# Chapter 1. Introduction

Multi-objective optimization is essential for fields like engineering, finance, or logistics, where decision-makers often must balance conflicting goals. (1) Traditional optimization methods fail to deal with such situations effectively since they cannot handle multiple goals simultaneously. In this regard, metaheuristic algorithms like NSGA-II and SMPSO have valuable approaches when it comes to tackling multi-objective optimization problems. (2)

# Project scope

This project aims to explore how combining the NSGA-II and SMPSO algorithms can enhance multi-objective optimization. Combining the genetic algorithm techniques of NSGA-II with the swarm intelligence of SMPSO, a hybrid algorithm that efficiently searches for solutions while maintaining a diverse set of options is created.

# Project objectives

The principal project objectives that should be implemented are the following:

* Analyze the mechanisms of NSGA-II and SMPSO algorithms. (2)
* Design an integration framework to combine NSGA-II and SMPSO components effectively – GAP simulator.
* Validate the algorithms performance on benchmarks.
* Evaluate the effectiveness of the algorithms with coverage, hypervolume (HV), Trailing Suction Hopper Dredger (TSHD) metrics.
* Check the impact of various parameter settings regarding the performance of the algorithms.
* Implement the superposition of NSGAII and SMPSO algorithms and analyze the results in comparison with individual implementation of the algorithms
  1. Contribution

This dissertation project advances the field of multi-objective optimization by

* Proposing a framework to combine the NSGA-II and SMPSO algorithms.
* Evaluating how well this integrated algorithm performs on standard test problems.
* Offering insights into how genetic algorithms (GA) and particle swarm optimization (PSO) techniques work together in multi-objective optimization. (3) (4)
* Integrate the GAP simulator for generating the needed results
  1. Scope and limitations
* The aim of this project is the integration of NSGA-II and SMPSO algorithms and the superposition of both algorithms for multi-objective optimization. (2)
* The evaluation of performance is focused on benchmark problems and may not cover all possible scenarios.
* The study assumes fixed parameter settings and does not explore dynamic parameter adaptation strategies.

# Paper structure

In the second chapter, the used technology stack is described, together with the jMetal library concept.

In the third chapter, the theoretical part of the project is detailed: Pareto concept, genetic algorithms, algorithm’s comparisons.

In the fourth chapter, the practical part of the project is explained, together with the analyses and charts containing the results of the superposition.

The last chapter is about the conclusions of this work and some ideas for future work.

* 1. Used technologies

The technologies that were used in the implementation of this project were the following:

* Sourcetree – version control
* IntelliJ IDEA – software tool
* Java language – backend implementation
* JDK
* jMetal library – library used for working with evolutionary algorithms
* Excel – tool used for generating the graphics
* excalidraw – tool used for creating diagrams

# Chapter 2. Technology stack description

This chapter provides detailed information regarding the important tools used in this project and how they helped in implementing the scope of the project.

## 2.1. IntelliJ IDEA

How does it work? IntelliJ IDEA is not only like a code instrument, but it is an entire platform that is meant to optimize all development processes. It provides a smooth and complete environment for numerous programming languages including Java, Kotlin, Python, and others with its intuitive interface and full suite of kit.

A good thing about it is its great debugging function that allows programmers detect and eliminate bugs in an easy and quick way. The integrated debugger provides the real-time insights on the code behaviour which helps the developers detect faults and amend them with the accuracy. Actually, IntelliJ IDEA's powerful analysis tools give a breakthrough analysis into code, performance and code quality as well as potential vulnerabilities hence developers can be proactive in code enhancement. It also enables seamless collaborations by integrating with popular collaboration tools such as Git, GitHub, and Bitbucket hence making the version control and team collaboration possible.

In fact, IntelliJ IDEA is no longer just an IDE; It is a powerful ally that spotlights creative potential and turns it into outstanding software solutions.

Installing IntelliJ IDEA is a straightforward process:

1. Download IntelliJ IDEA: Visit the official JetBrains website and download the needed (Windows, macOS, or Linux).
2. Run the Installer.
3. Select Installation Options: During the installation process, you may be prompted to select installation options such as the installation location and creating desktop shortcuts. Customize these options according to your preferences.
4. Complete the Installation: After configuring the installation options, proceed with the installation process. Once completed, you'll have IntelliJ IDEA installed on your system.

Running a project in IntelliJ IDEA:

1. Open IntelliJ IDEA: Launch IntelliJ IDEA by double-clicking on the desktop shortcut or selecting it from the installed applications list.
2. Open or Create a Project: If you already have a project, you can open it by selecting "File" > "Open" and navigating to the project directory. If you're starting a new project, select "File" > "New" > "Project" and follow the prompts to create a new project.
3. Configure Project Settings (if necessary): Depending on the type of project you're working on, you may need to configure project settings such as SDKs, libraries, and frameworks. IntelliJ IDEA provides wizards and tools to assist with this configuration.
4. Run the Project: Once your project is open, navigate to the main class or file containing the entry point of your application. Right-click on the file and select "Run <Your\_Project\_Name>".
5. View Output: IntelliJ IDEA will compile and run your project. You can view the output, including any errors or log messages, in the "Run" or "Console" tab at the bottom of the IntelliJ IDEA window.

## 2.2. Sourcetree

SourceTree is a good open source Git client with a good user interface and it offers users a smooth experience while managing Git repositories. SourceTree, which is the invention of Atlassian, is designed to deal with the complexities of version control by use of a visual interface through which users effortlessly bind to their Git operations such as commit, push, pull and merge. Its easy to use, together with other properties such as visual diffing and branching, make both beginner and experienced developers able to sail through their projects with ease. And it can be used with either git or Mercurial repositories. Whether you are a solo worker or part of a group, SourceTree will simplify version control and make it easy to get the best out of it for all individuals.

Every time a user makes certain changes in the code, he/she has the opportunity to do the commit operations, then push to the branch on which he/she is working on a certain task in the project. When the user wants to merge all the code on his branch with the source code, he makes a pull request.

Creating pull requests is often used in team software project development. Team members reviewing the code in the pull request have the opportunity to see the differences between the previous code and the newly written code that is to be changed and added to the source code. The pull request process, followed by the review, ends with the merge of the code to the source code located on the main branch, usually called master or main.

A screenshot of a computer

Description automatically generated

Fig. 1 Sourcetree interface (5)

## 2.3. Java language

Java is the language which is valued for its ease and convenience of using, and also for its stability, reliability and robustness. Java, invented by Sun Microsystems in the mid of the 1990s, has nowadays grown into one of the most used languages which are able to develop different type of apps, such as enterprise-level software, mobile apps and web apps. Java independence from the hardware types is provided by the Java Virtual Machine (JVM), and as a result the same code can run on any device as long as it has a JVM. This makes Java to be the ideal choice for the cross-platform development. The strength of Java as a language lies in its engineering through object-oriented programming principles which makes it very suitable for developing modular and scalable code base to be used in various where have been set up records.

The language syntax is similar to that of C, C++, C#, using:

* semicolon to mark the end of a line of code
* braces to group multiple lines of code that form a method, class, or namespace
* square brackets to define a vector or matrix

When we create a Java application, for example a console type, the Main.java class is created automatically and has the following appearance:

A screenshot of a computer

Description automatically generated

Fig. 2 Java Main class

* *public class HelloWorld {:* The next line states the helloworld class. The Java style is determined by the very first class definition, and the file name should also match the class name. The keyword public thus implies that the class is available for use from other classes.
* *public static void main(String[] args) {:* The first sentence tells that the method being executed is a main method which as primary entry of the Java program. It is a unique approach that the JVM emergency procedures for launching the program execution. "public" indicates that it is possible for a method to be accessible outside the class during runtime. "static" is the term used to refer to the fact that the particular method is attributed to the class as a whole, whereas "instance" means that these are methods of the class's instances. there is no return value stated in the signature of this method. The function name is main, and the String[] args is the parameter to main method, it is an array string of object that can hold arguments from the command line.
* *System.out.println("Hello, World!");*: The line displays the following string "*Hello, World!*" to console. System is the predefined class in the Java library. “out” is a static member of the “System” class, which represents the standard output stream. println is a method of the PrintStream class (to which out refers), used to print a line of text. "*Hello, World!*" is the string that will be printed to the console. The semicolon ; terminates the statement, indicating the end of the line of code.

