Optimization internship report
BDR Thermea Group

Dimitri Klockenbring

Under the supervision of Christophe Prud'homme and Aurélien Massein

August 24, 2021

Contents

1 Introduction

1.1 Presentation of the company

BDR Thermea Group is a leading european company (third in Europe) specialized in the making and selling of domestic and industrial heating appliances. The firm was created in 2007 by the merger of several leading european brands (De Dietrich, Remeha). Based in the Netherlands, the firm has customers in more than 100 countries and has 6300 employees around the world. It strives for a near-zero carbon footprint by developping products that serve energy and cut carbon emissions.

1.2 Objectives of the internship and used tools

During this internship, we have been looking to implement and test evolutionnary algorithms in order to solve complex multi-objective optimization problems. We had to statistically compare multiple possible configurations for different algorithms and find the most adequate parameters. Then we had to test the chosen solution in a digital simulation with a numerical model.

We had specific objectives to achieve along the way : first we did a bibliographic research to get familiar with some state-of-the-art algorithms. Then we worked on implementing some of these algorithms, run tests in parallel and analyze the data. For the analyzis, we had to come up with a protocol for evaluation.

We developped our tools using the Python language. We used the libraries Pymoo, FMPy and Joblib to implement the algorithms, and for the statistics we used Pandas, Seaborn and Matplotlib.

We used an Azure Devops Git repository temporarily provided by BDR Thermea to store and update our code.

2 The optimization problem

2.1 Studied products

During this internship, we have been looking to optimize two different products :

- a thermodynamical water-heater [1], fig. 1a, which we will refer to as TWH
- a heatpump [2], fig. 1b, which we will refer to as IDU3FS

(a) TWH (b) IDU3FS

Figure 1: Studied products

We considered that both products are configured by three parameters :

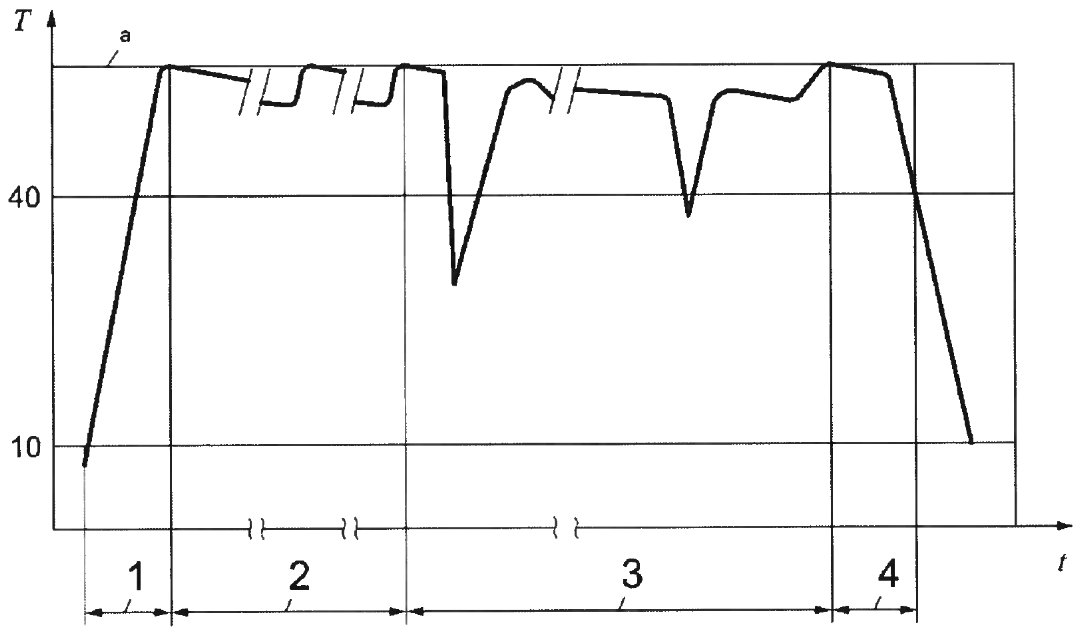
- T_{set} is the setpoint temperature, the maximal temperature to reach in order to stop the heating process
- $\Delta T_{\text{hysteresis}}$ is the temperature hysteresis, as $T_{min} = T_{\text{set}} - \Delta T_{\text{hysteresis}}$ is the minimal temperature to reach in order to start the heating process
- H_{gauge} is the height-location (from the top) of the temperature sensor mount onto the water heater which measures and regulates the tank's water temperature

There are two main objectives we want to optimize :

- the coefficient of performance (COP_{DHW}) that is definied by the EN16147:2017 norm [3]
- a number of stars (Stars) defined by the LCIE 103-15/C specifications [4], that is calculated based on the performances measured during the EN16147:2017 test protocol

2.2 The EN16147:2017 norm

The first objective we are interested in is the COP_{DHW} defined by the EN16147:2017 norm [3]. This norm consists in a test protocol performed on the product under real conditions, following the steps shown in fig. 2, [3]. It is mandatory to sell the product in the European Union market.



Légende

1	[Étape C] remplissage et période de mise en température (voir 7.7)	T	température
2	[Étape D] Puissance absorbée en régime stabilisé (voir 7.8)	t	temps
3	[Étape E] Puisages d'eau (voir 7.9)	a	température de consigne
4	[Étape F] Eau mitigée à 40 °C et température d'eau chaude de référence (voir 7.10)		

Figure 2: EN16147:2017 standard test protocol steps [3]

It can last around seven days, depending on the parameters. Here is a quick overview of the different steps and some of their associated measures [3] :

- step C, warming-up : filling the tank and heating the water up to T_{set}. We define t_{h}, the duration of temperature.
- step D, steady state : we let the water cool down naturally (due to the water tank's heat losses) until the heating process restarts, then wait until T_{set} is reached, and repeat this process. No water is drawn during this step. We define P_{es}, the absorbed power in steady state (in kW)
- step E, water drawings : for the test protocol, one must choose a profile of use among the ones in [3]. These profiles are meant to simulate a use of the product with a certain intensity. The water drawing cycle lasts 24 hours, however t_{TTC} the duration of this step can last more as we are also waiting as described in the previous step.
- step F, water emptying : all the water is drawn out of the tank. During this step, we keep track and measure the water drawn volume and the outlet temperature until the drawn water reaches $40°C$. We define V_{40} as the volume of the mixed water at $40°C$, and θ'_{WH} the average water outlet temperature; detailed in [3].

For this study, we chose the profiles M and L. After step E, we define the Coefficient Of Performance of Domestic Hot Water COP_{DHW} as

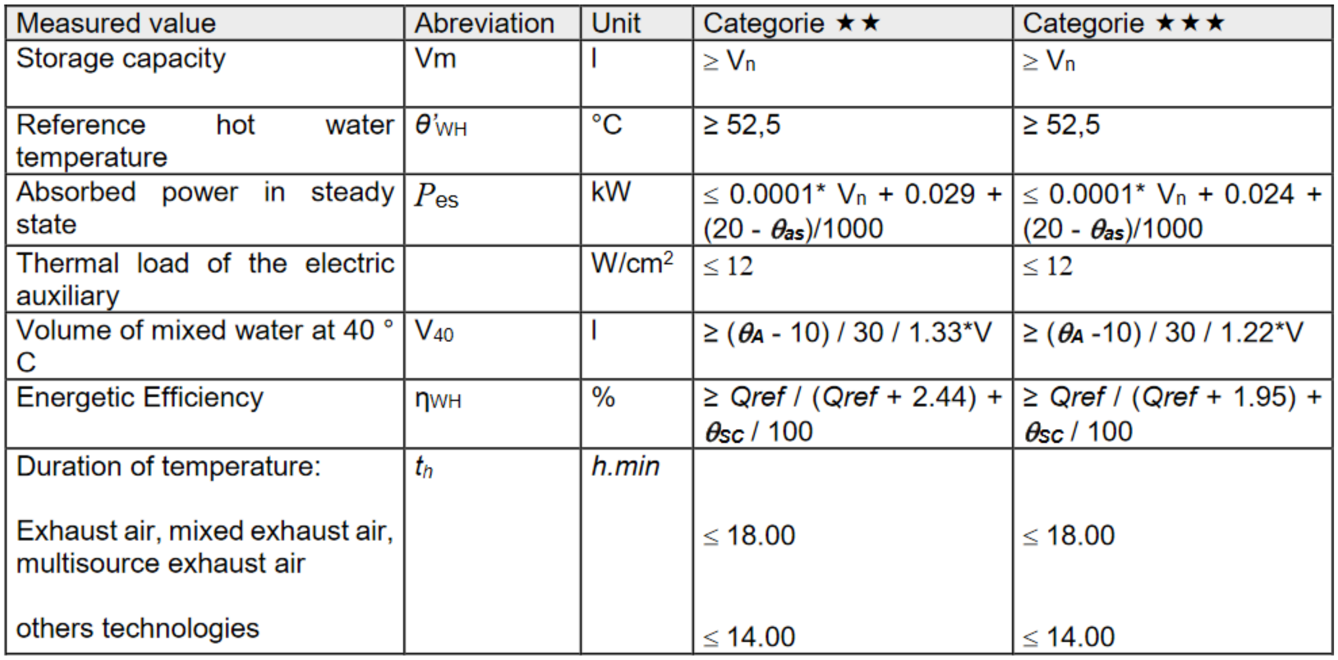
$$\text{COP}_{\text{DHW}} = \frac{Q_{\text{LP}}}{W_{\text{EL-LP}}}$$

where Q_{LP} is the total useful energy during the whole drawing profile, in kWh, and $W_{\text{EL-LP}}$ the raw electrical power consumption during the whole step, in kWh
Technically $W_{EL-LP}(t_{TTC}, P_{ES})$ depend on t_{TTC} and P_{ES} as a corrective factor is applied; this highlights the mathematical complexity underlying in the norm's performance computations.

Finaly, a percentage of energetic efficiency η_{wh} is defined [see details in 4]. This value will also be used to define the stars grading in the LCIE 103-15/C specifications.

2.3 The LCIE 103-15/C specifications

The second objective we are interested in is the stars notation from the LCIE 103-15/C specifications [4]. The grade provides an additional selling point for the product, especially in the French market, as its alternative name is "NF Électricité Performance". It is calculated based on the results of the EN16147:2017 test protocol. For each of the variables in fig. 3, threshold values are defined for each star notation. The products gets one star if the EN16147:2017 test protocol is valid (mandatory), and two or three stars if all the variables pass the thresholds in the corresponding column.



