7.94	7256	7.86	7356	15.29	5361	10.60	5178	8.26	7817	32.45	3927
uniform$_{12-102}$	7.30	7256	7.20	7356	14.11	5361	9.98	5178	7.67	7817	56.35	3927
uniform$_{16-104}$	7.85	7256	7.74	7356	15.07	5361	10.31	5178	8.21	7817	31.84	3927
uniform$_{17-166}$	7.94	7256	7.88	7356	15.51	5361	10.48	5178	8.42	7817	41.65	3927
uniform$_{19-154}$	7.92	7256	7.89	7356	15.38	5361	11.17	5178	8.25	7817	29.30	3927
average	**7.79**	**7256**	**7.714**	**7356**	**15.072**	**5361**	**10.508**	**5178**	**8.162**	**7817**	**38.318**	**3927**

Table 4.12: Table regrouping the objective values observed on five different random instances from the test sets with three objective aggregations : the initial one, the one where the service time is more important - time$_{1000}$ - and the one where the monetary costs are more important - cost$_{10}$. ST is the observed service time for a specific aggregation on a specific instance while MC is the sum of the monetary costs. The compared incumbents are always the ones obtained with the three hours runs.

number of slots per aisle is linked to the number of aisles which is itself linked to the battery life of the shuttles. Finally, the charging time is not linked to any other parameter. This variable tree is depicted in figure 4.7.

The goal would be that the choice of one value for a specific parameter would influence the choices for the other parameters. For example the batteryLife must be at least as great as the longest path through the warehouse which depends on the size of the stock - i.e. nAisles*nSlotsPerAisle. Thus, setting a value to batteryLife should influence the choices for nAisles and nSlotsPerAisle. In the same manner, the number of mobiles would influence their type as each mobile is more costly if its type is "fast" rather than "normal". We could thus have a large number of "normal" mobiles or a smaller number of "fast" mobiles for the same cost.

These relations exist when defining the variable tree like this because of how GGA is implemented. Each child node has 90% of chance to inherit the label of its parent node which defines from which tree the value of the offspring for this node will be inherited - i.e. the tree from the competitive part of the population or the one from the non-competitive part. In other words one can say that a good existing relation between the linked parameters would tend to survive as a block rather than individually, which was the case when considering a variable tree where

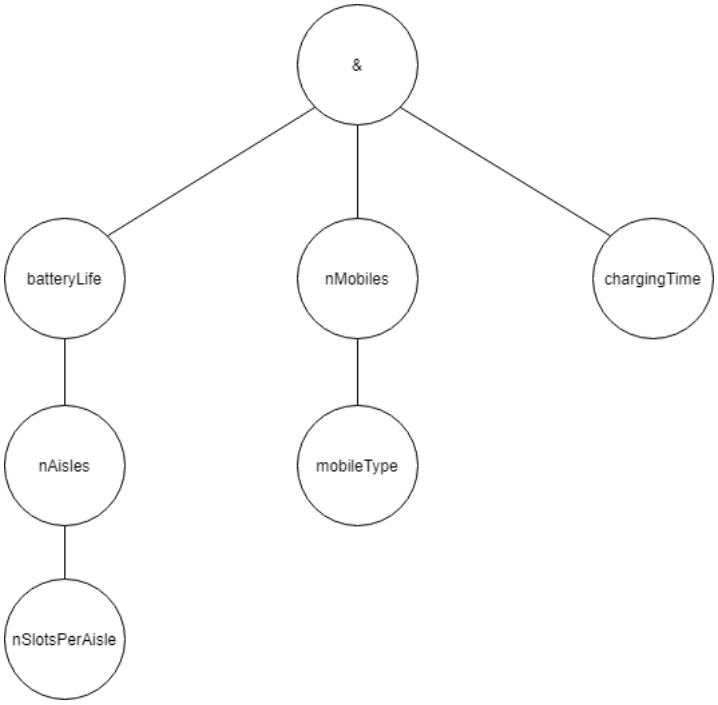


Figure 4.7: Variable tree with links between the parameters

all the parameters were independent - see chapter 3 for a full reminder of GGA.

Table 4.13 regroups the incumbents that were obtained when the runs were completed. They were executed with the same parameter space as in the first experiment. The Gaussian incumbent is almost identical to the one presented in table 4.2. The only noteworthy change is that the type of the mobiles was set to "normal" instead of "fast". This is probably a consequence of having linked the mobileType parameter to the nMobiles one. Concerning the incumbent obtained on the uniform training set the noteworthy change concerns the dimensions of the warehouse. In table 4.2 both nAisles and nSlotsPerAisle were close and thus formed a square rather than a rectangle but this is not the case here although the total number of pallet spots is nearly the same. We thus observe a concrete impact of the variable tree on the found incumbents.

