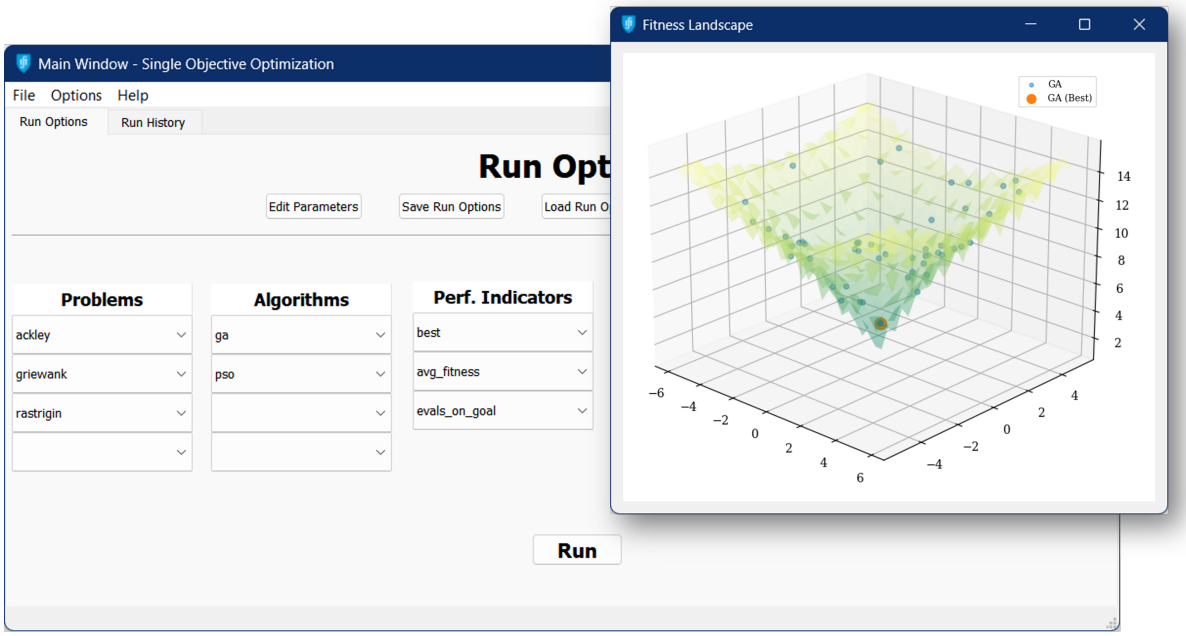
TÉCNICO LISBOA



Single and Multi-Objective Optimization using Computational Intelligence Techniques

A Desktop Application

Tomás Pereira de Brito Líbano Monteiro

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Nuno Horta
Prof. Rui Neves

Examination Committee

Chairperson: Prof. Pedro Tomás
Supervisor: Prof. Nuno Horta
Member of the Committee: Prof. Ricardo Martins

May 2024

Dedico esta tese à Isabel, minha afilhada,
para que saiba que o querer e o apetecer são coisas diferentes.

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to say a word of appreciation to my dissertation supervisor, Prof. Nuno Horta. I thank him for his understanding and even support of my other commitments in life, so I did not have to abandon them during the duration of the dissertation. Also, for ensuring my laziness would not get a hold on me, scheduling regular meetings. It is hard to strike a balance between these two sides, and I think he did it gracefully.

I also would like to express my gratitude to all my friends who accompanied me on this journey in Técnico, especially my friends of Xagueira. Without them, I would still be stuck trying to pass the Chemistry-I course of the first semester. For David and Isabel a huge thanks, who did not let me quit and carried me when I was more focused on other commitments.

Thank you to all the kids of Candeia, who showed me that I am not the center of the universe and helped me set my priorities in life in the right order, starting with the family.

So, thanks to my family for being what a family should be - not perfect but unconditionally loving. To my parents, brothers, sisters, and to the new addition to the family, my nephew Isabel. Thank you for reminding me that the joy of a large family is worth having to share a bedroom and a noisy home (most of the time, at least).

Lastly, I would like to thank God, as if it weren't for Him, I would not be here.

This work was hosted at Instituto de Telecomunicações, funded by Fundação para a Ciência e Tecnologia–Ministério da Ciência, Tecnologia e Ensino Superior (FCT/MCTES) through national funds and, when applicable co-funded European Union (EU) funds under the project UIDB/50008/2020.

Resumo

Com o aumento do poder computacional, as técnicas de computação inteligente têm sido progressivamente adoptadas para resolver problemas de otimização com um ou vários objectivos. Várias frameworks gerais de otimização foram desenvolvidas para tentar estabelecer um conjunto abrangente de ferramentas que facilitem a investigação nesta área. São oferecidas algumas interfaces gráficas de utilizador, mas muitas vezes carecem de características importantes para serem efetivamente úteis, como a extensibilidade e a personalização, ou não são gratuitas. Além disso, com Python a tornar-se uma das linguagens de programação mais utilizada no mundo, seria interessante ter uma interface gráfica baseada em Python, onde o código do utilizador pudesse ser facilmente integrado, promovendo um desenvolvimento mais rápido na área de investigação da otimização.

Este trabalho pretende atender a esta necessidade, desenvolvendo uma aplicação de desktop baseada em Python a partir de uma framework de otimização existente, pymoo. Reúne mais de 100 problemas e 14 algoritmos para otimização mono e multi-objetivo, bem como 11 indicadores de performance e 9 critérios de terminação. Mais de 40 operadores de vários tipos são fornecidos, permitindo uma grande personalização dos algoritmos. Os processos de otimização podem ser iniciados através do clique de um botão, visualizados através de técnicas adequadas e os resultados guardados. Com um código open-source e o seu equilíbrio entre facilidade de utilização e personalização, a aplicação revela-se um recurso valioso tanto para utilizadores principiantes como experientes no campo da otimização.

Palavras-chave: Computação Inteligente, Otimização Mono-Objetivo, Otimização Multi-Objetivo, Pymoo, Aplicação de Desktop, Interface Gráfica de Utilizador, Python.

Abstract

With the increase in computational power, computational intelligence techniques have been progressively adopted to solve single and multi-objective optimization problems. A series of general optimization frameworks were developed to try to establish a comprehensive set of tools to facilitate research in this area. Some graphical user interfaces are offered, but they often lack important characteristics to be effectively useful, such as extensibility and customization, or are behind a paywall. Furthermore, as Python becomes one of the most used programming languages in the world, it would be of interest to have a Python-based graphical interface, where user code could be easily integrated, fostering faster development in the optimization research area.

This work aims to tackle this issue by developing a Python-based desktop application on top of an existing state-of-the-art optimization framework, pymoo. It brings together more than 100 problems and 14 algorithms for single and multi-objective optimization, as well as 11 performance indicators and 9 termination criteria. More than 40 operators of various types are provided, allowing for great customization of the algorithms. Optimization processes can be set in motion with a click of a button, visualized through appropriate techniques, and the results saved. The open-source nature of the project and the application's balance between ease of use and customization make it a valuable resource for both novice and experienced users in the field of optimization.

Keywords: Computational Intelligence, Single-Objective Optimization, Multi-Objective Optimization, Pymoo, Desktop Application, Graphical User Interface, Python.

Contents

List of Tables

List of Figures

List of Code Snippets

Acronyms

ACO Ant Colony Optimization.

CV Constraint Violation.

DE Differential Evolution.

EA Evolutionary Algorithm.

EC Evolutionary Computation.

FE Function Evaluations.

GA Genetic Algorithm.

GD Generational Distance.

GPL General Public License.

GUI Graphical User Interface.

HV Hypervolume.

IGD Inverted Generational Distance.

MaOOP Many-Objective Optimization Problem.

MOEA/D Multi-Objective Evolutionary Algorithm Based on Decomposition.

MOEA Multi-Objective Evolutionary Algorithm.

MOOP Multi-Objective Optimization Problem.

MOO Multi-Objective Optimization.

NSGA Non-dominated Sorting Genetic Algorithm.

OOP Object-Oriented Programming.

PF Pareto Front.

PI Performance Indicator.

PSO Particle Swarm Optimization.

SOOP Single-Objective Optimization Problem.

SOO Single-Objective Optimization.

TSP Traveling Salesman Problem.

Chapter 1

Introduction

1.1 Motivation

Optimization is now an intrinsic part of our lives. From the search for the shortest route between home and the supermarket using GPS, to the goal of building the most satisfying and reliable smartphone from the user's perspective at the smallest cost, or the most effective way to schedule flights for airlines, optimization shapes the very fabric of our daily routines.

What were for many centuries problems solved through human reason and intuition are now replaced or aided by algorithms using intelligence computation techniques. The initial focus was on developing algorithms that tackle single-objective optimization problems (SOOPs).

However, numerous real-world optimization problems involve several conflicting objectives, making them inherently multi-objective (MOOPs). In this way, instead of a single best solution, a set of solutions that correspond to the best trade-offs between objectives emerges, called the optimal Pareto Front (PF).

With the rise of computational power, researchers have begun to propose algorithms that harness the advantages of evolutionary computation in MOOPs. As multiple solutions are to be found, not just a single one, Evolutionary Algorithms (EA) are frequently used [1, 2] to achieve or approximate the optimal PF, due to their population-based methods. Today, they are a common practice to solve well-known MOOPs in areas such as circuit design [3], scheduling [4], computational finance [5], and many others.

In the realm of academic research, optimization through evolutionary computation stands as one of the most actively investigated domains, with a focus on improving efficiency, performance, and resource utilization. As evidenced by numerous scholarly works [6, 7], the pursuit of developing and implementing optimization techniques through evolutionary computation has garnered substantial attention and continues to be a pivotal area of study.

Over the years, several frameworks have been developed to assist in the implementation of various algorithms, as well as in the definition of a series of test problems, performance measures, and visualization techniques. This is the case for frameworks such as MOEA [8], written in Java [9], PlatEMO [10], in MATLAB [11], pymoo [12] and DEAP [13], written in Python [14].

Some of the frameworks already include a Graphical User Interface (GUI), such as PlatEMO and MOEA. However, as Python becomes one of the most widely used programming languages among users around the world [15], there is a lack of a GUI that could not only facilitate the process of studying and discovering new algorithms in this language, but also be an entry point for several users with lower programming skills or know-how in optimization. Furthermore, existing GUIs are often limited in what they allow the user to do, as they are not being developed in a way to harness the fullness of the framework in the backbone of the application. This highlights the need for the work proposed in this thesis.

1.2 Topic Overview

The use of general population-based optimization methods, inspired by biological evolution, to solve problems originated before the creation of computers. This concept of mimicking evolutionary processes was already shown by Alan Turing's proposed method of genetic search in 1948 [16].

The use of computational power to implement these techniques began in the 1950s and 1960s [16]. John Henry Holland introduced genetic algorithms (GA) in the 1960s, and it was further developed at the University of Michigan in the 1970s [17]. However, only the big jump in computational power allowed for greater development in this area, with genetic programming emerging in the 1990s, advocated by John Koza, among others [18].

Some robust algorithms that are still in use were proposed in the late 1990s, such as Particle Swarm Optimization (PSO) by Kennedy and Eberhart [19], and the Non-dominated Sorting Genetic Algorithm (NSGA) [20], with the latter getting a great performance improvement with the upgraded NSGA-II [1] in 2000, one of the most established algorithms to date.

With further advancements in computational power, researchers could explore areas now made accessible by the greater resources available. Suited to tackle many-objective optimization problems (with 4 or more objectives), Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D) [2] was released in 2007, and later followed by NSGA-III [21], released in 2016.

With the development of multiple algorithms with different methodologies, the need for a systematic approach to use and evaluate them became more prominent. In this regard, several frameworks began to be developed, among them jMetal [22] and MOEA [8] , Java-based [9] frameworks for multi-objective optimization, both introduced in 2011. In 2019, jMetalPy [23] was released as the Python version of the original jMetal framework.

DEAP [13], a Python-based framework [14] for evolutionary algorithms, was first introduced in 2012. In 2017, PlatEMO [10], was released, based on the MATLAB [11] platform. MATLAB software is used to provide a powerful GUI. In 2020 pymoo, a Python-based framework for multi-objective optimization, was first introduced. It provides a variety of multi-objective optimization algorithms, including NSGA-II and NSGA-III, and is designed to be flexible and extensible.

The timeline of Figure 1.1 presents important events in optimization, in chronological order. The steps in which the development occurred are clearly seen. Evolutionary Algorithms are first slowly de-

veloped until the mainstream availability of computers. After that, a series of single-objective optimization algorithms are proposed. With an increase in computational power, more attention is paid to MOOPs, and some algorithms tailored to them emerge, harnessing the power of evolutionary computation to this effect. With multiple options available, the first optimization frameworks are developed, in conjunction with stronger algorithms for many-objective optimization problems (MaOOPs).

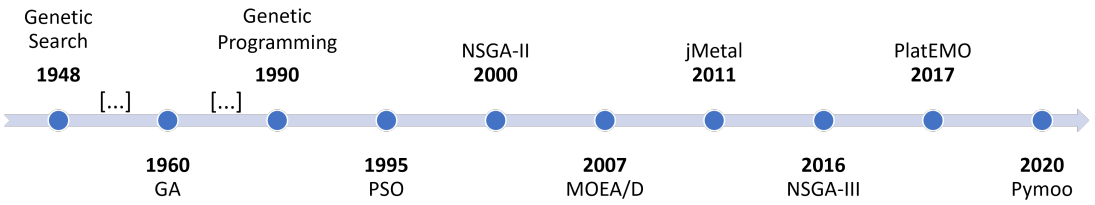


Figure 1.1: Timeline with developments in optimization.

1.3 Objectives and Deliverables

This work aims to create a desktop application for optimization algorithms. It should not be created from scratch as that is beyond the scope of time and resources of the thesis. Rather, it should have as a backbone an existing optimization framework to be seamlessly integrated. Then it should build a graphical interface on top of it, so it fulfills the needs of what is not yet available on the market. The application should meet the following requirements.

1. **Cost-free**. The desktop application should be open source, so further development and research can be performed without any hindrances to it. Its use should be open to the public with the most permissible license possible.

2. **Python-based**. As Python becomes one of the world's most used programming languages, there is a growing need for resources in this language. This means that the optimization framework chosen to be the backbone of the application also needs to be Python-based.

3. **Test multiple state-of-the-art algorithms on various benchmarking problems**. Also concerns the framework chosen to be the backbone of the application. It should not only have current state-of-the-art algorithms for single and multi-objective optimization but also have a perspective of continuous development. Furthermore, it should allow for thorough testing on multiple established problems.

4. **Visualization techniques**. The app should also allow the user to see the results through adequate visualization techniques. Not only showing the best, median, and worst values in a table format but also using various plotting modes.

5. **Easily extensible**. The application should be programmed in a modular way, where future updates and developments can easily be done. Whether it is to add a new algorithm, or a new functionality

to the app, the code should be well documented and modeled so it is easy to understand where it should be altered to achieve the respective goal.

6. **Customizable and easy to use**. These are often conflicting objectives, as when the range of customization options increases, so does the difficulty in learning how to use them. However, the app should strike a good balance between the two. On the one hand, users with limited knowledge in optimization and programming should be able to quickly explore and visualize different algorithms. On the other hand, it should also support great customization, such as easily introducing user-coded algorithms into it and allowing for various modifications to be made at the small level, such as personalized algorithm operators or termination criterion for the run.

1.4 Thesis Outline

Following the introduction, the rest of the document is defined as follows. The background and related work in the field are explored in Chapter 2, which presents the fundamental concepts for SOOPs and MOOPs, the available optimization frameworks, and GUI library options for the application.

In Chapter 3, the application's architecture is detailed, from backend to frontend, and some key aspects of its functionality. A demonstration of the usefulness of the app is done in Chapter 4. It outlines the reliability of results, and shows practical examples of how the application can assist in single and multi-objective optimization. Chapter 5 states the achievements of the work, as well as the possible ways in which it could be improved in the future.

Chapter 2

Background and Related Work

In this chapter, an insight into the fundamentals of single and multi-objective optimization is given in Section 2.1, as well as its early approaches, and the usefulness of Evolutionary Algorithms. In sections 2.2 to 2.4. In Sections 2.2 to 2.4 some of the most established EAs used in the field are detailed, as well as the necessity for different performance metrics and a variety of test problems that mimic the specific challenges posed by different real-life problems. The core concepts of Python are explored in Section 2.5. Section 2.6 explores existing frameworks in the optimization world, highlighting the advantages and disadvantages of each. In Section 2.7, possible GUI frameworks to use in the thesis application are detailed.

2.1 Fundamental Concepts

This section presents the fundamental concepts of single and multi-optimization and their early approaches. Moreover, it asserts in what cases and why evolutionary algorithms become useful, as well as detailing its generic structure.

2.1.1 Single-Objective Optimization

Generally, a single-objective optimization problem can be defined as follows:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_j(x) \leq 0 \qquad j = 1, \ldots, J \\ & h_k(x) = 0 \qquad k = 1, \ldots, K \\ & x_i^L \leq x_i \leq x_i^U \quad i = 1, \ldots, N \\ & x \in \Omega \end{aligned} \tag{2.1}$$

where $f(x)$ represents the objective function, x_i represents the i-th variable to be optimized, x_i^L and x_i^U its lower and upper bound, g_j the j-th inequality constraint and h_k the k-th equality constraint.

The objective function f is supposed to be minimized by satisfying all equality and inequality constraints. It should be noted that if the objective function is to be maximized $(\max f(x))$, one can redefine

the problem to minimize its negative value $(\min -f(x))$.

The search space represents all possible values that the variable x can take. The objective space represents the same for $f(x)$. Although the latter is always one-dimensional in SOOPs, it is not always the case, as will be detailed in Section 4.3.

To solve the SOOP, its different aspects have to be considered to know which is the best algorithm for the particular case. A large number of variables can cause the complexity of calculating the derivatives to grow exponentially, causing significant difficulties to gradient-based methods.

Variable types should also be taken into account as well, as there are algorithms tailored for discrete or continuous variables. If there is uncertainty in the objective function, specialized techniques could be required to evaluate performance, such as repeating the evaluation for different random initializations and averaging the resulting values.

Finally, the problem constraints have to be taken into consideration, as from the end-user perspective, the best solution is irrelevant if it is unfeasible. As dealing with constraints is a subject of its own, the following section describes common approaches to constrained problems.

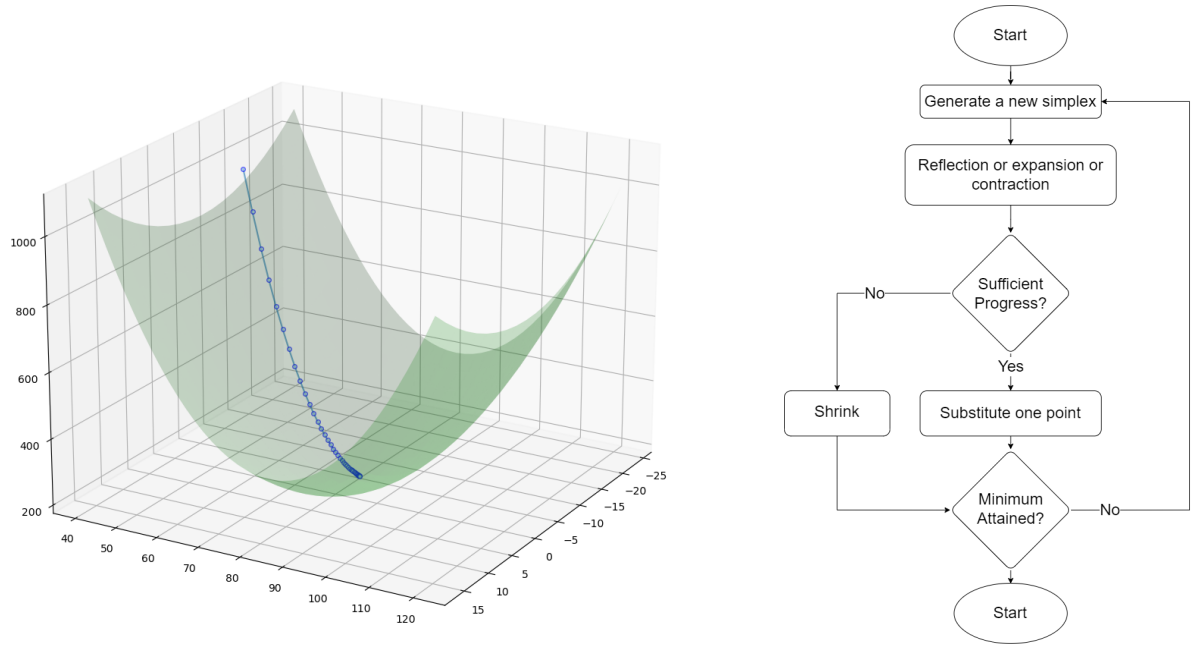
Constraints

During the optimization process, various techniques can be deployed to ensure the solutions are feasible or at least have the least constraint violation value.

- **Feasibility First**. Make the algorithm prioritize feasible solutions by sorting individuals, always favoring those that meet constraints over those that do not, regardless of the degree of infeasibility. This straightforward method is greedy but easy to implement.
- **Penalty Method**. The optimization problem is modified by integrating a penalty into the objective values, transforming it into a problem that can be solved by most algorithms without constraints.
- **Constraint Violation (CV) as an Objective**. Constraints are alternatively handled by considering violations as separate objectives, thus transforming the issue into a multi-objective problem requiring resolution of an additional goal.
- ϵ**-Constraint Handling**. Implements a dynamic threshold to assess the feasibility of solutions, a feature that must be incorporated by the algorithm itself.
- **Repair Operator**. Modifies the generated solution to meet all or most constraints.

Early Approaches

The concept of optimization has a long history dating back to ancient civilizations, where people used heuristics and trial-and-error methods to solve practical problems. The first systematic approaches to solving optimization problems can be traced back to the 18th and 19th centuries with the development of calculus and the idea of finding the maximum or minimum of a function. Some early contributions include the Gradient Descent and the Nelder-Mead algorithm, seen in Figure 2.1.



(a) (b)

Figure 2.1: Illustration of the Gradient Descent process in a two-dimensional search space to minimize the objective function value corresponding to the vertical axis (a) and Nelder-Mead Flowchart (b).

- **Gradient Descent**. An iterative optimization algorithm that finds the minimum of a function by following the negative gradient of the function. It was introduced by Cauchy in 1847 [24] and has been widely used in machine learning and optimization. Process illustrated in Figure 2.1(a) Some drawbacks of this technique are that it requires the computation of gradients, which can be computationally expensive, and it can get stuck in local minima.

- **Nelder-Mead**. A gradient-free optimization algorithm that uses a simplex (a set of n+1 points in n-dimensional search space) search method to find the minimum of a function. It was introduced by Nelder and Mead in 1965 [25] and is suitable for optimization problems with discontinuous or noisy objective functions. Its flow chart is presented in Figure 2.1(b)). However, this method can be slow to converge and requires a large number of function evaluations. It is sensitive to the initial simplex, as it has to be representative of the search space and not too large.

2.1.2 Multi-Objective Optimization

In contrast to SOO, multi-objective optimization involves finding the optimal solution for multiple, conflicting objective functions, altering equation (2.1) to $f(x) \longrightarrow f_m(x),\ m = 1, .., M$, with $M > 1$.

$$\begin{aligned} \min \quad & f_m(x) && m = 1, \ldots, M \\ \text{s.t.} \quad & g_j(x) \leq 0 && j = 1, \ldots, J \\ & h_k(x) = 0 && k = 1, \ldots, K \\ & x_i^L \leq x_i \leq x_i^U && i = 1, \ldots, N \\ & x \in \Omega. \end{aligned} \tag{2.2}$$

These objective functions may represent different trade-offs or conflicting goals, and finding the optimal solution requires balancing these conflicting objectives. An individual dominates the other if it is better in every objective function. In Figure 2.2, the objective is to maximize the functions $f1$ and $f2$. The individual B dominates D and C, and is dominated by A.

MOOPs can be more complex than SOOPs because there may not be a single optimal solution that is the best for all objectives. Instead, there may be a set of solutions that are Pareto-optimal, meaning that they are not dominated by any other individual. These solutions are called the Pareto-optimal front (Figure 2.2(b)). These fronts can be locally optimal (not dominated by neighbors in the region) or globally optimal (Figure 2.2(c)), and have different shapes that present various challenges to the search.

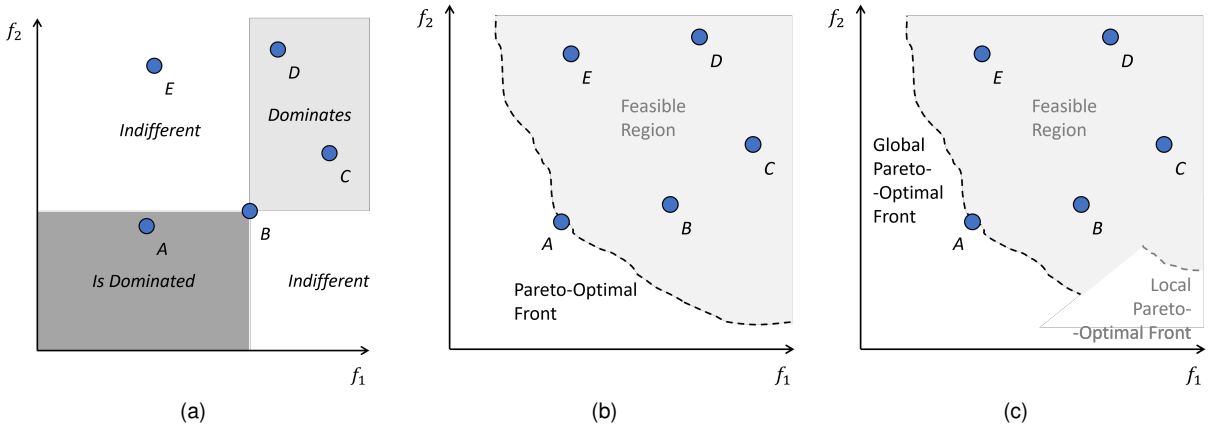


Figure 2.2: Illustrative examples in objective space minimizing $f1$ and $f2$. Possible relations of solutions (a), Pareto-optimality (b), and local and global Pareto-optimal fronts (c).

Many-Objective Optimization Problems

Another important aspect to mention is the difficulties that arise when the number of objectives increases. Many-objective optimization problems are defined as a subset of MOOPs with more than three objectives. This section explores common challenges in this area.

- **A large portion of the population is non-dominated**. It is recognized that as the number of objectives grows, a greater proportion of a randomly generated population remains non-dominated [26, 27]. Given that most Multi-Objective Evolutionary Algorithms (MOEAs) prioritize non-dominated solutions, when dealing with MaOOP, the opportunity to generate new solutions within a generation is limited.

- **Evaluating diversity and performance is computationally expensive**. Identifying how close solutions are to each other in a population or how well they perform in terms of convergence and diversity requires significant computational resources in high-dimensional spaces.

- **Difficulty to recombine solutions**. In scenarios where only a handful of solutions exist within a vast dimensional space, these solutions are typically far apart. This distance creates difficulty in the effectiveness of the recombination operator, a crucial search mechanism in MOEAs. Offspring

derived from such distantly related parents are also likely to be far away from their parents, suggesting the need for specialized recombination strategies, such as mating restrictions, to manage many-objective problems effectively.

- **Representation and visualization of trade-off surface is challenging**. Representing a higher-dimensional trade-off surface requires more points in an exponential way. Thus, a large population size is needed to represent the resulting Pareto-optimal front. This causes a decision-maker to have difficulty comprehending and making an adequate decision to choose a preferred solution. Also, visualization cannot be made directly as in one- to three-dimensional objective spaces, so workarounds have to be used.

Early Approaches

As the study around SOOPs was more developed, some early approaches to solving MOOPs involved transforming them back into a SOOP, using decomposition methods such as weighting and constraint methods [28], and solving it with already known SOOP algorithms.

- **Weighting method**. Assigns a weight or importance to each of the objective functions, and then solves the resulting single-objective optimization problem. The weights reflect the relative importance of the different objectives and can be used to trade off between the objectives in order to find the best overall solution. The main drawbacks of this technique are that it cannot generate all Pareto-optimal solutions with non-convex trade-off surfaces. This is illustrated in Figure 2.3(a), where for any given weights $w1$ and $w2$, solutions C and B are unreachable. Moreover, it requires *a priori* knowledge of the problem to select reasonable weights for each of the objectives.

- **Constraint method**. Converts the MOOP into a SOOP by transforming $n-1$ of the n objectives into constraints and solves the remaining objective subject to the constraints. In Figure 2.3(b), the objective is to minimize $f1$, subject to the constraint of $r < f2 < r'$. The lower bound r is then updated and the process is repeated to obtain different Pareto solutions. This method has the disadvantage of being computationally expensive and requires a suitable set of values for r to be known *a priori*.