Measured value	Abreviation	Unit	Categorie ★★	Categorie ★★★
Storage capacity	Vm	l	$\geq$ V$_n$	$\geq$ V$_n$
Reference hot water temperature	θ'_{WH}	°C	$\geq$ 52,5	$\geq$ 52,5
Absorbed power in steady state	P_{es}	kW	$\leq$ 0.0001* V$_n$ + 0.029 + (20 - θ_{as})/1000	$\leq$ 0.0001* V$_n$ + 0.024 + (20 - θ_{as})/1000
Thermal load of the electric auxiliary		W/cm^2	$\leq$ 12	$\leq$ 12
Volume of mixed water at 40 ° C	V$_{40}$	l	$\geq$ (θ_A - 10) / 30 / 1.33*V	$\geq$ (θ_A -10) / 30 / 1.22*V
Energetic Efficiency	η_{WH}	%	$\geq$ *Qref* / (*Qref* + 2.44) + θ_{sc} / 100	$\geq$ *Qref* / (*Qref* + 1.95) + θ_{sc} / 100
Duration of temperature:	*t$_h$*	*h.min*		
Exhaust air, mixed exhaust air, multisource exhaust air			$\leq$ 18.00	$\leq$ 18.00
others technologies			$\leq$ 14.00	$\leq$ 14.00

Figure 3: Thresholds to reach to get two or three stars [4]

2.4 Reformulation of the constrained multi-objective optimization problem

The main goal we want to achieve is to maximize the COP_{DHW} as well as the stars grade, depending of the parameters $T_{\text{set}}, \Delta T_{\text{hysteresis}}, H_{\text{gauge}}$. In this study, we also worked with discretized parameters, so the design space and the optimization problem can be defined as such :

- Let $\mathbb{T}_{\text{set}} = \{315.15 + 1 \cdot k \mid k \in [\![0, 15]\!]\}$ every 1 Kelvin degree
- Let $\Delta\mathbb{T}_{\text{hysteresis}} = \{2 + 1 \cdot k \mid k \in [\![0, 28]\!]\}$ every 1 Kelvin degree
- Let $\mathbb{H}_{\text{gauge}} = \{0 + 0.05 \cdot k \mid k \in [\![0, 18]\!]\}$ every 0.05 meter

We have $Card(\mathbb{T}_{\text{set}}) = 16, Card(\Delta\mathbb{T}_{\text{hysteresis}}) = 29, Card(\mathbb{H}_{\text{gauge}}) = 19$. We can now define the design space $\mathcal{X}$ as

$$\mathcal{X} = \mathbb{T}_{\text{set}} \times \Delta\mathbb{T}_{\text{hysteresis}} \times \mathbb{H}_{\text{gauge}}$$

such that $Card(\mathcal{X}) = 8816$

We an then define our multiobjective optimization problem :

$$\underset{u \in \mathcal{X}}{\text{argmin}}\, f_1(u), \text{where } f_1 : u \in \mathcal{X} \mapsto -\text{COP}_{\text{DHW}}(u)$$

$$\underset{u \in \mathcal{X}}{\text{argmin}}\, f_2(u), \text{where } f_2 : u \in \mathcal{X} \mapsto -\text{Stars}(u)$$

to which we add the constraint $\forall u \in \mathcal{X}, \text{Stars}(u) \geq 1$ (the norm EN16147:2017 has to be passed) :

$$g(u) \leq 0 \quad \forall u \in \mathcal{X}, \text{where } g : u \in \mathcal{X} \mapsto 1 - \text{Stars}(u)$$

In a first approach to perform this optimization, one could define a single objective optimization problem. This could be done by choosing an adequate fitness function $f : \mathbb{R}^2 \mapsto \mathbb{R}$ that would attribute a unique value to each $(\mathrm{COP_{DHW}}, \mathrm{Stars})$ couple. For example, we could use a weighted sum. This would present the benefit of simplifying the problem, however, this approach would require to judge in advance which of the objectives is the most important and to what extent (Plus, both objectives don't have the same nature). This is not an easy thing to do, and the priority of an objective over the other may vary depending on the context (in markets outside of France, the stars grade can be less important than the $\mathrm{COP_{DHW}}$). This is why for this study we chose to avoid this approach and prefered the search of a non-dominated front, also known as Pareto front [5]. Let's first give a definition of *Pareto dominance*.

Let $x = (x_1, \cdots, x_n), y = (y_1, \cdots, y_n) \in \mathbb{R}^n$. We suppose that we want to minimize $x_1, \cdots, x_n, y_1, \cdots, y_n$. We say that x *dominates* y, resp. y is *dominated* by x, if

$$x_i \leq y_i \quad \forall i \in [1, \cdots, n]$$

and

$$\exists i \in [1, \cdots, n] \text{ such that } x_i < y_i$$

In other words, x is at least as good as y in every objective and does strictly better in at least one objective.

Now let $Y \subset \mathbb{R}^n$ be an objective space, the *Pareto front* of X is the set of all $x \in X$ that are not dominated by any other point in X. This can be visualized in the following example :

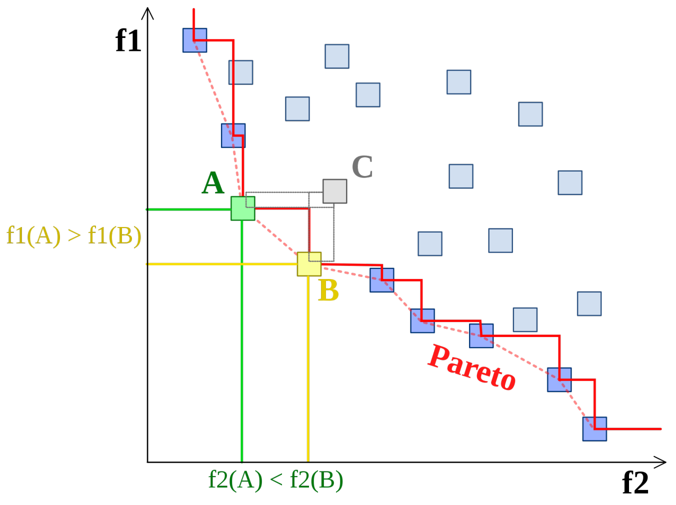


Figure 4: Example of a Pareto front with two objectives to minimize [5]

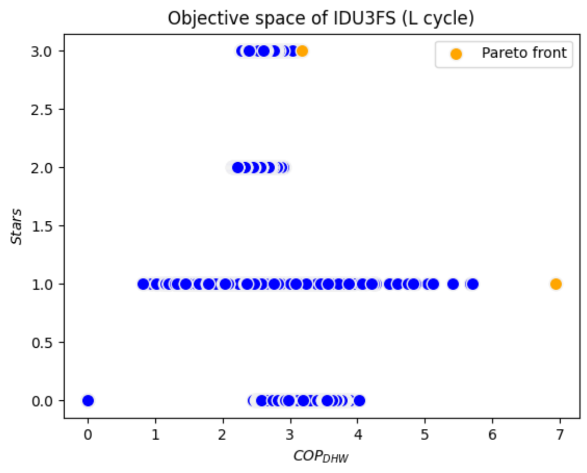
2.4.1 A modified objective : floating stars

Because Stars can only take values in $\{0, 1, 2, 3\}$, we can expect at most four individuals in the Pareto front. We tried to create a new indicator to get more solutions. The idea was to add a floating number between zero and one to the stars grade of each individual, in order to see how close we are from the getting the next grade, and also how much the threshold are satisfied, their satisfaction margin. To do this we first managed to define a "proximity measure" of each variable to the next threshold, then we defined a weighted sum of these proximities that we added to the original stars grade. The result can be visualized in sec. 2.5

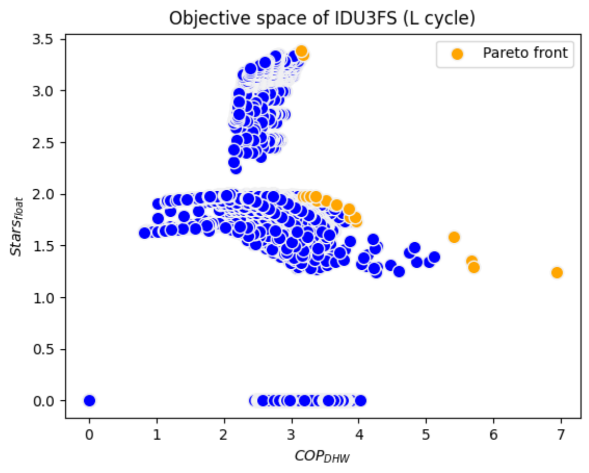
2.5 FMU Simulation, design space, data visualization

As explained in sec. 2.2, the test protocol of the EN16147:2017 norm is very long, and the water drawings in step E make the situation hard to understand. Therefore, we used functional mock-up interface (FMI) simulations [6] in order to explore the objective space. We used functional mock-up unit (FMU) files developped with the software Dymola. The language used in Dymola is *Modelica* and C in the underlying language used in the FMU runs. The FMU provides a deterministic approximation of the results of the test protocol given the profile of use (L or M). A test simulated this way can take around 20 minutes, but it depends greatly on the settings $u \in \mathcal{X}$. Methods such as metamodelisation [7] can't be used because the model is still very complex (around 40000 equations), involves proprietary DLLs with unknown code, and each step of the test is itself described with complex algorithms, especially the water drawing phase at step E, during the first 24 hours where a lot of thresholds and requirements need to be checked and satisfied.

We can now visualize the objective space and its Pareto-optimal front. If we keep the original stars objective, we observe as stated in sec. 2.4.1 that there are only two solutions in the front fig. 5a. The floating stars grade that we built allows for a larger choice fig. 5b.



(a) Objective space of IDU3FS without floating stars



(b) Objective space of IDU3FS with floating stars

Figure 5: Objective space of IDU3FS

We can also observe the individuals of the Pareto set in the original design space $\mathcal{X}$. In fig. 6 we hid the points that faild the EN16147:2017 test protocol :