All that remains is to compare the results of the new incumbents with the ones of the incumbents of table 4.2 that are gathered in table 4.3. The comparison is depicted in table 4.14. The conclusion is obvious : this new variable tree did not improve the results, neither in terms of the mean of the service time nor in terms of robustness. Various reasons could explain what happened. First, it could be due

Parameter	nMobiles	nAisles	nSlotsPerAisle	batteryLife	chargingTime	mobileType
			Gaussian			
GGA	200	40	42	4400	17206.70	normal
			Uniform			
GGA	200	14	116	2900	963.72	fast

Table 4.13: Table regrouping the incumbents obtained by running GGA on the Gaussian and uniform training sets for three hours each when the variable tree was defined as in figure 4.7

.

Instance set	**Gaussian**		**Uniform**	
Variable tree	Default	Linked	Default	Linked
GGA - train	2240 ± 27 (31 - 31)	2578 ± 48 (56 - 56)	2250 ± 21 (27 - 27)	2699 ± 297 (48 - 48)
GGA - test	4613 ± 17654	6714 ± 23796	4200 ± 13864	6460 ± 20587

Table 4.14: Results obtained by evaluating the incumbents of GGA obtained with the default variable tree and with the variable tree of figure 4.7 for three hours each on the Gaussian and uniform sets. They are presented as "mean ± std (nInstances - nRuns)" where nInstances is the number of distinct instances on which the incumbent has been evaluated and nRuns is the number of runs achieved on these instances. For the test section these two values are equal to 1000.

to a wrong ordering of the parameters in the tree. For example, it would maybe be better to have the battery life at the bottom of the left branch rather than at the top. Second, the hyper-parameters that were used for this experiment - the same as the ones that were discussed in section 4.2 - were maybe not restrictive enough to observe a sufficient link between the dependent parameters - it could be interesting to redo this experiment with P = 0.95 or P = 0.99 for example. Finally, it might be possible that introducing dependencies between the parameters reduces the exploration capability of GGA as the linked parameters are less prone to individual modifications. If this is the case it might be necessary to increase it in another way, for example by modifying A in order for the individuals to survive for a longer time.

Further experiments would be needed to draw further conclusions but this last section pointed out the difficulty to set the hyper-parameters of the configurators in a concrete example. This problem will be further discussed in chapter 5.

Chapter 5

Future work

5.1 Introduction

In this thesis we reviewed five existing algorithms for automatic configuration in depth and tested four of them through various experiments in the last section. However the field of parameter tuning algorithms is wide and only a very small part of what could be done was presented through this report. This final part aims at pointing out the topics of future work. The three first sections mention several parts that were already studied but in a limited manner allowing further improvements of the library or of the simulation. The final section will rather look at topics that would require more research which makes it of particular interest.

5.2 Improving the existing work

The Java library that was designed alongside this report could be improved in many points. The full implementation of such a library takes time in order for it to truly reflect the needs of the user in an elegant way. Introducing more flexibility, a more intuitive API or better data structures/implementation designs are parts of the library that could be improved for a future version. Introducing multi-threading versions of the algorithms could also boost the performances.

Besides that it could be interesting to extend the number of studied parameters and thus to extend the simulation that served as the target algorithm for this thesis. All the parameters mentioned in table 2.1 are interesting candidates which should also be optimized when designing the warehouse for a real customer. The existing implementation should straightforwardly lead to the study of the following parameters : the number of floors, the number of docks and the number of charging sites. It should also be possible to improve the underlying mechanisms of the simulation

such as the collision manager or the mobile, mission selector with smarter techniques.

Finally the experimental phase could be developed further to deeply study what was already discussed in chapter 4. For example, conducting experiments that would last for weeks on a smaller parameter space would probably lead to a global optimum or trying to find better hyper-parameters for the configurators would maybe improve the found incumbents. Moreover, studying the impact of adding new parameters to the target algorithm would also be an experiment of interest.

5.3 Reviewing, implementing and testing other existing algorithms

Although this work aimed at studying what seemed to be the state-of-the-art algorithms at the time of writing, several other methods exist in the literature - and some of them were mentioned in section 3.4.1. Automatic algorithm configuration is also a growing field on which several people are making research and new, better-performing procedures will for sure be designed in the next few years.

If the target algorithm would eventually be extended one might also need to find other algorithms - or to refine the study of those presented in this thesis - in order to handle larger parameter space sizes or a greater number of parameters. We saw for example from chapter 4 that some algorithms are better suited for large parameter spaces.