2.1.3 Evolutionary Algorithms

Traditional search procedures explore an optimum using information from the function being searched. Usually, this information comes in the form of the function's gradient or higher-order statistics, and it is used to move quickly in a direction that is presumed closer to the global optimizer.

While in cases where the function is convex, these techniques are very useful, non-convexity brings difficulties to the efficiency of these methods. In some cases, they can be unable to find suitable solutions, due to a saddle point that stalled progress, becoming trapped in local optimums, or in the face of a black-box problem (the analytical description of the function is not known or is too expensive to compute,

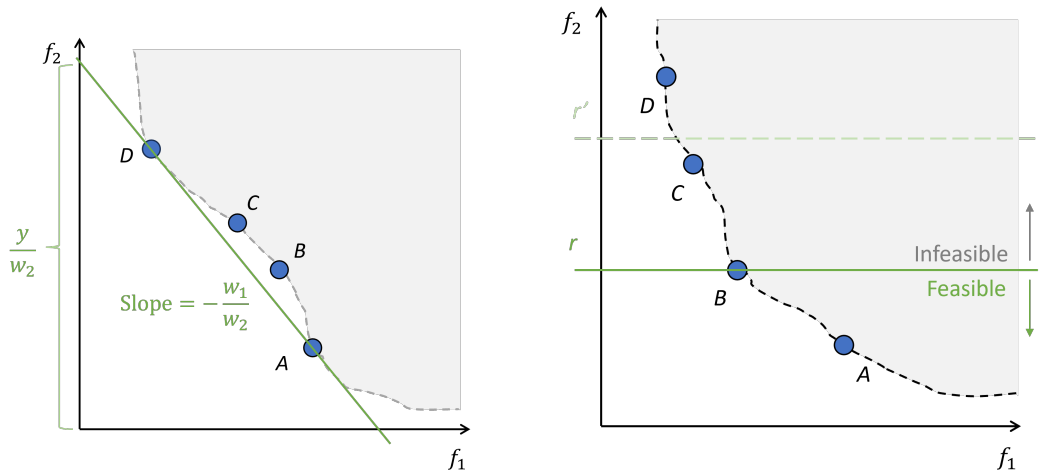


Figure 2.3: Graphical interpretation of the weighting (a) and constraint (b) methods to minimize $f1$ and $f2$.

but the value of $f(x)$ can be obtained). Moreover, when in the face of a MOOP, it is necessary to find the set of solutions that compose the Pareto set, adding a greater challenge to point-to-point optimization processes.

That is when Evolutionary Algorithms become most useful. They differ from traditional gradient searches in two fundamental ways: instead of a single point-to-point search, the algorithm uses a population of candidate solutions. This population evolves through a specific process, introducing randomness during the search. This makes them robust and able to handle large search spaces. Multiple alternative trade-offs can be generated in a single optimization run, with the synergies of the population being exploited during the search.

Evolutionary Algorithms follow the subsequent steps, illustrated in Figure 2.4.

1. **Set initial Population**. A collection of potential solutions to a given problem. The goal is to evolve the population over time to find the best possible solution or set of solutions.

2. **Create new individuals**. This process uses various tactics depending on the algorithm, such as mutation, crossover, and other methods involving randomness, to hopefully evolve into a better-performing generation.

3. **Choose the next generation**. This refers to the process of choosing which individuals between the old and newly generated population will enter the next generation, correlated to their **fitness**, determined by how well it performs on a given task or problem.

Steps 2 and 3 are repeated until a stop criterion is met. The usual stop criterion includes: stopping the search at a maximum number of iterations or time, convergence of a population to a stable state, target fitness achieved, or no improvement over previous generations.

It is important to carefully consider the stop criteria when using evolutionary computation algorithms, as they can have a significant impact on the quality and efficiency of the search process. In some

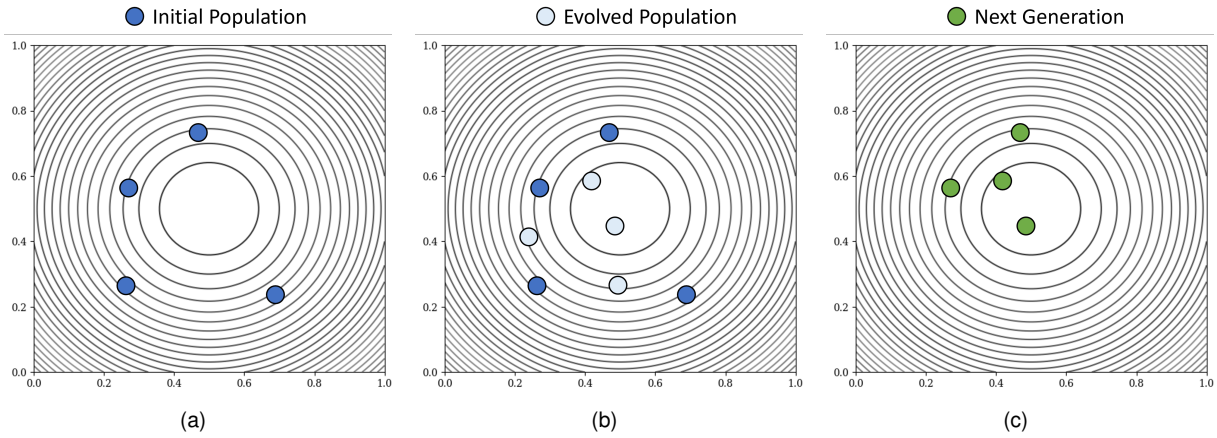


Figure 2.4: Generic steps of an EA in a two-dimensional search space with the level curves of the objective function and optimum at (0.5,0.5). Set initial population (a). Evolve Population (b). Choose the next generation (c).

cases, it may be necessary to use a combination of stop criteria to achieve the desired balance between accuracy and computational cost.

2.2 Established Evolutionary Algorithms

This section covers an overview of different EAs and how they implement the steps described in Section 2.1.3. This is by no means an extensive list, but rather a collection of some of the more established and proven concepts used in Evolutionary Computation (EC). The detailed algorithms include the Genetic Algorithm (GA) and its modification to MOOPs and MaOOPs - NSGA-II and NSGA-III, Differential Evolution (DE), Particle Swarm Optimization (PSO), and Ant Colony Optimization (ACO).

2.2.1 Genetic Algorithm

The generic genetic algorithm follows the subsequent steps.

1. **Initialization**: The initial population of chromosomes is generated randomly or using a specific sampling strategy. The chromosomes in the population represent potential solutions to the problem being optimized.

2. **Evaluation**: The objective function is used to evaluate the quality of each chromosome in the population. The objective function is a measure of how well a chromosome solves the problem. The chromosomes are ranked based on their score from the objective function.

3. **Survival**: It is often the core of the genetic algorithm used. For a simple single-objective genetic algorithm, the individuals can be sorted by their fitness, and survival of the fittest can be applied.

4. **Selection**: Parents are chosen from the population to participate in the reproduction process. Different selection strategies can be used to increase the convergence of the algorithm. For example,

some selection strategies may favor chromosomes that are more fit, while others may favor chromosomes that are less fit but have more diversity, preventing premature convergence to a local optimum.

5. **Crossover**: The genetic information from the parents is combined to create one or more offspring (a new solution). Crossover involves selecting a point in the chromosomes and swapping the genetic information on either side of the point to create offspring.

6. **Mutation**: Random changes are made to the offspring's genetic information to introduce new diversity into the population. Mutation helps to prevent the population from becoming stagnant and helps to explore new regions of the search space.

These steps are repeated for a fixed number of generations or a predetermined stop criterion is met. GAs are often used to find approximate solutions to optimization problems, but do not guarantee finding the global optimal solution.

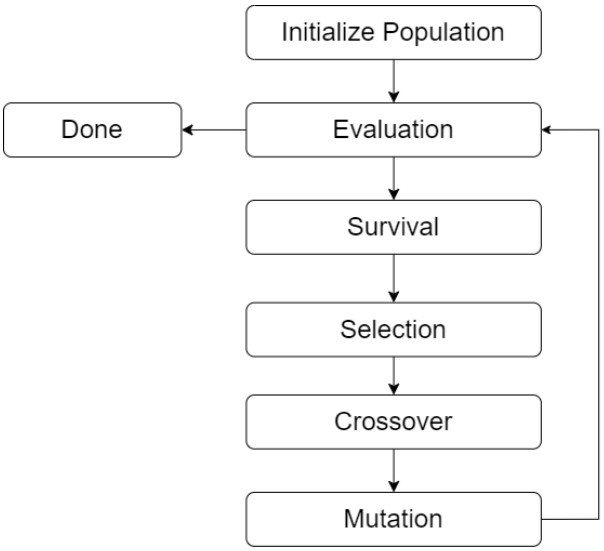


Figure 2.5: Flow chart of a generic GA.

2.2.2 NSGA-II

A renowned adaptation of the Genetic Algorithm worth mentioning is the Non-Dominated Sorting Genetic Algorithm-II. Tailored for Multi-Objective Optimization, the algorithm follows the same procedure as in Figure 2.5, with modified mating and survival selection. In NSGA-II, first, individuals are selected frontwise. These fronts are identified through the following process.

1. For each solution x in the population P, the number of solutions that dominate it, n_x, and the set of solutions it dominates, S_x, are calculated.

2. Select every solution with $n_x = 0$ to the first rank and remove them from P. Then go to the respective set of dominated solutions, S_x, and subtract 1 from the domination count of each of them.

3. Repeat step 2 until group P is emptied, so every solution will have an associated rank.

All individuals from the first front (of rank 1) are selected, then all from the second front (of rank 2), and so on, until a front is reached that needs to be split because not all individuals are allowed to survive. The solutions are then selected on the basis of the crowding distance (Manhattan distance in the objective space).

To calculate this crowding distance, the solutions in the population are first sorted in ascending order for each objective function. The boundary solutions (those with the smallest and largest function values) are assigned an infinite crowding distance value to ensure that they are selected.

For all other intermediate solutions, the crowding distance is calculated as the average distance between the two adjacent solutions along each objective. This is done by normalizing the values of the objective function and then computing the absolute difference between the values of the function of the adjacent solutions. The total crowding distance for a solution is the sum of the individual crowding distances for each objective.

The sorting and selection procedures are illustrated in Figure 2.6, where F_1 and F_2 are the objective functions to be minimized, P is the set of individuals from the older generation, Q the newly formed individuals, and F the different Pareto fronts.

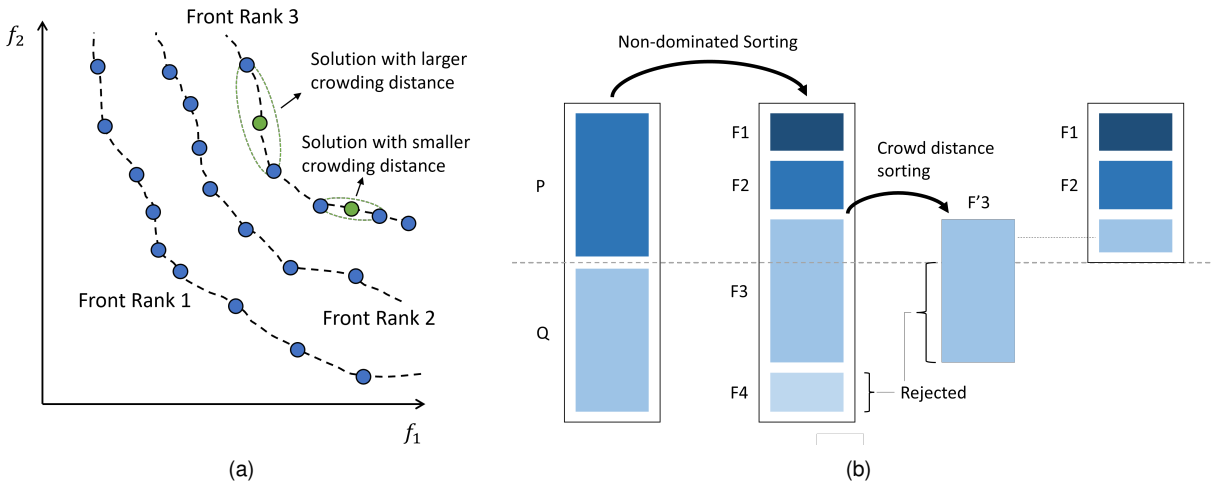


Figure 2.6: NSGA-II sorting process in a two dimentional objective space (a) and selection process (b).

2.2.3 NSGA-III

Designing a robust algorithm that could solve MaOOPs became possible through the exponential rise in computational power. To address the specific issues raised in this scenario, detailed in Section 2.1.2, NSGA-III was created, modifying the underlying algorithm NSGA-II.

To initialize the algorithm, the user now has to provide a set of reference points. These will be the focus of the optimization, so instead of trying to obtain all of the Pareto-optimal front, NSGA-III will focus on getting solutions to adequately represent the parts of the objective space in which the user is focused (the ones where the reference points are located).

NSGA-III behaves as its predecessor, but with a different process to select the members from the last front that will make it to the next generation. Instead of a crowding distance using the Manhattan distance in objective space, the following method is used.

At each iteration, the objective space is normalized. To achieve this, the ideal point is calculated, which contains the minimum value of all the objectives for the current population. Let it be denoted by $\bar{z} = \left(z_1^{\min}, z_2^{\min}, \ldots, z_M^{\min}\right)$. The new objective functions are now defined as $f_i'(\mathbf{x}) = f_i(\mathbf{x}) - z_i^{\min}$. The extreme points of each objective are those that minimize the Achievement Scalarizing Function (ASF) with the respective axis.

$$\mathrm{ASF}(\mathbf{x}, \mathbf{w}) = \max_{i=1}^{M} f_i'(\mathbf{x})/w_i, \tag{2.3}$$

Where x is a given population member, w the axis for one of the M objectives. When $w_i = 0$, it is replaced by a small value, 10^{-6}. These M extreme points form a M-dimensional linear hyperplane. After computing the interception of this hyper-plane with the translated axis, with a_i representing the intercept value on the i-th objective, the objective functions can be normalized as follows.

$$f_i^n(\mathbf{x}) = \frac{f_i'(\mathbf{x})}{a_i - z_i^{\min}} = \frac{f_i(\mathbf{x}) - z_i^{\min}}{a_i - z_i^{\min}}, \quad \text{for } i = 1, 2, \ldots, M \tag{2.4}$$

The reference points are projected into the normalized objective space, and the reference directions are drawn as a straight line between them and the ideal point (Figure 2.7(a)). To select the surviving solutions from the split front, NSGA-III first fills in the underrepresented reference directions (Figure 2.7(b)). If the reference direction does not have any assigned solution, then the solution with the smallest perpendicular distance in the normalized objective space is selected. In case a second solution for this reference line is added, it is assigned randomly.

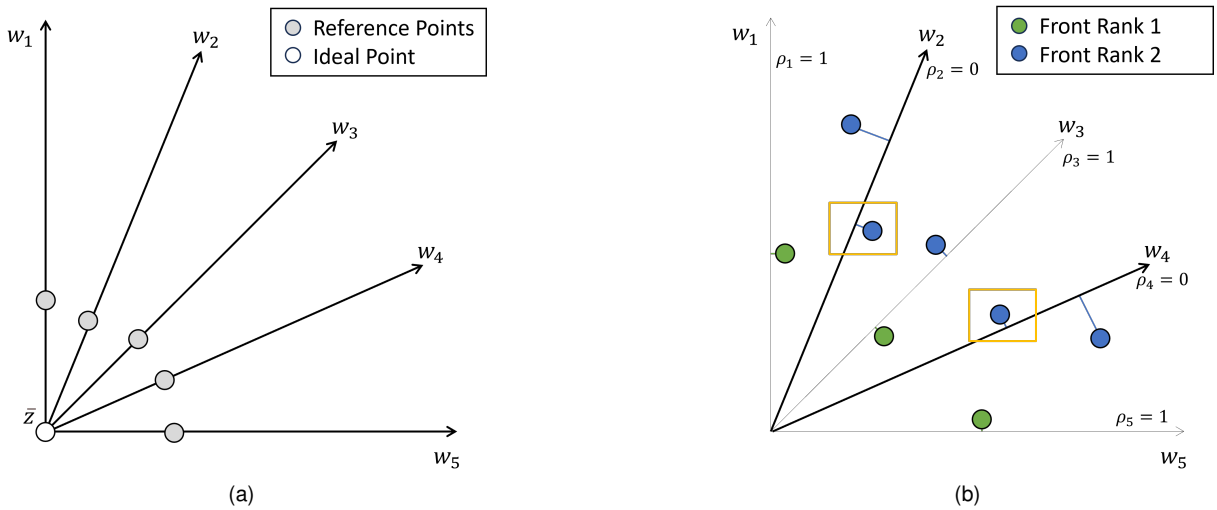


Figure 2.7: Selection method of NSGA-III. Drawing of reference directions in the normalized objective space (a) and selected solutions of the split front inside the yellow squares (b).

2.2.4 Differential Evolution

Differential evolution is a population-based optimization algorithm that was first proposed by Rainer Storn and Kenneth Price in 1995 [29]. It is a robust and efficient optimization method that has been applied to a wide range of optimization problems.

The key difference between DE and other Evolutionary Algorithms is that it uses differential mutation, a process in which the difference between two individuals is used to generate a new candidate solution.

The basic DE algorithm can be summarized as follows:

1. **Initialization**. Create a population of candidate solutions (randomly or input chosen by the user), $X_1, X_2, ..., X_N$. Each individual is represented as a vector of real-valued parameters.

2. **Mutation**. For each target vector, X_i, generate a mutant vector V_i, using a combination of three other randomly selected individuals (X_{r_0}, X_{r_1} and X_{r_2}) from the population (illustrated in Figure 2.8(a)):

$$V_i = X_{r_1} + F \cdot (X_{r_2} - X_{r_3}) \tag{2.5}$$

3. **Crossover**. For each of the components of target vector X_i, change its value to the respective mutant vector component, with a probability of CR (Crossover Rate). The CR is set to one in a random dimension, to guarantee that the newly generated mutant V_i', is always different than the target vector X_i. Figure 2.8(b) illustrates all 3 possible vectors resulting in the crossover in 2 dimensions.

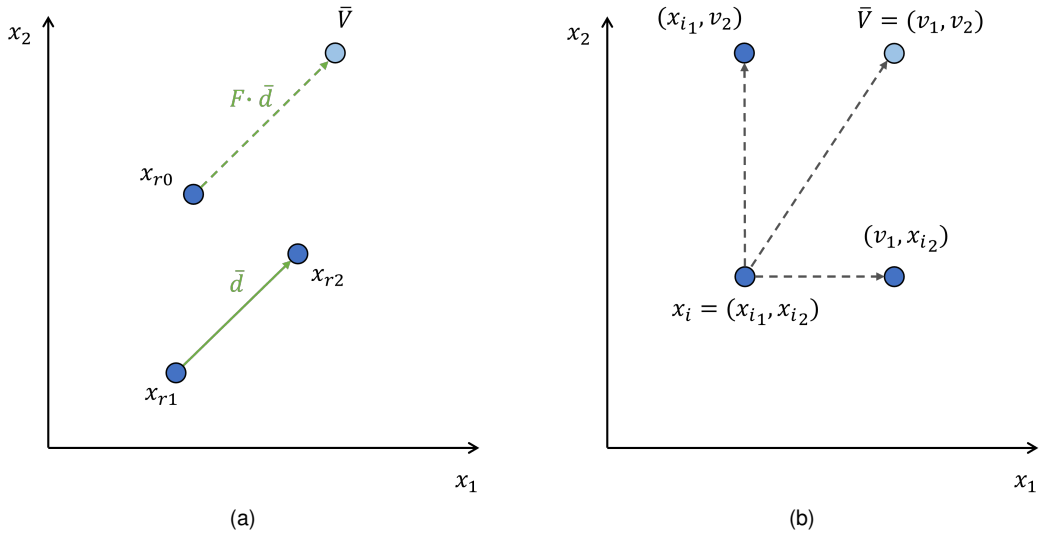


Figure 2.8: Illustration of the DE mutation (a) and crossover (b) operations in a search space with two dimensions.

4. **Survival**. Compare the new mutant vector V_i', with the target vector, X_i. The individual selected to be in the next generation (X_i') is the one with the better fitness value.

$$X_i' = \begin{cases} V_i & \text{if } f(V_i') < f(X_i) \\ X_i & \text{otherwise} \end{cases} \tag{2.6}$$

where X_i' is the new individual, $f(V_i)$ is the fitness of the mutant, and $f(X_i)$ is the fitness of the original individual.

Steps 2, 3, and 4 are repeated until the termination condition is met (e.g., a maximum number of iterations has been reached or desired fitness achieved).

2.2.5 Particle Swarm Optimization

Particle Swarm Optimization was proposed in 1995 by Kennedy and Eberhart [19] based on the simulation of social behavior. It differs from GAs in that instead of a population that generates offspring and can then be replaced by it, it operates in a swarm of particles that are evolved (move) to a new position at each generation, according to its velocity.

Given the following variables:

- $X_d^{(i)}$ is the d-th coordinate of the position of the i-th particle in the search space.
- $V_d^{(i)}$ is the d-th coordinate of the velocity of the i-th particle. The velocity of a particle determines how it moves through the search space.
- ω is the inertia weight, which determines how much the current velocity of a particle is influenced by its previous velocity.
- $P_d^{(i)}$ is the d-th coordinate of the personal best position of the i-th particle, which is the best position that the particle has found so far.
- $G_d^{(i)}$ is the d-th coordinate of the globally (or sometimes locally) best solution found by the swarm.
- c_1 and c_2 are weight values that are used to balance the influence of the particle's personal best position and the swarm's global best position on the particle's velocity update.
- r_1 and r_2 are random values that are used in the velocity update equation to introduce randomness and exploration into the algorithm.

The PSO can be described by the following steps:

1. **Initialization**. A set of possible solutions to the optimization problems is randomly generated or given by the user, as well as their initial velocities.
2. **Velocity calculation**. The velocity of each particle (illustrated in Figure 2.9) is calculated according to the following equation:

$$V_d^{(i)} = \omega V_d^{(i)} + c_1 r_1 \left(P_d^{(i)} - X_d^{(i)} \right) + c_2 r_2 \left(G_d^{(i)} - X_d^{(i)} \right). \tag{2.7}$$

3. **Particle update**. Each particle then moves to the new position using its updated velocity:

$$X_d^{(i)} = X_d^{(i)} + V_d^{(i)}. \tag{2.8}$$

Steps 2 and 3 are repeated until the termination criteria are met.

The cognitive behavior of the swarm is represented by the term $c_1 r_1 \left(P_d^{(i)} - X_d^{(i)}\right)$, which includes the personal best solution found by the particle. The term $c_2 r_2 \left(G_d^{(i)} - X_d^{(i)}\right)$ expresses the social behavior by using the best solution found globally (or locally) in the swarm for the velocity update. Cognitive and social components must be well balanced (c_1 and c_2 weights) to ensure that the algorithm performs well on a variety of optimization problems.

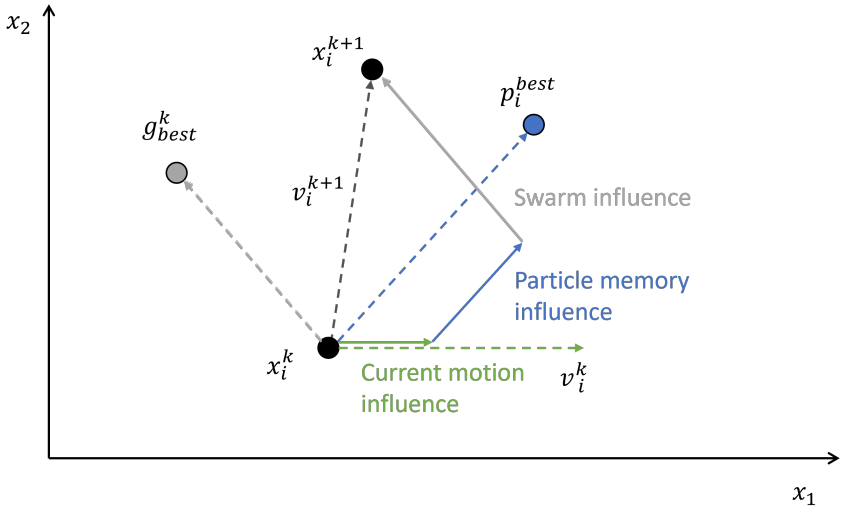


Figure 2.9: PSO velocity vector construction.

2.2.6 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic optimization algorithm that is inspired by the foraging behavior of ants. By simulating this behavior, ACO algorithms are used to solve optimization problems. The working principles are illustrated in Figure 2.10.

The fundamental idea behind ACO algorithms is that ants communicate with each other using chemical pheromones. When an ant finds food, it leaves a chemical trail of pheromones behind it as it returns to the nest. The other ants can then follow this trail of pheromones to locate the food source. With time, the pheromone trails become more concentrated, indicating the presence of a more efficient path to the food source. ACO is tailored to combinatorial optimization problems, widely used in routing and scheduling problems.

The steps involved in an ACO algorithm include the following:

1. **Initialization**. The algorithm begins by randomly generating a set of possible solutions to the optimization problem (called ants). These solutions are represented as paths through the search space.

2. **Pheromone update**. Trails are usually updated when all the ants have completed their solution. The algorithm updates the pheromone trail for each solution according to its quality. If a solution is particularly good, the pheromone trail on that solution is strengthened, indicating to other ants

that this is a good path to follow. This process can be mathematically represented as follows.

$$\text{new ph}(i,j) = ev \cdot \text{ph}(i,j) + \sum_{k}^{m} \Delta \text{ph}(i,j)^k, \tag{2.9}$$

where $\text{new ph}(i,j)$ is the updated pheromone, ev is the pheromone evaporation coefficient, m is the number of ants and $\Delta \text{ph}(i,j)^k$ is the amount of pheromone deposited by k th ant, typically given by

$$\Delta \text{ph}(i,j)^k = \begin{cases} Q/L_k & \text{if ant } k \text{ uses edge } (i,j) \text{ in its tour} \\ 0 & \text{otherwise,} \end{cases} \tag{2.10}$$

where L_k is the cost of the k th ant's tour (typically length) and Q is a constant.

3. **Solution construction**. Each ant selects its next move based on current pheromone trails and a heuristic function that estimates the quality of each possible solution. The heuristic function takes into account factors such as distance to the goal, availability of resources, and presence of obstacles. This process can be represented mathematically as:

$$p(i,j) = \frac{\text{ph}(i,j)^\alpha \times h(i,j)^\beta}{\sum_{k \in \text{All}} \text{ph}(i,k)^\alpha \times h(i,k)^\beta}, \tag{2.11}$$

where, for edge (i,j), $p(i,j)$ is the probability of an ant choosing it, $\text{ph}(i,j)$ its pheromone concentration, $h(i,j)$ its the heuristic value, α and β the pheromone and heuristic influence factors.

Steps 2 and 3 are repeated until the termination criteria are met.

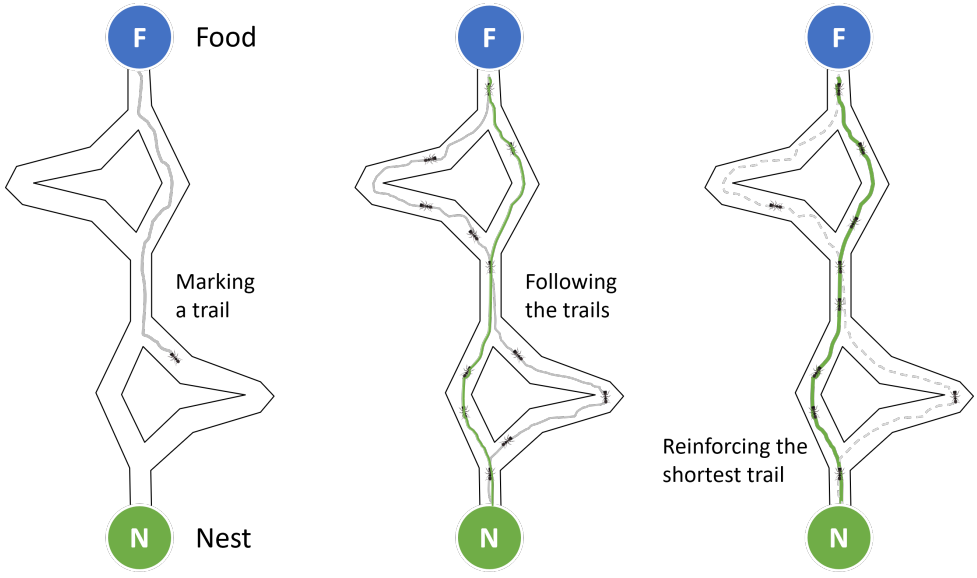


Figure 2.10: Working principals of ACO.

2.2.7 Summary

A summary of the detailed algorithms is presented in Table 2.1.

Table 2.1: Summary of the detailed algorithms.