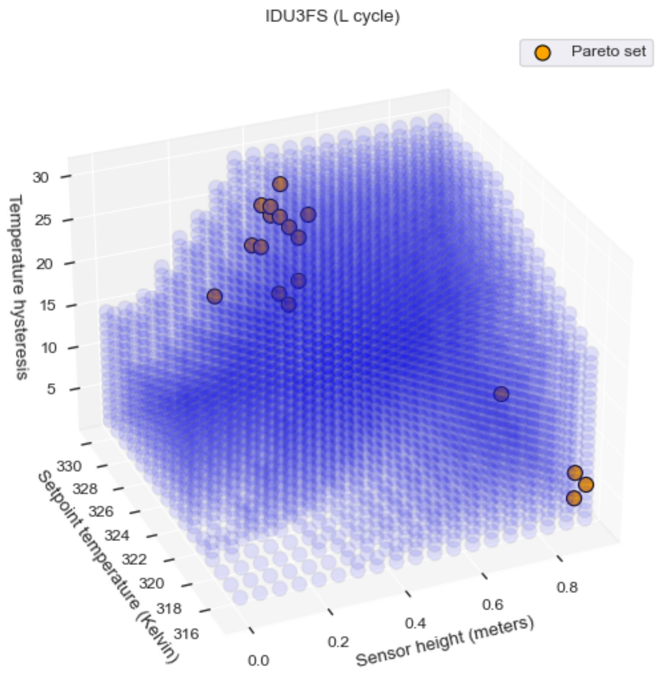
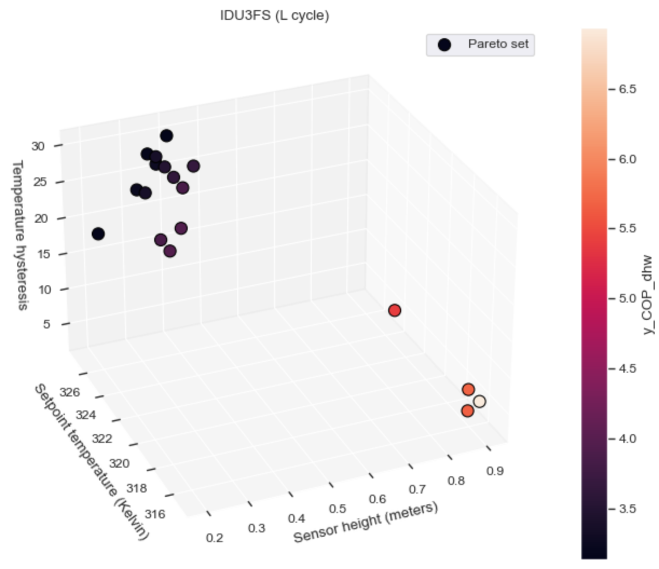
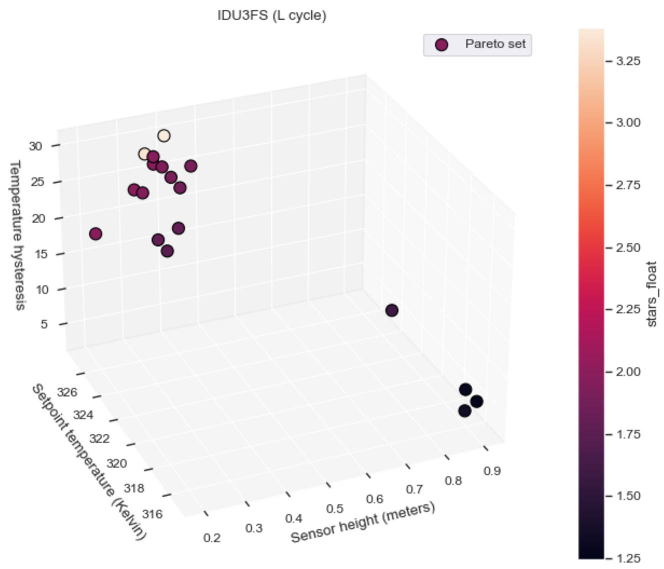


Figure 6: Design space and Pareto set of IDU3FS with floating stars

We observe that the Pareto front corresponds to two distinct areas in the original design space $\mathcal{X}$, and there is also a trade-off between these two zones : the highest stars values have lower associated COP_{DHW} values, and vice-versa (fig. 7)



(a) COP_{DHW}



(b) Stars (floating)

Figure 7: IDU3FS (L cycle) COP_{DHW} and Stars trade-off

3 Method and results

3.1 Proposed approach

As we have seen, the studied problem is complex : the functions we are working with are not analytical, so classical optimization methods such as gradient descent are not as helpful. If we were to use a gradient descent, the result would greatly depend on our starting point : this problem and space is not convex, and we will get stuck in local minimal, so we would have to execute several gradient descent algorithms at different and random starting points. Futhermore, the real laboratory test values of $\mathrm{COP}_{\mathrm{DHW}}$ and Stars (given a specific use cycle) are not deterministic, as the results EN16147:2017 test protocol may vary. However, these values are deterministic in our context because we are using an FMU to compute them (*Modelica* language is generating a deterministic system).

We are going to use evolutionary algorithms to solve our problem. These are metaheuristical algorithms [8] popularized by John H. Holland from 1975 through *genetic algorithms* [9]. They are inspired from natural selection and are useful to solve otherwise complex optimization problems. The overall approach is the following, as describes in [10] :

- Step 1 : Generate an initial population of random individuals (generation 0). For this step, a *sampling* operator is used.
- Step 2 : Select the fittest individuals to breed the next generation. For this step, a *selection* operator is used.
- Step 3 : Mate the individuals selected. We get a new set of individuals called the offspring. For this step, a *crossover* operator is used.
- Step 4 : Apply random mutations to the the whole population (offspring included). For this step, a *mutation* operator is used.
- Step 5 : Repeat steps 2, 3, 4 until a termination criterium is reached. Each tipe, a *survival* operator is used if needed to choose individuals to be dropped (when the population is full)
- Step 6 : At the end, select the fittest individual among all generations

Many such algorithms exist and share the same structure. They are mainly distinguished by the operators that are used to perform these steps. Examples of such operators are listed in [11]. Our approach for this study will be to select the adequate evolutionary algorithm and its parameters for our problem, from already implemented algorithms, in Python language [12]. In a first step, we will run these algorithms on the same problems and data sets, using the results of the FMU simulations that are already available to us. We will statisticaly compare the algorithms before retaining one of them and the best set of parameters we found for it. In a second step, we are going to run the chosen algorithm by directly calling the FMU. In order to do this we will have to parallelize the evaluations of the individuals, because every single evaluation will be much slower than in the first step (around 20 minutes to evaluate one individual). As we specified our problem is limited to 8816 solutions, and our datasets explored all solutions: therefore we also aim to minimize the time needed to find a Pareto front close to the true (and known) Pareto front.

3.2 State-of-the-art evolutionary algorithms

There are a lot of evolutionary algorithms available, but the ones we chose to run for this study are the following :

- MOEA/D [13] : Multiobjective Evolutionary Algorithm Based on Decomposition (2007)
- NSGA-II [14] : Non-dominated Sorting Genetic Algorithm II (2002)
- NSGA-III [15] : Non-dominated Sorting Genetic Algorithm III (2014)
- U-NSGA-III [16] : Unified Non-dominated Sorting Genetic Algorithm III (2016)
- C-TAEA [17] : Constrained Two-Archive Evolutionary Algorithm (2019)

The reason for this selection is the fact that these algorithms have been implemented in Pymoo [18], the Python library we chose for this study. Other tools were available, mainly the DEAP [19] and jMetalPy [20] libraries, that could have been used as well. Pymoo was interesting because it is a recent project that was developped under the supervision of K. Deb, the inventor of NSGA2, by his PhD student. We have to emphasize that Pymoo is a recent and a one-man developed library, so it may be prone to errors; as it was not long or thoroughly tested.

Unfortunately, we encountered difficulties to run MOEA/D and C-TAEA properly : these algorithm would crash very often after a certain number of generations, and we were not able to find the reason for this behavior. We can also regret that Pymoo doesn't offer any implementation for Particle Swarm Optimization [21]. Therefore we focused on the three NSGA algorithms. We also implemented an algorithm for random search, in order to be able to compare it with the smarter methods.

We will give a quick description of an iteration of our main algorithm, NSGA-II [1]. Let N be the size of the population. At the start of each iteration, we begin with a combined population $R_t = P_t \cup Q_t$ of size $2N$ (the parents and their offspring). The population R_t is sorted in successive non-dominated fronts $F_1, F_2, \cdots$ using fast non-dominated sorting procedure (see [14]). To choose individuals for the nex population P_{t+1}, NSGA-II will put the most emphasis on the individuals from the first fronts (this is called *elitism*) : if there are less than N individuals in F_1, all individuals from F_1 will be part of P_{t+1}. We continue filling P_{t+1} with the individuals from $F_2, F_3 \cdots$. At some point, we will likely reach a front F_l such that $Card(F_1 \cup \cdots \cup F_l) > N$. This will be the last set from which we will take individuals. To choose the individuals to select from F_l, a crowded-comparison operator (see [14]) is used. This operator tries to guarantee diversity by selecting in priority the points with fewer neighbours in the non-dominated front. The new population P_{t+1} is now complete. Then we select individuals to mate to generate the new offspring Q_{t+1} and the procedure restarts untill a termination criterion is reached.
The overall complexity of NSGA-II is $O(MN^2)$, where M is the number of objectives to optimize and N the size of the population. This complexity is mostly due to the non-dominated sorting procedure.

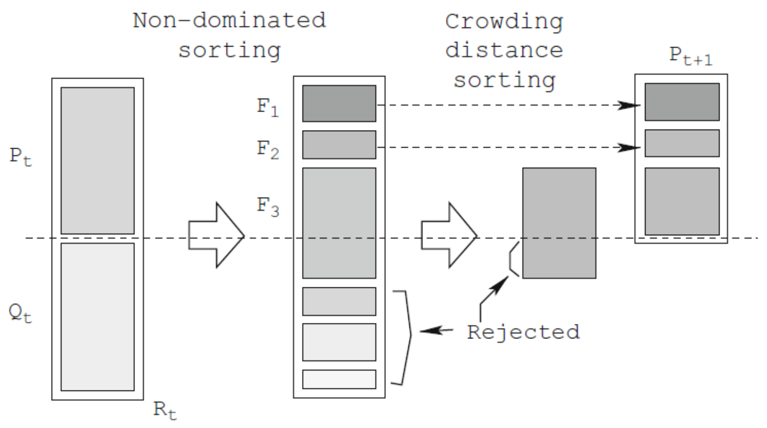


Figure 8: Principle of NSGA-II

3.3 Implementation of the problem and algorithms

We chose to rely on the implementations from the Python Pymoo library [18] to test our algorithms.

To do the implementation, we first had to define our optimization problem. Pymoo provides a `Problem` class in which we can set all the characteristics of our problem. The `__init__` method of this class provides a few useful parameters :

- `n_var` : the number of attributes, in this case three $(T_{\text{set}}, \Delta T_{\text{hysteresis}}, H_{\text{gauge}})$
- `n_obj` : the number of objectives, two in this case $(\text{COP}_{\text{DHW}}, \text{Stars})$
- `n_constr` : the number of constraints, only one here $(\text{Stars} \geq 1)$
- `x_l` and `x_u` : the minimal and maximal values for the attributes

[1]NSGA-III is a variant for problems with many objectives, and U-NSGA-III is an attempt at creating an algorithm adapted to both problems with few and problems with many objectives.

- `elementwise_evaluation` : we set it to `True` to indicate that individuals in our population will be evaluated one at a time [2].

We choose to discretise our algorithm through enumeration, to handle the model and product physical limitations on our parameters.
We set
`x_l` $= [0, 0, 0]$
and `x_u` $= [Card(\mathbb{T}_{\text{set}}), Card(\Delta\mathbb{T}_{\text{hysteresis}}), Card(\mathbb{H}_{\text{gauge}})] = [16, 29, 19]$
This way, we don't have to provide the steps of our discretization to Pymoo. We made a `attributes_from_coord` function to obtain the actual values of the attributes.