### 2.3.1. Comparison with other programming languages

The list with the most globally used programming languages in 2023 are the following:

A graph of a number of blue and white bars

Description automatically generated with medium confidence

Fig. 3 The most globally used programming languages in 2023 (6)

Java is often compared with programming languages like Javascript, C++ and C#. Some comparison between these programming languages are described below:

* Java vs. C++:
  + Memory Management: Java has automatic memory management (garbage collection), while C++ requires manual memory allocation and deallocation.
  + Performance: C++ offers better performance than Java, as it is closer to the hardware and allows for more control over memory and CPU usage.
  + Language Complexity: C++ is considered more complex than Java, with features like pointers, operator overloading, and multiple inheritance, whereas Java emphasizes simplicity and safety.
* Java vs. C#:
  + Platform: Java is designed to be platform-independent through its "write once, run anywhere" approach with the Java Virtual Machine (JVM). On the other hand, C# primarily targets the Windows platform, although it's possible to use it with other platforms through Mono or .NET Core.
  + Syntax: Both languages have similar syntax, owing to their shared heritage from C and C++. However, C# tends to have more modern language features, such as LINQ (Language Integrated Query) and properties, which can lead to more concise code compared to Java.
  + Memory Management: Both languages use automatic memory management (garbage collection), which simplifies memory management for developers.
  + Concurrency: Both languages provide support for multithreading and concurrency. Java offers features like synchronized blocks and the java.util.concurrent package, while C# has built-in support for asynchronous programming with the async/await keywords.

## 2.4. jMetal library

jMetal is a framework based on Java language which is created for multi-objective optimization (MOO). It includes metaheuristic algorithms and provides a wide range of tools and features to solve optimization problems with conflicting objectives. Some details about jMetal, including why it's used, its performance, and aspects related to its development can be described as the following:

* Multi-objective optimization support: The major claim for jMetal toolkit is its competence to solve problems with more than one conflicting objectives, which is very common in a real world.
* Extensive algorithm library: It brings together the different species of metaheuristic algorithms like NSGA-II, NSGA-III, MOEA/D, SPEA2 and so on
* Flexibility and extensibility: the jMetal architecture possesses high flexibility and extendibility that makes it easy to interject the new problems, algorithms, operators, and other components in the system.

* Open-source and community-driven: Like being open source, the jMetal is the platform for the two-way communication and contributions from the optimization research community for the energetic research progress.

* Comprehensive documentation and examples: It has a well designed documentation, tutorial, and samples that could be very useful during learning and applied usage of the framework.

* Scalability: jMetal is a form of a software which could transfer those optimization problems of varying degree of complexities as well as boundaries. Hence, its ability to handle both small and large scale applications is its many key features.

* Efficiency: The performance of some algorithms may differ due to the fact that the tasks are different in their nature, and jMetal essentially tries to provide timesaving and effective implementations of generic algorithms for multiple objective problems.
* Parallelization support: This is because some jMetal algorithms actually give some input support that is needed for users to profit from parallel computing resource so as to provide faster convergence and improved performances.

* Continuous improvement: As the jMetal community is focused on the performance and efficiency enhancements, each release includes the featured optimization and modifications to be implemented as follows.

* Modularity and abstraction: jMetal introduces a modular architecture, and it provides a clear structure that allows developers to understand the metaheuristic search components, allowing easy extension and maintenance.

* Compatibility: It works with the multiple Java development platforms of IntelliJ IDEA, Eclipse, and others, so, the developers from different levels have an equal chance to learn it.

The current project structure is the following:

A table of text on a white background

Description automatically generated

Fig. 4 Project development structure (7)

### 2.4.1. How to install and use it?

jMetal can be free downloaded from the official website [jMetal-github](https://github.com/jMetal/jMetal). For adding it to the project:

* Create a folder named "main" within the "src" folder of your project.
* Inside the "main" folder, create a subfolder named "java".
* Move the downloaded JAR files into the newly created "java" folder.

A screenshot of a computer

Description automatically generated

Fig. 5 jMetal library structure

Using the jMetal library in the project is easy as it only requires to import the needed files when a new class is created and extends another class from the library to use its facilities.

A screenshot of a computer program

Description automatically generated

Fig. 6 SMPSO algorithm class

To run the developed algorithms, a MainClass has to be implemented. Here, we can run the implementation and generate the needed results.

A computer screen shot of a program

Description automatically generated

Fig. 7 MainClass.java

For the purpose of this dissertation project, the main files that are needed for the implementation are the following:

* NSGAII algorithm, which is placed in the multiobjective directory of jMetal library
* SMPSO algorithm, which is also placed in the multiobjective directory of jMetal library
* AbstractEvolutionaryAlgorithm class
* Algorithm class
* TODO: to be added after the implementation

A diagram of a computer

Description automatically generated

Fig. 8 UML class diagram of jMetal (7)

### 2.4.2. Algorithm types

jMetal library contain information and development regarding many algorithms which are divided into the following categories:

* Multi-objective algorithms
* Single-objective algorithms

These two categories of algorithms are also divided into other algorithms types, for example:

* Evolutionary algorithm
* Genetic algorithm
* Swarm-based algorithm
* Parallel algorithm
* Differential evolution algorithm

The following two images display the algorithms and categories find in the jMetal library:

A screenshot of a computer

Description automatically generated

Fig. 9 Single-objective algorithms (7)

A table with text and images

Description automatically generated with medium confidence

Fig. 10 Multi-objective algorithms (7)

In the context of this paper work, NSGAII and SMPSO algorithms were used to develop the scope of this project.

### 2.4.3. Quality Indicators

jMetal provides a suite of quality indicators that are essential for comparing the performance of algorithms use solve multi-objective optimization. These indicators help evaluating various aspects of algorithm generated solution sets, such as convergence, diversity, and overall solution quality relative to Pareto front.

The following metrics (quality indicators) are used in this version of jMetal:

* Generational distance: illustrates how close the algorithm solution is to the actual optimal solutions. (8)
* Inverted generational distance: Similar to GD, but it also considers the diversity of the solutions by measuring the distance from the reference Pareto front to the solutions. Lower IGD values indicate better convergence and diversity.
* Hypervolume: measures the space covered by the solutions found by the algorithm. A higher hypervolume value means the solutions provided by the algorithm are closer to the set of optimal solutions which is ideal.
  + PISA Hypervolume: Provides an accuracy-efficiency trade-off, and it is useful when we have only two or three objectives. However, its performance decreases with a higher-dimensional objective space.
  + WFGH Hypervolume: Simultaneously tackles problems of many objectives and is preferred for its effectiveness and the capability to handle objective spaces which are big without problematic loss in performance. Normalized hypervolume: A variant of the hypervolume indicator that is normalized based on a reference point, making it easier to compare across different problem instances.
* Epsilon: measures how much the set of optimal solutions needs to be adjusted to match the reference optimal set. Lower values indicate better performance.
* Generalized spread: evaluates how evenly and widely the solutions are spread across the entire range of objectives on the Pareto front.
* Error ratio: The error rate measures how many solutions the algorithm identified as optimal (Pareto front) are actually not part of the true optimal set. The 0 error rate means all identified solutions are truly optimal, which is the best outcome, while the 1 error rate means none of the identified solutions are truly optimal, indicating poor performance. (7)
* Set coverage: compares two sets of solutions by calculating the solutions in percentage in one set that are either equal to solutions in the other set or dominated by the solutions. This helps determine how one set of solutions performs relative to the other.

### 2.4.4. Problems

Problems in jMetal represent the optimization challenges that algorithms are designed to solve. They are defined by their decision variables, objective functions, and constraints, providing a structured way to evaluate the performance of different optimization strategies.

Types of Problems

Single-objective Problems: These focus on optimizing one specific goal. The aim is to either maximize or minimize this single objective.

Multi-objective Problems: These involve optimizing two or more conflicting goals at the same time. The aim is finding solutions providing a good balance through the objectives, known as the Pareto front. (9)

A screenshot of a computer

Description automatically generated

Fig. 11 jMetal problem structure

Examples of problems:

jMetal offers a variety of benchmark problems that the optimization community uses to test and compare different algorithms. Examples include:

* ZDT (Zitzler, Deb, and Thiele) Series: Represents a set of synthetic problems with varying characteristics, such as the form of Pareto front and and how solutions are distributed.
* DTLZ (Deb, Thiele, Laumanns, and Zitzler) Series: Another set of synthetic problems designed for testing algorithms on many-objective optimization tasks.