5.4 Optimizing the parameters of the configurators

One task that could largely be extended is the one that would aim at finding the best "hyper-parameters" of the configuration process. As for machine learning models, parameter tuning algorithms are generally accompanied by several parameters themselves that could be interpreted as the hyper-parameters of these algorithms.

As it is possible to use model selection techniques to select the best parameters for an SVM to classify a certain data set it would also be possible to start a configurator multiple times on the same configuration, instance pairs to deduce the best hyper-parameters for a particular target algorithm A under study. Of course this would be time-consuming as the configuration of an algorithm can already be seen as a hyper-optimization process leading us to the optimization of a

hyper-optimization process.

In this thesis only six parameters, each with a reasonably small domain, were studied and this allowed to have bearable running times. However when configuring bigger algorithms such as the CPLEX resolver - one common application that was studied in the reviewed papers - days were necessary to obtain sufficiently good results. This would be transformed into months in order for the parameters of the configurators to be in turn optimized. While this time might be reduced to some weeks with supercomputers it would not be affordable for anyone that would use an optimized configurator. This is why this improvement would probably only concern small target algorithms for now until more efficient ways of doing model selection in the context of automatic configuration will see the light of day.

Even though the results were already quite interesting, with more time, I would probably have tried to guess better values for the hyper-parameters depending on the results with the default ones. For example we saw in chapter 4 that IFRace needed to be parameterized accordingly to the application at hand while in this thesis I only considered the default way of defining the time budget of each iteration and the number of configurations to sample at each of them. Besides the hyper-parameters themselves it would have been possible to study the impact of having other instance features or more features than one in SMAC or to study the impact of having other variable trees in GGA as started in section 4.5.

5.5 Extending the studied algorithms

Finally the part that has the bigger room for improvement is probably including new features into the existing state-of-the-art algorithms or equivalently designing better algorithms that would be extensions of the aforementioned ones.

5.5.1 Improving the core of the programs

The procedures that are studied in this thesis already demonstrated their effectiveness through several experiments. However the authors of the corresponding papers always mentioned that grey parts exist about their work and that they will look at them in further versions of their implemented solutions.

Often the ideas that arise in these papers are mechanisms that are already implemented in the other methods. For example the authors of paramILS wrote about including statistical tests in a newer version of their algorithms, as done in IF-race. The authors of SMAC on the other hand would like to integrate the

adaptive capping techniques of paramILS into their own algorithm.

This proves that even the state-of-the-art can still be improved and that it would be a good idea for these changes to be inspired from other existing algorithms.

5.5.2 Adding new features to design new procedures

A last but not least way of upgrading the existing procedures is to add new features to them as barely discussed in section 3.5.3. This would include auto-tuning algorithms - or more simply embedding tuning algorithm - and multi-optimization but it might go further than that. The authors of future automatic configuration algorithms might for example find brand new ways of intensifying the search, initializing the algorithms or fitting more complicated models - for example inspired by other machine learning or statistical methods - to design a new generation of state-of-the-art algorithms.

Chapter 6

Conclusion

Through this thesis I tried to demonstrate that using automatic configuration algorithms to design a warehouse under a certain kind of demands is possible. I first defined the problem from a conceptual view to the concrete target algorithm that has been used in the experiments. This definition was accompanied by the different parameters that might be optimized in a warehouse such as the number of shuttles to use or the size of the stock. However, due to the time constraint, I only studied six of them which are the number of shuttles, the number of aisles in the stock, the number of pallet slots per aisle, the type of the mobiles - declined in "slow", "normal" and "fast" -, the charging time of the batteries and their battery life. They were all included in a rather simple discrete time event simulation algorithm - simple in the sense that the decisions made inside it could be largely improved in the future - which served as the target algorithm for the rest of the thesis.

After having defined what will be studied through this work it was important to present the different state-of-the-art configurators that are implemented in the Java library that is linked to this thesis. This allowed - I hope - the reader to better grasp the idea behind these algorithms in order to convince him or her that they are suited for solving the problem at hand. Five main algorithms or frameworks were described : GGA which uses the power of genetic algorithms, ParamILS which relies on an iterated local search method, SMAC which implements a random forest regression model to guess which part of the parameter domains should be explored, IFRace which counts on a racing procedure and on statistical tests to discover the best configurations and finally GGA++ which is a model-based extension of GGA. Alongside these descriptions I also gave an overview of other methods that exist in the literature and I ended the chapter by discussing the pros and cons of these parameter tuning algorithms as well as the concrete library implementations.