The rest of the problem is mainly defined by the function evaluating individuals. This function calculates the values of the objectives and constraints of a given individual. First it calls the `attributes_from_coord` function to get the attributes of the individual, then it searches the values of the objectives in a CSV file that we provide. This CSV contains all the results of the FMU simulations performed in the design space $\mathcal{X}$. The values of the objectives are then stored in the `out` attribute of the class `Problem` : `out["F"]` contains the objectives and `out["G"]` the constraints. We also added a `unique_evals` attribute to this class, that returns the total number of different individuals that were evaluated. The full code of the problem we defined is available in appendix sec. 6.1.1.

The implementations of the algorithms provided by Pymoo could be used as-is, but we added `wrapper` function to do some slight tweaks and add functionalities. For example, the wraper automaticaly converts the floating operators of a given algorithm into the corresponding integer operator (functions for the conversion are provided by Pymoo), and it also generates a text description of the algorithm (names and parameters of the operators) that will be used for the statistics.

The parameters we worked with for each algorithm are the following :

- `pop_size` (NSGA-II) or `ref_dirs` (NSGA-III, U-NSGA-III) : the size of the population used for the evolution. In NSGA-II and U-NSGA-III, reference directions (points in the objective space) are needed to help the algorithm select individuals if the population gets too big (in our case, this is not really an issue, but it might become one if we add more objectives to the problem). By default, the population size in NSGA-III and U-NSGA-III is set to the number of reference directions.
- `crossover` : we used the default `SimulatedBinaryCrossover` operaor provided in Pymoo. We varied the probability of crossover `prob` $\in \{0, 0.25, 0.50, 0.75, 1\}$ and the value of `eta` $\in \{15, 30\}$ (a bigger `eta` value provides an offspring closer from the parents)
- `mutation` : we used the default `PolynomialMutation` operaor provided in Pymoo. We varied the probability of mutation `prob` $\in \{$`None`$, 0, 0.25, 0.50, 0.75, 1.0\}$ (`None` is interpreted as 1/`pop_size` which is a commonly recommanded value) and the value of `eta` $\in \{20, 30\}$ (a bigger `eta` value provides a mutation closer from the original offpsring)
- `termination` : we used the `Termination` class from Pymoo to define our own termination criterion, which in this case is very simple : we set a limit of 4000 unique evaluations or 500 generations : as the size of our design space 8816, we want to explore at most half of it. We also put a stop to the algorithm if every point of the known Pareto front have been found.

We then launched the execution of the algorithms with all these different parameters on four different problems : IDU3FS (M cycle), IDU3FS (L cycle), TWH (M cycle), TWH (L cycle). In order to guarantee the reproducibility of the results and to do some statistics, we ran each different parameter with 100 different seeds : each seed generates a unique pseudorandom scenario for the running of the algorithm, which can later be reproduced identically using the same seed. In total, we did $139,200$ unique executions of each of the three algorithms. For each of them, we created a unique CSV file containing the history of the running of the execution, identified by a unique hash key. We ran the processes in parallel on 34 cores with the Python library Joblib.

All $417,600$ executions in parallel took about a week (178 hours), which on average makes an execution every $\sim 1.5s$

[2]When using the FMU simulations later, we will have to change this to be able to run multiple FMUs in parallel.

3.4 Statistics and choice of an algorithm

3.4.1 Measures performed on each algorithm execution

We have chosen the following measures to evaluate the quality of an algorithm :

- Measures of convergence :
  - *Inverted Generational Distance* (IGD) : this measure gives an idea of how close the Pareto front that was found is close from the true front. Let Z be the true Pareto front and A the front found by the algorithm, then the IGD is defined as :

    $$\text{IGD}(A) = \frac{1}{|Z|}\left(\sum_{i=1}^{|Z|} d_i\right)$$

    where d_i is the euclidean distance of z_i to its nearest reference point in A. So if $\text{IGD} = 0$, it means that all points of Z have been found.
  - Number/percentage of individuals found ($n\_found/percent\_found$) from the exact Pareto front.
- Measures of performance :
  - Number of unique evaluations (`unique_evals`)
  - Number of generations (`n_gen`)
  - Number of cycles (`n_cycles`) : a calculated estimation of the total number of groups of 34 tasks that were executed. We first compute an estimation for each individual generation before summing all up. In practice, we have been able to verify that this number is close to $\left\lceil \frac{\texttt{unique\_evals}}{\texttt{n\_gen} * 34} \right\rceil * \texttt{n\_gen}$

In fig. 9, for illustration purposes, we show a side-by-side comparison of the running of NSGA2 and a simple random search, with a population of size 100 and the same seed. We can see that NSGA2 can find points in the true Pareto front much quicker. However in this example NSGA2 didn't explore explore enough of the bottom right area of the objective space (this may or may not happen depending on the seed), resulting in a bigger IGD. This shows the strength of evolutionary algorithms, as they are able to exploit their best individuals to find better solutions quicker, but also their weakness, since such algorithms can get "stuck" in a specific area depending on the set mutation probability.

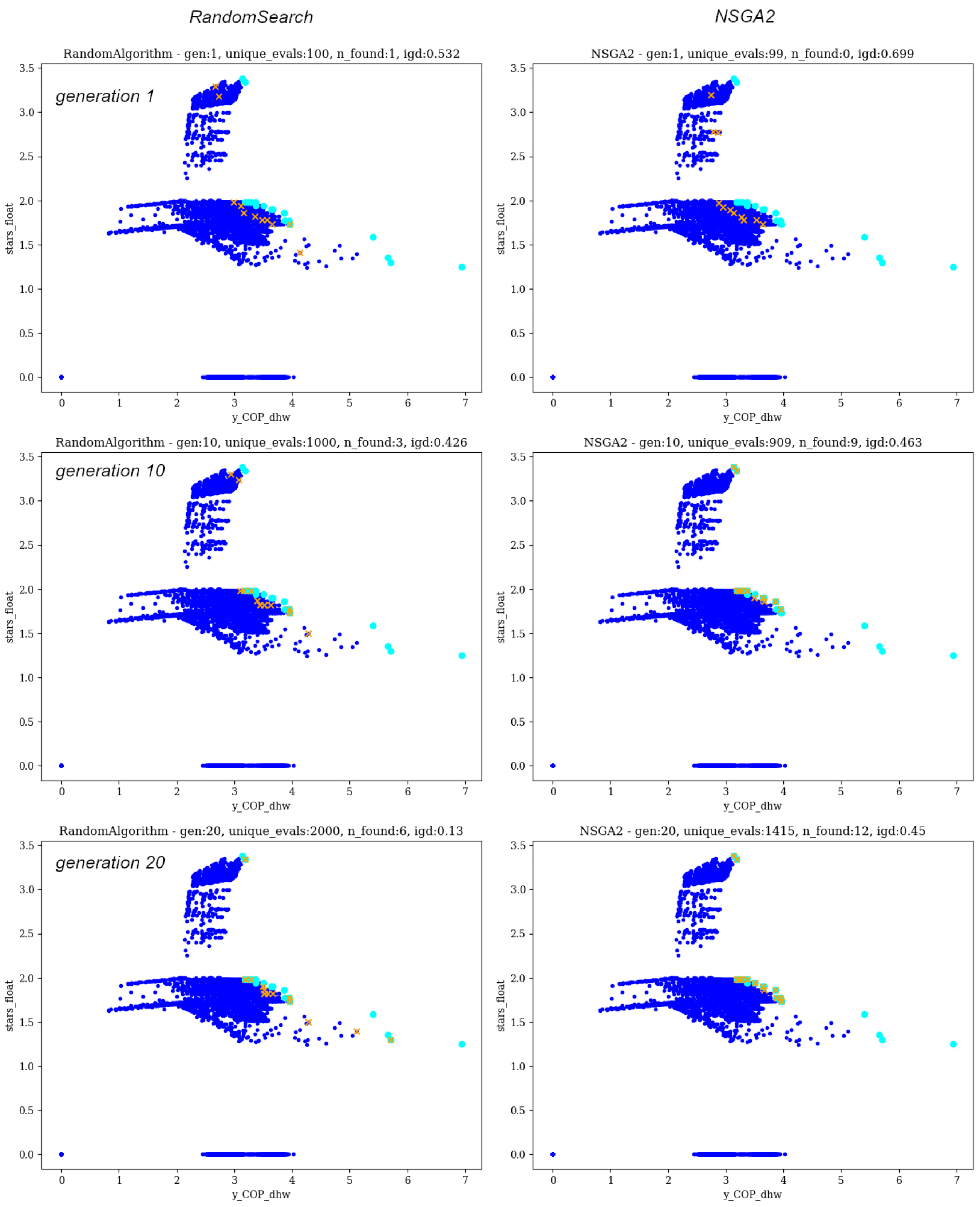
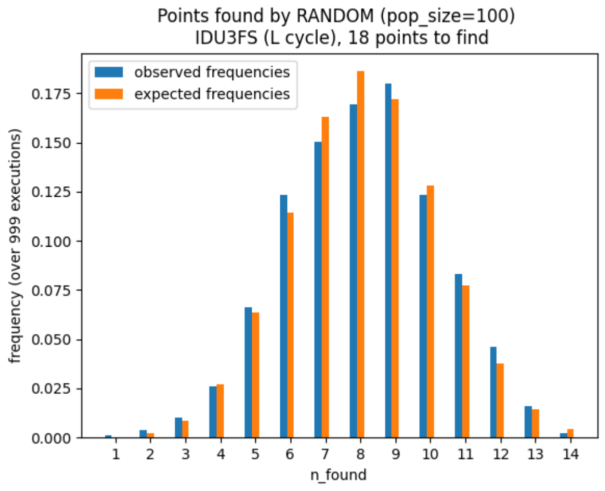
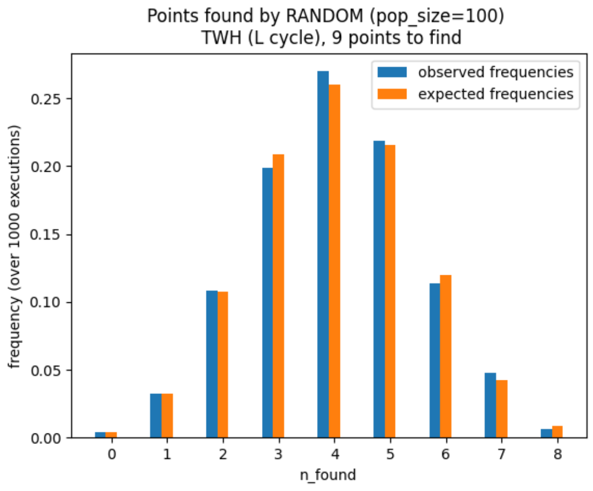


Figure 9: Example of a random search compared to a NSGA2 search, for a given random seed

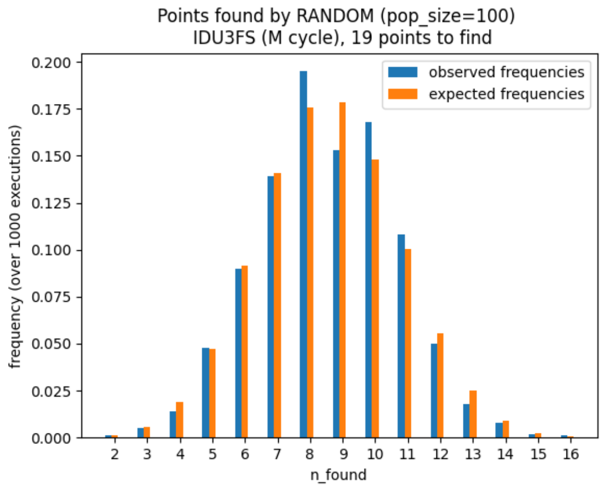
We can further the comparison by the looking at frequencies of the different numbers of individuals found for the random search fig. 10 and NSGA-II fig. 11. For these executions, we set the population size to 100 and the maximal number of unique evaluations to 4000, which is about half the size of the design space $\mathcal{X}$. In fig. 10, we also showed the expected frequencies which follow an hypergeometric law [22]. These figures show that NSGA-II is much better than a random search at finding points from the Pareto front. However, we observe that it is not perfect either : there are often a few points that are harder to find. In the first problem for example (IDU3FS, L-cycle), the last four points of the Pareto front (figs. 11a, 5b) are usually harder to find, although they are still found way more often than in a random search.