All these problems implement the Problem interface which has the following structure:

A screenshot of a computer code

Description automatically generated

Fig. 12 Problem interface in jMetal (7)

# Chapter 3. Theoretical part of the implementation

## 3.1. Pareto concept

The Pareto concept, named after economist Vilfredo Pareto, is fundamental to multi-objective optimization and widely used in economics, engineering, and decision-making. It is used for improving part of the solution that cannot be done without worsening another part. The efficiency of this method is found in the following concepts:

* Pareto optimality: A solution is Pareto efficient if no alternative solution can enhance any objective without deteriorating at least one other objective. These solutions are considered efficient or non-dominated. (10)
* Pareto dominance:

A solution X can outperform another solution Y if X is at least as effective as Y in all objectives and superior in at least one. In a minimization problem, for any two solutions X and Y:

* + If f1(X) ≤ f1(Y) and f2(X) < f2(Y), then X dominates Y.
  + If f1(X) < f1(Y) and f2(X) ≤ f2(Y), then X dominates Y.
  + If neither X dominates Y nor Y dominates X, then they are non-dominated. (10)
* Pareto front: represents the collection of all best solutions of Pareto. It shows the adjustments between different conflicting objectives. Choosing any point on the Pareto front means that improving one objective will inevitably worsen at least one other objective. (10)

Defining the Pareto front through the following graphic:

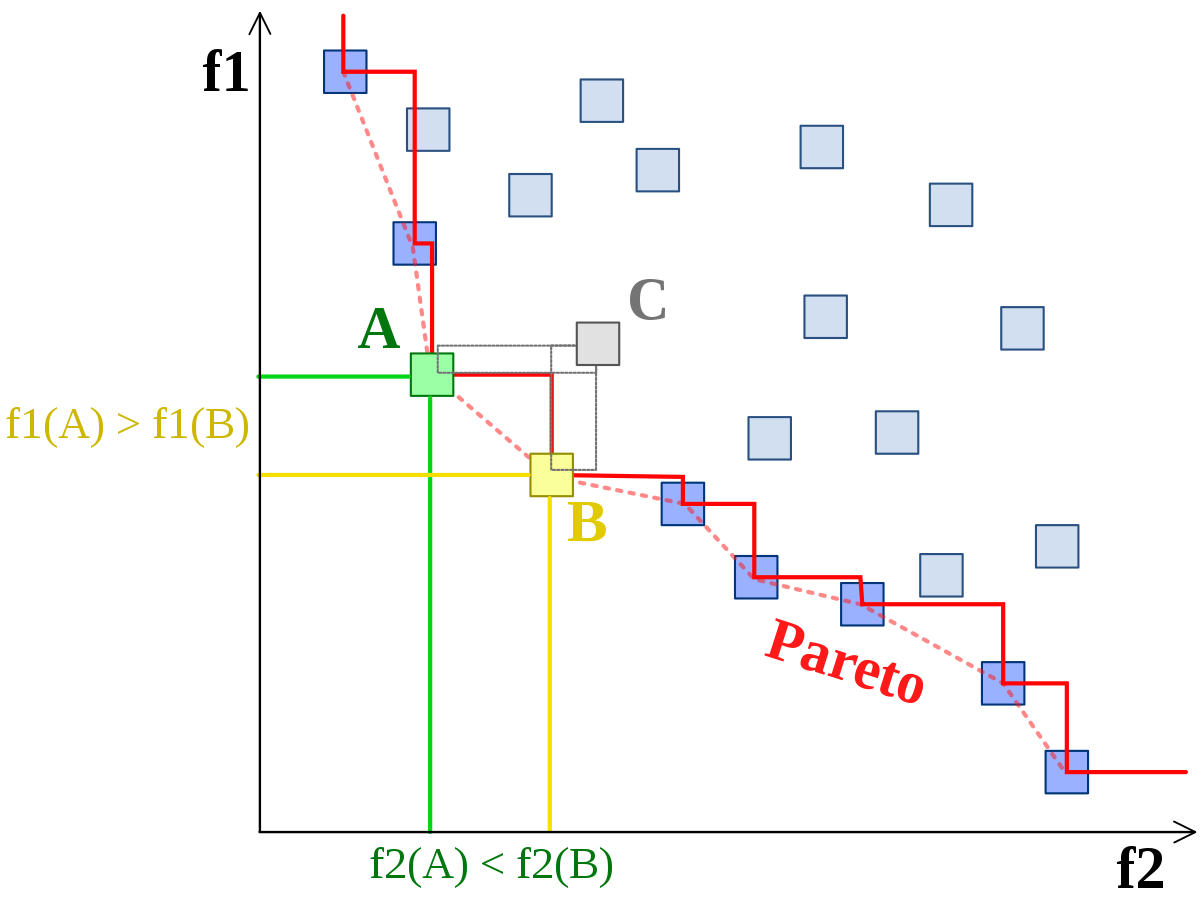


Fig. 13 Pareto front (11)

### 3.1.1. Multi-objective genetic algorithms (2)

A genetic algorithm begins with a starting group of individuals, often generated randomly, and continues until a termination condition is satisfied (like finding a satisfactory solution or hitting a time limit). After assessing the initial group, the algorithm proceeds to its main cycle. If the termination condition has not been fulfilled, the selection phase begins, where specific individuals are chosen to become parents. These parents generate offspring via crossover (recombination) and/or mutation. The offspring make up the new population, which is then evaluated. Depending on the algorithm, the next parent population may consist of these offspring, or a combination of offspring and previous parents (e.g., through elitism). (12)

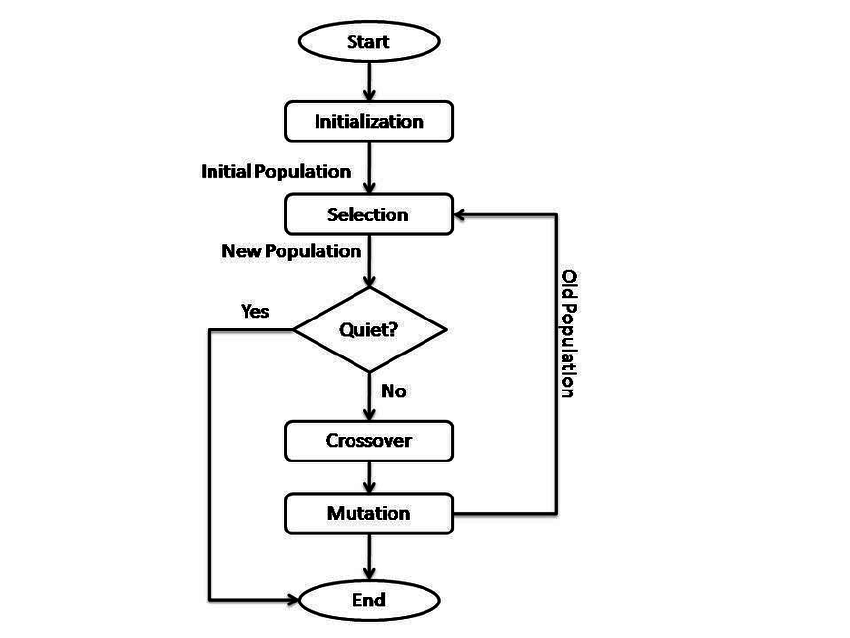


Fig. 14 Genetic algorithm (12)

### 3.1.1.1. Non-dominated Sorting Genetic Algorithm (12)

The Non-dominated Sorting Genetic Algorithm (NSGA) is part of the multi-objective optimization algorithms derived from natural selection and evolution. It is implemented to tackle problems with multiple conflicting goals and is widely used in engineering, economics, and other areas where decisions must balance various criteria.

At its core, NSGA maintains a group of potential solutions, referred to as individuals or chromosomes, each embodying a possible solution to the optimization problem. Unlike traditional single-objective genetic algorithms that evaluate solutions based on one objective, NSGA evaluates them based on multiple objectives at the same time. This helps NSGA discover solutions that aren't outperformed by any alternative solution in any of the objectives., known as the Pareto-optimal front. (2)

Key innovation of NSGA is represented by its unique method of selection and reproduction. Traditional genetic algorithms use fitness proportionate selection, where individuals are chosen for reproduction based on their relative fitness. NSGA, on the other hand, uses non-dominated sorting to rank individuals according to how they compare to others in the population. This process sorts the population into several groups or "fronts." The first front consists of solutions that are non-dominated and not outperformed by any alternative solutions in the group. (12)

After sorting, NSGA uses a mix of elitist selection and diversity preservation to create the next generation of individuals. Elitist selection guarantees that the top solutions from the present group are preserved for the next generation. Meanwhile, diversity preservation methods, like crowding distance, encourage exploring the solution space and avoid the algorithm from becoming blocked in suboptimal solutions.