EA	Fundamental Characteristics	Use Cases	Related Work
GA	Uses concepts from genetics and natural selection to search for optimal solutions. Very adaptative, can solver a wide range of problems. Hyperparameters have to be well tuned.	Machine Learning hyperparameter tuning	Holland [30]
NSGA-II	Modification of GA tailored for MOOPs. Uses non-dominated sorting and genetic operators to find a set of Pareto optimal solutions. May be slow in many-objective optimization problems.	Engineering design	Deb et al. [1]
NSGA-III	Modification of NSGA-II tailored for MaOOPs. Uses a special selection to explore specific areas of the Pareto Front associated with the reference directions given by the user.	Design of Vehicles	Deb and Jain [21]
DE	Uses differences between candidate solutions to guide the search for optimal solutions. Easy to implement, good convergence properties. Sensitive to noise and outliers in the data.	Neural Network training, Clustering	Price et al. [29]
PSO	Simulates the social behavior of birds and other animals to find optimal solutions. Limited ability to handle constraints.	Temperature prediction, Leukemia diagnosis	Kennedy and Eberhart [19]
ACO	Inspired by the foraging behavior of ants. Tailored for combinatorial problems. Sensitive to control parameters.	Scheduling, Routing	Dorigo te al. [31]

2.3 Performance Indicator

It is important to discuss by which measure will the evolution of the Peformance Indicator (PI) be counted over. In a first approach, it could be compelling to measure the evolution of the PI over time. When benchmarking the algorithm in a controlled and well-defined test function, the time to evaluate the function is reduced, so the algorithm overhead to generate the next generation could be problematic.

In practical scenarios, numerous optimization problems involve intricate and extensive mathematical equations or specialized software for evaluation. Utilizing third-party software frequently leads to functions that are computationally intensive and time-consuming to assess objectives or constraints.

For such problems, the computational overhead of the algorithm in identifying the next solutions to evaluate is typically insignificant. It is then advisable to measure the evolution of the PI over metrics such as the number of Function Evaluations (FE), or generations.

Concerning the choice of the PI itself, in a SOOP, it will most likely involve knowing the direct distance between the objective value of the solutions found to the global optimum, given that there is only one objective. However, in MOOPs, it is necessary to have a well-defined PI, as the solutions are not necessarily directly comparable, if one is not dominated by the other (Figure 2.11(a)). Moreover, it is desirable not only to have convergence with the true Pareto front (Figure 2.11(b)), but also to maintain good diversity between solutions (Figure 2.11(c)).

For this, a series of PIs are presented to measure these two objectives, convergence to the true PF and diversity among solutions. In addition, both cases of a problem with a known and unknown PF are addressed.

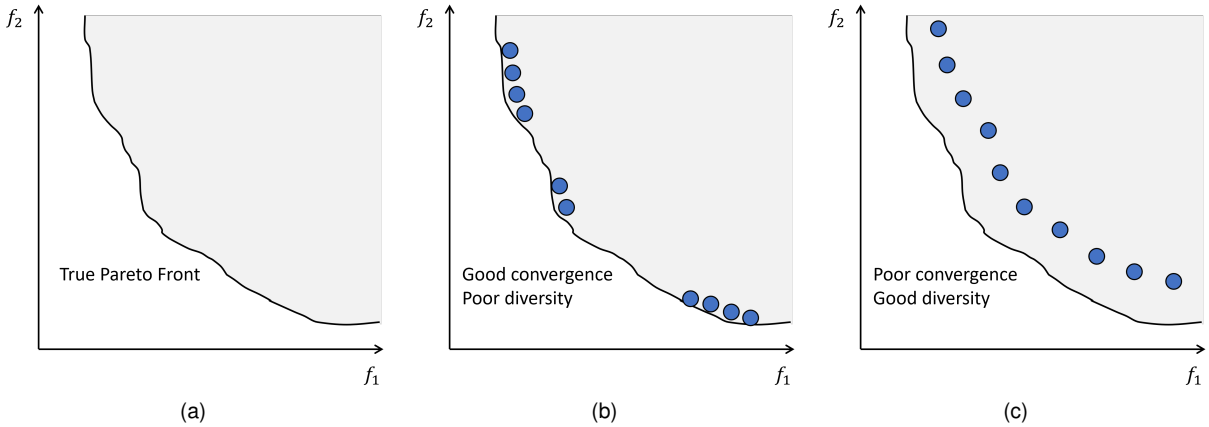


Figure 2.11: The two different goals regarding the PF minimizing $f1$ and $f2$ (a). Convergence to the true PF (b) and Diversity among solutions (c).

2.3.1 Known Pareto Front

It can be the case, if an algorithm is being tested in a controlled environment, for benchmarking purposes, to use it in well-defined problems. Problems for which the solution, its Pareto-optimal front, is already known. In this case, the following PIs can be used.

Generational Distance (GD)

GD measures the average distance from any point in the solution set given by the algorithm to the nearest reference point, usually the PF [32]. Given a set of points found by the algorithm used, described by the objective vector set $A = \{a_1, a_2, \ldots, a_{|A|}\}$ and the reference point set is $Z = \{z_1, z_2, \ldots, z_{|Z|}\}$. Then

$$\mathrm{GD}(A) = \frac{1}{|A|}\left(\sum_{i=1}^{|A|} d_i^p\right)^{1/p}, \tag{2.12}$$

where d_i represents the Euclidean distance $(\mathrm{p} = 2)$ from a_i to its nearest reference point in Z.

Inverted Generational Distance (IGD)

The IGD performance indicator [33] inverts the generational distance and measures the average distance from any point on the PF Z to the nearest point in the solution set A

$$\mathrm{IGD}(A) = \frac{1}{|Z|}\left(\sum_{i=1}^{|Z|} \hat{d}_i^p\right)^{1/p}, \tag{2.13}$$

where $\hat{d}_i$ represents the Euclidean distance $(\mathrm{p} = 2)$ from z_i to its nearest reference point in A

Generational Distance Plus (GD+) and Inverted Generational Distance Plus (IGD+) are variations of GD and IGD, respectively, which have been proposed by Ishibushi et al. [34] in order to address some of the limitations of the original GD and IGD PIs.

Both GD+ and IGD+ use a modified distance measure that takes into account the direction of the

difference between the approximation set and the Pareto front. Specifically, GD+ and IGD+ use a measure of distance that is only non-zero when the approximation set is worse than the PF in at least one objective.

This makes GD+ and IGD+ weakly Pareto compliant, meaning that they only penalize solutions that are not Pareto-optimal, rather than also penalizing solutions that are strictly better than the Pareto front in some objectives but worse in others. Figure 2.12 illustrates the difference between the IGD and IGD+ indicators. Their mathematical formulations are described below.

Generational Distance Plus (GD+)

$$\mathrm{GD}^{+}(A) = \frac{1}{|A|}\left(\sum_{i=1}^{|A|} {d_i^{+}}^2\right)^{1/2}, \tag{2.14}$$

where for minimization $d_i^{+} = \max\{a_i - z_i, 0\}$ represents the modified distance from a_i to its nearest reference point in Z with the corresponding value z_i.

Inverted Generational Distance Plus (IGD+)

$$\mathrm{IGD}^{+}(A) = \frac{1}{|Z|}\left(\sum_{i=1}^{|Z|} d_i^{+2}\right)^{1/2}, \tag{2.15}$$

where for minimization $d_i^{+} = \max\{a_i - z_i, 0\}$ represents the modified distance from z_i to the closest solution in A with the corresponding value a_i.

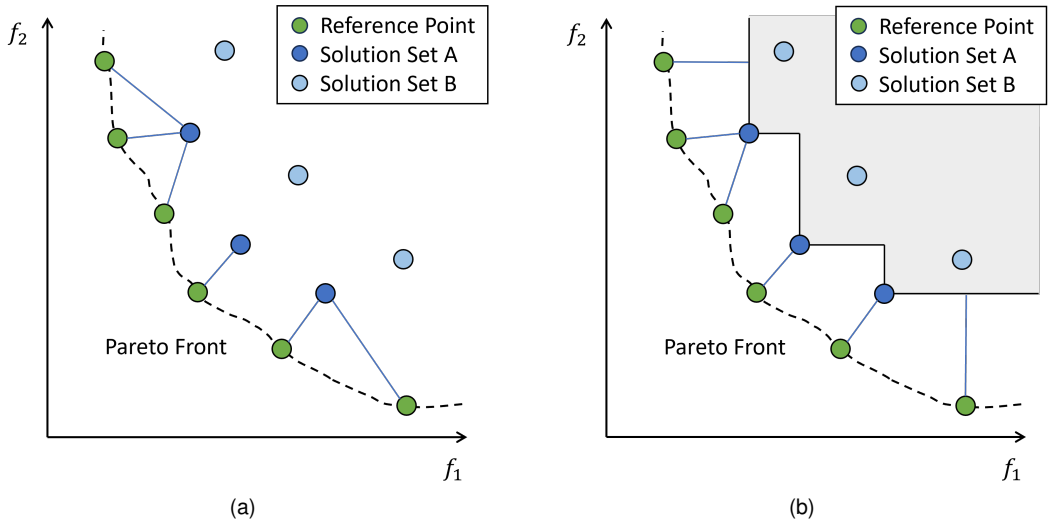


Figure 2.12: Illustration of the process to obtain the IGD (a) and IGD+ (b) PIs in an objective space to minimize f_1 and f_2.

2.3.2 Unknown Pareto Front

In real-life problems, the optimal Pareto front is unknown (or else there would be no problem, since the solution would already be known). It is then necessary to evaluate the algorithm's performance even in this case. A standard indicator uhypervolume, explained in the following.

Hypervolume

For hypervolume (HV), only a reference point needs to be provided. Calculates the area/volume, dominated by the set of solutions provided concerning a reference point. This process is illustrated in Figure 2.13, where HV is the gray area of the figure. It is important to note that while the other performance indicators detailed until now were to be minimized, the HV is to be maximized.

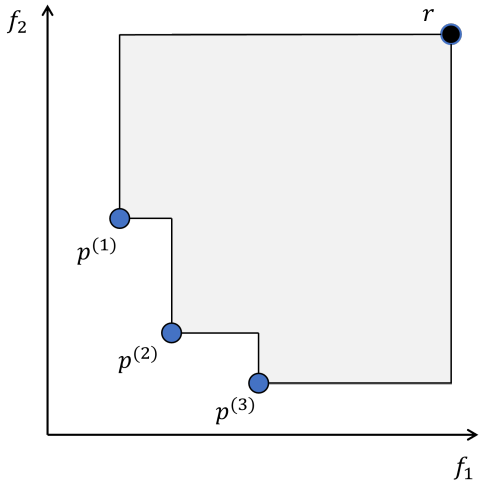


Figure 2.13: Hypervolume calculation minimizing both objectives.

2.3.3 Summary

A summary of the detailed performance indicators is presented in Table 2.2.

Table 2.2: Summary of the detailed performance indicators.

PI	Fundamental Characteristics	Needs Known PF	Measures	Related Work
GD	Average distance from any point in the solution set given by the algorithm to the nearest point on the Pareto-front.	✓	- Accuracy	Veldhuizen [32]
IGD	Average distance from any point on the Pareto-front to the nearest point in the solution set found by the algorithm.	✓	- Accuracy - Diversity	Coello and Reyes [33]
GD+	Modification of GD that is weakly compliant.	✓	- Accuracy	Ishibuchi et al. [34]
IGD+	Modification of IGD that is weakly compliant.	✓	- Accuracy - Diversity	Ishibuchi et al. [34]
HV	Calculates the area which is dominated by the provided set of solutions concerning a reference point. Computationally expensive in higher dimensions.	✗	- Accuracy - Diversity	Fonseca et al. [35]

2.4 Benchmark Problems

As EAs are used to solve a wide range of real-life optimization problems, it becomes necessary to use a variety of test problems, with different types of challenges, to evaluate their performance in those specific conditions. The objective of this section is not to present a comprehensive list of all the available renown benchmark problems, but rather to explain the need for them and give examples of each case.

Optimization problems can be separated into two different types.

- **Combinatoric.** The objective is to find the best combination of items within a certain set, where the order may matter, depending on the problem. A well-known benchmark for this case is the Traveling Salesman Problem (TSP), where the objective is to find the shortest path to pass in every city exactly once (Figure 2.14(a)). Mathematically, the optimization problem can be formulated as follows.

$$\begin{array}{lll} \min & \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} c_{ij} x_{ij}) & \\ \text{s.t.} & \sum_{i=1, i\neq j}^{n} x_{ij} = 1 & \text{for all } j, \\ & \sum_{j=1, j\neq i}^{n} x_{ij} = 1 & \text{for all } i, \\ & \sum_{i\in S, j\notin S} x_{ij} \geq 1 & \text{for all } S \subset \{1, 2, \ldots, n\}, S \neq \emptyset, S \neq \{1, 2, \ldots, n\}. \end{array} \tag{2.16}$$

- **Numeric.** Problems that involve finding a solution by optimizing a numeric value, such as a mathematical function. An example is the Ackley function, which is multimodal and non-convex (Figure 2.14(b)). It is defined with the mathematical equation below.

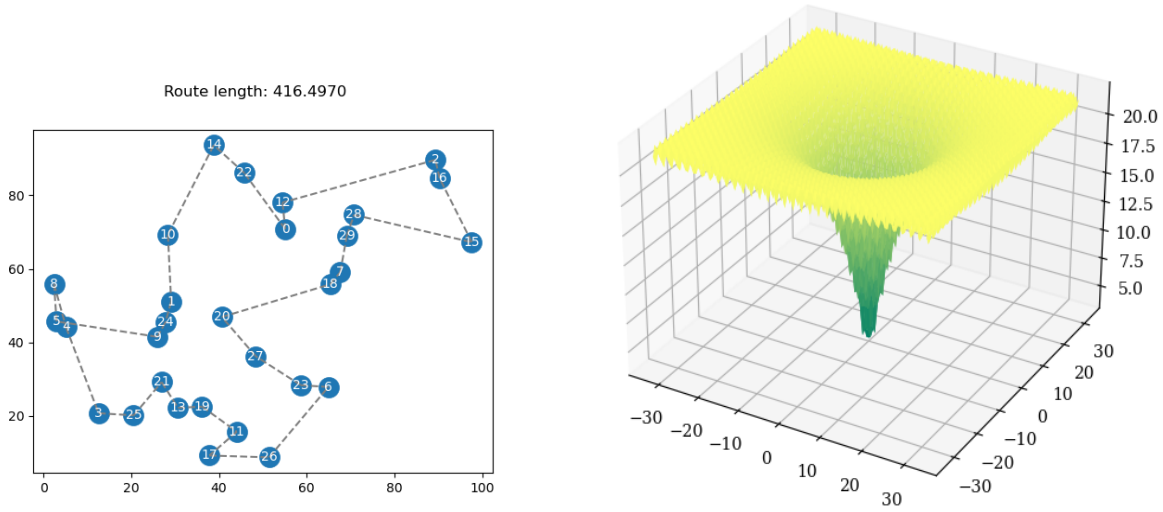
$$f(x) = -a \exp\left[-b\sqrt{\frac{1}{n}\sum_{i=1}^{n} {x_i}^2}\right] - \exp\left[\frac{1}{n}\sum_{i=1}^{n} \cos(cx_i)\right] + a + e. \tag{2.17}$$

In both cases, the problems can be further distinguished between SOOPs and MOOPs. While the first can have challenges with respect to its optimization surface, the latter faces additional challenges with respect to its PF shape. Assuming the main difficulty for which the EAs are developed, to tackle non-convexity and multi-objective optimization, below are some of the additional challenges that are faced in each of these areas.

2.4.1 Optimization Surface

- **Local minima:** Non-convex optimization problems often have multiple local minima, which are points that are locally optimal but not globally optimal. This means that an optimization algorithm might get stuck at a local minimum and be unable to find the global minimum. This is the case of the **Rastringin** problem, with its fitness landscape presented in Figure 2.15(a), and defined as:

$$\begin{aligned} f(x) =& 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right] \\ & -5.12 \leq x_i \leq 5.12 \quad i = 1, \ldots, n. \end{aligned} \tag{2.18}$$



(a) (b)

Figure 2.14: A grid representing a possible solution for the combinatorial problem TSP (a) and the fitness landscape of a two-dimensional numeric problem, Ackley, with the search space represented in the horizontal axes and the objective value in the vertical axis.

- **Plateaus:** Non-convex optimization problems can also have plateaus, which are regions of the search space where the objective function is relatively flat and the algorithm's progress slows down. This can make it difficult for the algorithm to progress and find the global minimum, as in the **Rosenbrock** problem of Figure 2.15(b). Its mathematical formulation is given by:

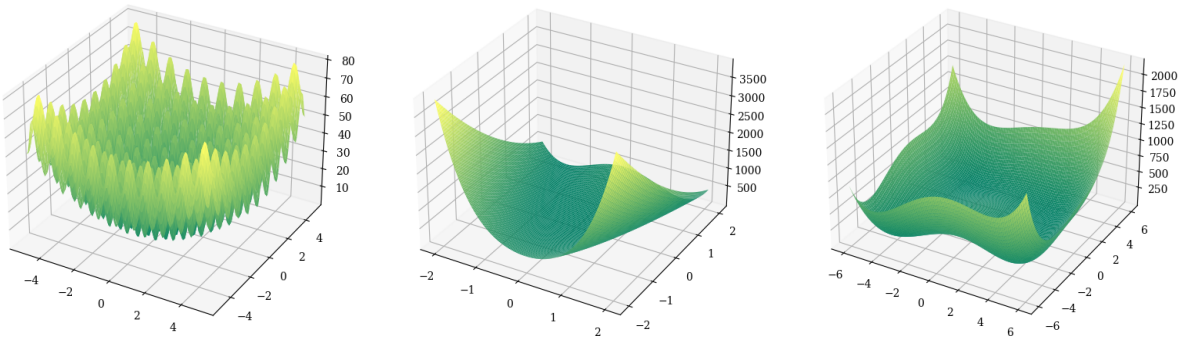
$$f(x) = \sum_{i=1}^{n-1} \left[100 \left(x_{i+1} - x_i^2 \right)^2 + \left(x_i - 1 \right)^2 \right] \\ -2.048 \leq x_i \leq 2.048 \quad i = 1, \ldots, n. \tag{2.19}$$

- **Saddle points:** Non-convex optimization problems can also have saddle points, which are points where the objective function has a local minimum in some dimensions and a local maximum in others. Saddle points can be particularly challenging because they can trap the algorithm and prevent it from progressing. Figure 2.15(c) presents the fitness landscape of the **Himmelblaus** problem, defined by the following equation.

$$f(x_1, x_2) = \left({x_1}^2 + x_2 - 11 \right)^2 + \left(x_1 + {x_2}^2 - 7 \right)^2 . \tag{2.20}$$

2.4.2 Pareto Front Shape

In multi-objective optimization, the shape of the PF can vary depending on the problem being solved, and some shapes can be more difficult to optimize than others. Some common Pareto front shapes that can be challenging include:



(a) (b) (c)

Figure 2.15: Fitness landscape of SOOP evidencing different challenges. Vertical axis corresponds to the objective value, horizontal axes are the two dimensional search space. Rastrigin - widespread local minima (a). Rosenbrock - Plateaus (b). Himmelblaus - saddle points (c).

- **Convex Pareto fronts**. These can be difficult to find because they require the algorithm to explore a large area of the search space. This is the case of problem **ZDT1**, with its parent front depicted in Figure 2.16(a), and defined by:

$$\begin{aligned} f_1(x) &= x_1 \\ g(x) &= 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h\left(f_1, g\right) &= 1 - \sqrt{f_1/g} \\ 0 \leq x_i &\leq 1 \quad i = 1, \ldots, n. \end{aligned} \tag{2.21}$$

- **Non-convex Pareto fronts**. These can be challenging because they require the algorithm to find solutions in multiple disconnected regions of the search space. It is the case of **ZDT2** (Figure 2.16(b)), with its mathematical equation detailed below.

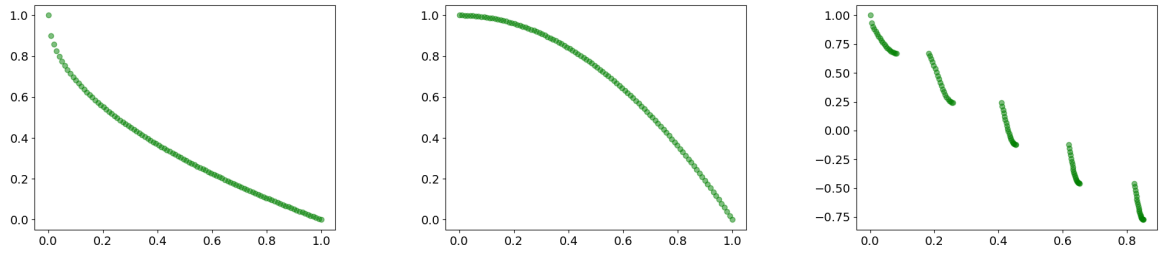
$$\begin{aligned} f_1(x) &= x_1 \\ g(x) &= 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h\left(f_1, g\right) &= 1 - \left(f_1/g\right)^2 \\ 0 \leq x_i &\leq 1 \quad i = 1, \ldots, n. \end{aligned} \tag{2.22}$$

- **Multi-modal Pareto fronts**. Pareto fronts having multiple distinct peaks or modes. These present a difficulty to converge because they require the algorithm to find solutions in multiple disconnected regions of the search space, such as **ZDT3**, depicted in Figure 2.16(c). Its mathematical

formulation is:

$$\begin{aligned} f_1(x) &= x_1 \\ g(x) &= 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h\left(f_1, g\right) &= 1 - \sqrt{f_1/g} - \left(f_1/g\right)\sin\left(10\pi f_1\right) \\ 0 &\leq x_i \leq 1 \quad i = 1, \ldots, n. \end{aligned} \tag{2.23}$$

- **Unscaled objectives**. In almost all real-life problems, the different objectives are not on the same scale, which can lead the algorithm to incorrectly prioritize the minimization of one over the other, based on their absolute value.



(a) (b) (c)

Figure 2.16: MOOP PF evidencing different challenges - minimizing both objectives. ZDT1 - convex PF (a). ZDT2 - concave PF (b). ZDT3 - multi-modal PF (c).

2.5 Python - Concepts and Advantages

As one of the objectives of this work is to build an application using this programming language, it is important to understand its main advantages and fundamental concepts.

Python is a high-level, interpreted programming language that is known for its simplicity and readability. One of the key concepts in Python is the use of indentation to indicate blocks of code. Another important concept in Python is the use of variables. Variables are used to store data and can be used to reference that data throughout the program. Python uses a simple syntax for declaring variables, without the need for a specific keyword such as 'var' or 'int'.

It is considered one of the main programming languages in the world right now [15] due to its versatility and wide range of applications. It is used in various fields, including web development, data analysis, machine learning, and scientific computing. Python's popularity is due to its simplicity, readability, and the fact that it is easy to learn. Python is also known for its rich library support, which includes pre-written code for various tasks, saving time and effort for developers. Additionally, Python's vibrant community has developed thousands of third-party packages, which extend Python's functionality even further.

Python also supports Object-Oriented Programming (OOP), which is a programming paradigm that uses classes to represent real-world concepts and organize code in a way that is easy to understand, maintain, and extend. The main ideas behind OOP include abstraction, encapsulation, inheritance, and

polymorphism [36]. For the understanding of this work, the important concepts to grasp are described in the following.

- **Class**. A class is a blueprint or template for creating objects in Python. They define the properties (attributes) and behaviors (methods) that objects of that class will have. Inheritance allows you to create a new class based on an existing one. The existing class is called the parent class, and the new class is called the child class. The child class inherits attributes and methods from its parent class, which promotes code reuse and organization.

- **Object**: An object is an instance of a class, which means it is a specific representation of the class's structure and behavior. Objects are created by instantiating a class, which involves allocating memory for the object and executing the class's constructor method.

- **Instantiation**. Instantiation is the process of creating a new object from a class in Python. It involves allocating memory for the object and executing the class constructor method, which initializes the object's attributes and sets up its behavior.

- **Methods**. Methods are functions that are associated with a specific object or class in Python. They are used to perform actions or operations involving the object's attributes. Methods are defined as part of the class the object belongs to, and they are executed using instances of that particular class.

- **Attributes**. Attributes are variables that are associated with an object or class in Python. They are used to store information or state about the object or class. Attributes can be class attributes, which are variables that are inherited by every object of a class, or instance attributes, which are variables that are unique to each object of the class.

- **Dictionary**. Python dictionaries are a data structure that allows you to store key-value pairs. They are similar to hash tables in other programming languages, but Python dictionaries are more flexible and easier to use. The keys in a dictionary must be unique, and they can be any immutable type, such as strings or numbers.

2.6 Optimization Frameworks

There are several optimization frameworks available. The process of choosing one can vary according to the user's needs. One might be set on a programming language that is more comfortable with and then see the available options. In another case, the user might need to use the optimization process commercially, so the license type will be the most important. In a research case, an active community can help solve the problems raised by the exploration and extension of the framework. Furthermore, for some specific real-world use cases, there are specialized frameworks tailored for them.

Specific Optimization Frameworks

The **Analog IC Design Automation** (AIDA) framework [37] belongs to the latter case. This framework allows the user to execute an analog IC design process that starts from circuit-level specifications to a detailed physical layout, emphasizing design optimization and transfer through the use of advanced search techniques, precise circuit-level simulations, layout design rules, and parasitic extraction capabilities.

It is designed to support skilled analog IC designers by automating the most labor-intensive and repetitive tasks in the usual analog IC design process. Has two main components, AIDA-C and AIDA-L, focused on the optimal sizing and full layout, respectively. In addition, a graphical user interface is provided to assist in the design process, not requiring the user to have deep programming skills to be able to use the software. Figure 2.17 shows the result of the optimization process to achieve different circuits, with the obtained Pareto set contemplating various trade-offs between smaller size and greater gain.

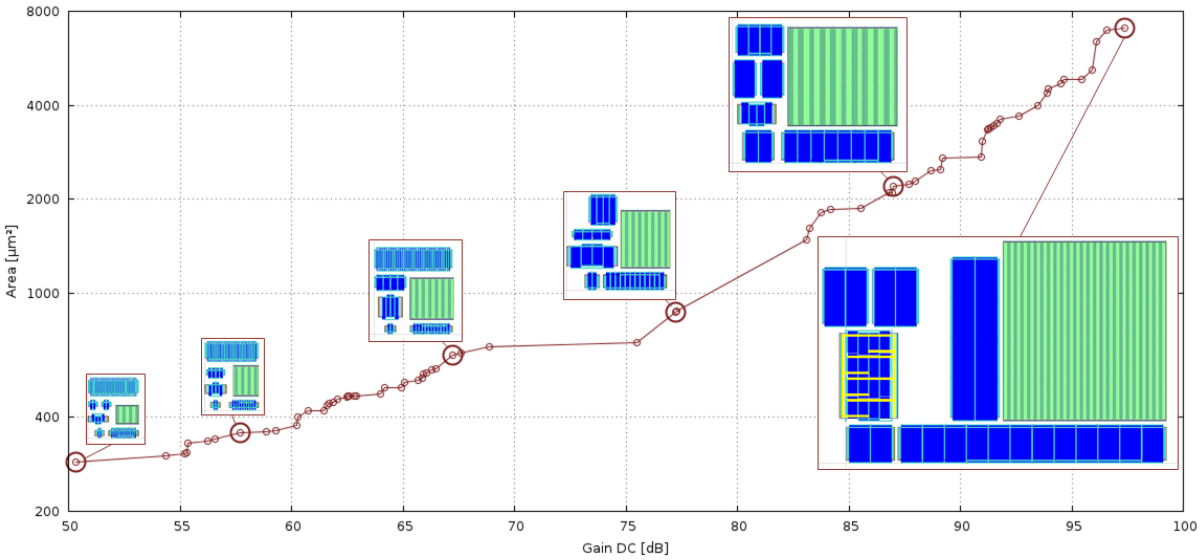


Figure 2.17: Pareto set obtained through AIDASoft framework [37]. Vertical axis displays area to be minimized, horizontal axis displays gain to be maximized.

General Optimization Frameworks

As the aim of the work is to build a more general application for optimization, it is of interest to explore in more detail the most established general optimization frameworks available, among which the following can be found.

- **jMetal** [22]. Developed in Java [9], includes a wide set of resources, including state-of-the-art multi-objective algorithms, solution encodings, benchmark problems, quality indicators, and utilities to perform experimental studies. Was ported to Python [14] in 2019, as **jMetalpy** [23].
- **MOEA** [8]. Written in Java, specializes in multi-objective optimization. The library is highly cus-

tomizable and has tools for decision-making. Includes a simple GUI that offers different visualization methods in one-to-one runs (one algorithm to one problem).