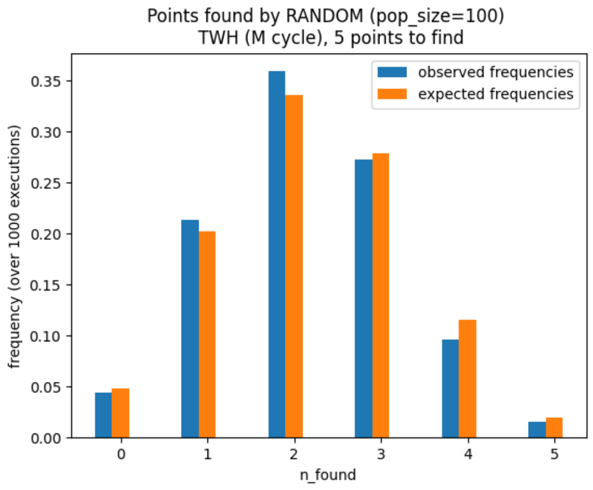
(a) RandomSearch, IDU3FS-L



(b) RandomSearch, TWH-L

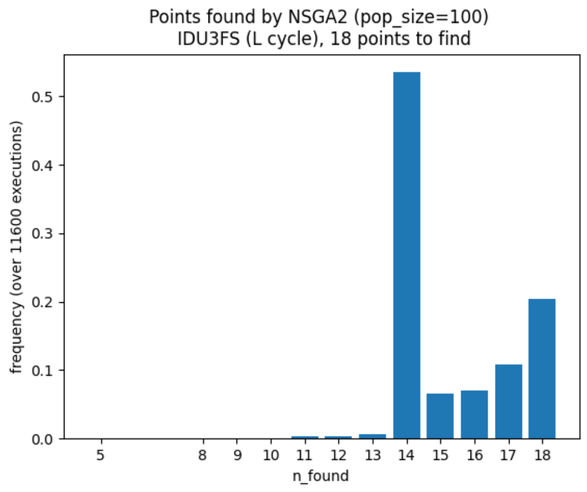


(c) RandomSearch, IDU3FS-M

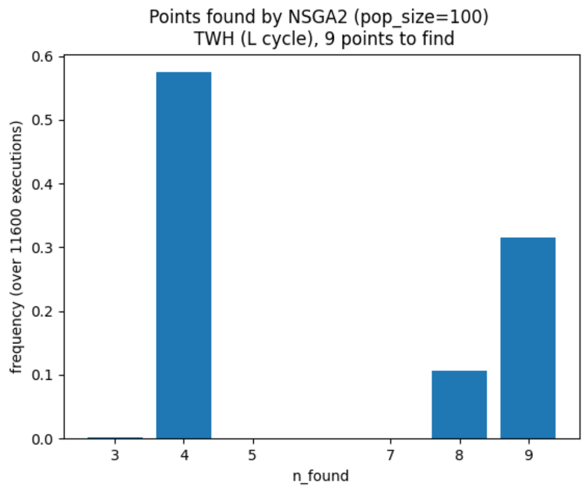


(d) RandomSearch, TWH-M

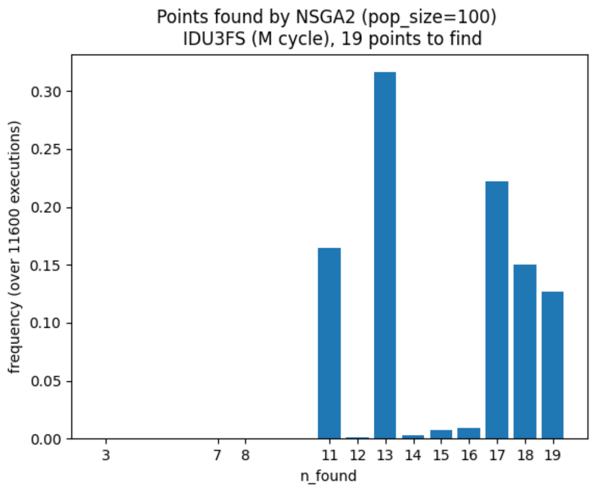
Figure 10: Frequencies of solutions found by random search (`pop_size` = 100)



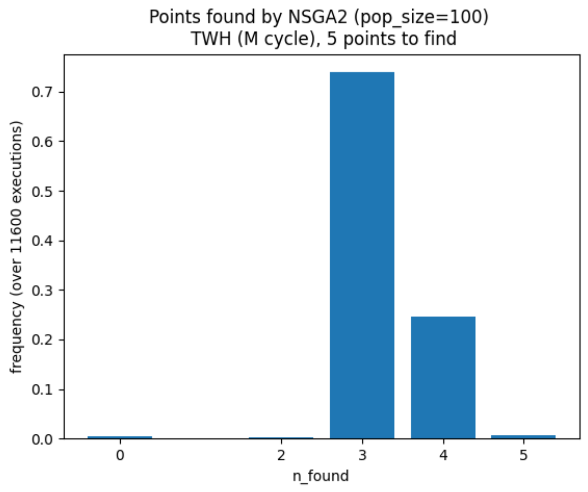
(a) NSGA-II, IDU3FS-L



(b) NSGA-II, TWH-L



(c) NSGA-II, IDU3FS-M



(d) NSGA-II, TWH-M

Figure 11: Frequencies of solutions found with NSGA-II (`pop_size` = 100)

For NSGA-III and UNSGA-III, we found very similar histograms. We can make an overall comparison between all the algorithms, in fig. 12 we chose to compare them on the IDU3FS (L cycle), with a population size of 100. Since we didn't observe any significative difference between the different versions of NSGA for this problem, we chose to focus on the state-of-the-art NSGA-II algorithm for the rest of the study.

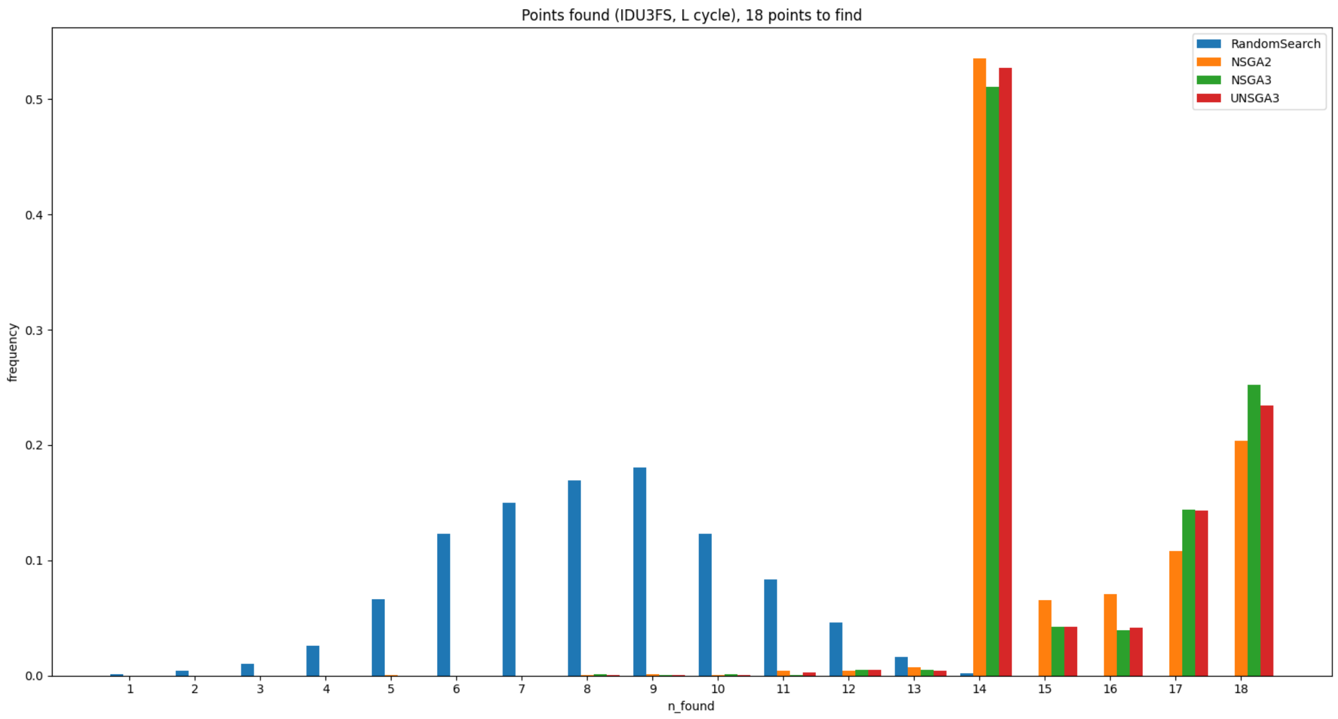


Figure 12: Comparing the accuracty of all algorithms (`pop_size`=100), IDU3FS (L cycle)

3.4.2 Virtual termination criterion and success rate

Using the history files we stored when running the algorithms, we have been able to simulate a custom termination criterion. We chose an IGD of 0.05 as our new termination criterion : for each algorithm run, we extracted the first line of the history where this condition was met (and marked the execution as successful), or we kept the last line in case of failure (and marked the execution as failed).

This allowed us to compute a success rate for each algorithm configuration. We defined it as the proportion of seeds (over 100 seeds) for which the termination criterion was met.

3.4.3 Influence of the population size

In order to choose a best parameter for NSGA-II, we can first look at the influence of the population size on the performances. In fig. 13, each dot represents a configuration of NSGA-II on the problem (IDU3FS, L cycle). The x coordinate represents the success rate of this configuration as we just defined. The y coordinate represents the 95% quantile of `unique_evals` over all successful executions. This informs us on the speed of the algorithm when it converges.