In the reproduction phase of NSGA, offspring are created by using genetic operators like crossover and mutation to selected parent individuals. These operators help explore and utilize the solution space, enabling NSGA to produce a variety of high-quality solutions over successive generations.

### 3.1.1.2. Non-dominated Sorting Genetic Algorithm II

NSGA-II (Nondominated Sorting Genetic Algorithm II) is a type of genetic algorithm within the evolutionary algorithm’s family, designed specifically for addressing multi-objective optimization challenges. It’s an enhanced version of the original NSGA (Nondominated Sorting Genetic Algorithm), with enhancements to boost its performance and convergence.

NSGA-II works on principles inspired by natural selection and genetics. It keeps a population of solutions which are potential, often called individuals or chromosomes, and evolves this population over multiple generations. The objective is to identify solutions that form the Pareto-optimal front, where no solution can be improved in one objective without compromising another.

A diagram of a flowchart

Description automatically generatedFig. 15 NSGA II algorithm process (12)

The main components of the algorithm which make the difference between NSGA and its own are the following:

Step 1: Calculation of Dominance Counter

For every individual in the population, a dominance counter is calculated. This counter shows how many other individuals in the population outperform the current individual.

Step 2: Identification of Pareto Fronts

After calculating the dominance counters in Step 1, all individuals in the first Pareto front (Front I) will have a dominance counter of 0, as no other individual in the population outperforms them. For each individual in Front I, the algorithm finds the individuals they dominate and decreases their dominance counters by 1. This process identifies who is dominated by those in Front I and adjusts their counters accordingly. Once this is done, the individuals with a dominance counter of 0 will form the second Pareto front (Front II).

Iterative Process for All Fronts

This process of identifying Pareto fronts is repeated for each subsequent front. For each front, the algorithm finds individuals with a dominance counter of 0, and these individuals form the next Pareto front. The dominance counters of individuals dominated by those in the current front are decreased. This continues until every individual in the population has been assigned to a Pareto front.

Termination of the Process

Termination process continues until there are no individuals left in the population to be assigned to the Pareto front. At this point, the entire population has been divided into fronts based on their Pareto dominance relationships. This marks the end of the process, and the algorithm proceeds to the next stage, typically involving the selection and reproduction of individuals to generate the next generation of solutions.

NSGA-II improves on the original NSGA by addressing issues like high computational complexity and the tendency to produce too many solutions in crowded areas of the Pareto front. It offers better convergence and maintains diversity, making it a popular option for tackling multi-objective optimization issues in diverse domains.

This algorithm has a computational complexity of O(MN^2), where M stands for the number of objectives and N signifies the number of individuals in the population. While this adjustment lowers the computational complexity, it increases the memory usage from O(N) in NSGA to O(N^2) in NSGA-II. (1)

## 3.2. Particle Swarm Optimization algorithms (13)

The Particle Swarm Optimization (PSO) algorithms represent nature-inspired optimization techniques that are inspired by organisms, particularly birds or fish schooling. (4)

In this algorithm, a group of potential solutions, known as particles, moves through the entire search space by adjusting their positions based on their own experiences and the experiences of their neighbors. Every particle's movement is decided by two primary factors:

• its optimal individual position and

• the best overall position

By continuously updating their positions, particles explore the search space and gradually move towards optimal solutions. PSO algorithms are popular because they are simple, efficient, and capable of handling complex, high-dimensional problems. They have been successfully used in areas like engineering, finance, and machine learning, where they are excellent at finding nearly optimal solutions for difficult optimization challenges.

### 3.2.1. Particle Swarm Optimization algorithm description

Particle Swarm Optimization (PSO) uses a group approach to find the best solutions. Particle Swarm Optimization (PSO), developed by Kennedy and Eberhart in 1995, has achieved widespread recognition because of its simplicity, efficiency, and effectiveness. (4)

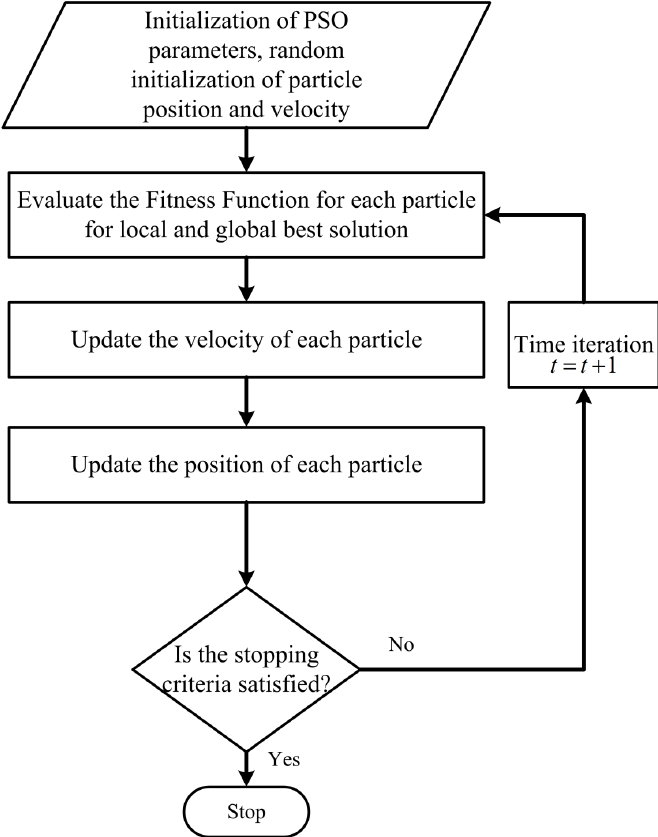


Fig. 16 PSO algorithm process (13)

Initialization

The algorithm begins by creating a collection of particles, everyone representing a possible resolution to the problem. These particles are spread out randomly throughout the exploration area, and every particle is given a random direction and speed.

Fitness Evaluation

After initialization, each particle's fitness is assessed using a performance metric that evaluates the quality of the solution it represents. This goal function is unique to the problem and encapsulates the optimization targets.

Initialization of Particle's Memory

Each particle maintains records of two key positions: its personal best position (Pbest) and the overall best position (Gbest). At the start, both of these positions are set to where the particle is currently located.

Particle Movement

In each cycle of the algorithm, particles adjust their positions and speeds based on their current, personal and the global best position. The motion of each particle is influenced by two main factors:

* Cognitive element: This element directs the particle toward its individual optimal position, encouraging exploitation of promising areas of the search domain.
* Social element: This element directs the particle towards the optimal position discovered by any other particle in the swarm, facilitating exploration of the entire search space.

The new velocity of each particle is calculated using a combination of its current velocity, cognitive component, and social component. The particle then updates its position based on the new velocity, potentially moving towards more promising regions of the search space.

Updating Particle's Memory

After moving, each particle checks if its new position is better than its previous personal best (Pbest). If it is, the particle updates its Pbest to this new position. Additionally, the personal best position Pbest is compared to the global best position Gbest and if a particle's Pbest is better than the current Gbest, the Gbest is updated to this new, better position.

Termination Criterion

The PSO algorithm continues iterating up to the point when the stopping condition is met. Common reasons to stop are getting to reach the maximum possible number of iterations, finding a good enough solution, or hitting a predefined computational limit.

### 3.2.2. Strength Pareto Multi-objective Particle Swarm Optimization algorithm

The SMPSO algorithm is an improved version of the classic Particle Swarm Optimization (PSO) algorithm. (4) Introduced by Durillo and Nebro in 2009, SMPSO enhances the traditional PSO by including Pareto dominance and methods to maintain diversity among solutions.

In SMPSO, particles move through the search space by adjusting their positions, while also taking into account Pareto dominance among solutions.