Once these two components were defined it was time to conduct some ex-

periments to understand how these algorithms must be used and to identify the strengths and the weaknesses of each of them for the application at hand. For this purpose I compared the different configurators on three sets of instances to study their capabilities under different workload constraints. The uniform instance set was used to see if they were capable of finding a configuration that is well suited for a continuous and regular arrival of the trucks while the Gaussian instance set would rather study their capability to find a configuration that is able to handle a large amount of missions in a small amount of time. Finally, a third set combining the two preceding ones was used to evaluate if it was possible to find a configuration that would handle both cases as they could both happen in a real environment. Runs of only three hours were used to compare the algorithms because of the time constraints that rule this thesis. We saw that it was not enough to explore a sufficiently large part of the initial parameter space but nevertheless that the results were already quite convincing and logical with respect to the constraints of the instances. In this experiment we discovered that IFRace performed worse than its pairs and that it might be due to a poor choice of its hyper-parameters. We also saw that ILS was performing quite well on all sets of instances and that SMAC was finding the most robust configurations.

To complete this experimental phase, I studied if improvements over the results could be obtained by running the algorithms for a longer time or by reducing the parameter space size. It seems obvious that allocating more time to the configurators would have led to better results than what was presented for the three hours runs as the space would be much more explored and the incumbents would be evaluated on far more different instances, leading to a better robustness. However, I also concluded that reducing the parameter space size could not always be beneficial as the lower granularity of the space could induce the configurators to find suboptimal values for the parameters. In an other section I showed that modifying the objective aggregation to fit the user's constraints in terms of monetary costs for example indeed impacts the obtained results but sometimes to a lesser extent. This would allow future users to find a good balance between the service time of the trucks and the monetary costs of the warehouse depending on their needs. In a last experiment, which was initially designed to study the impact of the hyper-parameters of the configurators on the results, I tried to evaluate to which extent defining dependencies between the parameters would help the configuration algorithms to find blocks of values that go well together. The results of this experiment showed that it is very difficult to guess values for the hyper-parameters that would fit the problem at hand. The state-of-the-art is admittedly efficient and powerful but also complicated.

Finally, I described in chapter 5 what are the future prospects of this work.

As this thesis is far from being complete, it would be interesting to improve the work that was done in one hand but also to explore new possibilities on the other hand such as studying other existing parameter tuning algorithms or extending the studied ones. It would also be possible to improve the target algorithm - i.e. the simulation - to handle more parameters or to design smarter subroutines to better simulate the environment of a a real warehouse.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Holger H Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.

[2] David Silver. Cooperative pathfinding. *Aiide*, 1:117–122, 2005.

[3] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[4] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 142–157. Springer, 2009.

[5] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Manuel López-Ibánez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[7] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *International workshop on hybrid metaheuristics*, pages 108–122. Springer, 2007.

[8] Radu Marinescu and Rina Dechter. And/or branch-and-bound for graphical models. In *IJCAI*, pages 224–229. Citeseer, 2005.

[9] William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998.

[10] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020.

[11] Sidney Siegel. Nonparametric statistics for the behavioral sciences. 1956.

[12] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[13] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. Sequential parameter optimization. In *2005 IEEE congress on evolutionary computation*, volume 1, pages 773–780. IEEE, 2005.

[14] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin P Murphy. An experimental investigation of model-based parameter optimisation: Spo and beyond. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 271–278, 2009.

[15] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[16] Frank Hutter. *Automated configuration of algorithms for solving hard computational problems*. PhD thesis, University of British Columbia, 2009.

[17] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. Model-based genetic algorithms for algorithm configuration. In *IJCAI*, pages 733–739, 2015.

[18] Wei-Yin Loh. Classification and regression tree methods. *Wiley StatsRef: Statistics Reference Online*, 2014.

[19] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[20] Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren. *Learning and Intelligent Optimization: 10th International Conference, LION 10, Ischia, Italy, May 29–June 1, 2016, Revised Selected Papers*, volume 10079. Springer, 2016.

[21] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[22] Deng Huang, Theodore T Allen, William I Notz, and Ning Zeng. Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of global optimization*, 34(3):441–466, 2006.

[23] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin Murphy. Time-bounded sequential parameter optimization. In *International Conference on Learning and Intelligent Optimization*, pages 281–298. Springer, 2010.

[24] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations research*, 54(1):99–114, 2006.

[25] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756. Citeseer, 2010.

[26] Yuri Malitsky and Meinolf Sellmann. Stochastic offline programming. *International Journal on Artificial Intelligence Tools*, 19(04):351–371, 2010.

[27] Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. Mo-paramils: A multi-objective automatic algorithm configuration framework. In *International Conference on Learning and Intelligent Optimization*, pages 32–47. Springer, 2016.

[28] Hojjat Rakhshani, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. Mac: Many-objective automatic algorithm configuration. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 241–253. Springer, 2019.

[29] Ronald Aylmer Fisher et al. The design of experiments. *The design of experiments.*, (2nd Ed), 1937.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**