Although we observe that it takes fewer unique evaluations to succeed when the population is small, the success rates we have with population sizes 50 an 100 are too low compared to a bigger population. This should encourage us to always choose a population as big a possible. For the following, we will do the statistics for `pop_size` = 200.

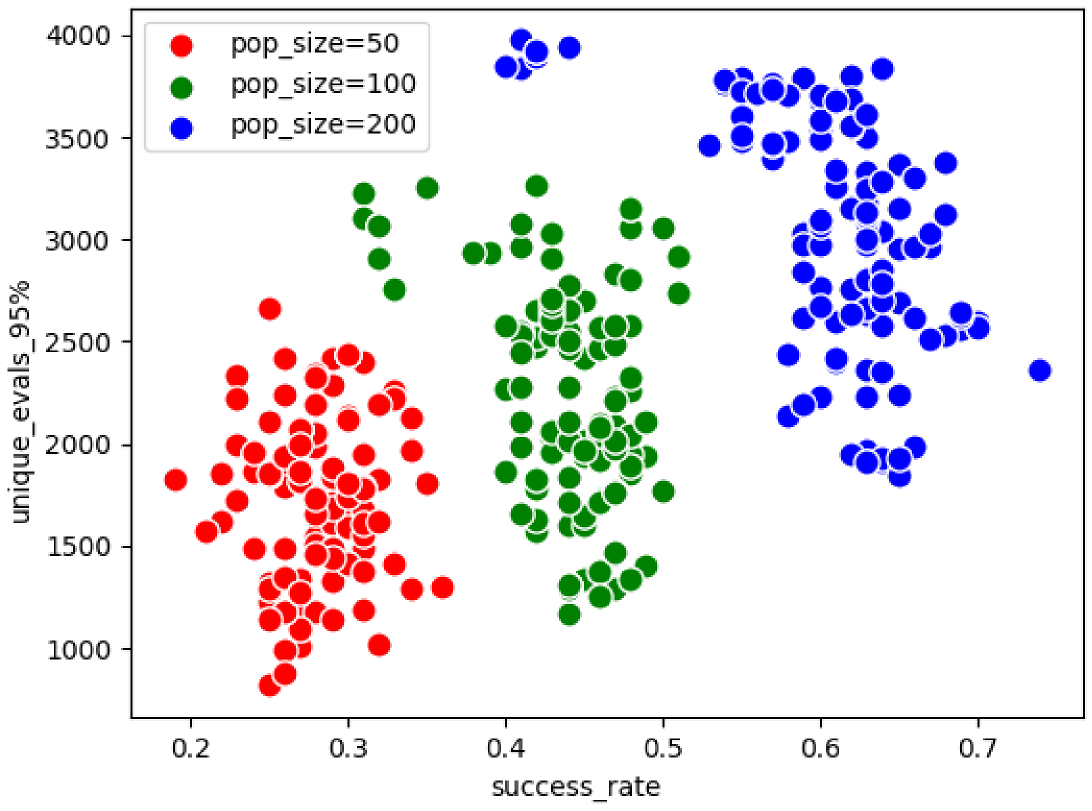
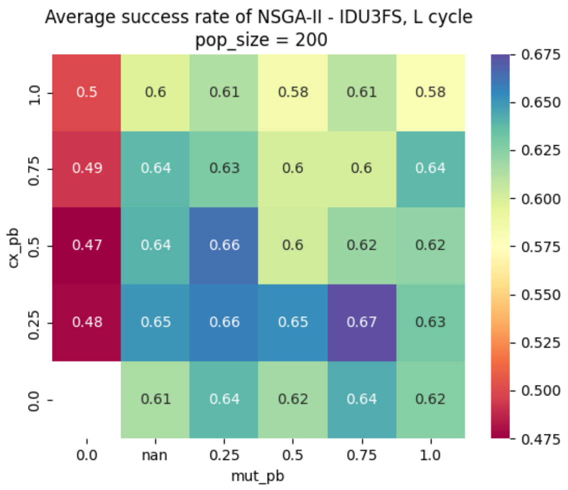


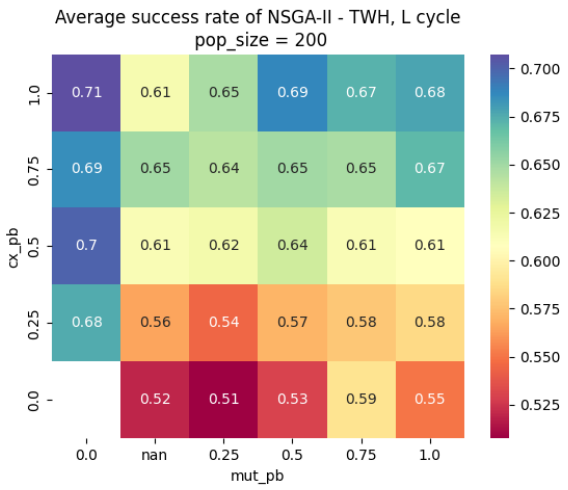
Figure 13: Speed and success rate of NSGA-II for different population sizes (IDU3FS, L cycle)

3.4.4 Influence of mutation and crossover probabilities

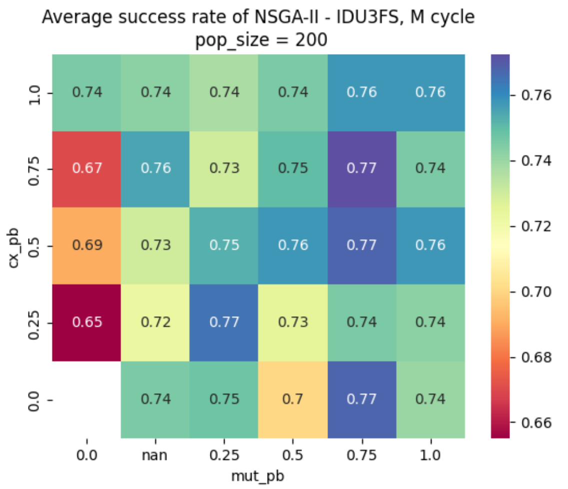
Before committing to a configuration for NSGA-II, we can try to get an idea of the influence of other parameters on the performances of the algorithm. More specifically, let's look at the influence of the probability of crossover `cx_pb` (a parameter of the `SimulatedBinaryCrossover` operator) and the probability of mutation `mut_pb` (a parameter of the `PolynomialMutation` operator). In fig. 14, we displayed the average success rate of each (`cx_pb`, `mut_pb`) couple. For each couple, the average is done on four configurations (because there are two possible values for `eta` in each operator). Unfortunately, the situation is quite different for each problem, so it is difficult to conclude. For example, the setting `mut_pb` $= 0$ seems to be the best in {fig. 14b}, but it seems to be the worst in {fig. 14a}. Maybe there was not enough data to see the corelations, in any case there doesn't seem to exist one best configuration in general.



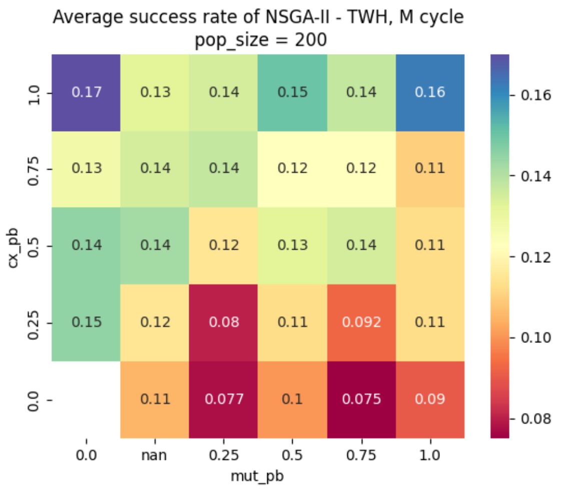
(a) NSGA-II, `pop_size` = 200 IDU3FS-L



(b) NSGA-II, `pop_size` = 200 TWH-L



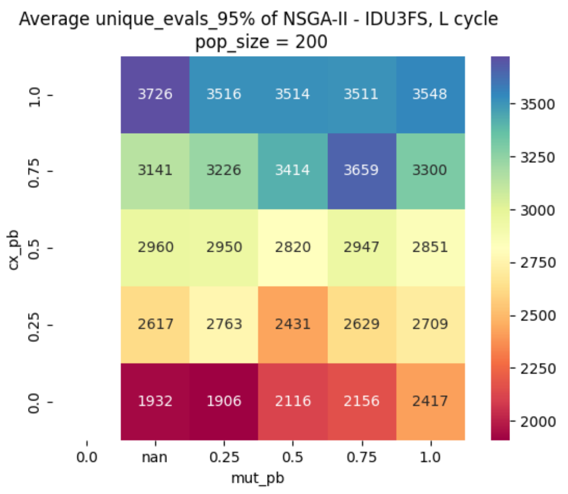
(c) NSGA-II, `pop_size` = 200 IDU3FS-M



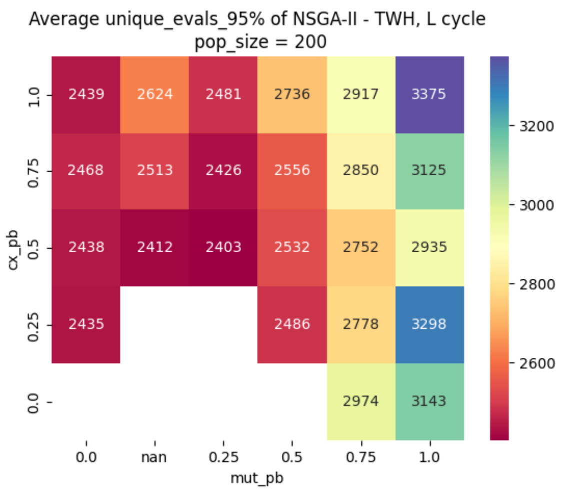
(d) NSGA-II, `pop_size` = 200 TWH-M

Figure 14: Average success rate of NSGA-II (`pop_size` = 100)

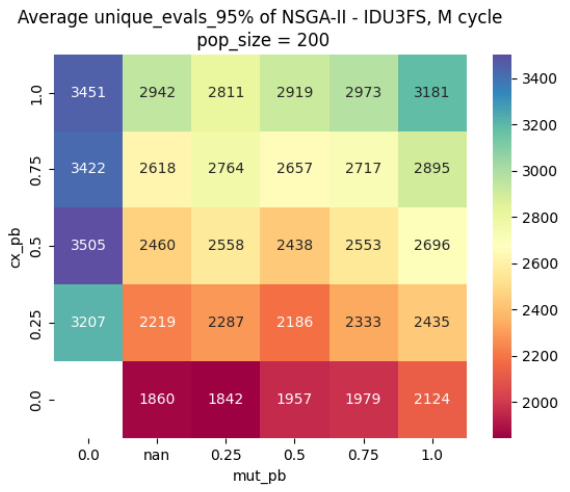
The same comparison can be done to see the influence of these probabilities on the number of unique evaluations. In {fig. 15}, we show the average 95% quantile on `unique_evals` of each configuration, computed using only the successful executions. Here we can see an other problem, success was never atteigned for the TWH with the M cycle. This could mean that our definition of a success was too restrictive in this case. However, we can see overall that smaller values of `cx_pb` and `mut_pb` are associated with smaller numbers of evaluations. This makes sense, as bigger probabilities lead to more diversity in the population, so more unique evaluations.