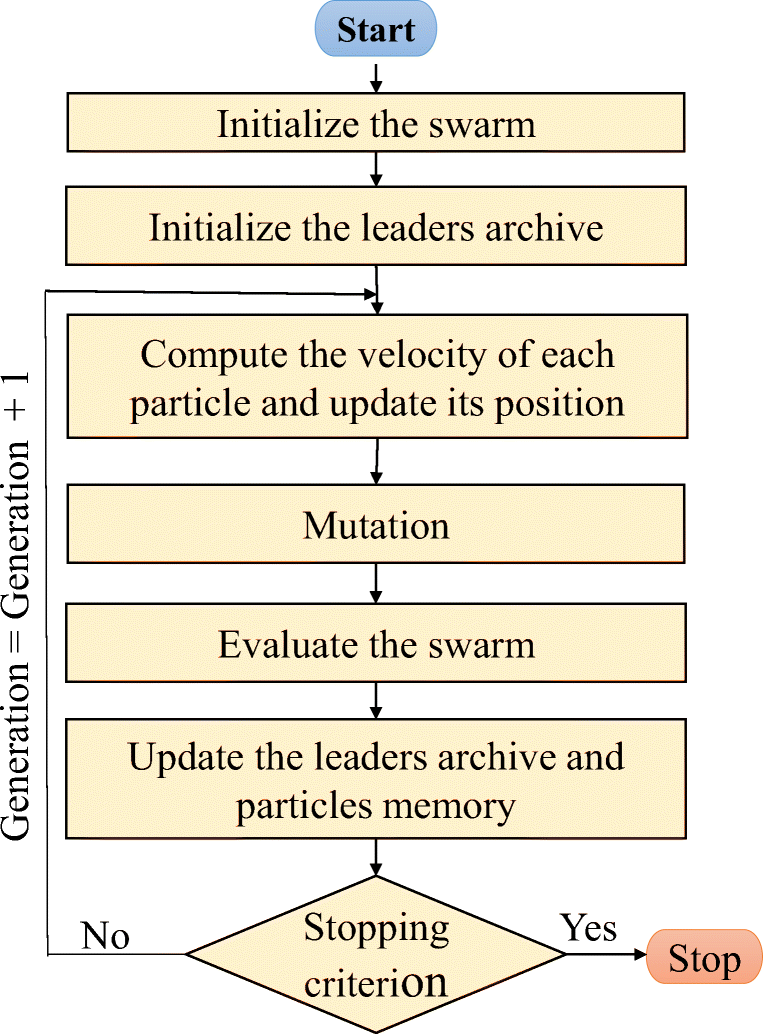


Fig. 17 SMPSO algorithm process (14)

The comparison between PSO and SMPSO in the algorithm process are the following:

Initialization:

Like PSO, SMPSO begins with a random initialization of a group of particles within the search space. Each particle is represented by a possible solution to the multi-objective optimization problem. (4)

Fitness Evaluation:

Unlike PSO, which evaluates fitness based on a single objective, SMPSO assesses each particle's fitness using multiple objective functions at the same time. This enables SMPSO to tackle multi-objective optimization problems, optimizing solutions with respect to conflicting goals.

Pareto Dominance:

In SMPSO, particles keep a set of personal best solutions (Pbest) and a global best solution (Gbest). Unlike PSO, which compares solutions using a single fitness value, SMPSO uses Pareto dominance to evaluate solutions. SMPSO ensures that the population only includes non-dominated solutions, forming the Pareto front, by maintaining these dominance relationships.

Velocity and Position Update:

In SMPSO, the equations for updating velocity and position are similar to those in PSO but are modified to include Pareto dominance and diversity preservation mechanisms. Particles adjust their velocities based on their personal and global best solutions, as well as their interactions with other particles in the swarm. However, in SMPSO, the velocity update also takes into account Pareto dominance relationships among solutions to steer the search towards non-dominated areas of the objective space.

Diversity Preservation:

A major difference between SMPSO and classic PSO is the inclusion of diversity preservation mechanisms in SMPSO. With these mechanisms, the exploration of the entire Pareto front can promote solution diversity. Methods like crowding distance or grid-based diversity measures are used to maintain a varied group of solutions that are non-dominated in the population.

Termination Criterion:

The termination criteria for SMPSO are usually similar to those for PSO. The algorithm stops running when a stop condition is reached, like a maximum number of iterations, finding a good set of non-dominated solutions, or exceeding a computational limit.

Compared to classic PSO, SMPSO offers several advantages for solving these multi-objective optimization problems.

Firstly, SMPSO handles problems with multiple conflicting objectives more effectively by keeping a diverse set of non-dominated solutions along the Pareto front. Additionally, SMPSO promotes thorough exploration of the entire Pareto front through diversity preservation mechanisms, whereas PSO may have difficulty adequately exploring the objective space in multi-objective scenarios. (15)

As a conclusion, SMPSO is better suited for multi-objective optimization tasks, offering improved convergence towards Pareto-optimal solutions and better exploration of the solution space compared to classic PSO.

## 3.3. Algorithm’s comparisons

Evolutionary algorithms for multi-objective optimization generate sets of solutions that often cannot be easily ordered to determine the best one, so comparing the performance of exploration algorithms becomes a complex task. To enhance objectivity in the comparison process, it is necessary that experiments involving heuristic algorithms adhere to standardized experimental procedures. These procedures typically involve the following steps:

* defining the objectives or aims of the experiment.
* selecting appropriate performance measures, referred to as metrics, to evaluate the algorithms' performance.
* designing and executing the experiment according to established protocols.
* analyzing the collected data and drawing meaningful conclusions based on the results.
* publishing the outcomes and conclusions of the experiment to contribute to the collective understanding of algorithm performance.

While delving into the theoretical aspects and providing specific examples of experimental procedures exceeds the scope of this work, it is essential to outline the metrics used for evaluating the algorithms. These metrics serve as quantitative indicators to assess various aspects of algorithm performance and effectiveness. These metrics are also described above in the chapter [2.4.3. Quality Indicators](#_2.4.3._Quality_Indicators).

For the purpose of this project, the following metrics were developed:

* Coverage of two sets:

The one that compares the performances of two tested algorithms by determining the proportion of individuals from the Pareto front determined by one algorithm that are dominated by individuals from the Pareto front.

A = individuals from the Pareto front of the first algorithm, and

B = individuals from the Pareto front of the second algorithm.

Let ai denote the individual i from set A, and bj denote the individual j from set B. We define the coverage function, C(A, B) as follows:

*C*(*A*,*B*)=∣*B*∣∑*bj*∈*B*​*ai*​∈*A* where that *ai*​ dominates *bj*​​

From the formula, it is evident that the metric takes values in the interval [0, 1]. For example, if

C(A, B) = 0.5, it shows that a half of individuals from set B are dominated by individuals from set A. However, for accuracy, it is necessary to calculate C(B, A) as well, because C(A, B) = 1 - C(B, A). (9)

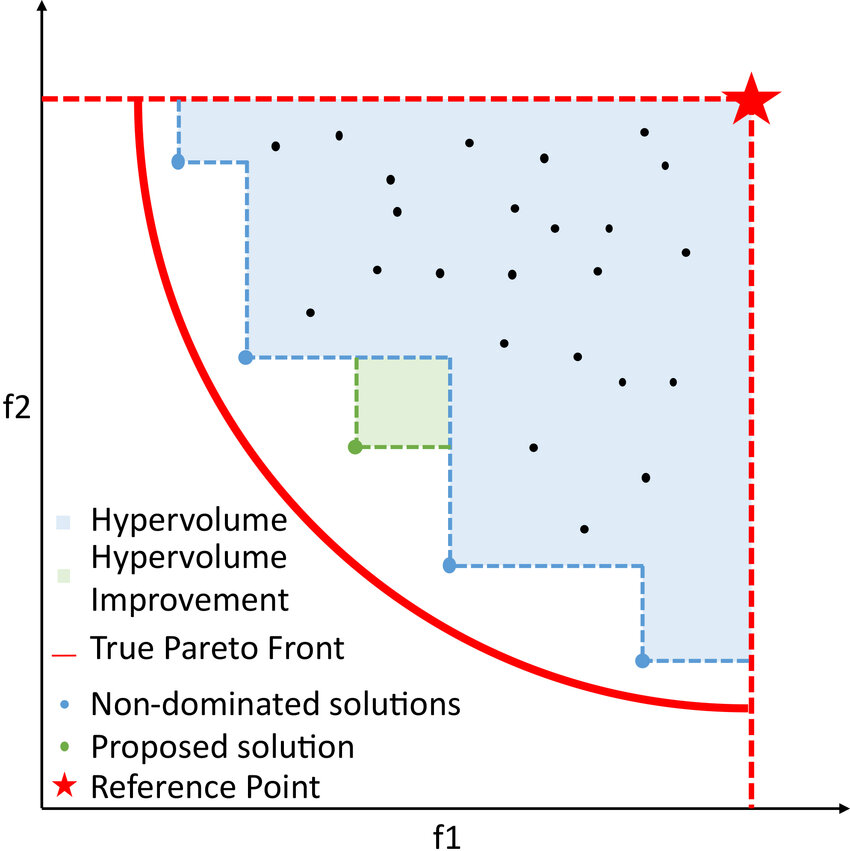
* Hypervolume: measures the volume of the objective space enclosed between the Pareto front and a reference point, with the reference point representing the worst possible solution for each objective.