- **PlatEMO** [10]. Developed in Matlab [11]. Includes a powerful GUI for performing experiments in parallel and generating results in the format of Excel or LaTeX tables by one-click operation.
- **PyGMO** [38]. Developed in Python, focuses on massively parallel optimization. It is built around the idea of providing a unified interface to optimization algorithms and optimization problems and making their deployment in massively parallel environments easy.
- **DEAP** - Distributed Evolutionary Algorithms in Python [13]. DEAP is an evolutionary computation framework for rapid prototyping and testing of ideas. Does not focus on multi-objective optimization, although multi-objective algorithms can be developed due to its modularity and extensibility.
- **pymoo** [12]. Python-based, provides state-of-the-art single and multi-objective optimization algorithms, as well as tools for visualization, decision-making, and parallelization.

This is not, by any means, an extensive list, but rather a representation of the frameworks available in current times. The summary is presented in Table 2.3, which includes the type of license and fundamental characteristics related to the proposed work.

Table 2.3: Summary of the general optimization frameworks.

Name	License	First Release	Language	Visualization	Decision Making	GUI
MOEA	GNU LGPL	2011	Java	✓	✓	✓
jMetal	MIT	2015	Java	✓	✗	✗
PlatEMO	MIT	2016	Matlab	✓	✓	✓
DEAP	LGPL-3.0	2012	Python	✗	✗	✗
PyGMO	GPL-3.0	2013	Python	✗	✗	✗
pymoo	Apache 2.0	2020	Python	✓	✓	✗

2.7 Graphical User Interfaces

2.7.1 Connected to a General Optimization Framework

First, it is important to look at existing general frameworks that already provide a graphical interface. In the case of the Java-based MOEA framework, a simple GUI is provided to illustrate its potential. Alongside several tutorial pre-programmed examples that show the MOEA capabilities at solving different problems, a Diagnostic Tool is provided. This tool, shown in Figure 2.18, allows the user to run one algorithm on one problem at a time. In addition, a window is displayed to analyze and compare the results.

In the case of PlatEMO, the developers make use of the capabilities of the MATLAB software, which is able to incorporate the visual aspects with the code itself. Starting from the MATLAB editor, the GUI is

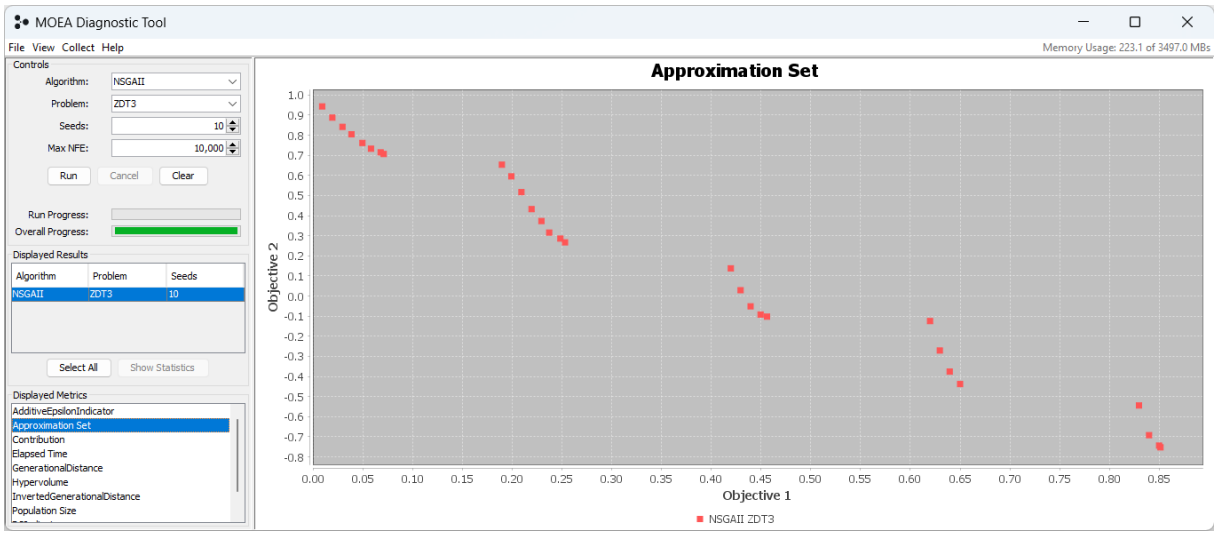


Figure 2.18: MOEA Diagnostic Tool.

divided into three modules. The Test Module (Figure 2.19) lets the user test one algorithm on a problem with specified parameter settings, and analyze the result, studying the performance of the algorithm from various aspects. The Application Module allows the user to create his own problem and solve it through the available algorithms.

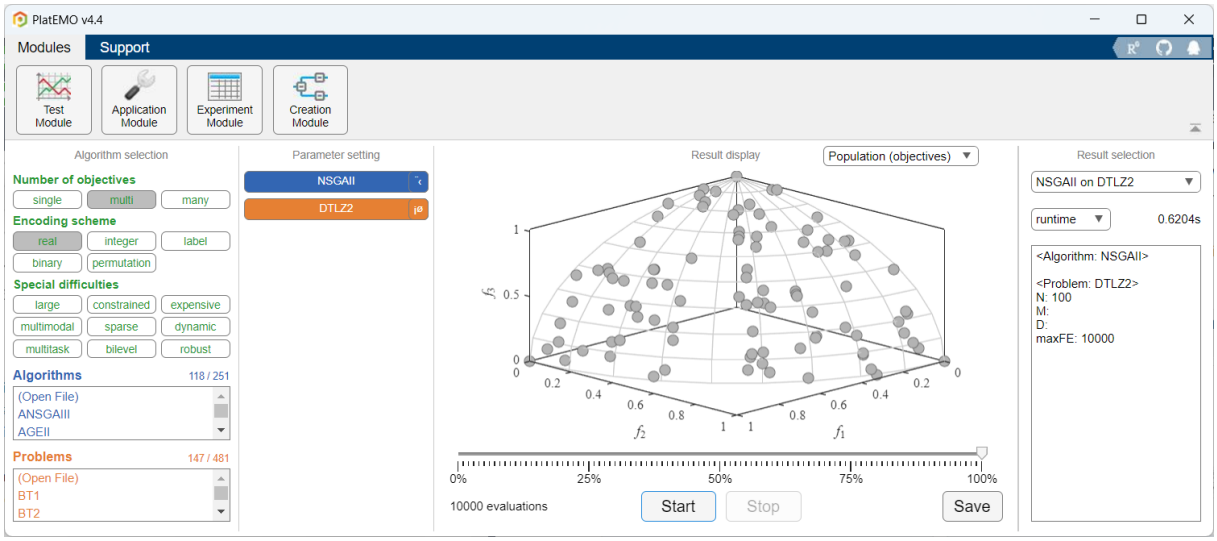


Figure 2.19: PlatEMO GUI Test Module.

Finally, the Experiment Module (Figure 2.20) allows the evaluation of multiple algorithms in multiple problems, creating a table to observe the statistical results of the run. The Creation Module present in the top left corner of both Figures 2.19 and 2.20 is in development. Table 2.4 provides a summary of the key characteristics between the available GUIs and the proposed solution.

Now it is possible to understand where the proposed solution can provide benefits. Besides being Python-based, it is supposed to allow the user to quickly test an Algorithm through multiple operators, and easily integrate self-coded options to the App. Furthermore, it should be available cost-free and

should not have resistance to academic, personal, or commercial use.

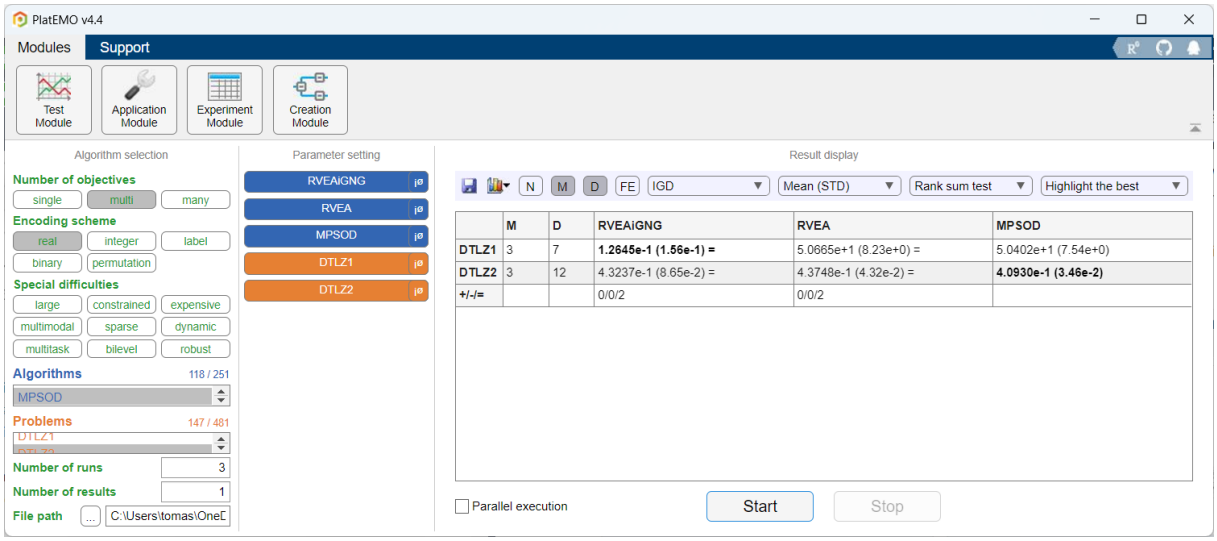


Figure 2.20: PlatEMO GUI Experiment Module.

Table 2.4: Characteristics of available optimization frameworks with a GUI and proposed solution.

Name	Free	Python-based	Runs Multiple Algorithms at Once	Tests Different Operators	User Code Integration
MOEA	✓	✗	✗	✗	✗
PlatEMO	✗	✗	✓	✗	✓
Proposed Solution	✓	✓	✓	✓	✓

2.7.2 Python Libraries for GUIs

As mentioned previously, the aim of this work is to provide a GUI for a Python-based optimization framework. With that being said, various approaches can be taken. There are some robust and well-documented software in the market, such as Microsoft's Power BI or Excel. However, these options have some major problems. They are well suited to present data that already exist, but the integration with Python is not the primary objective, nor serving as a graphical user interface with multiple input types and variables. Therefore, it becomes evident that a solution where the entire framework and GUI would be 100% Python-based is advisable. For this purpose, some of the existing Python GUI libraries are explored in the following list.

- **Tkinter** [39]. An open-source Python GUI library well known for its simplicity. It comes pre-installed in Python, which means that there is no additional work to get it. These features make it a great choice for beginners and intermediates, but it is not capable of carrying out large-scale projects and has few customization options.
- **Kivy** [40]. An OpenGL ES 2 accelerated framework designed for the creation of new user interfaces. Kivy is an open-source Python library for developing multi-touch applications. It is designed

to be easy to use and is ideal for creating applications that require a touch interface.

- **wxPython** [41]. A Python wrapper for the wxWidgets C++ GUI toolkit. It offers a wide range of widgets and tools that make it easy to create complex applications.
- **PyQt** [42]. Developed by Riverbank Computing, PyQt is one of the most popular Python frameworks for GUI. It is built around the Qt framework, which is a cross-platform framework used to create various applications on different platforms. It offers QtDesigner (Figure 3.3), a tool that provides visual elements that the developer can implement by drag and drop. You can also opt to create the element by code, which enables you to easily develop small- and large-scale applications.
- **PySide** [43]. Similar to **PyQt**, it also is a Python binding for the Qt framework. Developed by Qt, it offers a more permissive license but was released later to the general public.

Table 2.5 provides a summary of the libraries mentioned, with the characteristics described being retrieved up to November 23, 2023.

Table 2.5: Summary of the GUI libraries.

Name	License	Latest Stable Version	Release Date	First Release Date	Graphical Designer Tool
Tkinter	Python	(Python) 3.12.0	Oct 27, 2023	1991	✗
Kivy	MIT	2.2.1	Jun 17, 2023	2011	✗
wxPython	wxWindows	4.2.1	Jun 7, 2023	1998	✗
PySide	LGPL	6.5.1	Oct 27, 2022	2009	✗
PyQt	GPL	6.6.0	Oct 30, 2023	1998	✓

2.8 Conclusion

This chapter explored the fundamental concepts of single and multi-objective optimization. The specific challenges in each case were detailed. It presented some of the most established EAs, the PIs, as well as the different types of difficulties brought up by specific benchmark problems.

After grasping these theoretical concepts, the options for the proposed work and the available alternatives were analyzed. First, the state-of-the-art optimization frameworks available for general and specific use were presented in Section 2.6. A look at current alternatives for optimization through a GUI was done. Finally, the options of Python libraries for GUIs were explored.

With these foundations laid out, it is now possible to understand in more detail what the proposed solution can offer to be innovative in regard to its alternatives. Also, after exploring the different options for optimization frameworks to be the backbone of the application, as well as GUI libraries in Python, it is now possible to select the best options with which the proposed work should be built on, and delve into the architecture of the application.

Chapter 3

Architecture

In this chapter, the application architecture is detailed. Figure 3.1 illustrates the main components of its frontend, backend and testing modules. First, it is important to recognize the importance of software architecture. It plays a vital role in software development by serving as a roadmap for the system, allowing developers to understand and handle the implications of making specific modifications. A sound architecture simplifies the process by confining most changes to a single element or a small set of elements, thus minimizing the need for architectural alterations. The following sections should mention

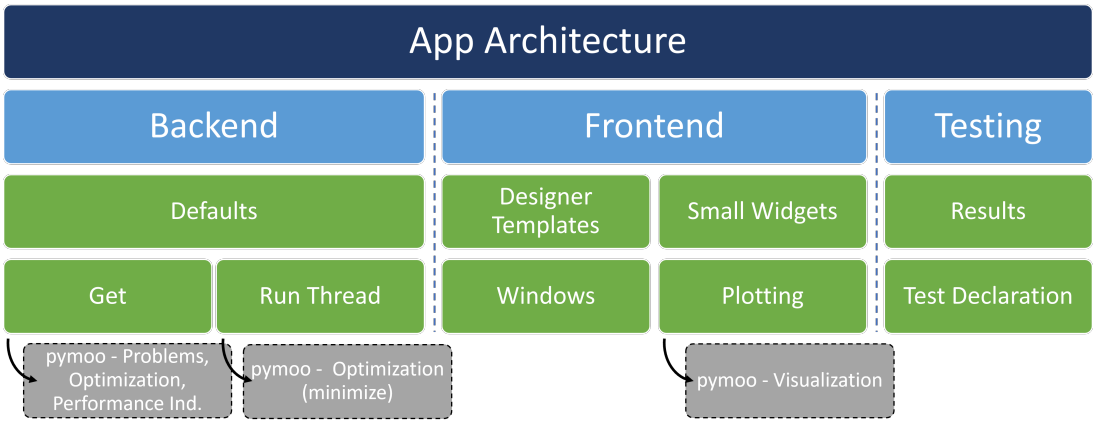


Figure 3.1: Overview of the application architecture.

which of the requirements delimited in Section 1.3 are they addressing, and how they achieve them. So in the end, it should be clear that the application meets the following: **1. Cost-free; 2. Python-based; 3. Test multiple state-of-the-art algorithms on various benchmarking problems; 4. Visualization techniques; 5. Easily extensible; 6. Customizable and easy to use**.

Furthermore, as this work intends to be the ground for future exploration and development, it is highly important that the architecture is well documented, to allow for a future contributor to clearly understand it and improvements to be easily done.

The chapter is organized as follows. In Section 3.1, a general overview of the architecture is given, with its choices for the optimization framework behind the application and the GUI library explained. The general display of the code structure is also given, laying out the main blocks that make up the

architecture of the app. Sections 3.2 and 3.3 provide details of the frontend and backend of the application, respectively. Lastly, Section 3.4 refers to other important aspects of the architecture, such as the frontend-backend interaction, application testing, scalability and robustness, and how to integrate user-coded classes in the app.

3.1 Overview

3.1.1 Building Choices

Python Optimization Framework

As seen in Section 2.6, there are multiple Python optimization frameworks available. However, pymoo [12] stands out as one of the most comprehensive, as it offers state-of-the-art optimization algorithms and benchmark problems, meeting **requirements 2 and 3**. In addition, its many visualization techniques will help with **requirement 4**.

Powered by *anyoptimization* (a Python research community), it is developed and maintained by Julian Blank, who is affiliated with the Computational Optimization and Innovation Laboratory (COIN) and supervised by Kalyanmoy Deb, a renowned scientist in the multi-objective optimization field, and one of the authors of NSGA. Pymoo also has an active community that interacts through a discord server, making it possible to ask for help or suggestions while using it. Its architecture is illustrated in Figure 3.2, and it is of great importance, as many of its components will be integrated into the desktop application. The links to the framework can be seen in Figure 3.1, shown at the beginning of the chapter.

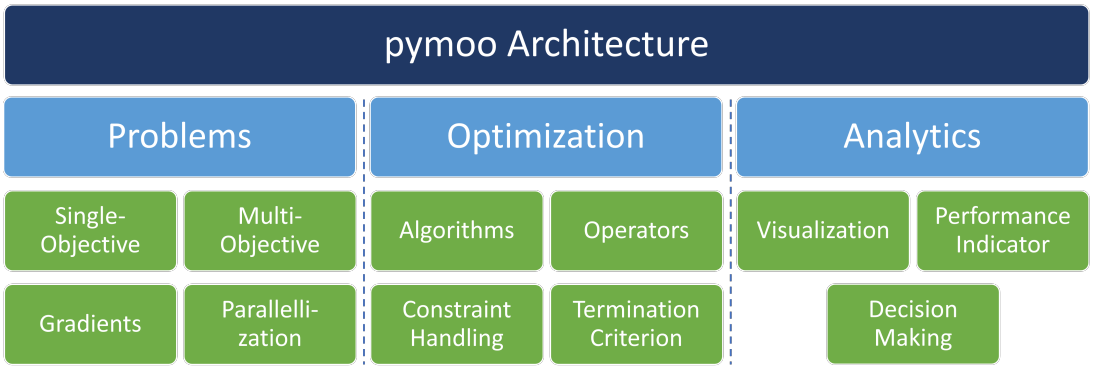


Figure 3.2: Overview of the pymoo architecture.

Pymoo is under the Apache License. It is a very permissive license which allows users to use the software for any purpose, to distribute it, to modify it, and to distribute modified versions of the software under the terms of the license, without concern for royalties. For this reason, it will help achieve **requirement 1** of being cost-free.

GUI Library

For the GUI library, PyQt is chosen, as it provides the most comprehensive customization tools that are going to be needed in this work. Furthermore, the graphical designer tool (Figure 3.3) provides a great aid in easily changing the look of the app and adding more components, which is very important in the development process, and also helps to meet **requirement 5**, to make it easily extensible to future developers' desires.

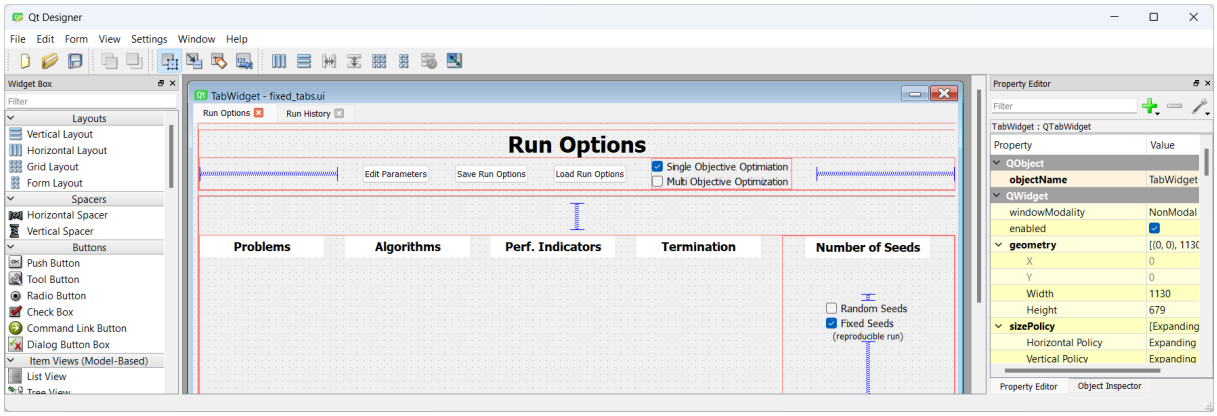


Figure 3.3: Designer Tool from PyQt creating the graphical component for a window of the application.

PyQt5 was selected despite its newer version, PyQt6, as a way to tackle its steeper learning curve regarding other options. As PyQt5's first release was in 2013 but is still maintained to date, there are a greater amount of examples available to learn from. Its General Public License (GPL) also guarantees end users the four freedoms to run, study, share, and modify the software, allowing **requirement 1** to be met.

3.1.2 General Code Structure

A well-structured code is essential to meet **requirement 5**. To make the application extensible, modularity is very important. It should be easy to know where each part is located, and the components separated, with their communication well defined, so that adding or modifying one of them is easy. For a desktop application, the main modules are normally the frontend and backend, explained below in general terms.

- **Frontend**. User-facing part responsible for presenting the interface and interacting with users. Includes elements such as the graphical user interface, user input processing, and client-side logic. The frontend is designed to provide a seamless and intuitive user experience, focusing on aspects such as responsiveness, usability, and visual appeal.

- **Backend**. Handles the core functionality of the application, such as data processing, business logic, and communication with external services or databases. It is responsible for managing data, performing computations, and ensuring the overall functionality of the application.

Having this in mind, the application was properly divided between these two components, which interact through well-defined interfaces, enabling them to communicate and exchange data efficiently while maintaining a clear separation of concerns.

In addition, the testing module (the last one in Figure 3.1) is created to assist during the development process, as it ensures that the current changes do not alter the desired app functionalities. Figure 3.4 displays the number of code lines in each folder of the project, corresponding to the three modules.

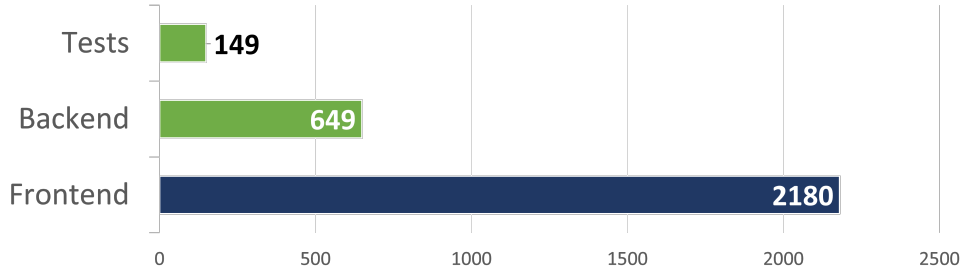


Figure 3.4: Lines of code developed in each module.

It comes as no surprise that the most intense work was done at the frontend level. This is because all the algorithms, problems, terminations, and other classes related to the optimization process itself are inside the pymoo library. Therefore, when the application code is analyzed, more code lines are required in the frontend module.

3.2 Frontend

In this module, everything the user directly interacts with is included. Figure 3.5 zooms in on its components and the respective files where they are coded. The following sections go through their details, explaining how they were designed, and how they relate to each other.

As a general rule, graphical elements are first conceived using the PyQt designer tool mentioned in Section 2.7.2. Figure 3.3 illustrates one of the graphical components of the app's starting window, created in it and then stored in the *designer_templates* folder as a *.ui* file. Afterward, these files are imported into the respective class through the framework's useful function *loadUi()*, as shown in Code Snippet 1. As a general disclaimer for the rest of the thesis, these Code Snippets sometimes present small modifications of the original code for better readability.

Code Snippet 1 Initializing a Window with its graphical component from the respective *.ui* file.

```
class MainTabsWidget(QTabWidget):
    def __init__(self, run_options: dict,
                 parameters: dict, moo: bool)
        super().__init__()

        loadUi(DESIGNER_FIXED_TABS, self)
        [...]
```

Finally, the graphical components are connected to the respective functions they are to perform,

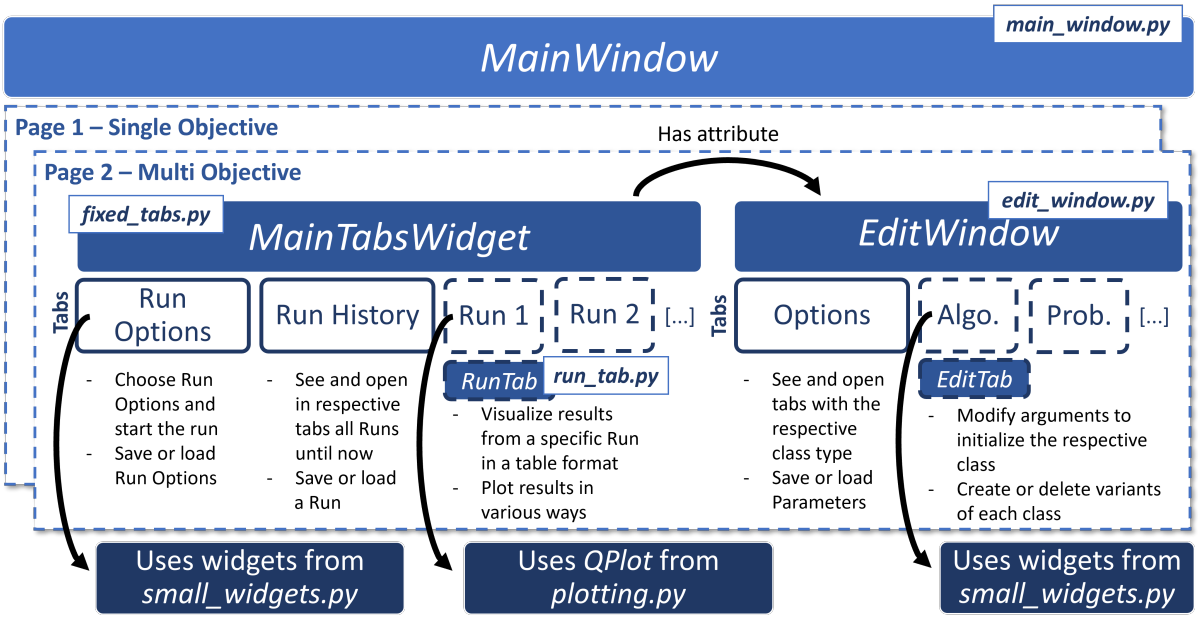


Figure 3.5: Frontend components.

building on the once-static interface into the actual functioning app.

3.2.1 Main Window

Shown in Figure 3.6, this is the window where the app naturally opens, with its class *MainWindow* defined in the *main_window.py* file. To tackle not only single but also multi-objective optimization, it has two pages, one for each type of optimization. The pages contain the same widget class, *MainTabsWidget* defined in the *fixed_tabs.py* file, instantiated twice, for SOOPs and MOOPs. In the Menu Bar of the app, on the top of Figure 3.6, three drop-downs: “File”, responsible for the saving and loading of components of in the app; “Options”, allowing the user to switch between MOOPs and SOOPs, or edit the parameters through the Edit Window; “Help”, which gives access to a tutorial and information about the project.

3.2.2 Main Tabs Widget

This widget, held by the virtual container, is shown in Figure 3.6 inside the dashed green square. It is coded in the *fixed_tabs.py* file, where the functionalities of the two fixed tabs, “Run Options” and “Run History”, are developed. Additional “Run” tabs are created after the end of a run. Because it can be instantiated a variable number of times and contains a larger number of options and functionalities, this tab is coded as a class in itself, *RunTab*, in the *run_tab.py* file. The following sections detail the respective tabs.

Besides what is visually displayed in the widget, this class stores as attributes the *run_counter*, which counts how many runs were performed, and the *edit_window*, where the Edit Window of the respective single or multi-objective optimization is stored.