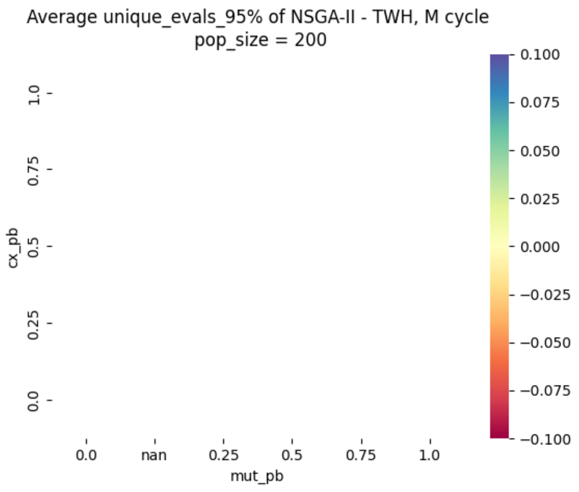
(a) NSGA-II, `pop_size` = 200 IDU3FS-L



(b) NSGA-II, `pop_size` = 200 TWH-L



(c) NSGA-II, `pop_size` = 200 IDU3FS-M



(d) NSGA-II, `pop_size` = 200 TWH-M

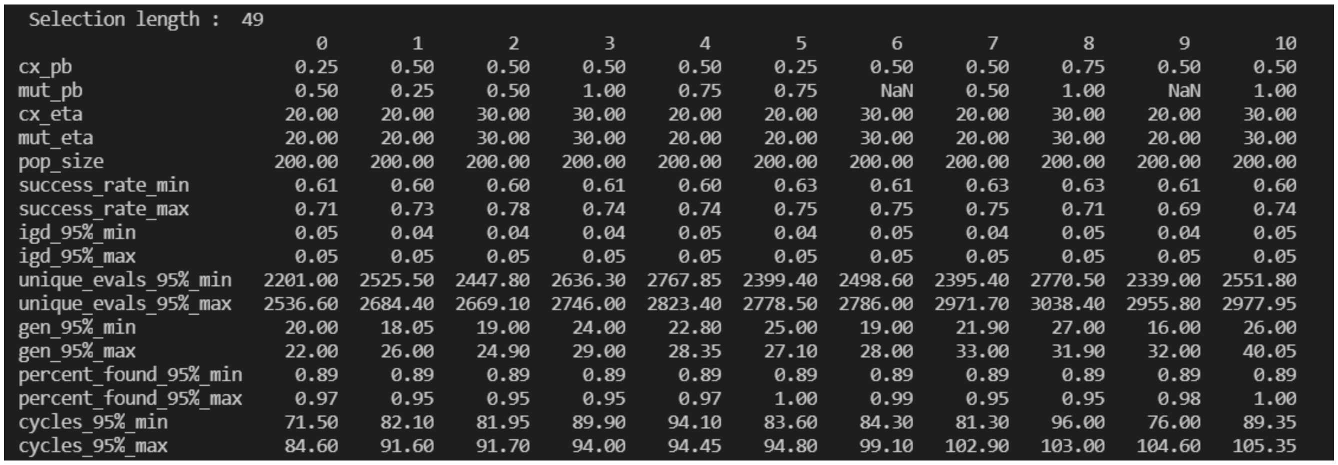
Figure 15: Average 95% quantile of `unique_evals` of NSGA-II when `success_rate` $\geq 60\%$ (`pop_size` = 100)

3.4.5 Selection of a configuration for NSGA-II

We can now select a configuration for our algorithm. Since the example involving TWH with the M cycle was so restrictive, we did not take it into account to do our selection. To do the ranking, we proceeded as such :

- For each configuration and each of the three problems, we store the 95% quantile of the corresponding `cycles` measure (defined in sec. 3.4.1), `cycles_95%`. The quantile is computed using only the seeds for which the algorithm was successful.
- For each configuration, we store the worst (biggest) quantile among all three problems, `cycles_95%_max`
- We sort all the configurations by ascending value of `cycles_95%_max`
- We filter to keep only the configurations with a success rate above 60

In fig. 16 we show the result of this ranking procedure. The first configuration is the one we chose to run with the FMU.



```
 Selection length :  49
                           0        1        2        3        4        5        6        7        8        9       10
cx_pb                   0.25     0.50     0.50     0.50     0.50     0.25     0.50     0.50     0.75     0.50     0.50
mut_pb                  0.50     0.25     0.50     1.00     0.75     0.75      NaN     0.50     1.00      NaN     1.00
cx_eta                 20.00    20.00    30.00    30.00    20.00    20.00    30.00    20.00    30.00    20.00    30.00
mut_eta                20.00    20.00    30.00    30.00    20.00    20.00    30.00    20.00    30.00    20.00    30.00
pop_size              200.00   200.00   200.00   200.00   200.00   200.00   200.00   200.00   200.00   200.00   200.00
success_rate_min        0.61     0.60     0.60     0.61     0.60     0.63     0.61     0.63     0.63     0.61     0.60
success_rate_max        0.71     0.73     0.78     0.74     0.74     0.75     0.75     0.75     0.71     0.69     0.74
igd_95%_min             0.05     0.04     0.04     0.04     0.05     0.04     0.05     0.04     0.05     0.04     0.05
igd_95%_max             0.05     0.05     0.05     0.05     0.05     0.05     0.05     0.05     0.05     0.05     0.05
unique_evals_95%_min 2201.00  2525.50  2447.80  2636.30  2767.85  2399.40  2498.60  2395.40  2770.50  2339.00  2551.80
unique_evals_95%_max 2536.60  2684.40  2669.10  2746.00  2823.40  2778.50  2786.00  2971.70  3038.40  2955.80  2977.95
gen_95%_min            20.00    18.05    19.00    24.00    22.80    25.00    19.00    21.90    27.00    16.00    26.00
gen_95%_max            22.00    26.00    24.90    29.00    28.35    27.10    28.00    33.00    31.90    32.00    40.05
percent_found_95%_min   0.89     0.89     0.89     0.89     0.89     0.89     0.89     0.89     0.89     0.89     0.89
percent_found_95%_max   0.97     0.95     0.95     0.95     0.97     1.00     0.99     0.95     0.95     0.98     1.00
cycles_95%_min         71.50    82.10    81.95    89.90    94.10    83.60    84.30    81.30    96.00    76.00    89.35
cycles_95%_max         84.60    91.60    91.70    94.00    94.45    94.80    99.10   102.90   103.00   104.60   105.35
```

Figure 16: Original sorting configurations of NSGA-II

3.5 Simulation with an FMU (functional mock-up unit)

We chose to test the chosen configuration of NSGA-II to optimize the $\mathrm{COP}_{\mathrm{DHW}}$ and floating Stars of the IDU3FS. The configuration we used is the following :

- Optimization problem :
  - Product : IDU3FS
  - Cycle : L
  - Objectives : $\mathrm{COP}_{\mathrm{DHW}}$, Stars (floating)
- Algorithm : NSGA-II
  - `pop_size` : 200,
  - `crossover`: `SimulatedBinaryCrossover(prob=0.25, eta=20.0)`
  - `mutation` : `PolynomialMutation(eta=20.0, prob=0.50)`
  - termination criterion : we set a time limit of 36 hours

To be able to use the calculation power of our machines, we used the Joblib library in the `_evaluate` function of our problem class to compute the evaluations in parallel and store them in a data frame. This way, each individual had to be evaluated at most one time during the run of the algorithm.

To do the simulation, we used the function `simulate_fmu` provided by the FMPy Python library [@:noauthor_home_nodate]. The FMU files were provided to us by BDR Thermea.

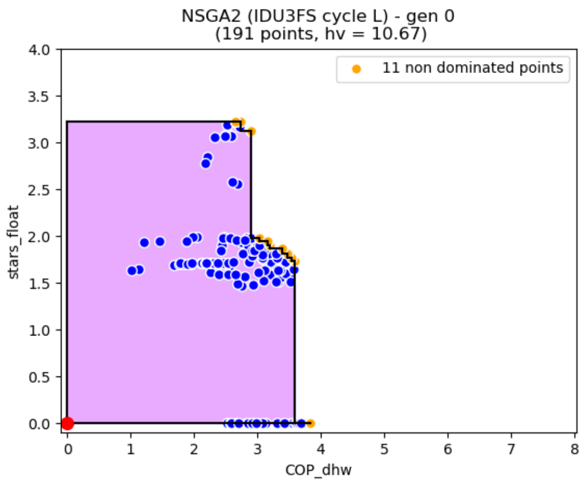
To simplify, we set the step size to 300 seconds and we fixed the stop time to 14 days. [3] Because of this simplified use of the FMU, the objectives we evaluated for each individual were slightly different from the precalcuated ones we had at the beginning, so the exact Pareto front could not be known. Therefore, we could not use an indicator such as the IGD to track the progress of the algorithm. We chose to replace it with an hypervolume indicator ([11]), which doesn't require to known the Pareto front in advance. This indicator is not suited for problems with many objectives as the computational cost quickly explodes as the dimension increases, however it was practical to use it for our case. As shown in fig. 17, the hypervolume is an area between the non-dominated front and a point of reference that must be provided, in this cas $(0, 0)$.

We ran the simulation on 34 processors for a total of 36 hours. in total, 3875 unique individuals were evaluated in 78 generations, which makes an average of 28 minutes by generation and 34 seconds per individual. The evolution of the population with the non-dominated front is shown in fig. 17, and the evolution of the unique evaluations and hypervolume with respect to each generation is shown in fig. 18. We can see that the hypervolume doesn't increase

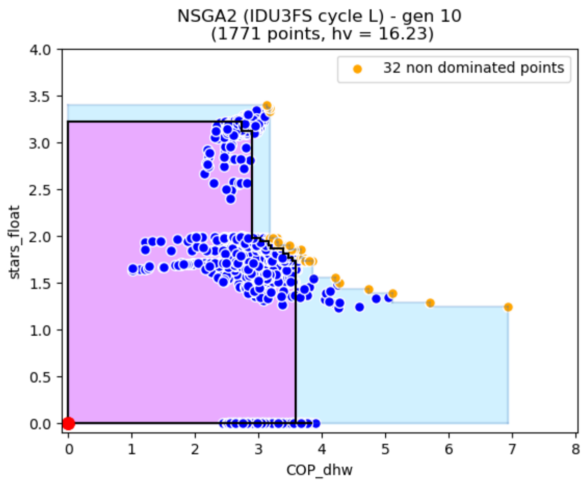
[3] To further optimize the simulation, it is possible to monitor the status of the model variables during the execution and to stop it earlier if no more calculations are required

after generation 28, which corresponds to 21 hours of simulation and 2731 unique evaluations (31% of the design space $\mathcal{X}$).