Fig. 18 Hypervolume of a convex minimization problem (16)

# Chapter 4. Integration of theoretical notions into practical part of the application

## Project architecture

A diagram of a simulation

Description automatically generated

Fig. 19 Architecture of the application

## NSGAII algorithm

The NSGAII algorithm was developed through jMetal library, which was integrated in the project. The algorithm described at chapter 3.1.1.2. has the following diagram representation:

A screenshot of a computer

Description automatically generated

Fig. 20 NSGAII UML diagram

## SMPSO algorithm

A screenshot of a computer screen

Description automatically generatedThe SMPSO algorithm was developed through jMetal library, which was integrated in the project. The algorithm described at chapter 3.2.2 has the following diagram representation:

Fig. 21 SMPSO UML diagram

## Superposition of the algorithms

Multi-objective evolutionary optimization (MOEO) is a powerful method for solving problems that have multiple conflicting goals. In MOEO, a group of potential solutions evolves over time, with each solution balancing different objectives. The goal is to find solutions (individuals) that are closer to the Pareto-optimal front, meaning no solution is better in all aspects than any other. The main challenge is balancing how quickly the solutions improve (convergence speed) with maintaining a variety of solutions (diversity). To tackle this, a new approach called superposition of populations has been introduced to enhance MOEO.

A few advantages of the superposition are the following ones:

* Improved diversity:
* Multi-objective algorithms are used to maintain a diverse group of solutions to comprehensively cover the Pareto front. NSGA-II accomplishes this by utilizing non-dominated sorting and the crowding distance mechanisms, that help distribute the solutions effectively.
* SMPSO uses a crowding distance mechanism and a velocity clamping technique to maintain diversity, ensuring that particles explore various regions of the search space.
* Improves Convergence:
* While maintaining diversity, multi-objective algorithms aim to converge toward the Pareto front. NSGA-II achieves this balance through elitism, crowding distance sorting, and tournament selection, promoting convergence toward the true Pareto front.
* SMPSO incorporates a speed constraint mechanism that prevents particles from moving too fast, allowing for better convergence towards the Pareto front.
* Scalability to high-dimensional problems:
* Multi-objective algorithms are designed to handle high-dimensional search spaces effectively. Both NSGA-II and SMPSO can efficiently handle problems with numerous decision variables by thoroughly exploring the search space.
* Both NSGA-II and SMPSO are population-based algorithms that maintain a diverse set of solutions, which can help in exploring high-dimensional search spaces more thoroughly.
* Potential for parallel implementation:
* Multi-objective algorithms can be parallelized to exploit modern computing architectures effectively. NSGA-II and SMPSO can benefit from parallelization techniques such as parallel evaluation of solutions, parallel selection, and parallel update of the population.
* Parallel implementations of NSGA-II and SMPSO have been explored in various research studies, leveraging multi-core CPUs, GPU computing, and distributed computing environments for improved performance.

These characteristics make multi-objective algorithms like NSGA-II and SMPSO ideal for tackling intricate optimization challenges with numerous conflicting goals. By balancing diversity and convergence while being scalable and amenable to parallel implementation, these algorithms offer efficient and effective solutions across various domains.

To explain better the superposition concept, in the followings subchapters, the implementation of NSGAII and SMPSO superposition is described.

## Superposition structure

For implementing the superposition concept, the following classes were added in the project:

* AbstractSuperPositionGA
* AbstractSuperPositionPSO
* AbstractParticleSwarm
* AbstractGeneticAlgorithm
* SuperPositionNSGAII
* SuperPositionSMPSO
* SuperPositionCombinator1
* AbstractSuperPositionGAPSOCombinator1
* GeneticAlgorithmWrapper
* ParticleSwarmWrapper

### AbstractGeneticAlgorithm class

In jMetal 6.0, the AbstractGeneticAlgorithm class serves as the base class for genetic algorithm implementations. It provides common functionality and structure for implementing various genetic algorithms:

A screenshot of a computer

Description automatically generated

Fig. 22 AbstractGeneticAlgorithm UML diagram

1. Class Structure: The AbstractGeneticAlgorithm class is typically an abstract class, meaning it cannot be instantiated directly. It contains common attributes and methods shared by genetic algorithms.
2. Attributes:
   * problem: A reference to the problem being solved. It encapsulates the optimization problem and its characteristics.
   * population: The current candidate solutions.
   * selectionOperator: The operator used to select parents for reproduction.
   * crossoverOperator: The operator used to create offspring solutions from selected parents.
   * mutationOperator: The operator used to introduce variability in the population.
   * Other parameters related to algorithm configuration, such as population size, probability parameters, etc.
3. Methods:
   * setMaxPopulationSize: Sets the maximum size of the population for the genetic algorithm.
   * getMaxPopulationSize: Returns the maximum size of the population.
   * getSelectionOperator: Returns the operator used by the algorithm. The selection operator determines which individuals are chosen for reproduction.
   * getCrossoverOperator: Returns the operator used by the algorithm that combines pairs of individuals to produce offspring.
   * getMutationOperator: Returns the operator used by the algorithm. The mutation operator introduces variations by altering individual offspring.
   * AbstractGeneticAlgorithm **(constructor)**: Initializes the genetic algorithm with a specified problem to solve. It sets up the problem instance that the algorithm will work on.
   * createInitialPopulation: Creates the starting group of individuals for the algorithm. It generates a list of solutions to start the algorithm.
   * selection: Applies the selection operator iteratively to form the mating pool from the initial set. This pool is used for creating offspring.
   * reproduction: Applies the crossover and mutation operators iteratively to create the offspring population from the mating pool. It ensures that the population size remains consistent and valid.
   * checkNumberOfParents: Verifies that the population size is divisible by the number of parents required by the crossover operator. It ensures the population has the correct number of parents for valid crossover operations.
   * Other helper methods for logging, statistical analysis, etc.

### 4.4.1.2. GeneticAlgorithmWrapper class

This class is a wrapper for a genetic algorithm class to access easier the other private or protected members of the genetic algorithms, such as population, stop condition and so on. Its role is to help further in the process of the superposition implementation. The class has the following code: (7)

|  |
| --- |
| public class GeneticAlgorithmWrapper**<**S**>**  **{**  protected AbstractGeneticAlgorithm**<**S**,** List**<**S**>>** algorithm**;**  public GeneticAlgorithmWrapper**(**AbstractGeneticAlgorithm**<**S**,** List**<**S**>>** algorithm**)**  **{**  **this.**algorithm **=** algorithm**;**  **}**  public List**<**S**>** createInitialPopulation**()**  **{**  **return** algorithm**.**createInitialPopulation**();**  **}**  public List**<**S**>** evaluatePopulation**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**evaluatePopulation**(**population**);**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithm**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** selection**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**selection**(**population**);**  **}**  public List**<**S**>** reproduction**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**reproduction**(**population**);**  **}**  public List**<**S**>** replacement**(**List**<**S**>** population**,** List**<**S**>** offspringPopulation**)**  **{**  **return** algorithm**.**replacement**(**population**,** offspringPopulation**);**  **}**  public void initProgress**()**  **{**  algorithm**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithm**.**updateProgress**();**  **}**  public Problem**<**S**>** getProblem**()**  **{**  **return** algorithm**.**getProblem**();**  **}**  public List**<**S**>** result**()**  **{**  **return** algorithm**.**result**();**  **}**  public int getMaxPopulationSize**(){** **return** algorithm**.**maxPopulationSize**;** **}**  public int getMaxIterations**()** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "maxIterations"**);**  myField**.**setAccessible**(true);**  **return** **(**int**)** myField**.**get**(**algorithm**);**  **}**  public void setIterations**(**int n**)** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "iterations"**);**  myField**.**setAccessible**(true);**  myField**.**set**(**algorithm**,** n**);**  **}**  protected Field getField**(**Class clazz**,** String fieldName**)** **throws** NoSuchFieldException  **{**  **try**  **{**  **return** clazz**.**getDeclaredField**(**fieldName**);**  **}**  **catch** **(**NoSuchFieldException e**)**  **{**  Class superClass **=** clazz**.**getSuperclass**();**  **if** **(**superClass **==** **null)**  **{**  **throw** e**;**  **}**  **else**  **{**  **return** getField**(**superClass**,** fieldName**);**  **}**  **}**  **}**  **}** |