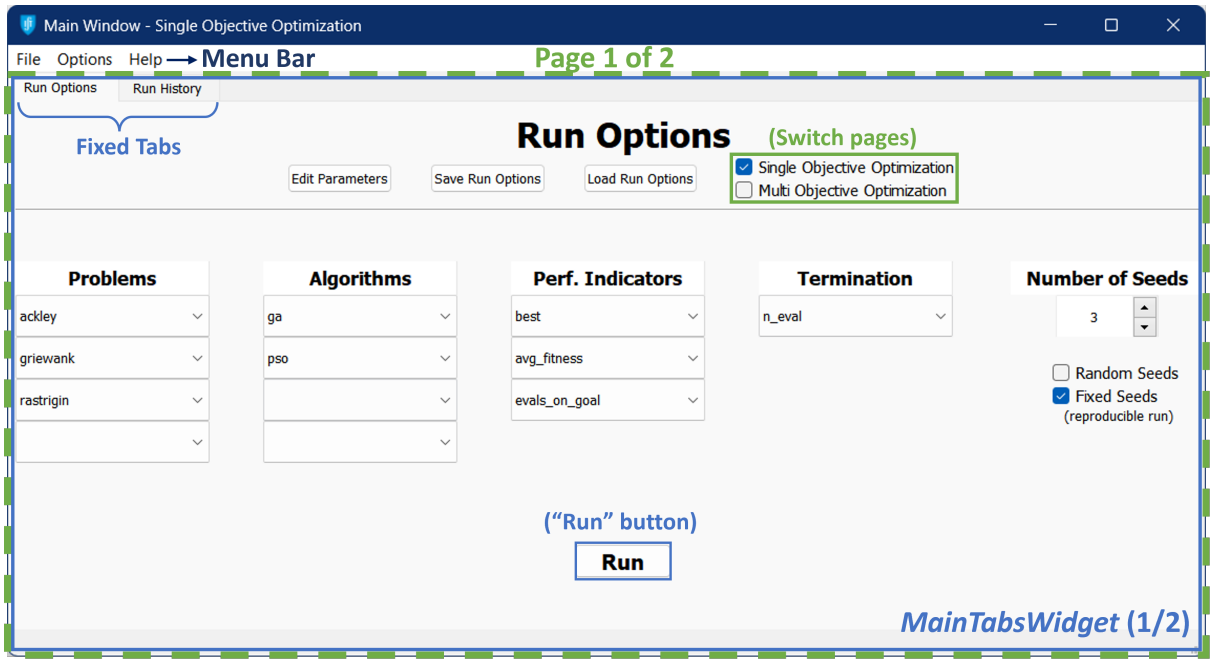


Figure 3.6: Main Window showing page 1, for single-objective optimization, and opened in the "Run Options" tab.

Run Options Tab

It is the visible tab of Figure 3.6. The user can choose multiple problems to solve using multiple algorithms. In optimization, it is a common practice to execute each problem-algorithm combination multiple times to obtain a more precise result. This is necessary because several algorithms involve processes that rely on probabilities, resulting in varying outcomes. Consequently, it is also required that the user define a number of seeds.

The optimization ends when the selected termination criteria are met and the final result is then evaluated through the chosen performance indicators. All these options the user can select in this tab will be refered to as Run Options. Additionally, the optimization process through all these options will be refered as a Run.

Each column (which is, in fact, a table widget with only one column), except for the number of seeds, is made up of a series of combo box widgets, of the class *MyComboBox* coded in *small_widgets.py*.

MainTabsWidget is instantiated with two Python dictionaries: *run_options* and *parameters*. The *parameters* dictionary (exemplified later in Code Snippet 4) is analyzed to see what are all the options available regarding the problems, algorithms, performance indicators and terminations.

Consequently, the combo boxes are instantiated with their initial items and respective table, setting them to be able to add or remove rows in all but the termination table below the "Termination" header of Figure 3.6, as it is set to have the same termination criteria for all combinations.

The class also has two methods, *dictToTables* and *tablesToDict*. They are used to convert the table widgets into a dictionary or to set the dictionary values into the combo boxes. Each key is the class type (algorithms, problems, etc.), and the values are strings with the chosen IDs at each column. For

example, the options of Figure 3.6 would be translated into the dictionary presented in Code Snippet 2.

Code Snippet 2 Python dictionary matching options in Figure 3.6.

```
run_options_soo = {
    'problem': ['ackley', 'griewank', 'rastrigin'],
    'algorithm': ['ga', 'pso'],
    'indicator': ['best', 'avg_fitness', 'evals_on_goal']
    'termination': ['n_eval'],
    'seeds': 3
}
```

The most important button in this widget is the "Run" button, associated with the *runButton* method. First, it calls the *getRunThread* method, which tries to instantiate a *RunThread* object. To do that, it loops through the tables: Problems→Algorithms→Perf. Indicators and Terminations. At each iteration, it gives specific arguments that can be used to instantiate the next object type with them.

For example, certain algorithms need the number of variables in the problem to be instantiated. If the same algorithm is supposed to solve two problems with different number of variables, it has to be instantiated at least twice with those different values. In case one of the objects cannot be instantiated, the exception is shown to the user, and the run is aborted.

All these specific arguments that can be used to customize the same Run Option into different instantiations of the same class are detailed later in Table 3.3 of the Edit Window section, but the general idea of how this is made possible is shown in Code Snippet 3.

Code Snippet 3 Loops in *getRunThread* method to instantiate the objects.

```
# PROBLEMS
for prob_id in prob_ids:
    convert_dict = {}
    prob_object = tabs[PROB_KEY].getObjectFromID(prob_id, convert_dict)
    if isinstance(prob_object, Exception):
        return None

    # get specific args from problem for next instantiations
    n_obj = prob_object.n_obj if prob_object.n_obj else None
    n_var = prob_object.n_var if prob_object.n_var else None
    kwargs = {'n_obj':n_obj,'n_var':n_var,'prob_id':prob_id,
              'prob_object':prob_object}
    convert_dict.update(kwargs)

    # ALGOS (can use n_var, n_obj, prob_id as args)
    for algo_id in algo_ids:
        algo_object = tabs[ALGO_KEY].getObjectFromID(algo_id, convert_dict)
        if isinstance(algo_object, Exception):
            return None
        [...]
```

By permitting one object to modify the subsequent one, the application moves closer to achieving **requirement 6**, as this feature can be used to make custom algorithm initializations, behaving differently according to certain external variables.

If all objects are instantiated for the chosen options, a *historyFrame* widget (detailed in the next section) is created, establishing the chain that will lead to the optimization process itself.

More actions are available through the tab buttons that call their respective methods, namely:

- **Edit Parameters**. Calls the Edit Window to edit the parameters for each of the IDs chosen in the combo boxes.
- **Save Run Options**. The Run Options are turned into a dictionary through the *tableToDict* method. After that, it is saved in a pickle file as a serialized Python object. In this way, they can be loaded later, allowing the user to seamlessly continue the work.
- **Load Run Options**. Loads a pickle file with a dictionary containing previously saved Run Options, and inserts them into the combo boxes through the *dictToTable* method. The loading process makes sure that there are no invalid IDs that will call objects not represented in the Edit Window, making the app more robust. This option helps to meet **requirement 6**, as it makes it easy to return to previously selected Run Options, avoiding a repetitive task each time the app starts.
- **Random/Fixed Seeds**. Allows the user to choose between fixed seeds, making the run reproducible, or random ones, so a different run is guaranteed if there are probabilistic elements in it, even if the Run Options are the same.

Run History Tab

In this tab, shown in Figure 3.7, the history of the Runs is displayed. As mentioned above, when the "Run" button is pressed and a *RunThread* instantiated, a *historyFrame* widget (highlighted in the figure) is added, which is also coded in the *fixed_tabs.py* file.

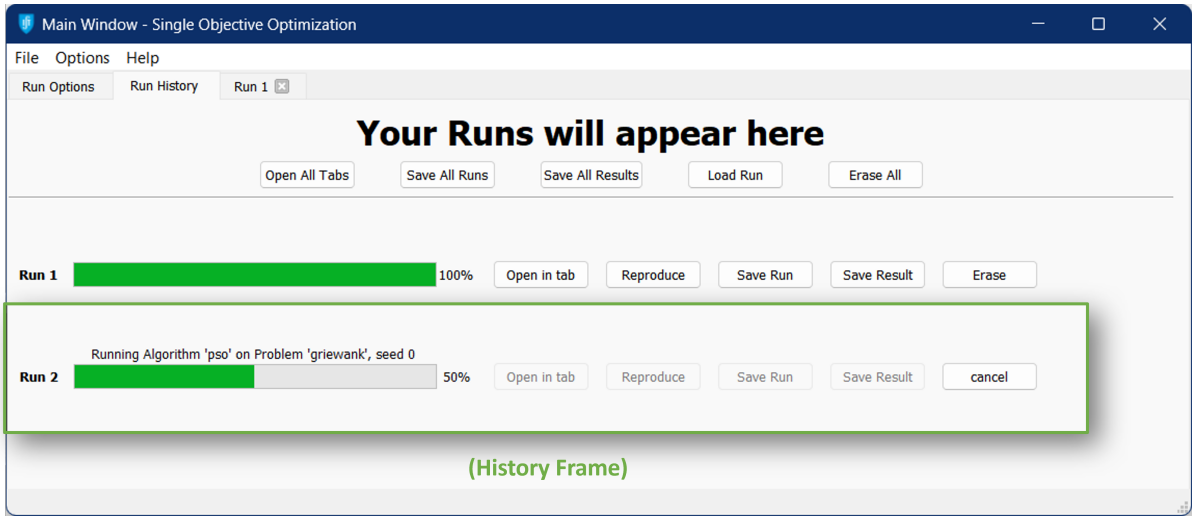


Figure 3.7: "Run History" tab with two History Frames.

This frame will then create and store a *RunThread* as its attribute, detailed in the backend section. This will allow multiple Runs to be set in motion by the user simultaneously, which will help to meet

requirement 3. A progress bar is displayed, detailing the progress on the Run and allowing the user to cancel it. Once completed, a specific "Run" tab (detailed in the next section) is created, and the following buttons are made available, calling their respective methods.

- **Open in tab**. Opens the created tab where the user can see the results of the Run.
- **Reproduce**. Returns the application to the state it was in right before the "Run" button was clicked. The Run Options and Parameters are set as the same as the Run. This makes it possible for the user to reproduce it by simply pressing the "Run" button again.
- **Save Run**. Transforms the Run Object into a Python dictionary and saves it as a pickle file so that it can be loaded into the app later.
- **Save Result**. Takes the result of the Run, presented as a table with the values of the performance indicators on each generation, for every problem-algorithm-seed combination, and saves it after changing to the *.csv* file format.
- **Erase**. Erases the History Frame and the corresponding data with it.

In addition, the tab presents a "Load Run" button at the top, which can load a previously saved Run. The loading process ensures that the pickle file has the appropriate format, so it can be used to instantiate a *RunThread* object, making the application more robust.

The app limits the loaded Runs to 6 to prevent a poorer performance due to an extensive use of memory. It asks the user to erase a Run from history before loading or setting another. As was mentioned, it is possible to save the Run before erasing it from the app's memory and loading it again, so that the work is not lost.

Run Tab

As can be seen in Figure 3.8, the tab is created with the respective Run number. It presents the user with two options to analyze the results: a table with the results for their chosen options displayed, or a window with a plot for a specific type.

After the *RunThread* is finished, the results are stored in a Pandas [44] *Dataframe* object with the fixed columns: *seed*, *problem*, *algorithm*, *n_eval*, *n_gen*, and one more for each performance indicator. This *Dataframe* (shown later in Table 3.4) is rather extensive, as it contains the values of each generation for each problem-algorithm combination.

The table that the user will see will only display results concerning the final generations of each problem-algorithm combination. Another *Dataframe* is created and stored as the *stats_seeds_df* attribute, referring only to these final values. For each performance indicator, the best, median, worst, and average values are calculated. Because it will later be relevant to know what the specific seed was that led to the best, worst, and median cases, it is also stored in the new *Dataframe*. An example of the final structure can be seen in Table 3.1.

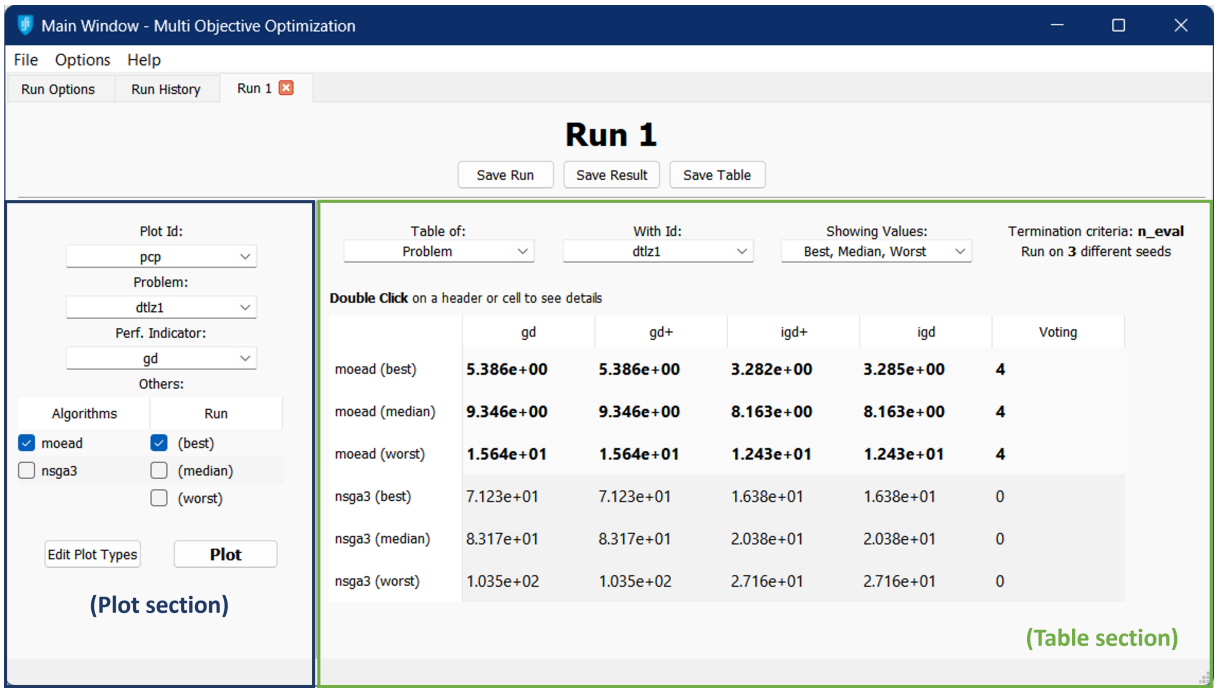


Figure 3.8: Run Tab.

Table 3.1: Structure of *stats_seeds_df*.

problem	algorithm	igd								[...]
		best		median		worst		average		
		(value)	(seed)	(value)	(seed)	(value)	(seed)	(value)	(seed)	
dtlz1	moead	6.62	2	10.81	5	9.04	1	8.83	NaN	
[...]										

The user can select the values to be presented for a specific problem or performance indicator. Furthermore, they can decide to see either the average value across seeds or the worst, median, and best values. The table that will display these results is obtained using grouping and slicing methods from the Pandas library in the original *stats_seeds_df Dataframe*. An additional voting column is constructed counting the number of times a given algorithm performed better than the others. The table can be saved in a *.csv* file through the “Save Table” button.

These functionalities serve **requirement 6**, both on the customization side, since the table can be displayed in different ways according to the user preferences, but also on its ease-to-use side, allowing all these different indicators to be displayed without further effort.

Other functionalities of this tab include the following buttons or clicks that call their respective methods.

- **Save Result**. Saves all the values from the original *Dataframe* given by the *RunThread* in a *.csv* file.
- **Double click on a table header**. Given the ID of the problem, algorithm, or performance indicator,

it presents the arguments with which it was instantiated, so that the user can quickly remember what they were.

- **Double click on a table cell**. Each value (except for the average) is associated with a specific run. Clicking on it will display the value of the correspondent seed.

- **Edit Plot Types**. Button in the plot section highlighted in Figure 3.8. It calls the the tab in the Edit Window where the user can change the initialization arguments for each plot type.

- **Plot**. Also in the plot section highlighted in Figure 3.8. The button calls a *QPlot* class, defined in *plotting.py*. Passes as initializing arguments the chosen problem, performance indicator, algorithms, and run type. The details of this class will be laid out in the next section.

QPlot

This class, coded in *plotting.py*, is mainly concerned with **requirement 4** - adequate visualization techniques. It provides a parent class that servers as a guide to implement specific visualization techniques to allow the user to have adequate ways to understand the results in single and multi-objective optimization. As it will later be detailed, the *RunThread* stores not only the results of the performance indicator at each generation of the population, but also the final population values in the search and objective spaces.

The user selects the performance indicators, algorithms, and run (best, median, or worst), to plot regarding a certain problem and plot type. To find the population of a specific run, the *stats_seeds_df* is used to identify the seed corresponding to the specific problem-algorithm-run combination. With the combination problem-algorithm-seed, it is now possible to obtain the respective population stored in *RunThread* object.

To develop a specific plot type, a child class should be developed, implementing the abstract method *createCanvas* that should return a Matplotlib [45] *FigureCanvasQTAgg*. Depending on the chosen plot type, the respective child class of *QPlot* is called, among the following.

- **QProgress**. This plot type is used in single and multi-objective optimization. It is helpful to understand how the algorithm came to a certain final value. To do this, it first slices the *stats_seeds_df* to the chosen problem, algorithms, and PI. For each algorithm, it calculates the average and standard deviation across seeds for the PI value at each generation. The plot is then made with a thick line representing the average value, and the shadow painting corresponds to the standard deviation (Figure 3.9(a)).

- **QFitnessLandscape**. Only appropriate for SOOPs with one or two dimensions in the search space. Based on pymoo's *FitnessLandscape* class, is used to simultaneously visualize the search and objective spaces. The search space is represented in the horizontal plane, and the objective space is plotted on the vertical axis. The solutions found by the selected algorithms are naturally located on the fitness surface, plotted as spheres (Figure 3.9(b)).

- **QPCP**. Parallel coordinates is a powerful plot type to analyze multi and many-objective optimization, also based on a pymoo's class *PCP*. Visually represents the trade-offs between different objectives (Figure 3.9(c)). It can plot the problem Pareto front sampled at the given reference directions, and the algorithms final generation for a given run (best, median, worst). Each objective is plotted on a separate vertical axis, all axes are parallel to each other, and the range of each objective is normalized. Solutions are represented as lines connecting points on these axes, where each point on a line corresponds to the performance of that solution with respect to one of the objectives. This allows for easy comparison of multiple solutions and understanding how changes in one objective affect the others.

- **QParetoSets**. For two or three objective problems, the Pareto sets can be plotted as a 2D or 3D graph, with each axis corresponding to one objective (Figure 3.9(d)). The selected problem Pareto front is sampled with regard to the optimal solutions corresponding to the given reference points. The final generation of the selected algorithm run (best, media, worst) can also be plotted in this objective space.

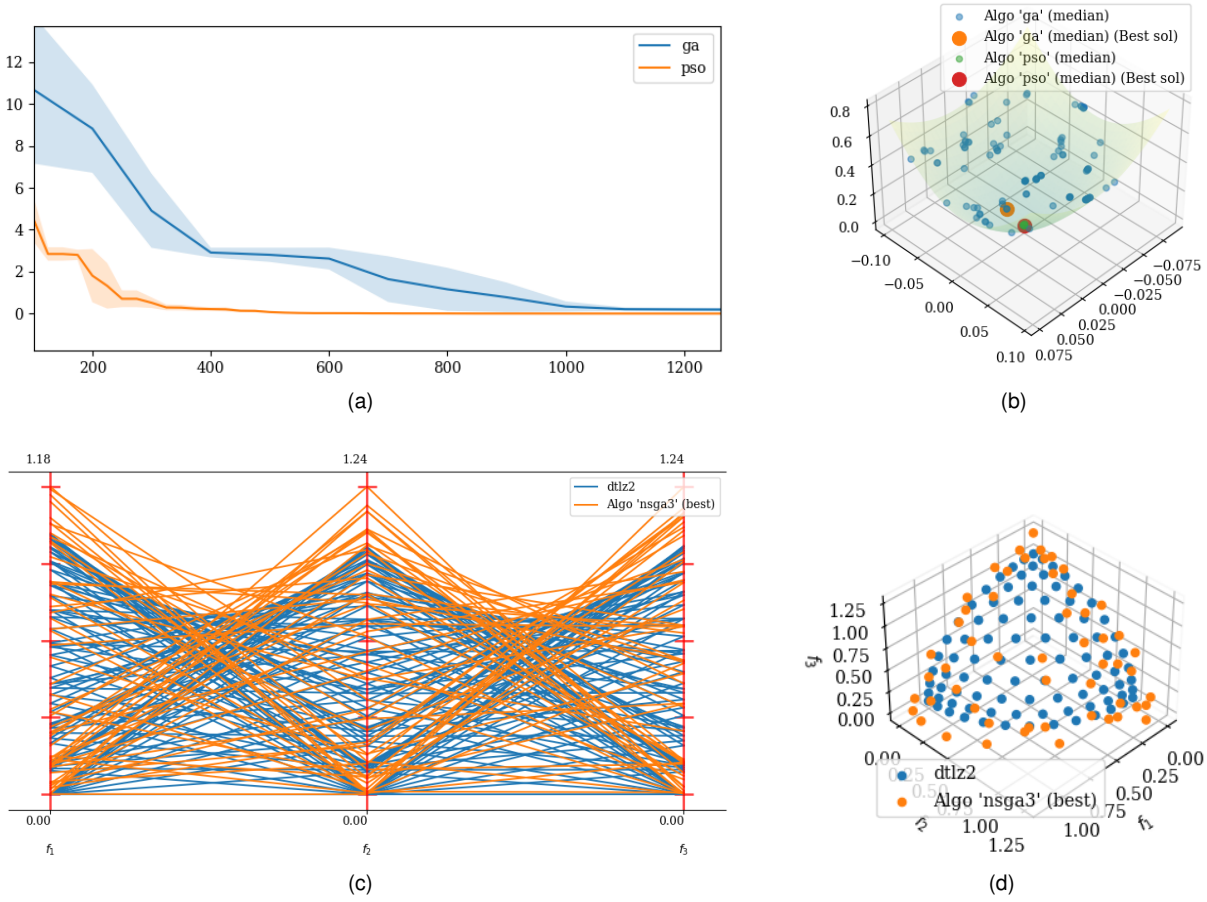


Figure 3.9: Plotting types available in the application: Progress (a), Fitness Landscape (b), Parallel Coordinates (c) and Pareto Sets (d).

3.2.3 Edit Window

While the main window provides the graphical interface to select the options of the run, there is still the need to give the user an option to not only change the parameters that will be used in each problem, algorithm, performance indicator, termination criterion and plots, but also create new ones. This is an essential part of **requirement 6**, making the optimization process as customizable as possible through the app without losing its ease of use.

For this purpose, the Edit Window was developed. It is coded in the file *edit_window.py* as the class *EditWindow*. The user can find it by selecting "Edit Parameters" in the Main Window Menu Bar or by clicking on the Edit Parameters button of the "Run Options" tab. The window on top of Figure 3.10 will then be displayed.

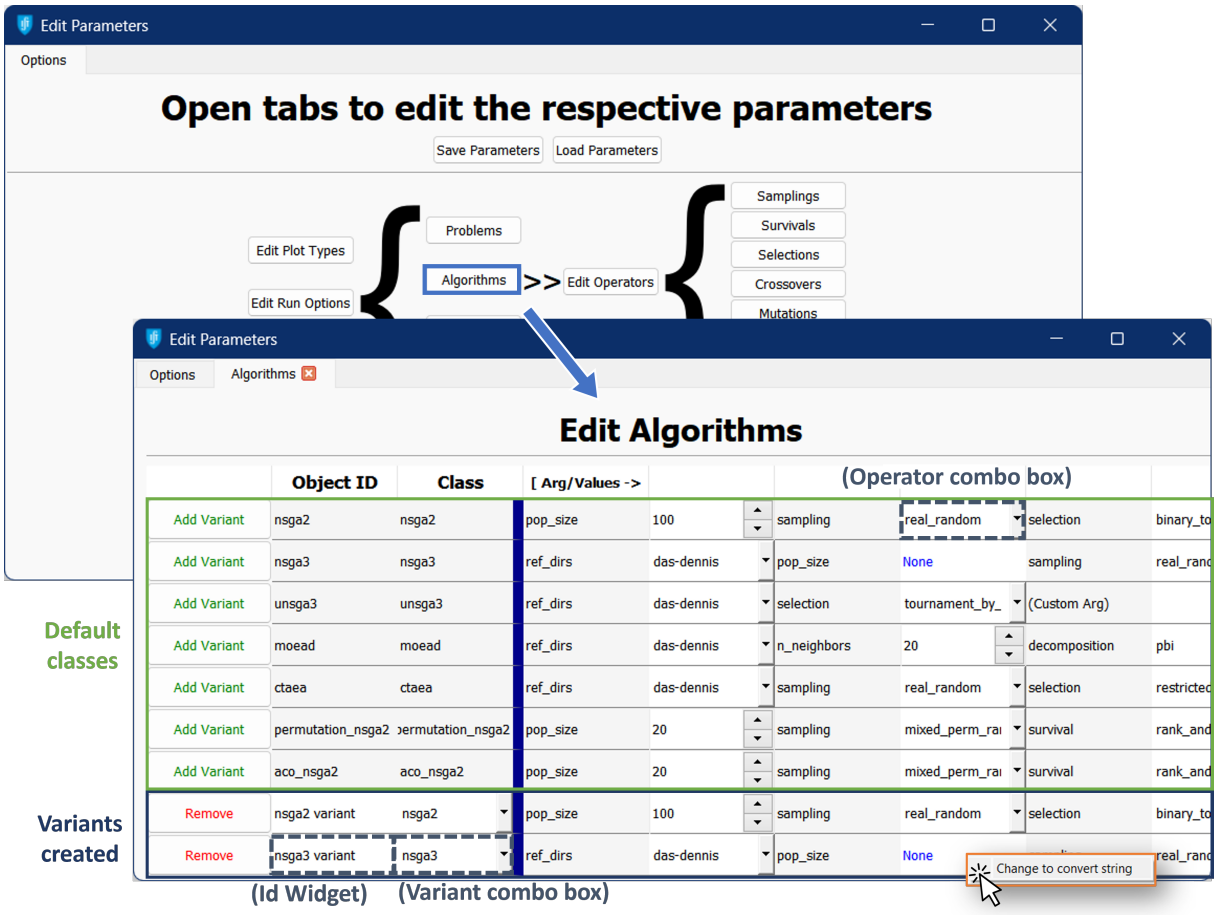


Figure 3.10: Edit Window.

This class is instantiated twice, one for SOOPs and the other for MOOPs. As with *MainWindow*, it needs a dictionary as an argument, called *parameters*, to be instantiated. This dictionary must have a structure as that shown in Code Snippet 4.

A tab of class *EditTab* is created for each first-level key in the dictionary using the *dictToTabs* method, and the respective second-level dictionary is passed as an argument. The reverse is also possible through the *tabsToDict* method, which saves the information of the tabs in a dictionary and saves it in a

.pickle file. This dictionary can later be loaded through the button “Load Parameters” using the already mentioned *dictToTabs* method.

Code Snippet 4 *parameters* dictionary structure.

```
parameters = {
    # tab level
    'problems': {
        # rows of the table inside the tab
        'prob1': {'class': 'prob1', 'arg1': 0.2, 'arg2': True},
        'prob1_variant': {'class': 'prob1', 'arg1': 0.3, 'arg2': False}
        [...]},
    'algorithms': {
        'algo1': {'class': 'algo1', 'arg1': '1/n_var(convert)'},
        'algo2': {'class': 'algo2', 'arg1': 'rnd_sampling', 'arg2': 14}
        [...]}
    [...]}
```

In the fixed tab “Options”, the user can open the tab with the plot classes mentioned in the previous section, or any of the specific types of class present in the Run Options: problems, algorithms, performance indicators, and terminations.

Furthermore, all Genetic Algorithm operators (Figure 2.4) have their own dedicated tab: sampling, survival, selection, crossover, mutation. The additional components often used in MOO, decomposition and reference directions, are also present.

When any of these buttons is pressed, the respective tab opens, as the algorithm tab shown at the bottom of Figure 3.10. The structure of this tab is detailed in the next section.

Edit Tab

As mentioned above, the tab receives as an initialization argument the respective second-level dictionary of *parameters*. Now, this dictionary is converted to the table shown in Figure 3.10 with its various components highlighted in different colors, by the following process.

If the dictionary key, which will represent the ID of the class to be instantiated, is equal to the class name (the string associated with the ‘class’ key in the dictionary - Code Snippet 4), it is considered a default (highlighted in green). Else, it will be considered a variant of the correspondent class (highlighted in dark blue).

The difference between the default and variant rows is that the first one cannot be erased from the table, and through the “Add Variant” button, the user can create another row copying the arguments from the default. The variant ID widget (highlighted in a dotted gray box), of class *MyLineEdit* coded in *small_widgets.py*, guarantees that there are no repeated IDs (adding (1), (2)... if the name is not unique).

On the other hand, variant rows can have their class changed between the default ones through the variant class combo box (highlighted in a dotted gray box), and their row can be erased through the “Remove button”. These buttons use the ability of the class *MyComboBox*, the same as that used in the Main Window “Run Options” tab, to copy and remove its row of the table.

The rest of the arguments regarding a dictionary key, which correspond to the initialization arguments of the respective class, are represented by different widgets depending on their type. All these widgets are also coded in *small_widgets.py*. Table 3.2 presents a description of all the widgets in that file and their possible utilizations in the app.

Table 3.2: Widgets in small_widgets.py and their respective utilizations in the app.