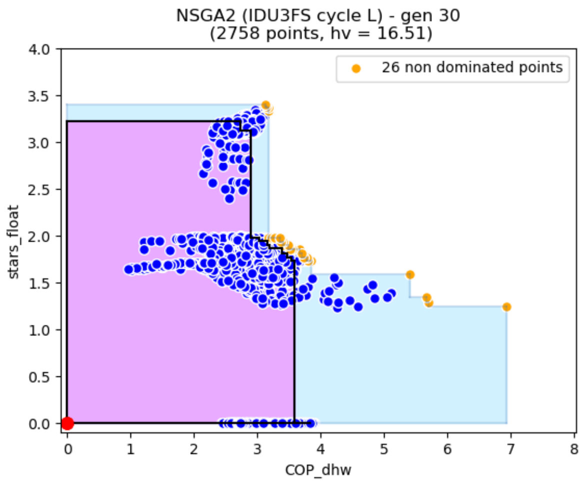
In fig. 18, we also observe that after about 20 generations, there are often fewer new individuals to evaluate than the 34 available processors. Therefore, our algorithm could be improved by injecting new random individuals in the population at each generation if the number of new individuals is smaller than 34.



(a) Initial population



(b) Generation 10



(c) Generation 30

Figure 17: Evolution of the population and Pareto front of NSGA-II

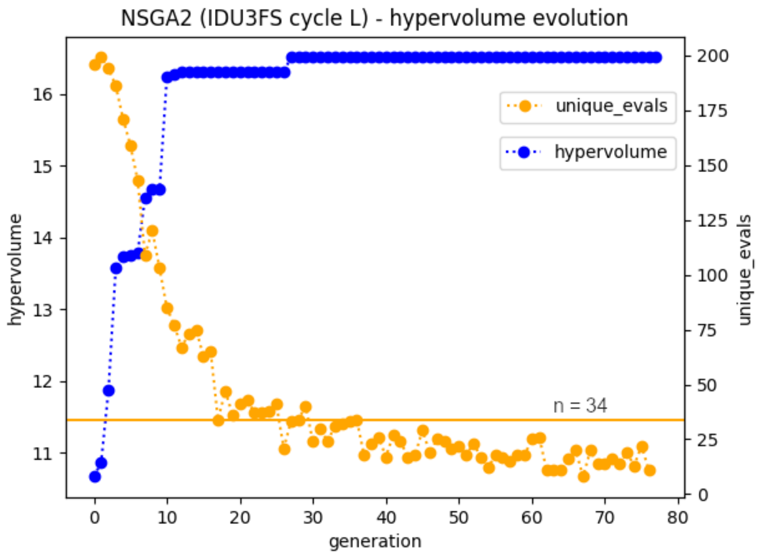


Figure 18: Unique evaluations by generation of NSGA-II and evolution of the hypervolume

4 Conclusion

In conclusion, we have seen that evolutionary algorithms such as NSGA-II can be used to solve complex multi-objective optimization problems that would otherwise be very hard to solve. Such algorithms can efficiently explore a design space and quickly find relevant solutions in the form of a Pareto front. In this study, our design space contained 8816 points, and we were able to find a satisfiying Pareto front by exploring only 30% of the data. Although it is not clear if the algorithms themselves can be further optimized, there is room for practical improvement, mainly by managing parallelization better.

When implementing the algorithms, it was difficult to choose which parameters to vary and to what extent. There were a lot of possible operators, each one with many parameters, and we could not test everything because it would explode the number of simulations. The simulations themslves were very long and therefore had to be well planified; sometimes, problems occurred after several days of calculations, forcing us to start the whole process again.

Doing the statistics to choose a configuration was also not easy, the results were nevever the same depending on the termination criteria, the required success rate, the percent of quantiles... There was a compromise to find between accuracy of the results and speed of the algorithm. Furthermore, we did not find any obvious relationship between parameters (such as the probabilities of crossover and mutation) and the performance of the algorithms, as the situation was very different for each problem. It could be something to work on in future works.

Finally, it is not certain that we should keep Pymoo for the implementation of the algorithms. Pymoo is robust but has a complicated architecture, with a lot of code factoring. Other alternatives such as the DEAP library or a custom implementation from scratch could be considered.

5 Bibliography

1. Kaliko thermodynamical water-heater (TWH), https://www.dedietrich-thermique.fr/nos-produits/chauffe-eau-thermodynamique/kaliko, (2021)
2. Strateo heatpump (IDU3FS), https://www.dedietrich-thermique.fr/nos-produits/pompe-a-chaleur/strateo, (2021)

3. EN16147:2017 norm, https://www.boutique.afnor.org/norme/nf-en-16147/pompes-a-chaleur-avec-compresseur-entraine-par-moteur-electrique-essais-determination-des-performances-et-exigences-pour-le-m/article/905399/fa188149?utm_source=UNM&utm_medium=lien-texte&utm_campaignc=AffiliationUNM, (2021)
4. LCIE 103-15/c specifications, https://www.lcie.fr/medias/cdc_103-15c_chauffe-eau_thermodynamique_autonome_a_accumulation_2018_06_version_fran%C3%A7aise.pdf, (2021)
5. Optimum de Pareto, https://fr.wikipedia.org/w/index.php?title=Optimum_de_Pareto&oldid=184312394, (2021)
6. Functional Mock-up Interface, https://en.wikipedia.org/w/index.php?title=Functional_Mock-up_Interface&oldid=1020746737, (2021)
7. Métamodèle, https://fr.wikipedia.org/w/index.php?title=M%C3%A9tamod%C3%A8le&oldid=178634870, (2021)
8. Métaheuristique, https://fr.wikipedia.org/w/index.php?title=M%C3%A9taheuristique&oldid=180927297, (2021)
9. John H. Holland: Genetic Algorithms. Scientific American. 267, 66–73 (1992)
10. Korstanje, J.: Genetic Algorithms in Python using the DEAP library, https://towardsdatascience.com/genetic-algorithms-in-python-using-the-deap-library-e67f7ce4024c, (2020)
11. Pymoo - Performance Indicator, https://pymoo.org/misc/performance_indicator.html#Hypervolume
12. Welcome to Python.org, https://www.python.org/
13. Zhang, Q., Li, H.: MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. IEEE Transactions on Evolutionary Computation. 11, 712–731 (2007). https://doi.org/10.1109/TEVC.2007.892759
14. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation. 6, 182–197 (2002). https://doi.org/10.1109/4235.996017
15. Jain, H., Deb, K.: An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach. IEEE Transactions on Evolutionary Computation. 18, 602–622 (2014). https://doi.org/10.1109/TEVC.2013.2281534
16. Seada, H., Deb, K.: A Unified Evolutionary Optimization Procedure for Single, Multiple, and Many Objectives. IEEE Transactions on Evolutionary Computation. 20, 358–369 (2016). https://doi.org/10.1109/TEVC.2015.2459718
17. Li, K., Chen, R., Fu, G., Yao, X.: Two-Archive Evolutionary Algorithm for Constrained Multiobjective Optimization. IEEE Transactions on Evolutionary Computation. 23, 303–315 (2019). https://doi.org/10.1109/TEVC.2018.2855411
18. Blank, J., Deb, K.: Pymoo: Multi-Objective Optimization in Python. IEEE Access. 8, 89497–89509 (2020). https://doi.org/10.1109/ACCESS.2020.2990567
19. DEAP documentation — DEAP 1.3.1 documentation, https://deap.readthedocs.io/en/master/
20. Benítez-Hidalgo, A., Nebro, A.J., García-Nieto, J., Oregi, I., Del Ser, J.: jMetalPy: A Python framework for multi-objective optimization with metaheuristics. Swarm and Evolutionary Computation. 51, 100598 (2019). https://doi.org/10.1016/j.swevo.2019.100598
21. Particle swarm optimization, https://en.wikipedia.org/w/index.php?title=Particle_swarm_optimization&oldid=1036646387, (2021)
22. Loi hypergéométrique, https://fr.wikipedia.org/w/index.php?title=Loi_hyperg%C3%A9om%C3%A9trique&oldid=183211444, (2021)

6 Appendix

6.1 Codes

6.1.1 Definition of the problem with Pymoo's `Problem` class

```
class COP_Stars_csv(Problem):

    attributes = [
        'x_tank_default_control_sensor_height',
        'x_tank_default_control_setpoint_temperature',
        'x_tank_default_control_temperature_hysteresis']
    objectives = ['y_COP_dhw', 'stars_float']

    def __init__(self, datafile, path):
        self.data, self.boundaries = format_data(
            datafile,
            path=path,
            attributes=self.attributes,
            compute_stars_float=True)
        self.known_inds = pd.DataFrame(
            columns=self.objectives,
            index = pd.MultiIndex.from_tuples(
                [np.zeros(len(self.attributes))],
                names=self.attributes) )
        self.datafile = datafile
        self.evaluated = {}
        super().__init__(n_var=len(self.attributes),
                n_obj=len(self.objectives),
                n_constr=1,
                xl = np.zeros(len(self.attributes)),
                xu = np.array(self.boundaries['length'].values),
                elementwise_evaluation=True,
                type_var=int)
        self.pf = -np.array(Pareto_set(
            self.data, self.objectives)[self.objectives].values)
        self.ps = np.array([list(ind) for ind in Pareto_set(
            self.data, self.objectives).index.values])

    def _evaluate(self, x, out, *args, **kwargs):
        ind = attributes_from_coord(x, self.boundaries)
        if ind not in self.known_inds.index :
            if ind in self.data.index :
                eval = tuple(self.data.loc[ind, self.objectives])
            else:
                eval = (0.0,)*len(self.objectives)
            self.known_inds.loc[ind, :] = eval
        eval = tuple(self.known_inds.loc[ind, self.objectives])
        obj_COP_dhw, obj_stars = eval[0], eval[1]
        constr_stars = 1 - obj_stars # 1 - stars <= 0
        out["F"] = [-obj_COP_dhw, -obj_stars]
        out["G"] = [constr_stars]
```

```python
def get_hash(self, x):
    ind = attributes_from_coord(x, self.boundaries)
    return self.data.loc[ind, 'Hash'] if ind in self.data.index else 0

def unique_evals(self) :
    return len(self.known_inds)-1

def _calc_Pareto_front(self):
    return self.pf

def _calc_Pareto_set(self):
    return self.ps

def plot(self):
    plt.scatter(
        self.data[self.objectives].values[:, 0],
        self.data[self.objectives].values[:, 1],
        color="blue", marker=".")
    plt.scatter(-self.pf[:,0], -self.pf[:,1], color="orange", marker="x")
    plt.xlim(-0.1, 6.93+0.1)
    plt.ylim(-0.1, 3.41+0.1)
    plt.show()
```