The diagram class is represented in the following image:

A screenshot of a computer program

Description automatically generated

Fig. 23 GeneticAlgorithmWrapper UML diagram



### AbstractSuperPositionGA class

The class structure is very simple: it has to be easily inherited by any other genetic algorithm developed in the jMetal library. With the run() method, the algorithm can create the initial population and then evaluate it. While the stop condition is not met, the selection, reproduction and evaluate processes are executed, so a new generation can be invoked. The class has the following code:

|  |
| --- |
| public abstract class AbstractSuperPositionGA**<**S**,** R**>** **extends** AbstractEvolutionaryAlgorithm**<**S**,** R**>** **implements** Runnable  **{**  protected GeneticAlgorithmWrapper**<**S**>** algorithmWrapper**;**  public volatile boolean isWaiting**;**  public IAction onNewGeneration**;**  public AbstractSuperPositionGA**(**GeneticAlgorithmWrapper**<**S**>** wrapper**)**  **{**  algorithmWrapper **=** wrapper**;**  isWaiting **=** **false;**  setProblem**(**algorithmWrapper**.**getProblem**());**  **}**  @Override  public void run**()**  **{**  List**<**S**>** offspringPopulation**;**  List**<**S**>** matingPopulation**;**  population **=** createInitialPopulation**();**  population **=** evaluatePopulation**(**population**);**  initProgress**();**  **while** **(!**isStoppingConditionReached**())** **{**  matingPopulation **=** selection**(**population**);**  offspringPopulation **=** reproduction**(**matingPopulation**);**  offspringPopulation **=** evaluatePopulation**(**offspringPopulation**);**  population **=** replacement**(**population**,** offspringPopulation**);**  onNewGeneration**.**invoke**();**  updateProgress**();**  isWaiting **=** **true;**  **while** **(**isWaiting**)**  **{**  Thread**.**onSpinWait**();**  **}**  **}**  **}**  @Override  protected List**<**S**>** createInitialPopulation**()**  **{**  **return** algorithmWrapper**.**createInitialPopulation**();**  **}**  @Override  protected List**<**S**>** evaluatePopulation**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**evaluatePopulation**(**population**);**  **}**  @Override  public boolean isStoppingConditionReached**()**  **{**  **return** algorithmWrapper**.**isStoppingConditionReached**();**  **}**  @Override  protected List**<**S**>** selection**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**selection**(**population**);**  **}**  @Override  protected List**<**S**>** reproduction**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**reproduction**(**population**);**  **}**  @Override  protected List**<**S**>** replacement**(**List**<**S**>** population**,** List**<**S**>** offspringPopulation**)**  **{**  **return** algorithmWrapper**.**replacement**(**population**,** offspringPopulation**);**  **}**  @Override  protected void initProgress**()**  **{**  algorithmWrapper**.**initProgress**();**  **}**  @Override  protected void updateProgress**()**  **{**  algorithmWrapper**.**updateProgress**();**  **}**  public int getMaxPopulationSize**()** **{**  **return** algorithmWrapper**.**getMaxPopulationSize**()** **;**  **}**  public void forceStoppingCondition**()**  **try** **{**  int maxIterations **=** algorithmWrapper**.**getMaxIterations**();**  algorithmWrapper**.**setIterations**(**maxIterations**);**  **}**  **catch** **(**Exception ex**)**  **{**  // TODO  **}**  **}**  **}** |

The UML diagram of the class with the other class connections is represented in the following diagram:

A screenshot of a computer program

Description automatically generated

Fig. 24 AbstractSuperPositionGA UML diagram

### 4.4.1.4. AbstractParticleSwarmOptimization class

In jMetal 6.0, the AbstractParticleSwarmOptimization class serves as the base class for particle swarm optimization (PSO) algorithms. It has the following attributes and classes in its development:

1. Attributes:

* swarm: the list of particles (solutions) that make up the swarm in the particle swarm optimization algorithm.

1. Methods:
   * getSwarm: Returns the current swarm of particles.
   * setSwarm: Sets the swarm of particles to a new list.
   * initProgress: An abstract method to initialize any progress-related variables or structures at the beginning of the optimization process.
   * updateProgress: An abstract method to update the progress of the optimization, typically called at each iteration.
   * isStoppingConditionReached: An abstract method that checks whether the stopping condition for the optimization has been met. Returns a boolean indicating whether to stop the optimization.
   * createInitialSwarm: An abstract method to create and return the initial swarm of particles.
   * evaluateSwarm: An abstract method to evaluate the swarm of particles and return the evaluated swarm.
   * initializeLeader: An abstract method to initialize the leader(s) of the swarm, which are typically the best-performing particles.
   * initializeParticlesMemory: An abstract method to set each particle’s memory, that stores the best position found by each particle.
   * initializeVelocity: An abstract method to set each particle’s velocity in the swarm.
   * updateVelocity: An abstract method to update each particle’s velocity based on its current state and the state of the swarm.
   * updatePosition: An abstract method to update particle’s position based on its velocity.
   * perturbation: An abstract method to apply perturbations to the swarm, introducing variations to avoid local optima.
   * updateLeaders: An abstract method to update the leaders of the swarm based on the latest evaluations.
   * updateParticlesMemory: An abstract method to update the memory of each particle with its current state if it is better than its previous state.
   * run: The method implementing the main loop of the particle swarm optimization algorithm. It initializes the swarm, evaluates it, and iteratively updates velocities, positions, and other relevant states until the stopping condition is reached.

A screenshot of a computer

Description automatically generated

Fig. 25 AbstractParticleSwarmOptimization UML class

### 4.4.1.5. ParticleSwarmWrapper class

This class is a wrapper for a PSO algorithm class to access easier the other private or protected members of the genetic algorithms, such as population, stop condition, etc. Its role is to help further in the process of the superposition implementation. It contains the main methods helpful for every particle swarm optimization algorithm that is using it. From creating the initial swarm to the final result. The class has the following code:

|  |
| --- |
| public class ParticleSwarmWrapper**<**S**>**  **{**  protected AbstractParticleSwarmOptimization**<**S**,** List**<**S**>>** algorithm**;**  public ParticleSwarmWrapper**(**AbstractParticleSwarmOptimization**<**S**,** List**<**S**>>** algorithm**)**  **{**  **this.**algorithm **=** algorithm**;**  **}**  public List**<**S**>** createInitialSwarm**()**  **{**  **return** algorithm**.**createInitialSwarm**();**  **}**  public List**<**S**>** evaluateSwarm**(**List**<**S**>** swarm**)**  **{**  **return** algorithm**.**evaluateSwarm**(**swarm**);**  **}**  public void initializeLeader**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeLeader**(**swarm**);**  **}**  public void initializeParticlesMemory**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeParticlesMemory**(**swarm**);**  **}**  public void initializeVelocity**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeVelocity**(**swarm**);**  **}**  public void updateVelocity**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateVelocity**(**swarm**);**  **}**  public void updatePosition**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updatePosition**(**swarm**);**  **}**  public void perturbation**(**List**<**S**>** swarm**)**  **{**  algorithm**.**perturbation**(**swarm**);**  **}**  public void updateLeaders**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateLeaders**(**swarm**);**  **}**  public void updateParticlesMemory**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateParticlesMemory**(**swarm**);**  **}**  public void initProgress**()**  **{**  algorithm**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithm**.**updateProgress**();**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithm**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** result**()**  **{**  **return** algorithm**.**result**();**  **}**  public List**<**S**>** getSwarm**()**  **{**  **return** algorithm**.**getSwarm**();**  **}**  public void setSwarm**(**List**<**S**>** swarm**)**  **{**  algorithm**.**setSwarm**(**swarm**);**  **}**  public int getMaxIterations**()** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "maxIterations"**);**  myField**.**setAccessible**(true);**  **return** **(**int**)** myField**.**get**(**algorithm**);**  **}**  public void setIterations**(**int n**)** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "iterations"**);**  myField**.**setAccessible**(true);**  myField**.**set**(**algorithm**,** n**);**  **}**  protected Field getField**(**Class clazz**,** String fieldName**)** **throws** NoSuchFieldException  **{**  **try**  **{**  **return** clazz**.**getDeclaredField**(**fieldName**);**  **}**  **catch** **(**NoSuchFieldException e**)**  **{**  Class superClass **=** clazz**.**getSuperclass**();**  **if** **(**superClass **==** **null)**  **{**  **throw** e**;**  **}**  **else**  **{**  **return** getField**(**superClass**,** fieldName**);**  **}**  **}**  **}**  **}** |