Widget	Used In	Description
ScientificSpinBox	- Int Argument	Spinbox adapted to allow scientific notation so large integer numbers can be represented
ScientificDoubleSpinBox	- Float Argument	Spinbox adapted to allow scientific notation and rational numbers
MyCheckBox	- Boolean Argument	A simple check box with TRUE or FALSE boolean values
MyTextEdit	- Variant and Default IDs - Argument Name - String/None Argument	Widget that allows free writing. If it is a Variant ID widget, when is out of focus guarantees that is unique and sends the signal to the respective combo boxes. Can be a convertible string or not, depending on user's choice.
MyComboBox	- Operator Argument - Run Options (Main Window)	Used when an algorithm has an argument with an operator name, and in the Run Options tables. Receives signals from the ID widgets to update items.

A signal, a PyQt event used for communication between objects, with the name *itemsSignal*, is emitted from the ID widgets whenever an ID is changed. The signal is propagated by the Edit Window through its own signals *runOptionsUpdates* and *operatorUpdates*.

These signals are respectively connected to the combo boxes from the "Run Options" tab of the Main Window (with the exception of the "Plot Type" combo box which is present in the *RunTab*), or the operator combo boxes (such as the one highlighted in a dotted gray box) among the *EditTabs*, depending on their respective class. When the signal is received, the items are updated to include the new ID. This guarantees that the items in the combo boxes are always up-to-date, preventing errors trying to get unexistent IDs from a table.

It is also worth noting that all widget styles are copied from a single reference frame made in the designer tool, *widgets_frame.ui* (Figure 3.11). This is made to allow for rapid changes in the appearance of the app without having to resort to code.

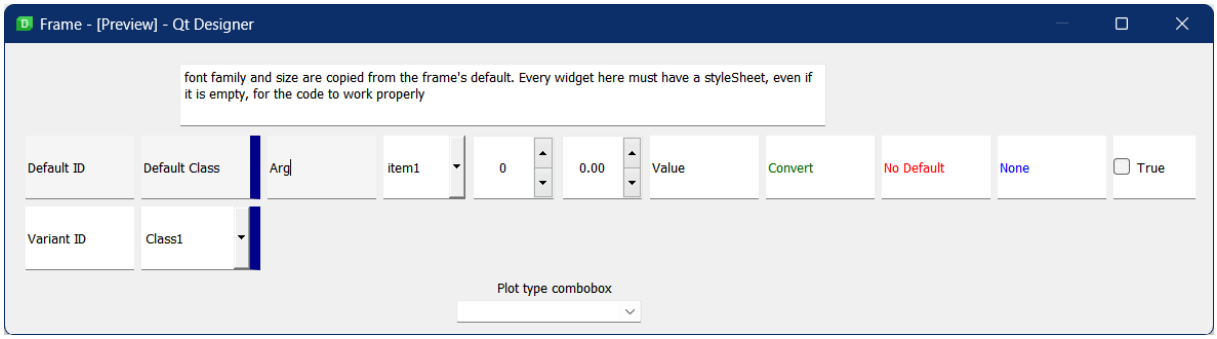


Figure 3.11: Widgets frame.

To help meet **requirement 6**, another feature was added to the application, allowing greater customization of the parameters. when right clicked, the *MyTextEdit* widget allows the user to change between a "plain string" or a "convert string" (process highlighted in orange at the bottom of Figure 3.10),

saving it as the boolean attribute *convert*.

The first option will assume the value of the argument to be just the sequence of characters, while the second one will try to substitute the keywords present in Table 3.3 for their respective values and evaluate the mathematical expression if possible. The respective substitution values are collected through the process already explained in Code Snippet 3.

Table 3.3: Keywords for *MyTextEdit* set to convert the string and their substitution.

Keyword	Substitution	Can be used to instantiate		
		Algorithms	Terminations	Perf. Ind.
None	python's "None" null value	✓	✓	✓
n_var	number of variables in the problem to be optimized	✓	✓	✓
n_obj	number of objectives in the problem to be optimized	✓	✓	✓
prob_id	id of the problem	✓	✓	✓
prob_object	problem object	✓	✓	✓
prob_pf	PF of the problem. If available, it is calculated with the reference dir. of the algorithm	✓	✓	✓
algo_id	id of the algorithm		✓	✓
algo_object	algorithm object		✓	✓
term_id	id of the termination			✓
term_object	termination object			✓

When converting the parameters into a dictionary, if *convert* is true, the additional characters '(convert)' are added to the end of the string, as seen in the dictionary of Code Snippet 4. Then, when the parameters are loaded back, if a string ends with that sequence, it takes it out and sets *convert* to true in the respective widget.

A last costumizable argument is added at the end of each table row, which can be set by the user to any string. In this way, if a specific argument the user wants to set that is not already named, for example, from the parent class of the row's class, it can be set in this way.

3.3 Backend

This module contains the code that provides the logic behind the app. Being an application for optimization, this will naturally be the main focus for the backend module. Figure 3.12 provides a closer look at its main components. It is divided into three files - *get.py*, *defaults.py*, and *run.py* which will be detailed in the following sections with the corresponding names.

3.3.1 Get

As seen previously, the GUI allows the user to choose between various Run Options, presented as unique strings. These strings are connected to a row in the respective Edit Window table, which in turn explicitly states the class and argument values for its initialization.

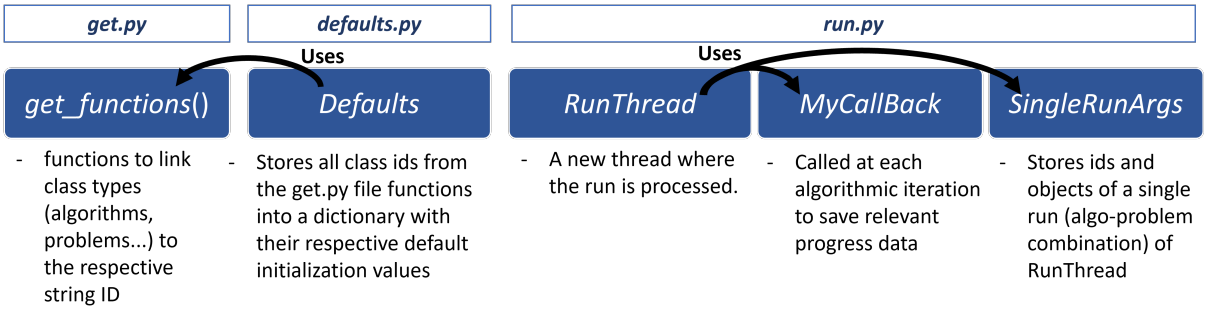


Figure 3.12: Backend components.

The *get.py* file contains multiple *get* functions (*get_algorithm*, *get_problem*, etc.). The code, shown in part in Code Snippet 5, is based on a file that existed in pymoo version 0.4.2, called *factory.py*.

Code Snippet 5 *get_algorithm* function from *get.py*.

```
def get_algorithm(name, *args, **kwargs):
    from pymoo.algorithms.moo.nsga2 import NSGA2
    [...]
    ALGORITHMS_SINGLE = {
        'ga': GA,
        [...]
    }
    ALGORITHMS_MULTI = {
        'nsga2': NSGA2,
        [...]
    }
    return returnObjectOrOptions(name, ALGORITHMS_SINGLE,
                                 ALGORITHMS_MULTI, *args, **kwargs)
```

These functions contain a dictionary for SOO options and another for MOO options. The dictionaries link a unique string to a pymoo class of the respective type. All available options are shown in Appendix B, including more than 100 problems, 14 algorithms, and 40 different operators.

The function requires a “name” argument that will be passed into the *returnObjectOrOptions* function. This function returns the SOO, MOO or joined dictionary, in case “name” is “soo_options”‘, “moo_options” or “all_options”, respectively. Otherwise, it will assume the “name” is a key in the dictionary, and it will instantiate the associated class with the given arguments.

Through these functions, the class strings presented in the app can be used to choose the class of the object to be instantiated. The values of the arguments used in the initialization are converted from the widgets of the respective row of the Edit Window.

3.3.2 Defaults

Having in mind **requirement 6** - the app’s customization and ease of use - when the user starts the app, it is necessary to provide him with ready-to-use algorithms, problems, and terminations. Furthermore, the respective arguments names and their default values should be presented, so as not to

require prior knowledge to know them in order to start the optimization process.

Later, these argument-value pairs, presented in the GUI as widgets of different types, can be easily changed for a different initialization of the objects. In this way, the user can not only quickly test multiple algorithms as their default values are already provided, but can also change the values through the app.

To allow all options to be readily available at the user's disposal, the *Defaults* class defined in this file is created. It can be instantiated to SOO or MOO. Then, it goes through the respective options provided by the *get* functions of problems, algorithms, terminations, and operators in the *get.py* file and stores their respective arguments with the default values in the *parameters* dictionary, one of the class attributes. This is done using its method *get_class_dict*, detailed in Code Snippet 6.

Code Snippet 6 Part of *get_class_dict* function of *defaults.py*.

```
def get_class_dict(self, get_name: str, cls: type):
    sig = inspect.signature(cls.__init__)
    # get all the arguments into a dictionary
    args_dict = {arg: par.default for arg, par in sig.parameters.items()}
    ret_dict = {CLASS_KEY: get_name}  # add an entry with the class string

    # loop to filter 'bad' arguments and get operators
    for arg, value in args_dict.items():
        if arg in OPERATORS:
            ret_dict[arg] = self.getOperator(arg, value)
        elif type(value) in ALLOWED_VALUE_TYPES:
            ret_dict[arg] = value
        [...]

    return ret_dict
```

There are some difficulties with the process of getting the arguments with the default values that are worth noting. What should be done when one of the arguments requires another object as its value? In the case of an argument with a value of string, int, float, or boolean type, the user can simply change it directly through the respective GUI widget. However, there is no direct way to allow the user to choose different objects for an argument value.

The assumption could be made that if one of the arguments required value is an object, then always initialize it with the same default object and do not list it in the app, making it impossible for the user to change it.

However, this would impose a serious limitation on operator customization, restricting the implementation of **requirements 6**. As the app should allow for testing of algorithms with multiple operators, the following workaround is used in this case. The *Defaults* class first stores the different operators. After the algorithms are stored, the names of the arguments are verified. If it matches one of the operators keywords (*sampling*, *survival*, *selection*, *crossover*, *mutation*, *decomposition*, and *ref_dirs*), substitute the value for the corresponding operator string associated with that object class. This process is done through the *getOperator* method called in Code Snippet 5.

The same is done in the frontend when instantiating an algorithm object. If it contains an operator as an argument, it first gets the correspondent object from the class string, and then the algorithm object

itself is initialized with it. This makes it possible for the user to easily experiment with multiple operators by simply selecting from the available ones in the respective combo box.

It is important to notice that the operator match is made only at the class level, so even if the algorithm specifies a different default initialization (Code Snippet 7), it is not taken into account. The user should create a variant for the operator and select it from the combo box instead.

Code Snippet 7 NSGA2 initialization in pymoo framework.

```
# nsga2 will be initialized with the default SBX values, not the
# ones selected here. Need to change them in the app.
class NSGA2(GeneticAlgorithm):
    def __init__(self,
        crossover=SBX(eta=15, prob=0.9),
        [...]
```

Another exception worth noting is the reference directions operator, used in algorithms such as NSGA-III. When testing on multiple problems, this operator needs to be adjusted for each one, as it depends on the number of objectives to determine the dimensions of it.

To allow the user to run the algorithm on multiple problems at once, the default values for the parameter *n_dim* is set to *n_obj*, which will later be replaced by the number of objectives of the respective problem, as mentioned in Table 3.3.

Lastly, knowing that some of the performance indicators available in the framework require the Pareto front of the problem to be calculated, the parameter *pf* is hard coded to the value *prob_pf*, which will later be changed to the Pareto front of the respective problem. This is done to tackle the same issue of reference directions. In this way, a user can simply define the same parameter for all performance indicators and it will automatically change to correctly calculate it for each problem.

3.3.3 Run

In this file, the *RunThread* class is defined. It is a child of the *QThread* class, so the optimization running in the background does not interfere with the graphical interface of the app. In addition, it makes it possible to perform multiple optimizations, since different *RunThread* objects are created when the user clicks the “Run” button. This is important to meet **requirement 3**. It makes it possible not only to run multiple algorithms on multiple problems with just one click but also to start various disassociated runs without having to wait for one to end to start the other.

As seen in the previous section, even if the optimization is done by the same algorithm, it can be necessary to have different instantiations for different problems, as the initialization arguments can vary according to it. The same happens with the performance indicators.

For this reason, *RunThread* receives as input a list with *RunArgs* objects, each containing the problem, algorithm, performance indicator, and termination instantiated for a single problem-algorithm combination.

The optimization is then performed for every combination and with all the different seeds using the *run* method, shown in Code Snippet 8. If the user chooses fixed seeds, they are the same each time;

otherwise, they are peaked at random. This can be seen in Code Snippet 8

Code Snippet 8 *run* method of *RunThread*.

```
def run(self):
    if self.fixed_seeds:
        seeds = np.arange(self.n_seeds)
    else:
        seeds = np.random.choice(100000, size=self.n_seeds, replace=False)

    for run_args in self.run_args_list:
        for seed in seeds:
            if self.canceled:
                return
            self.progressUpdate(run_args.algo_id, run_args.prob_id, seed)
            res = self.singleRun(run_args, seed)
            self.updateData(run_args, res, seed, res.algorithm.callback)
```

At the beginning of each single run (problem-algorithm-seed combination), a signal is emitted (*progressUpdate* method) with a string and a percentage to inform the optimization progress or if an error occurs.

During its execution, progress is saved through the object *MyCallBack*, that stores the necessary information. After a single run, it checks to see if the user cancels it through the GUI. If not, *data*, a *Dataframe* attribute of *RunThread*, is updated by the *updateData* method shown in Code Snippet 9, as mentioned in the QPlot section. This attribute, later used in various plots or to save the results of the run as a *.csv* file, has the structure shown in Table 3.4.

Code Snippet 9 *updateData* method of *RunThread* called after each single run.

```
def updateData(self, run_args: RunArgs, res: Result, [...]):
    [...]
    if self.data.empty:
        self.data = pd.DataFrame(single_run_data)
    else:
        self.data = pd.concat([self.data, pd.DataFrame(single_run_data)])

    if self.moo:
        feas = np.where(res.algorithm.opt.get('feasible'))[0]
        F = res.algorithm.opt.get('F')[feas] if len(feas) > 0 else np.nan
        X = res.algorithm.opt.get('X')[feas] if len(feas) > 0 else np.nan
    else:
        feas = np.where(res.algorithm.pop.get('feasible'))[0]
        F = res.algorithm.pop.get('F')[feas] if len(feas) > 0 else np.nan
        X = res.algorithm.pop.get('X')[feas] if len(feas) > 0 else np.nan

    key = (run_args.prob_id, run_args.algo_id, seed)
    self.best_gen[key] = {}
    self.best_gen[key]['X'] = X
    self.best_gen[key]['F'] = F
```

Additionally, a set of feasible solutions is stored in the Python dictionary best_gen, also a *RunThread* attribute. If the optimization concerns SOOPs, it will store the search and objective space values of

the population in the last generation, in the 'X' and 'F' entries of the dictionary, respectively. If the optimization concerns MOOPs, the dictionary will store the search and objective space values referring to the best Pareto set found throughout the optimization process, also depicted in Code Snippet 9.

Table 3.4: Structure of *data*.

problem	algorithm	seed	n_evals	n_gen	igd	igd+	[...]
dtlz1	moead	2	100	5	8.83	8.92	
[...]							

3.4 Other Aspects

3.4.1 Frontend-Backend Interaction

Now that the two main modules have been detailed, it is important to better understand and make explicit the ways in which they are connected to each other. It is important to have clear and concise communication between them if the code is to be easily extensible, as **requirement 5** commands. With this in mind, there are only two processes that require interaction between them: when the application is started and when the "Run" button is clicked.

In Figure 3.13, the process that occurs when the app is started is detailed.

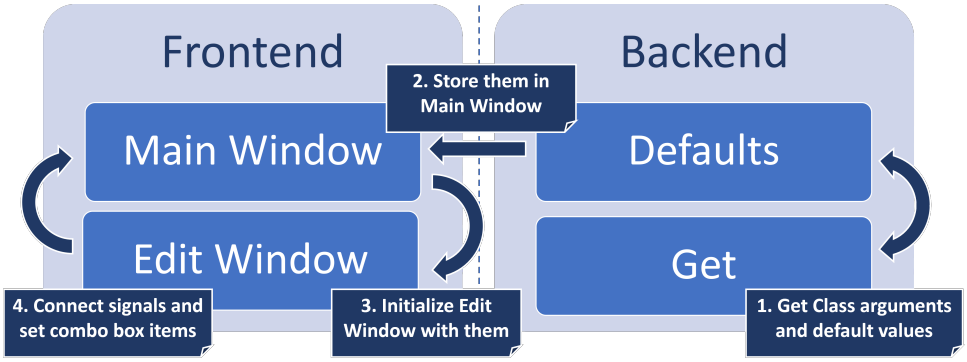


Figure 3.13: App start process.

As it is shown in Code Snippet 10, one of the arguments for the app initialization is set as the *Defaults* class attribute *parameters*. This means that the first step requires the acquisition of them from the options available in the *get.py* file, as mentioned before.

Code Snippet 10 *MainWindow* initialization.

```
class MainWindow(QMainWindow):
    [...]
    def __init__(self, run_options_soo = {}, run_options_moo = {},
                 parameters_soo = Defaults(moo=False).parameters,
                 parameters_moo = Defaults(moo=True).parameters)
```

After this, they are used by the Main Window to initialize the Edit Window, where the tables for each

type (problem, algorithm, etc.) are set in the respective tabs. The signals from the *MyTextEdit* widgets that contain the IDs are then connected to the respective combo boxes of the Main Window, and their initial items set.

In relation to the frontend-backend interaction, the process set in motion when the "Run" button is clicked is shown in Figure 3.14. The first step requires the selected ID from each combo box to be

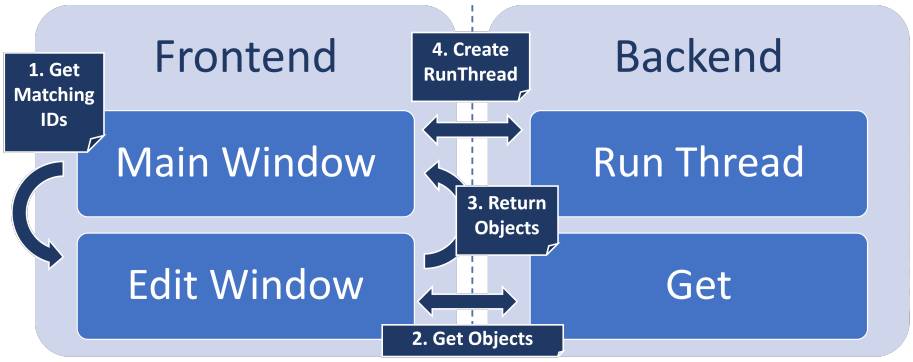


Figure 3.14: "Run" button process.

matched with the respective Edit Window table entry. Thereafter, the string that matches the class from file *get.py* is retrieved and the arguments' values converted into their respective types through the process described in Section 3.3.1. With the objects returned, Main Window proceeds to create *RunThread*, which is linked through a signal that emits the progress of the Run.

3.4.2 Testing

The *Test* class, shown in Code Snippet 11, is defined in file tests_declaration.py within *tests* folder. It takes as input a dictionary of Run Options such as the one in Code Snippet 2.

Code Snippet 11 *Test* class.

```
class Test():
    def __init__(self, options: dict, [...]):
        [...]
        # initialize main window with test options
        self.main_window = MainWindow(run_opt_soo, run_opt_moo,
                                       param_soo, param_moo)
        # get RunThread
        self.run_thread = self.main_window.activeTabs().getRunThread()
        self.run_thread.finished.connect(self.afterRun)

    def run(self):
        self.run_thread.start()

    def afterRun(self):
        data = self.run_thread.data
        data.to_csv(RESULTS_FOLDER + '/' + self.test_name, index=False)
```

Then, making use of the flexible Main Window initializing arguments, sets its combo boxes to the values in the dictionary. Afterward, it simulates a "Run" button click to set the optimization process in

motion for these specific Run Options. The finished signal from the created *RunThread* is connected to the method of the *Test* class that saves the results in a *.csv* file.

Three tests are declared for single-objective optimization: *soo_probs*, with all the available SOOPs in the app optimized by a single algorithm that handles constrained problems; *soo_algos*, where an unconstrained SOOP is optimized by all the available algorithms in the app; *soo_mixed*, where three algorithms optimize five different problems. Other three tests are created with the same logic for multi-objective optimization: *moo_probs*, *moo_algos*, and *moo_mixed*.

The *main* function of *run_tests.py*, seen in Code Snippet 12, starts all the tests provided in different threads. After waiting for all of them to finish, it compares every result file created with the correspondent benchmark file that contains the expected results obtained from a previous project version.

Code Snippet 12 *run_tests.py*.

```
def main():
    [...]
    # create pyqt app so windows can be instantiated
    app, tests = QApplication([]), []

    for test_options in TESTS_TO_RUN:
        test = Test(test_options)
        tests.append(test)
        test.run()
    [...]
    # compare the CSV files and write result
    for file in files:
        with open(RESULTS_FILE, 'a') as f:
            [...]
            result = ' passed!' if files_are_equal else ' failed!'
            string = date_time + ' Test ' + file + result
            f.write(string)
```

A *results.txt* file is created where it is written if each test passed or failed, depending on whether the file produced with the results of the test is identical to the reference obtained from a previous version of the project. The process can be seen in Figure 3.15, where the block on the right is the one detailed in Figure 3.14.

Whenever the application suffers significant changes during its development process, the tests can be run to ensure that the output remains the same among all these algorithms and problems.

3.4.3 Integrating Custom Classes

As stated in **requirement 5**, the application should be easily extensible. In this way, it can not only facilitate basic level studying of single and multi-objective optimization algorithms, but also allow for a more knowledgeable person to speed up their research.

For this goal, it is essential that user-programmed classes can be seamlessly integrated into it and tested against the existing ones in the pymoo framework. It should come as no surprise that the classes

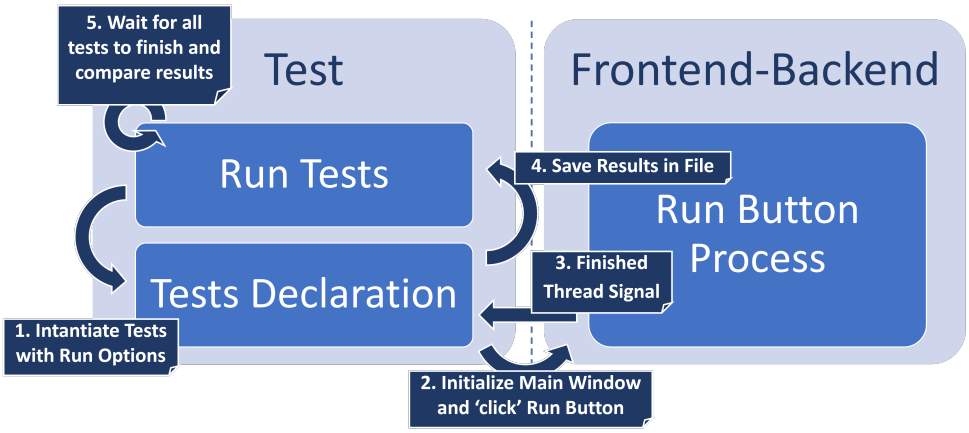


Figure 3.15: Run tests process.

should be developed as children of the parent classes defined in Pymoo, for compatibility purposes. Code Snippet 13) shows the process of adding a user coded *MyAlgorithm* class.

Code Snippet 13 Adding a user-coded algorithm into the app.

```
def get_algorithm(name, *args, **kwargs):
    from pymoo.algorithms.moo.nsga2 import NSGA2
    [...]
    from my_classes import MyAlgorithm

    ALGORITHMS_SINGLE = {
        'ga': GA,
        [...]
        'my_algorithm': MyAlgorithm
    }

    ALGORITHMS_MULTI = {
        ('nsga2', NSGA2),
        [...]
    }

    return returnObjectOrOptions(name, ALGORITHMS_SINGLE,
                                 ALGORITHMS_MULTI, *args, **kwargs)
```

As a general procedure, the following steps must be taken. Locate the type of class (problem, algorithm, termination, mutation, crossover, selection, sampling, decomposition, or reference directions). Import it and add it to the respective *get* function of the *backend/get.py* file, alongside the default id associated with it.

The algorithm is then going to appear in the app, with the default values. It is important to remember that it allows the user to change the values in arguments of type string, boolean, integer, or float, and keywords from Table 3.3. In the case of an algorithm, if one of the arguments is an operator, the app will retrieve the default class of that operator, provided that it is one of the operators in the respective list.

3.4.4 Robustness and Scalability

Besides all the specific requirements stated in Section 1.3, robustness and scalability are two factors that are always important to consider when building software. In light of this, this section defines these terms and explains what measures were taken to achieve them in a sufficient degree for the app's purposes.

Robustness pertains to the ability of a software system to remain stable and functional even when facing unexpected events or errors. It ensures that the system can handle various scenarios and recover gracefully, reducing downtime and increasing reliability.

As this desktop application influence is only on the local machine where it was installed, it has no connections to the internet or other machines. This means that there is no need to protect from "exterior" attacks, as there is no possible way for an outside person to interfere with it. Also, because of its nature of being a tool for optimization, there is no need to have a security set against ill-intentioned users.

With this being said, the main task of the app's architecture regarding robustness is to prevent what can be a well-intended miss-use of the application to crash it. For example, if the user tries to set an argument for a given class with an incompatible value, it should not crash when it tries to instantiate it. This is achieved throughout the app using Python's *try/except* statements, as seen in Code Snippet 14.

Code Snippet 14 *singleRun* method of *RunThread* with try/except statements.

```
def singleRun(self, run_args, seed):
    try:
        res = minimize(algorithm=run_args.algo_object, [...])
    except Exception as e:
        res = None
        self.canceled = True
        error_message = (f'Error while running {run_args.algo_id} on
                         '{run_args.prob_id}, seed {seed}:\n{e}'
                         'Please Make sure the algorithm isnt'
                         'incompatible with the problem.')
        self.progressSignal.emit(error_message, -1)

    return res
```

These are carefully located at the app's most sensible points where user input could break it: trying to optimize a problem through an algorithm (the one shown in Code Snippet 14); trying to instantiate a class with the arguments from the Edit Window; trying to convert a convertible *MyTextEdit* argument; trying to plot a specific problem with the selected plot type. All errors are handed gracefully, with the error message shown in a pop-up window so the user understands what needs to be altered.

In addition, the load points of the app (load Run Options, Parameters, or Run) are set with a verification process to add robustness. The dictionaries imported from the pickle files are checked to see if they have the expected structure (necessary keys, adequate number of levels, etc.) before being used to load the values into the app itself. This is to prevent a dictionary with a different type from being loaded into the wrong place, resulting in the app malfunctioning.

The other important aspect, **Scalability**, refers to the capability of a system to handle increased work-

load and user demand without sacrificing performance. In essence, it enables usage to grow seamlessly while maintaining optimal efficiency.

As the work involves a desktop application, the work load has to be performed on the single local machine. The scalability is therefore limited by the machine's characteristics and performance. There are, however, some techniques used to maximize what the local machine can handle at a given time.

First, the pymoo framework already offers out-of-the-box parallelization of evaluations whenever possible. This is done through a vectorized evaluation using matrices where each row represents a solution. Therefore, a vectorized evaluation refers to a column that includes the variables of all solutions. By using vectors, the objective values of all solutions are calculated at once.

Furthermore, the already mentioned *RunThread* allows the user to start multiple optimization processes at once, being limited by the maximum number of *historyFrame* spaces: seven.

3.5 Conclusion

In this chapter, the architecture of the application was presented. By going through its main modules, the reasoning behind the choices was detailed, pointing to the completion of the requirements outlined in Section 1.3.

One by one, they were addressed until all were met. Starting with **requirement 1**, the building choices ensured the application is cost-free as a consequence of their permissive licenses. Furthermore, they ensured all the application is Python-based, as demanded by **requirement 2**. Pymoo's extensive list of available algorithms and problems helped achieving **requirement 3**, along with the fact that multiple optimizations can start in parallel through the *RunThread*.

Various visualization techniques are provided in *RunTab*, through the *Plotter* class, fulfilling **requirement 4**. Not only is the architecture designed in a modular way, but also, through a simple mechanism, the user can integrate its own classes into the application. This achieves **requirement 5**, as the app becomes easily extensible.

Both parts of **requirement 6** are answered by various design choices. The optimization process is highly costumizable, as the Edit Window allows the user to change the initialization arguments, and even experiment between various operators or use convertible strings to make dependable arguments. On the other hand, the user is presented with multiple choices of algorithms and problems ready to run, with its default values retrieved. Also, the work can be saved and loaded, so repetitive tasks are eliminated and previous work can be continued.

Therefore, in a theoretical realm, it seems the application meets all the requirements at the architecture level. It is now important to access what the practical aspects of potential use-cases look like and see the application "in action" to ensure it is actually helpful to the end user.

Chapter 4

Results

In this chapter, a look at the usefulness of the application is taken to a number of use cases. The installation guide, presented in the *README.md* file of the repository, is shown in Appendix A. As part of the installation process, it is necessary to clone a GitHub repository to a local machine. In this repository, the code related to all the experiments is present in the respective sub-folder of the *thesis* folder, allowing the reader to easily verify the results by running the correspondent scripts or loading the saved pickle files of the respective Run into the app and reproduce it.

Each section will contain an explanation of the relevance of the given experiment, its process, and a discussion of the subsequent results. The objective of this chapter is not primarily to provide new insights or improvements to the single and multi-objective research domain, but rather to prove the methodology of such studies can be assisted by the use of the app.

4.1 App - Framework match

To assert what must be verified at the outset, a recall of the choice for the framework serving as the backbone of the application must be made. The pymoo framework has more than 37 contributors headed by Julian Blank, an applied scientist with more than seven years of experience in optimization.

For this reason, it is only reasonable to assume that the testing done to ensure the correctness of the algorithms, problems, and other features regarding the optimization process will always be better and more extensive in the framework than what one student could try to do for his thesis.

Therefore, it becomes evident what the first task is to ensure that the app obtains reliable results: verify that the results obtained through its use are exactly the same as the ones obtained with the direct use of the optimization framework through a simple Python script.

This is done through the *main.py* script in the *thesis/app_framework* folder. It uses a similar logic to the testing method defined in Section 3.4.2. A *main* function starts all the tests and, after all are over, compares the saved files with the respective expected ones, writing the results in the *results.txt* file.

The already mentioned *Test* class takes a dictionary of Run Options as an argument and instantiates a Main Window with them. It then simulates a “Run” button click, converting the widgets in the Edit

Window into their respective variable types, instantiating the classes, and starting the Run.

A new class *FrameworkTest* is created, shown in Code Snippet 15, which also takes a dictionary of Run Options as an argument.

Code Snippet 15 *FrameworkTest* class.

```
class FrameworkTest(Thread):
    def __init__(self, options: dict, n_seeds:int=1, n_evals:int=500):
        super().__init__()
        [...]
        self.problems = {prob_id: try_get_function(get_problem,prob_id)
                                  for prob_id in self.options[PROB_KEY]}
        self.algos = {algo_id: try_get_function(get_algorithm,algo_id)
                                  for algo_id in self.options[ALGO_KEY]}

    def run(self):
        self.problems = correctProbs(self.problems)
        for prob_id, prob in self.problems.items():
            self.algos = correctAlgos(self.algos, prob.n_obj)
            for algo_id, algo in self.algos.items():
                [...]
                for seed in range(self.options[SEEDS_KEY]):
                    res = minimize(algorithm=algo, [...])
                    self.updateData([...])

        self.data.to_csv(RESULTS_FOLDER + '/' + self.test_name, [...])
    [...]
```

However, this class directly instantiates the objects. It uses the *try_get_funtion*, which catches the exceptions raised, for example, when the arguments are not correctly set. Furthermore, some corrections are made through *correctProbs* and *correctAlgos* in specific cases. For example, the NSGA3 algorithm has no default value for the reference directions, which depend on the number of objectives of the problem. So, the initial *get_algorithm* call without arguments will fail, leaving the *algos['nsga3']* entry with a string indicating the failure to get the object. This is changed through *correctAlgos* by directly instantiating the algorithm with those arguments hard coded.

All the different test options mentioned in Section 3.4.2, which go through all available algorithms and problems in the app, were performed. The results were compared to the respective files generated by the app testing.

As there is no difference between the produced files for all the algorithms and problem classes in the app, it can be stated that the application has no conflicts with what would be expected through the direct use of pymoo.

4.2 Single-Objective Optimization

It is now important to understand how the app could assist and speed up in concrete optimization researches. Regarding single-objective optimization, the PSO algorithm is explored to see in what ways its performance is changed by the different initialization arguments. The study used as a reference is that

of Trelea [46]. It explores optimization in problems such as the Rosenbrock, Rastringin, and Griewank functions.

The methodology for evaluating performance is different from the general multi-objective study case. A certain fitness value is set as the goal, and the algorithm should obtain it in the least number of function evaluations possible. Table 4.1 provides the details of the problems to be optimized.

Table 4.1: Problem options for PSO optimization.

Name	Formula	Range of x	Optimal F	Number of Variables	Goal for F
Rosenbrock	$f_1(\vec{x}) = \sum_{i=1}^{n-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right)$	$[-30, 30]^n$	0	2	0.5
				30	100
Rastringin	$f_2(\vec{x}) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right)$	$[-5.12, 5.12]^n$	0	2	0.5
				30	100
Griewank	$f_3(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$[-600, 600]^n$	0	2	0.01
				30	0.1

As the goal depends on the problem id, there are two options to obtain the results through the app: either start separate Runs with the goal to be achieved explicitly set in the app, or program a personalized class to take the problem id into account when setting it, as in Code Snippet 16.

Code Snippet 16 Personalized performance indicator which takes *prob_id* into consideration.

```
class EvalsOnGoalPSO(EvalsOnGoal):
    def __init__(self, prob_id = 'prob_id(convert)'):
        if 'rosenbrock-30' in prob_id:
            goal = 100
        elif 'rastringin-30' in prob_id:
            goal = 100
        elif 'griewank-30' in prob_id:
            goal = 0.1
            [...]
        else:
            raise ValueError('Invalid prob_id to set PSO goal')
        super().__init__(goal=goal)
```

The algorithm is tested on two sets of fixed weights for its particle update equation (2.7), and on 20 different seeds, as according to the reference study. Set 1 has w = 0.6, and both c_1 and c_2 equal to 1.7. Set 2 has w = 0.73, and both c_1 and c_2 equal to 1.5. Pymoo's adaptive implementation is set to *false*, so the weights remain the same through the optimization. The velocity rate is set to 0.9, and *perturbe_best* argument set to *false*. The population varies between [3, 7, 15] and [15, 30, 60] for the 2 and 30 variable test functions, respectively.

As for the termination, a multi-criterion termination class is designed, which takes into consideration three aspects: if the goal of the respective function is met, if the best solution found is the same over the last 1000 function evaluations, and if the maximum number of 10000 generations is met. If any of these criteria are met, the run is terminated.

The “Run” button is then clicked for the 2 variable problem options, and after it finishes, the results shown in Table 4.2 are presented. The column E-FE - Expected Function Evaluations, is the only one not showing directly through the app, as it is the result of post-processing calculation through the equation below.

$$\text{Expected FE (E-FE)} = \text{Average FE (A-FE)}/\text{Success Rate (SR)}. \tag{4.1}$$

Table 4.2: PSO algorithm performance: Average Function Evaluations (A-FE), Sucess Rate (SR) and Expected Function Evaluations (E-FE) on 2 variable Rosenbrock, Rastringin and Griewank problems.

		Griewank-2			Rastrigin-2			Rosenbrock-2		
Pop Size	Set	A-FE	SR	E-FE	A-FE	SR	E-FE	A-FE	SR	E-FE
3	Set 1	**439**	0.30	1,462	134	0.10	1,335	**144**	0.35	**413**
	Set 2	469	0.35	1,340	**92**	0.25	**370**	189	0.40	473
7	Set 1	854	0.85	**1,004**	333	0.85	391	2,395	**1.00**	2,395
	Set 2	935	0.90	1,038	369	0.80	462	3,057	**1.00**	3,057
15	Set 1	1,256	0.95	1,322	470	0.90	522	1,055	**1.00**	1,055
	Set 2	1,424	**1.00**	1,424	532	**1.00**	532	1,114	**1.00**	1,114

As it was expected, when the population size increases, there is a larger success rate to met the goal, with the 15 member population always having the largest values. Figure 4.1(a) presents the best and worst PSO runs with population 3 and parameter Set 1. The seemingly smooth fitness landscape reveals a rugged surface with multiple local minima when zoomed in. This multiple local minima can trap the few members into early convergence. This problem is not immune to larger populations, especially when the valleys are more accentuated. As can be seen in Figure 4.1(b), which presents both successful and unsuccessful PSO runs with population 15 on the Rastrigin problem, even larger populations can fall in local minima. Rosenbrock, being unimodal (only one minimum), has the higher success rate. However, it can still be problematic for smaller populations, which will stall in the plane areas, as seen in Figure 4.1(c).

However, the higher success rate of larger populations comes at the cost of more function evaluations. This is why the best value of the average function evaluations is always found in the smaller population. The expected function evaluations values are computed to understand the best trade-off between success rate and average function evaluations. A population of size 3 reveals to be the best approach for Rosenbrock and Rastrigin problems, and size 7 for Griewank.

It is worth noting the difference between Set 1 and Set 2. As the first has a lower inertia weight value and a higher social weight value, its population should, on average, converge faster. This is verified as the algorithm takes on average 6% less function evaluations to achieve the goal. In contrast, the higher inertia value and lower social weight make Set 2 have a success rate 22% higher. Taking these differences into account, Set 2 achieves the best trade-off overall, with 13% fewer expected function evaluations.

To see if these conclusions apply to higher-order search spaces, as are the ones normally found in

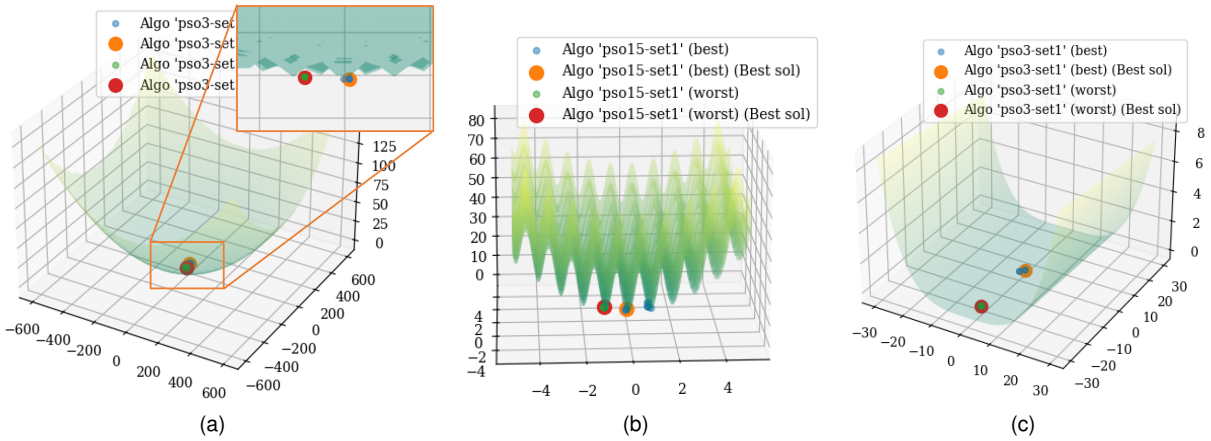


Figure 4.1: Fitness Landscape plotted for best and worst fitness values of PSO with Set 1 for the problems: Griewank, with population set to 3 (a), Rastrigin, population of size 15 (b), and Rosenbrock, population of size 3 (c).

real-world optimization problems, the algorithm is now tested in the same problems with the number of variables set to 30. and the respective population sizes. The results of this Run are saved in a *.csv* and translated into Table 4.3.

Table 4.3: PSO algorithm performance: Average Function Evaluations (A-FE), Sucess Rate (SR) and Expected Function Evaluations (E-FE) on 30 variable Rosenbrock, Rastringin and Griewank problems.

		Griewank-30			Rastrigin-30			Rosenbrock-30		
Pop Size	Set	A-FE	SR	E-FE	A-FE	SR	E-FE	A-FE	SR	E-FE
15	Set 1	8,636	0.40	21,591	**2,694**	0.25	10,776	**14,523**	0.30	48,408
	Set 2	**8,311**	0.60	13,852	4,973	0.20	24,863	26,128	0.95	27,504
30	Set 1	8,403	0.95	**8,845**	3,971	0.70	**5,672**	21,858	**1.00**	21,858
	Set 2	10,103	0.90	11,226	6,774	0.50	13,548	14,790	**1.00**	**14,790**
60	Set 1	12,750	**1.00**	12,750	6,578	**0.80**	8,222	18,138	**1.00**	18,138
	Set 2	16,153	0.95	17,003	10,478	**0.80**	13,097	19,311	**1.00**	19,311

In terms of population size, the same phenomenon occurs: the lowest population has all the best average function evaluation values. In contrast, the 60-member population has the highest success rate. This can also be seen in Figure 4.2, where the average fitness of the population is plotted on the function evaluations. In the starting point, the PSO with the lowest population converges quickly, but in the last stages it often misses the goal by getting stuck in a close local minimum. On the other hand, the 60-member PSO has a slower start, but better search space exploration leads to an overtake at approximately 9000 function evaluations.

A population of 30 members proves to achieve the best trade-off between exploitation and exploration, as noted in the original study [46]. It performs better in all functions, regarding the expected function evaluations.

Regarding the difference between Set 1 and Set 2, it also maintains what was asserted for the 2 dimensional search spaces. Set 1 takes on average 18% less function evaluations to achieve the goal.

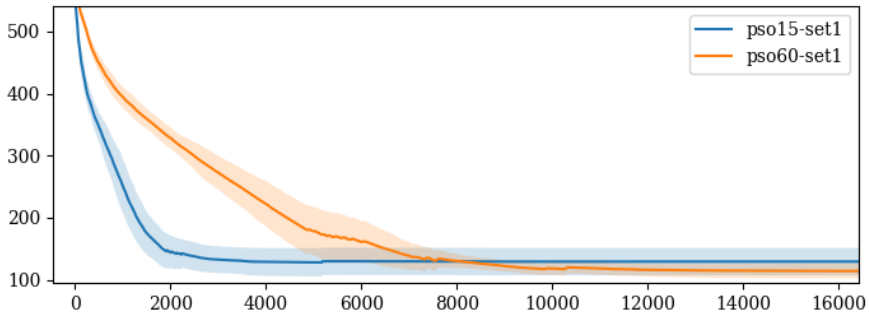


Figure 4.2: Progress of the average population fitness by function evaluations on the optimization of 30 objective Rastrigin problem by PSO with Set 1 and population size 15 (blue) and 60 (orange). Averaged across 20 seeds, shadow indicates the standard deviation.

Set 2 on the other hand has a success rate 23% higher. In general, these differences will cancel out, leading to approximately the same expected function evaluations. This means that there is no better set for all cases, but it depends on the specific function. Set 1 outperformed Set 2 on the Griewank and Rastrigin problems, while Set 2 performed better on Rosenbrock.

4.3 Multi-Objective Optimization

To understand what the proper methodology is for multi-objective optimization research and how the app could facilitate it, a well-known study presented in the article introducing the NSGA-III algorithm, by Deb and Jain [21], is taken as a reference. It tests NSGA-III against MOEA/D [2] on a series of problems from the DTLZ test suite [47].

MOEA/D decomposes a multi-objective problem into single-objective optimization sub-problems. It then evolves a population of candidate solutions over multiple generations, combining solutions from neighboring sub-problems to optimize them cooperatively. In this study case, both the Tchebycheff and penalty-based boundary intersection (PBI) decomposition methods are explored.

Tchebycheff uses (4.2) to calculate the metric by which each offspring is associated with one or more decomposition weight vectors. $\mathbf{z}^*$ is the utopian point and $\mathbf{w}$ the weight vector.

$$\mathrm{TCH}\left(\mathbf{x}, \mathbf{w}, \mathbf{z}^*\right) = \max_{i=1}^{M} w_i \left| f_i(\mathbf{x}) - z_i^* \right|. \tag{4.2}$$

For the PBI method, (4.3) is used instead, where a penalized distance measure of a point from the ideal point is formed by weighted sum (weight θ is another algorithmic parameter) of the perpendicular distance (d_2) from the reference direction and the distance (d_1) along the reference direction.

$$\mathrm{PBI}(\mathbf{x}, \mathbf{w}) = d_1 + \theta d_2. \tag{4.3}$$

Two variants of the MOEA/D algorithm are created in the app, “moead-tch” and “moead-pbi”. Their respective decomposition methods are set simply by selecting the corresponding item in the decomposition operator combo box.

The reference directions are based on the Riesz s-Energy [48] method, which provides a well-spaced

point set for any arbitrary number of points and dimensions. The dimensions are the same as the number of objectives of the respective problem, obtained by "n_obj", and the number of points is set to 100. The rest of the algorithms parameters are set according to the default values in the pymoo framework.

The algorithms are tested in DTLZ2 to DTLZ4 problems. Three variants of each are created in the app with the number of objectives and variables varying, respectively, between 3-12, 5-14, and 8-17.

The performance indicator chosen is the inverted generational distance [33], as it can measure both accuracy and diversity in the population. As the problem Pareto fronts are known, the calculation is done using the set of points in it matching the reference directions, and the distance to each point of the final population. From the app's perspective, this is allowed through the keyword "prob_pf", which allows the parameter to be correctly set for each problem.

All problem-algorithm combinations are ran on 10 different seeds. Finally, the termination type used in the study is the number of generations, set to 500. The "Run" button is then pressed and, when it finishes, the application displays the results showing the worst, median, and best values for each run, presented in Table 4.4. The best value of its type for a given problem is indicated in bold font.

Looking at Table 4.4, it is clear that MOEA/D-TCH performs worse than the other two algorithms in all tests. MOEA/D-PBI performs better in DTLZ2 for all objectives. In DTLZ3, the IGD values are closer, with NSGA-III performing slightly better than MOEA/D-PBI on 3 objectives, and the tendency inverting for 8 objectives. For DTLZ4, NSGA-III performs significantly better than both MOEA/D algorithms.

This corroborates what was found in the initial study done in the paper introducing NSGA-III [21]. Also, using the existing plotting options of the app, the obtained Pareto sets are plotted, shown in Figure 4.3 for DTLZ2 with 3 objectives. As expected, both NSGA-III and MOEA/D-PBI are converging to the reference points demonstrating good matches, while MOEA/D-TCH struggles to maintain a correspondence between the population members and reference points.

Table 4.4: Best, median and worst IGD values obtained for NSGA-III, MOEAD-TCH and MOEAD-PBI on 3, 5 and 8 objective DTLZ2 to DTLZ4 problems.

	DTLZ2			DTLZ3			DTLZ4		
Objectives:	3	5	8	3	5	8	3	5	8
MOEA/D-TCH	7.54E-02	4.21E-01	6.94E-01	7.29E-02	5.19E+00	4.83E+00	7.59E-02	6.91E-01	9.74E-01
	7.78E-02	4.42E-01	8.60E-01	7.60E-02	1.30E+01	5.16E+01	7.43E-01	8.76E-01	1.02E+00
	7.86E-02	4.50E-01	8.67E-01	1.05E+00	8.94E+01	8.48E+01	9.50E-01	1.12E+00	1.24E+00
MOEA/D-PBI	**3.33E-04**	**3.06E-03**	**6.92E-03**	7.85E-02	**9.25E-02**	**7.31E-02**	**2.59E-04**	3.21E-01	9.07E-01
	4.39E-04	**3.54E-03**	**7.56E-03**	2.71E-01	3.47E+00	**1.15E+00**	7.42E-01	9.93E-01	1.08E+00
	5.48E-04	**4.54E-03**	**1.02E-02**	1.71E+00	2.19E+01	**2.78E+00**	9.50E-01	1.12E+00	1.24E+00
NSGA-III	3.00E-02	1.24E-01	1.40E-01	**2.61E-02**	2.02E-01	2.11E-01	1.50E-02	**9.27E-02**	**1.43E-01**
	3.01E-02	1.24E-01	1.41E-01	**3.17E-02**	**3.98E-01**	2.47E+00	**1.51E-02**	**9.33E-02**	**1.43E-01**
	3.02E-02	1.24E-01	1.43E-01	**1.05E+00**	**1.85E+00**	6.13E+00	**1.53E-02**	**9.44E-02**	**1.44E-01**

In Figure 4.4, the value of the IGD indicator on the MOEA/D-PBI and NSGA-III algorithms is plotted by the number of function evaluations. The thick line indicates the average value and the shadows represent the standard deviation.

The y axis is in the logarithmic scale, and thus the lower and upper limits of the standard deviation

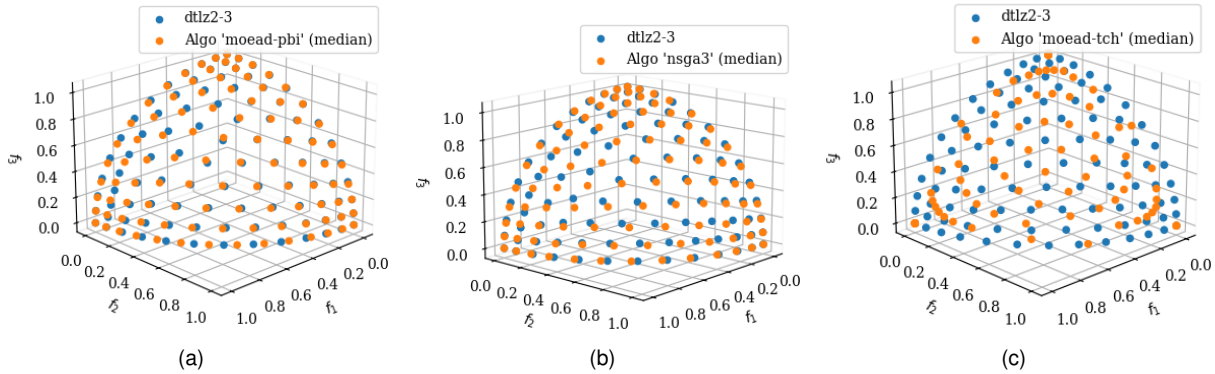


Figure 4.3: Reference points on the 3 objective DTLZ2 (blue) and final Pareto sets obtained (orange) for the median runs of MOEA/D-PBI (a), NSGA-III (b), and MOEA/D-TCH (c).

appear to be unequal. An observation similar to the original study can be made [21]: MOEA/D-PBI starts with a steeper progress, but is surpassed by the NSGA-III algorithm around the 10,000 function evaluations, maintaining a worst value from then on.

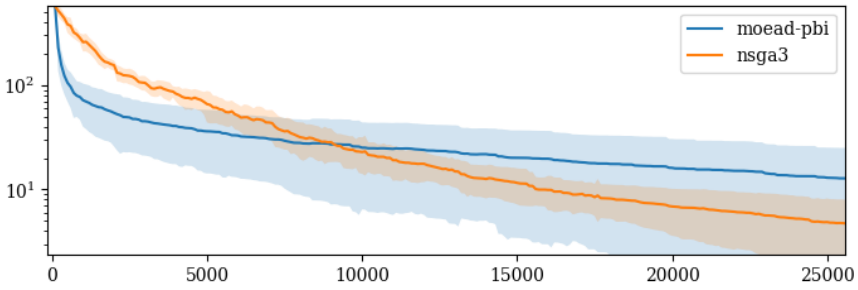


Figure 4.4: Progress of the IGD indicator by function evaluations on the optimization of 5 objective DTLZ3 problem by MOEA/D-PBI (blue) and NSGA-III (orange). Averaged across 10 seeds, shadow indicates the standard deviation.

Another aspect tested is the robustness of these algorithms with problems that are not normalized. For this, the same experience is conducted, but with a scaling factor in the DTLZ set where the i-th objective is scaled by 10^i.

To achieve this through the app, the class *ScaledDTLZ* is used to create the respective variants. The problems are optimized by the NSGA-III and MOEA/D-PBI algorithms with the same initialization arguments, on 5 seeds. The termination criterion is the same 500 generations.

After pressing the “Run” button and finishing the Run, the app shows the results in Table 4.5, where the best, median and worst values of the IGD indicator are presented. The better performance of NSGA-III relative to MOEA/D is clear when looking at the table, as the IGD values are inferior in all cases.

This fact, also mentioned in the original study [21], is given to the normalization procedure inherent to the NSGA-III selection process, detailed in Section 2.2.3. As MOEA/D includes no such mechanism, it will treat scaled-up objectives with a disproportionate attention, resulting in the loss of diversity among solutions, focusing on the optimization of just the objectives with the larger absolute values.

Table 4.5: Best, median and worst IGD values obtained for NSGA-III and MOEAD-PBI on 3, 5 and 8 objective scaled DTLZ2 to DTLZ4 problems.

	Scaled DTLZ2			Scaled DTLZ3			Scaled DTLZ4		
Objectives:	3	5	8	3	5	8	3	5	8
MOEA/D-TCH	4.38E+01	3.09E+03	2.28E+06	4.74E+01	3.31E+03	2.07E+06	4.76E+01	3.16E+03	1.93E+06
	4.42E+01	3.09E+03	2.28E+06	4.83E+01	3.31E+03	2.07E+06	4.85E+01	3.16E+03	1.93E+06
	4.46E+01	3.09E+03	2.28E+06	4.85E+01	3.31E+03	2.07E+06	4.85E+01	3.16E+03	1.93E+06
NSGA-III	**1.59E-02**	**1.19E+01**	**1.07E+04**	**4.75E-01**	**1.22E+02**	**1.11E+06**	**1.33E-02**	**9.35E+00**	**4.78E+04**
	2.41E-02	**1.74E+01**	**1.52E+04**	**1.10E+00**	**2.11E+02**	**1.23E+06**	**1.59E-02**	**1.06E+01**	**1.21E+05**
	3.47E-02	**2.31E+01**	**2.97E+04**	**1.76E+00**	**4.86E+02**	**1.78E+06**	**3.85E-02**	**1.36E+01**	**9.72E+05**

This becomes also evident when one looks at Figure 4.5. Although the NSGA-III solutions of the median run are well spread among objectives, the same does not occur for MOEA/D. As virtually every real-world optimization problem has unscaled objectives, this proves to be a significant advantage of NSGA-III.

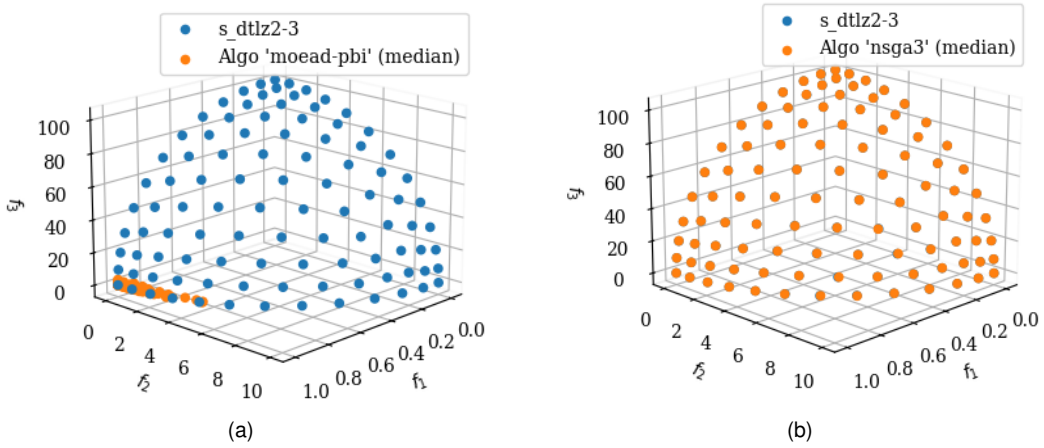


Figure 4.5: Reference points on the 3 objective DTLZ2 (blue) and final Pareto sets obtained (orange) for median runs of MOEA/D-PBI (a), and NSGA-III (b).

4.4 Worst-Case Scenario

The usefulness of the application was shown in the previous sections for general research cases in single and multi-objective optimization, making use of pymoo's wide range implemented problems and algorithms. However, it remains to be seen how or if it could help in the worst-case scenario - if the optimization at hand is to be made through a specific algorithm, problem and even visualization technique that is not already implemented in pymoo.

The following sections detail what is necessary to implement and integrate custom classes into the app.

4.4.1 Problem

For the problem, an adaptation of the Traveling Salesman Problem, detailed in Section 2.4 is done. It is now transformed into a multi-objective problem with mixed variable types, where the salesman not only has to choose the order of the cities to be traversed, but also the mode of transportation. A child class of the core pymoo *Problem* class is implemented, with the name *MultiobjectiveMixedTSP*.

For each type of transportation, two matrices are provided, the cost matrix and the time matrix. Naturally, one mode of transport can make the trip shorter but more expensive or vice versa. The problem now consists not only in finding the best order to cross each city once, but also the transportation types in each trip that will find the best Pareto front.

A solution *x* is coded as a *numpy* array with size $2c$, where c is the number of cities. The first c elements of the array are the city numbers, and the last c elements are the integer corresponding to the type of transportation. The evaluation method is thus programmed as shown in Code Snippet 17.

Code Snippet 17 *_evaluate* method of the multi-objective TSP class.

```
def _evaluate(self, x, out, *args, **kwargs):

    n_cities = self.n_cities
    path, transport_idx = x[:n_cities], x[n_cities:]
    time = 0
    cost = 0
    for k in range(n_cities - 1):
        i, j = path[k], path[k + 1]
        transport_k = self.transport_options[transport_idx[k]]
        time += self.Time[transport_k][i, j]
        cost += self.Cost[transport_k][i, j]

    # back to the initial city
    last, first = path[-1], path[0]
    transport_last = self.transport_options[transport_idx[-1]]
    time += self.Time[transport_last][last, first]
    cost += self.Cost[transport_last][last, first]

    out['F'] = np.array([time, cost])
```

The nadir point, later used as a reference point to calculate the hypervolume, is set as the worst value found in the time and cost matrices multiplied by the number of cities.

$$\text{Nadir} = \text{n\_cities} \cdot [\text{WorstTime}, \text{WorstCost}]. \tag{4.4}$$

A child of the class is created to be directly introduced in the app. This class randomly creates the city coordinates along the specified size. It assumes three modes of transportation: car, train, and plane. For the car, the time matrix is calculated based on the distance value. The train and plane time matrices are calculated by multiplying the matrix by $1/$factor, with the factor set by default as 5 and 10, respectively.

The cost matrix for the car is set with the values obtained with a uniform probability between $\pm$Per$\%$ of the original value of the time matrix, where Per is a percentage value given by the user, with 10 as

the default. The process for the train and plane matrices is the same, but in the end multiplied by the respective 5 and 10 factors. So, for this problem, it is expected that the cheapest path will have more car travels, and the fastest path will have more plane travels. Lastly, the problem is inserted into the application by adding it to the *get_problem* function options.

4.4.2 Algorithm

For the algorithm, there are two cases to be studied. The framework already provides operator classes for making a generic genetic algorithm capable of handling a simple permutation problem. These classes are then altered to fit the problem variable representation. *PermutationRandomSampling* is adapted in *MixedPermRandomSampling* and *OrderCrossover* in *MixedOrderCrossover*. *InversionMutation* is changed into *InversionFlipMutation* to not only mutate the path, but also the transportation. More details of each of these operators can be found in the framework website[1].

The class *PermutationNSGA2*, a child of *NSGA2*, is created with all the aforementioned operators set as default values. The survival and selection processes are copied from the default NSGA-II (Section 2.2.2) methods.

Furthermore, a new algorithm (ACO-NSGA2) is created that mixes NSGA2 and ACO (Section 2.2.6) methods, to see if it obtains better results from the heuristic-based approach that ACO provides, tailored to graph problems. In this regard, the NSGA-II survival operator *RankAndCrowding*, is significantly extended in *RankAndCrowdingACO*. After the usual surviving procedure, the population is then modified by the steps described in Section 2.2.6, as shown in Code Snippet 18.

Code Snippet 18 Survival operator class *RankAndCrowdingACO*.

```
class RankAndCrowdingACO(RankAndCrowding):
    [...]
    def _do(self, problem, pop, *args, n_survive=None, **kwargs):
        # do the normal rank and crowding
        pop = super()._do(problem, pop, [...])

        [...] # update pheromone, get probability matrix

        # update ants
        for ant in range(len(X_path)):
            probs = original_probs.copy()
            curr_city , X_path[ant, 0] = 0,0
            for i in range(n_cities-1):
                prob=probs[curr_city, :]/probs[curr_city, :].sum()
                new_city = np.random.choice(n_cities, p=prob)
                X_path[ant, i+1] = new_city
                # prevent going to same city
                probs_copy[curr_city, :] = 0
                probs_copy[:, curr_city] = 0
                curr_city = new_city
        [...]
        return pop
```

[1] https://pymoo.org/customization/permutation.html

In the original ACO, the probability matrix is calculated using the heuristic and pheromone values. The pheromone values, in turn, are calculated on the basis of the fitness of the ant. As this is a multi-objective problem, there is no direct fitness value. Therefore, a weight decomposition strategy is employed, normalizing the two objective functions and summing up the results. This is the value used to assess the fitness of an ant.

The heuristic value is based on the direct distance between the two cities. As time and cost values are strongly correlated with this value, as detailed in the problem section, it is expected that this algorithm will outperform *PermutationNSGA2*. However, it remains to be seen how the original crossover and mutation operators affect the search process. Two versions of the algorithm are to be tested, ACO-NSGA2 and ACO-NSGA2 (s) - simple, the latter maintaining only the transport flip part of the mutation and performing no crossover after the ACO procedure.

4.4.3 Plotting

The already implemented *QProgress* class can provide important aspects regarding the optimization task at hand, as the hypervolume indicator can be obtained and analyzed even though the exact optimal Pareto front is unknown.

However, by the nature of the problem, it would be very beneficial to visualize the path in a customized way, to quickly understand the type of solutions found by the algorithm. In this regard, a special *TSPplot* class is designed, as a child of the *QPlot* class mentioned in Section 3.2.2.

Depending on the *plot_best* argument value, it plots one of the specific solutions of the given Pareto front. For 'cost', it chooses the solution with the smallest cost; 'time' chooses the one with the lowest time. With 'ratio' as input, it uses the same decomposition strategy as mentioned in the previous section, and chooses the lowest value of the sum of the normalized objectives. For the given solution, the grid is plotted with the coordinates of the cities and different colors for the different transportation methods, as seen in Figure 4.8.

4.4.4 Run

The problems, set to 10, 20 and 60 cities, are optimized by the three detailed algorithms with a population of size 20, for 10 different seeds, and 4,000 function evaluations as the termination criteria. The after the "Run" button is clicked and the Run finished, the results for the normalized hypervolume are presented in the app, shown in Table 4.6.

Table 4.6: Average negative hypervolume value for the modified algorithms on MixedTSP-10, 20 and 60.

	MixedTSP-10	MixedTSP-20	MixedTSP-60
ACO-NSGA2 (n)	-0.897	-0.928	**-0.941**
ACO-NSGA2 (s)	**-0.903**	**-0.931**	-0.939
Permutation-NSGA2	-0.894	-0.897	-0.869

As expected, ACO-NSGA2 outperforms Permutation-NSGA2. The heuristic-based method is shown to converge better, as it uses the problem specific knowledge (the cities coordinates) to better evolve the solutions. With regard to the two ACO-NSGA2 types, the simple performs better in MixedTSP-10 and 20, and the normal one in MixedTSP-60.

Figure 4.6 presents the negative hypervolume values by function evaluations for MixedTSP-10, 20 and 60 cities. In the 10 cities problem, the three algorithms manage to converge quickly, with ACO-NSGA2 (s) achieving the best results. With 20 cities, ACO-NSGA2 is closer and Permutation-NSGA2 struggles to converge.

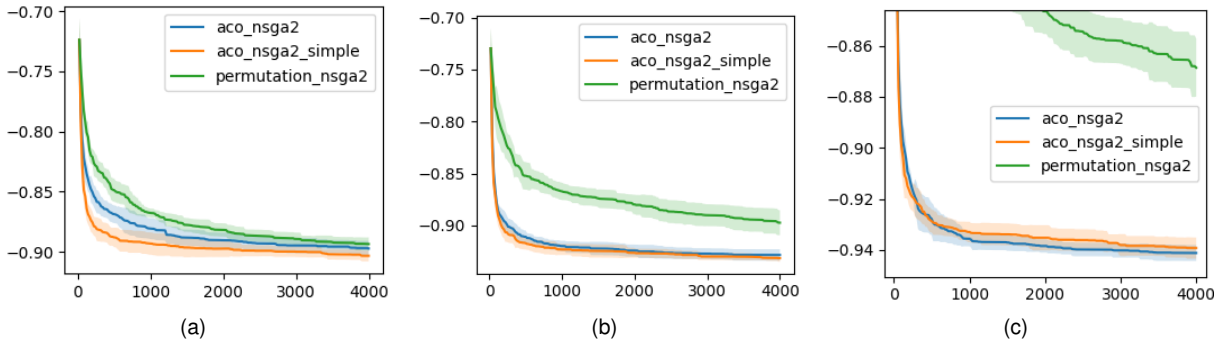


Figure 4.6: Negative hypervolume by function evaluations on ACO-NSGA2, ACO-NSGA2 (simple) and Permutation-NSGA2 for MixedTSP-10 (a), 20 (b), and 60 (c).

This is exacerbated in MixedTSP-60, with the Permutation-NSGA2 unable to find a good solution for the termination set, and ACO-NSGA2 (s) surpassing its twin before 500 function evaluations. This suggests that with a simpler problem, the mutations and crossovers of ACO-NSGA2 actually stall the convergence process, but as they become more complex, they are helpful to not get stuck in local minimums and converge to the true Pareto front. The Pareto sets obtained by the median runs of the algorithms are shown in Figure 4.7.

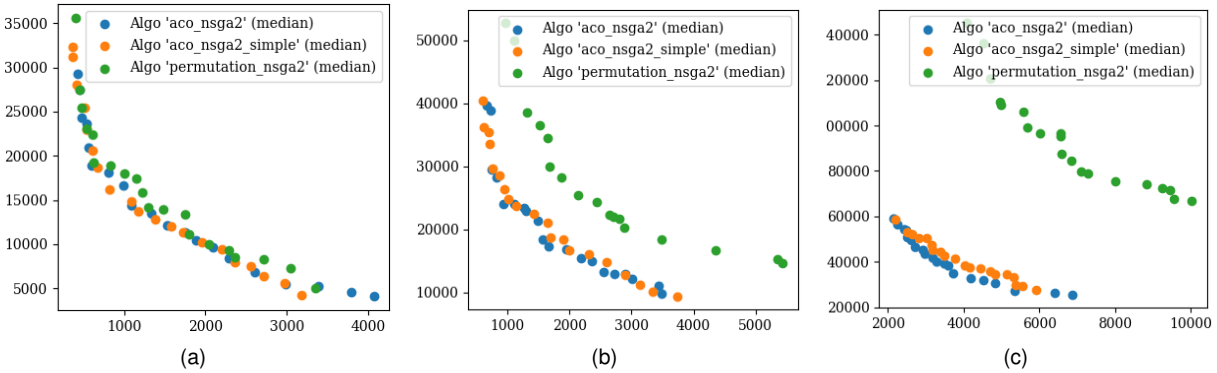
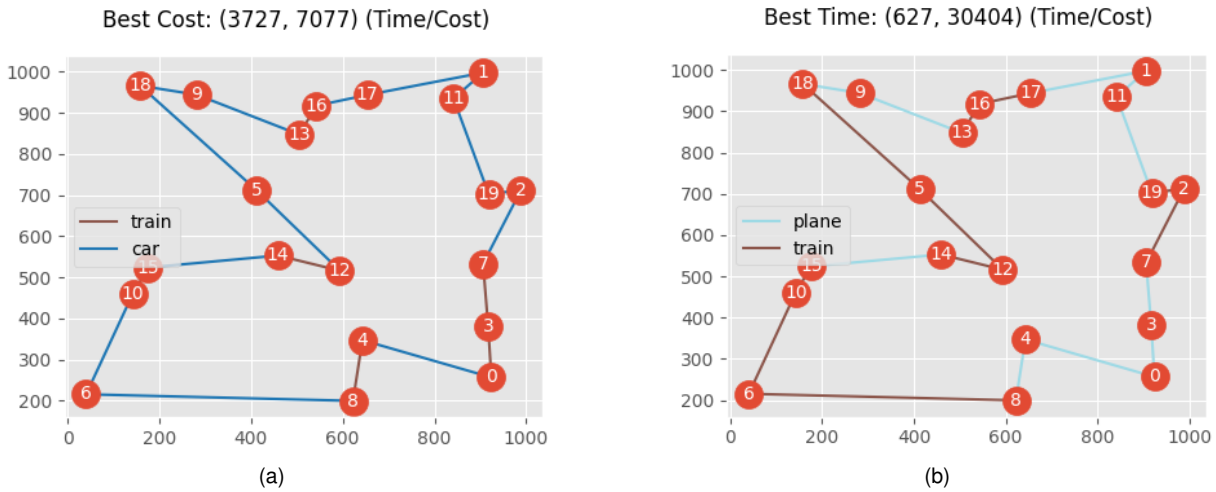


Figure 4.7: Final Pareto sets obtained by the median runs of ACO-NSGA2, ACO-NSGA2 (simple) and Permutation-NSGA2 on MixedTSP-10 (a), 20 (b), and 60 (c).

They are also in accordance with what can be perceived with the hypervolume convergence figures. In Figure 4.7(a), the Pareto sets are closer to each other. Figure 4.7(b) shows that both versions of ACO-NSGA2 converge well, with slightly better performance of the (s) version, while Permutation-NSGA2 struggles. In Figure 4.7(c), it becomes clear that ACO-NSGA2 surpasses the (s) version.

To analyze in more detail the best solutions found by the algorithms on MixedTSP-20, the plotting method devised in Section 4.4.3 is used, resulting in Figure 4.8.



(a) (b)

Figure 4.8: Graph of the solutions obtained by ACO-NSGA2 (simple) on MixedTSP-20 with the best path cost (a), and path time (b).

Figure 4.8(a) shows the path with the best cost. As expected, the most frequent transportation method is the car, the cheapest on average. However, there are still some connections by train. In contrast, Figure 4.8(b) shows the path with the best time, where the most frequent transportation method is the plane, but multiple connections are also traveled by train.

This can be explained in part by the mutations performed in the time and cost matrices, as the values are not directly proportional to the distance. So, if the car cost value is mutated to a higher value in relation to its reference and the train to a lower value in relation to its reference, they can become closer to one another. However, since the mutation percentage is set to only 10% and the multiplying factor to 5, it is found that the train cost will never be lower than the car. Therefore, it can only be concluded that the algorithm has yet to converge to the Pareto optimal front extreme with the lowest cost. The same analysis holds for the time objective function.

It stems from this observation that in order to achieve these Pareto front extremes, it would be necessary to raise the number of function evaluations set as the termination criterion, or change the mutation transport flip part to privilege paths with the same transportation mode, in order to obtain a quicker convergence.

It is worth noting that both paths, while being the extremes of the Pareto front, have the same sequence of visited cities. This is because the order of the path generally minimizes both objectives, so they are not competing in that regard. This suggests that the algorithm converged to the best order and is simply finding the best transportation. However, it is important to note that this can not always be the case if the time and cost matrices are not calculated using the distance between cities directly.

4.5 Conclusion

In this chapter, the wide range of use cases of where the application is useful was demonstrated. First assuring the fidelity of results with the original framework in Section 4.1, a single-objective optimization study was conducted in Section 4.2, analyzing the PSO behavior, calculating the average and expected function evaluations, and success rate. Furthermore, the fitness landscape was plotted to obtain insights into the optimization process.

In addition, a look at the differences between NSGA-III and MOEAD was conducted in Section 4.3, reviewing the proper methodology to guarantee reliable results. The progress of the IGD indicator and Pareto sets were plotted and discussed.

Finally, the worst-case scenario was dealt with in Section 4.4, where new algorithms, problems, and plotting methods had to be programmed. Through the breadth of the framework options and the ease of extending the app's functionalities, three algorithms were compared, using the app's own methods and a new personalized visualization technique.

Chapter 5

Conclusions

5.1 Achievements

The thesis had as its goal the creation of a desktop application for single and multi-objective optimization. To achieve it, the fundamentals of single and multi-objective optimization were explored in Chapter 2, including established Evolutionary Algorithms, benchmarking problems, and performance indicators.

Furthermore, the available optimization frameworks with and without an attached GUI were detailed to better understand what is already available in the market and where there is a lack of offerings and space for innovation. The library options to construct a Python-based graphical user interface were presented, in order to access the best option that would fit the needs for the work proposed in this thesis.

Chapter 3 provided the architecture of the application and the thought process that went into the building options to meet the requirements set in Section 1.3.

The various design choices, from the chosen optimization framework and GUI library, to the app's modularity and easily extensible nature, were explained and linked with the different objectives they were set to achieve.

To test the practicality of the application, Chapter 4 went through the methodologies of single and multi-objective optimization research, having as a reference renown studies in the literature. First, to guarantee the correctness of the results, it showed the correspondence between those obtained through the application and directly through the optimization framework.

Then, taking Trelea's single-objective PSO study [46] as a reference, it showed how it could be accelerated and facilitated through the use of the application, as well as the usefulness of the provided visualization techniques. For multi-objective optimization, the paper introducing NSGA-III by Deb and Jain [21] was taken as a reference, also demonstrating the app's ability to provide the necessary conditions for research in this area.

Lastly, in Section 4.4, it was demonstrated how the extensibility of the app allows it to be useful even in cases where the problem, algorithm, and adequate visualization techniques are not already provided

and have to be incorporated.

In summary, the developed application successfully meets the set requirements, providing a versatile and user-friendly tool for optimization. It demonstrates the potential of integrating an existing optimization framework with a custom-built graphical interface to create a solution that fulfills the needs not yet met by the market. The application's open-source nature and its balance between ease of use and customization make it a valuable resource for both novice and experienced users in the field of optimization.

5.2 Future Work

Further development of the application could be done in some areas. A comprehensive suite of automated tests for many of the app's functionalities such as buttons, save/load methods, and visualization techniques could be undertaken to substitute manual verification.

Also, it could integrate more options already available in the pymoo framework, such as dynamic problems, hyper-parameter optimization, decision-making strategies, and other visualization methods such as video capture. Customization of the constraint handling methods could be made available to the user in the graphical interface.

A deeper focus on speeding up the optimization process could be done by presenting costumized options for parallelization of function evaluations. In addition, the aesthetics of the application could be improved, including font, size, and colors, as it was not the main focus of this work nor within the student's natural realm of interests or talents.

The application could be made available as a ready-to-use executable, avoiding a more difficult installation process and more requirements for its use. Also, it could be explored the feasibility to turn it into a web app, so the optimization could be made online and not be dependable of the local computer characteristics. Lastly, the application can be taken to pymoo's active community to receive feedback and suggestions, and eventually turned into a collaborative project.

Bibliography

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-II," *Trans. Evol. Comp*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[2] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[3] R. Martins, N. Lourenço, and N. Horta, "Multi-objective optimization of analog integrated circuit placement hierarchy in absolute coordinates," *Expert Systems with Applications*, vol. 42, no. 23, pp. 9137–9151, 2015.

[4] H. Ishibuchi and T. Murata, "A multi-objective genetic local search algorithm and its application to flowshop scheduling," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 28, no. 3, pp. 392–403, 1998.

[5] D. Moniz, J. Pedro, N. Horta, and J. Pires, "Multi-objective framework for cost-effective otn switch placement using nsga-ii with embedded domain knowledge," *Applied Soft Computing*, vol. 83, p. 105608, 2019.

[6] T. H. W. Bäck, A. V. Kononova, B. van Stein, H. Wang, K. A. Antonov, R. T. Kalkreuth, J. de Nobel, D. Vermetten, R. de Winter, and F. Ye, "Evolutionary Algorithms for Parameter Optimization—Thirty Years Later," *Evolutionary Computation*, vol. 31, no. 2, pp. 81–122, 06 2023.

[7] J.-Y. Li, Z.-H. Zhan, and J. Zhang, "Evolutionary computation for expensive optimization: A survey," *Machine Intelligence Research*, vol. 19, no. 1, pp. 3–23, 2022.

[8] D. Hadka, "Moea framework," http://moeaframework.org/, 2015, accessed: November 22, 2023.

[9] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[10] Y. Tian, R. Cheng, X. Zhang, and Y. Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Computational Intelligence Magazine*, vol. 12, no. 4, pp. 73–87, 2017.

[11] D. J. Higham and N. J. Higham, *MATLAB guide*. SIAM, 2016.

[12] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.

[13] F.-A. Fortin, F.-M. De Rainville, M. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *Journal of Machine Learning Research, Machine Learning Open Source Software*, vol. 13, pp. 2171–2175, 07 2012.

[14] G. Rossum, "Python reference manual," 1995.

[15] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[16] A. E. Eiben and J. E. Smith, *Evolutionary Computing: The Origins*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 13–24.

[17] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 03 1998.

[18] D. B. Fogel, *An Introduction to Evolutionary Computation*. Wiley-IEEE Press, 1998, pp. 1–28.

[19] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948 vol.4.

[20] N. Srinivas and K. Deb, "Muiltiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994.

[21] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[22] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, 2011.

[23] A. Benítez-Hidalgo, A. J. Nebro, J. García-Nieto, I. Oregi, and J. Del Ser, "jmetalpy: A python framework for multi-objective optimization with metaheuristics," *Swarm and Evolutionary Computation*, vol. 51, p. 100598, 2019.

[24] A. L. Cauchy, "M'ethode g'en'erale pour la r'esolution des syst'emes d''equations simultan'ees," *Journal de l'Ecole Polytechnique*, vol. 17, pp. 141–161, 1847.

[25] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–313, 1965.

[26] M. Garza-Fabre, G. T. Pulido, and C. A. C. Coello, "Ranking methods for many-objective optimization," in *MICAI 2009: Advances in Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 633–645.

[27] K. Deb, *Introduction to Evolutionary Multiobjective Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 59–96.

[28] J. L. Cohon, *Multiobjective programming and planning*. Courier Corporation, 2004, vol. 140.

[29] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Berlin, Heidelberg: Springer-Verlag, 2005.

[30] J. H. Holland, *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[31] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 26, no. 1, pp. 29–41, 1996.

[32] D. A. V. Veldhuizen, "Multiobjective evolutionary algorithms: Classifications, analyses, and new innovations," Evolutionary Computation, Tech. Rep., 1999.

[33] C. A. Coello Coello and M. Reyes Sierra, "A study of the parallelization of a coevolutionary multiobjective evolutionary algorithm," in *MICAI 2004: Advances in Artificial Intelligence*, R. Monroy, G. Arroyo-Figueroa, L. E. Sucar, and H. Sossa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 688–697.

[34] H. Ishibuchi, H. Masuda, Y. Tanigaki, and Y. Nojima, "Modified distance calculation in generational distance and inverted generational distance," in *Evolutionary Multi-Criterion Optimization*, A. Gaspar-Cunha, C. Henggeler Antunes, and C. C. Coello, Eds. Cham: Springer International Publishing, 2015, pp. 110–125.

[35] C. M. Fonseca, L. Paquete, and M. López-Ibáñez, "An improved dimension sweep algorithm for the hypervolume indicator," in *Proceedings of the 2006 Congress on Evolutionary Computation (CEC 2006)*. Piscataway, NJ: IEEE Press, Jul. 2006, pp. 1157–1163.

[36] P. Wegner, "Concepts and paradigms of object-oriented programming," *ACM Sigplan Oops Messenger*, vol. 1, no. 1, pp. 7–87, 1990.

[37] R. Martins, N. Lourenço, A. Canelas, R. Póvoa, and N. Horta, "Aida: Robust layout-aware synthesis of analog ics including sizing and layout generation," in *2015 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2015, pp. 1–4.

[38] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: pygmo," *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, 2020.

[39] A. D. Moore, *Python GUI Programming with Tkinter: Develop responsive and powerful GUI applications with Tkinter*. Packt Publishing Ltd, 2018.

[40] R. Ulloa, *Kivy: interactive applications in python*. Packt Publishing Ltd, 2013.

[41] R. D. Noel Rappin, *wxPython in Action*. Manning, 2006.

[42] M. Summerfield, *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education, 2007.

[43] V. Loganathan, *PySide GUI application development*. Packt Publishing Ltd, 2013.

[44] W. McKinney, “Data structures for statistical computing in python,” 2010.

[45] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[46] I. C. Trelea, “The particle swarm optimization algorithm: convergence analysis and parameter selection,” *Information Processing Letters*, vol. 85, no. 6, pp. 317–325, 2003.

[47] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, “Scalable test problems for evolutionary multiobjective optimization,” in *Evolutionary multiobjective optimization: theoretical advances and applications*. Springer, 2005, pp. 105–145.

[48] J. Blank, K. Deb, Y. Dhebar, S. Bandaru, and H. Seada, “Generating well-spaced points on a unit simplex for evolutionary many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 1, pp. 48–60, 2021.

Appendix A

Installation

To use the desktop application on your local computer, the following simple steps are required.

1. If you do not have Git installed, download the zip file from this repository (`https://github.com/tomaslibanomonteiro/Thesis-code`) and extract it to a directory of your choice. Otherwise, issue the command to clone the project repository to your local machine.

   ```
   $ git clone https://github.com/tomaslibanomonteiro/Thesis-code
   ```

2. Although not required, it is recommended to create a virtual environment for the installation of the project dependencies. Installing a package manager such as Anaconda (`https://www.anaconda.com/download`) can facilitate this process. To create a new environment with Python 3.9 through Conda, execute the command below.

   ```
   $ conda create -n my_env python=3.9
   ```

3. Navigate to the project directory and activate the environment.

   ```
   $ conda activate my_env
   ```

4. Install the required dependencies using the `utils/requirements.txt` file of the project.

   ```
   $ pip install -r utils/requirements.txt
   ```

5. Run the `main.py` file to start the application.

   ```
   $ python main.py
   ```

Appendix B

Available Options in the App

Table B.1: Available options in the app for the respective *get* functions. The "(n)" at the end of the class indicates a new class created in the context of the thesis. Documentation in `https://pymoo.org/`.

***get* function**	**Count**	**Options**
Algorithms	14	GA, DE, NelderMead, PatternSearch, CMAES, PSO, BRKGA, NSGA2, NSGA3, UNSGA3, MOEAD, CTAEA, PermutationNSGA2 (n), ACO_NSGA2 (n).
Sampling	5	FloatRandomSampling, LHS, BinaryRandomSampling, PermutationRandomSampling, MixedPermRandomSampling (n).
Survival	3	FitnessSurvival, RankAndCrowding, RankAndCrowdingACO (n).
Selection	6	RandomSelection, TournamentByCVAndFitness, RestrictedMatingCTAEA, BinaryTournament, TournamentByCVThenRandom, TournamentByRankAndRefLineDist.
Crossover	11	SBX, DEX, PCX, UniformCrossover, HalfUniformCrossover, ExponentialCrossover, PointCrossover, OrderCrossover, EdgeRecombinationCrossover, MyNoCrossover (n), MixedOrderCrossover (n).
Mutation	5	NoMutation, PM, BitflipMutation, InversionMutation, InversionFlipMutation (n).
Ref. Dir.	4	UniformReferenceDirectionFactory, RieszEnergyReferenceDirectionFactory, LayerwiseRieszEnergyReferenceDirectionFactory, ReductionBasedReferenceDirectionFactory.
Decomp.	6	WeightedSum, Tchebicheff, PBI, ASF, AASF, PerpendicularDistance.
Termination	9	MaximumFunctionCallTermination, MaximumGenerationTermination, MinimumFunctionValueTermination, TimeBasedTermination, DefaultSingleObjectiveTermination, MinFitnessTermination (n), StaledBestTermination (n), PSOTermination (n), DefaultMultiObjectiveTermination.
Problems	115	Ackley, G1, G2, G3, G4, G5, G6, G7, G8, G9, G10, G11, G12, G13, G14, G15, G16, G17, G18, G19, G20, G21, G22, G23, G24, CantileveredBeam, GriewankExplicitLimits, Himmelblau, RandomKnapsackSingle (n), PressureVessel, RastriginExplicitLimits (n), RosenbrockExplicitLimits (n), Schwefel, Sphere, Zakharov, BNH, RandomKnapsackMulti (n), Carside, CTP1, CTP2, CTP3, CTP4, CTP5, CTP6, CTP7, CTP8, DASCMOP1, DASCMOP2, DASCMOP3, DASCMOP4, DASCMOP5, DASCMOP6, DASCMOP7, DASCMOP8, DASCMOP9, DF1, DF2, DF3, DF4, DF5, DF6, DF7, DF8, DF9, DF10, DF11, DF12, DF13, DF14, MW1, MW2, MW3, MW4, MW5, MW6, MW7, MW8, MW9, MW10, MW11, MW12, MW13, MW14, DTLZ1, DTLZ2, DTLZ3, DTLZ4, DTLZ5, DTLZ6, DTLZ7, ScaledDTLZ, C1DTLZ1, C1DTLZ3, C2DTLZ2, C3DTLZ1, C3DTLZ4, DC1DTLZ1, DC1DTLZ3, DC2DTLZ1, DC2DTLZ3, DC3DTLZ1, DC3DTLZ3, Kursawe, OSY, SRN, TNK, Truss2D, WeldedBeam, ZDT1, ZDT2, ZDT3, ZDT4, ZDT5, ZDT6, RandomMultiMixedTSP (n).
Perf. Ind.	11	BestFitness (n), AvgPopFitness (n), MinusGoalAchieved (n), EvalsOnGoal (n), MinusGoalAchievedPSO (n), EvalsOnGoalPSO (n), GD, GDPlus, IGD, IGDPlus, minusHypervolume (n).
Plot Types	5	QFitnessLandscape (n), QPCP (n), QParetoSets (n), QProgress (n), TSPplot (n).