The class has the following diagram:

A screen shot of a computer

Description automatically generated

Fig. 26 ParticleSwarmWrapper UML diagram

### 4.4.1.6. AbstractSuperPositionPSO

The class structure is very simple: it has to be easily inherited by any other PSO algorithm developed in the jMetal library. This is developed in the same way as AbstractSuperPositionGA described at chapter 4.4.1.3, but for a PSO algorithm. The class has the following code:

|  |
| --- |
| public abstract class AbstractSuperPositionPSO**<**S**,** R**>** **implements** Algorithm**<**R**>**  **{**  protected ParticleSwarmWrapper**<**S**>** algorithmWrapper**;**  public volatile boolean isWaiting**;**  public IAction onNewGeneration**;**  public AbstractSuperPositionPSO**(**ParticleSwarmWrapper**<**S**>** wrapper**)**  **{**  algorithmWrapper **=** wrapper**;**  isWaiting **=** **false;**  **}**  @Override  public void run**()**  **{**  createInitialSwarm**();**  evaluateSwarm**();**  initializeVelocity**();**  initializeParticlesMemory**()** **;**  initializeLeader**()** **;**  initProgress**();**  **while** **(!**isStoppingConditionReached**())** **{**  updateVelocity**();**  updatePosition**();**  perturbation**();**  evaluateSwarm**();**  updateLeaders**();**  updateParticlesMemory**();**  onNewGeneration**.**invoke**();**  updateProgress**();**  isWaiting **=** **true;**  **while** **(**isWaiting**)**  **{**  Thread**.**onSpinWait**();**  **}**  **}**  **}**  public void createInitialSwarm**()** **{**  List**<**S**>** initialSwarm **=** algorithmWrapper**.**createInitialSwarm**();**  algorithmWrapper**.**setSwarm**(**initialSwarm**);**  **}**  public void evaluateSwarm**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  List**<**S**>** evaluatedSwarm **=** algorithmWrapper**.**evaluateSwarm**(**currentSwarm**);**  algorithmWrapper**.**setSwarm**(**evaluatedSwarm**);**  **}**  public void initializeLeader**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeLeader**(**currentSwarm**);**  **}**  public void initializeParticlesMemory**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeParticlesMemory**(**currentSwarm**);**  **}**  public void initializeVelocity**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeVelocity**(**currentSwarm**);**  **}**  public void updateVelocity**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateVelocity**(**currentSwarm**);**  **}**  public void updatePosition**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updatePosition**(**currentSwarm**);**  **}**  public void perturbation**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**perturbation**(**currentSwarm**);**  **}**  public void updateLeaders**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateLeaders**(**currentSwarm**);**  **}**  public void updateParticlesMemory**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateParticlesMemory**(**currentSwarm**);**  **}**  public void initProgress**()**  **{**  algorithmWrapper**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithmWrapper**.**updateProgress**();**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithmWrapper**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** getSwarm**()**  **{**  **return** algorithmWrapper**.**getSwarm**();**  **}**  public void setSwarm**(**List**<**S**>** newSwarm**)**  **{**  algorithmWrapper**.**setSwarm**(**newSwarm**);**  **}**  public void forceStoppingCondition**()**  **{**  **try** **{**  int maxIterations **=** algorithmWrapper**.**getMaxIterations**();**  algorithmWrapper**.**setIterations**(**maxIterations**);**  **}**  **catch** **(**Exception ex**)**  **{**  // TODO  **}**  **}**  **}** |

The following diagram represent the fields and methods of this class, together with the connections with the other classes from the project.

A screenshot of a computer

Description automatically generated

Fig. 27 AbstractSuperPositionPSO UML diagram

### 4.4.1.7. SuperPositionNSGAII class

The implementation of this class is abstract and helps in the final result of the superposition method of NSGAII and SMPSO algorithms.

|  |
| --- |
| public class SuperPositionNSGAII**<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionGA**<**S**,** List**<**S**>>**  **{**  public SuperPositionNSGAII**(**GeneticAlgorithmWrapper**<**S**>** wrapper**)**  **{**  **super(**wrapper**);**  **}**  @Override  public String name**()**  **{**  **return** "SuperPositionNSGAII"**;**  **}**  @Override  public String description**()**  **{**  **return** "SuperPosition Non-dominated Sorting Genetic Algorithm II"**;**  **}**  @Override  public List**<**S**>** result**()**  **{**  **return** algorithmWrapper**.**result**();**  **}**  **}** |

The diagram of the class and its connections is represented through the following:

A screenshot of a computer

Description automatically generated

Fig. 28 SuperpositionNSGAII UML diagram

### 4.4.1.8. SuperPositionSMPSO class

The implementation of this class is abstract and helps in the final result of the superposition method of NSGAII and SMPSO algorithms.

|  |
| --- |
| public class SuperPositionSMPSO **<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionPSO**<**S**,** List**<**S**>>**  **{**  public SuperPositionSMPSO**(**ParticleSwarmWrapper**<**S**>** wrapper**)** **{**  **super(**wrapper**);**  **}**  @Override  public String name**()**  **{**  **return** "SuperPositionSMPSO"**;**  **}**  @Override  public String description**()**  **{**  **return** "SuperPosition Speed-constrained Multi-objective PSO"**;**  **}**  @Override  public List**<**S**>** result**()** **{** **return** algorithmWrapper**.**result**();** **}**  **}** |

The diagram of the class and its connections is represented through the following:

A screenshot of a computer program

Description automatically generated

Fig. 29 Superposition SMPSO UML diagram

### 4.4.1.9. SuperPositionCombinator1 class

The **SuperPositionCombinator1** class extends **AbstractSuperPositionGAPSOCombinator1** and acts as a combinator for combining two superposition-based optimization algorithms: a genetic algorithm (12) (**AbstractSuperPositionGA**) and a particle swarm optimization algorithm (**AbstractSuperPositionPSO**). The population from the first algorithm, in this case, NSGAII is combined with the swarm of the SMPSO algorithm, and a new population is generated in the end, after calculated the fronts. The class code is represented as the following:

|  |
| --- |
| public class SuperPositionCombinator1**<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionGAPSOCombinator1**<**S**>** **{**  public SuperPositionCombinator1**(**AbstractSuperPositionGA**<**S**,** List**<**S**>>** alg1**,** AbstractSuperPositionPSO**<**S**,** List**<**S**>>** alg2**)** **{**  **super(**alg1**,** alg2**);**  **}**    @Override  protected List**<**S**>** ApplySuperPosition**()**  **{**  // 1. COMBINE POPULATIONS  List**<**S**>** combinedPopulation **=** **new** ArrayList**<>();**  // 2. RANKING AND TAKING THE BEST FRONT - we make this step to take the best 100 individuals from the combined population  combinedPopulation**.**addAll**(**algorithm1**.**getPopulation**());**  combinedPopulation**.**addAll**(**algorithm2**.**getSwarm**());**  // 3. GENERATE NEW POPULATION - here we set the 100 best as the new generation  Ranking**<**S**>** fronts **=** computeRanking**(**combinedPopulation**);**  List**<**S**>** newPopulation **=** take**(**fronts**,** algorithm1**.**getMaxPopulationSize**());**  // 4. SET NEW POPULATION FOR EACH ALGORITHM  algorithm1**.**setPopulation**(**newPopulation**);**  algorithm2**.**setSwarm**(**newPopulation**);**  **return** newPopulation**;**  **}**  **}** |

Below, the diagram of the class is represented, together with the connections:

A screenshot of a computer

Description automatically generated

Fig. 30 SuperpositionCombinator1 UML diagram

### 4.4.1.10. AbstractSuperPositionGAPSOCombinator1

This class is the most important one, because it simulated the whole superposition process of a genetic algorithm, like NSGAII and a particle swarm optimization algorithm, like SMPSO algorithm. (12) There are two lists representing both algorithms and also two events onNewGenerationDone and onSuperPositionDone that are invoked after the algorithm has reached its best solution and the result of the superposition is saved. The run() method executed the combination of the algorithms for providing the superposition result. It depends if one algorithm or another has reached its stopping condition or not, so the process can continue or stop at some solution during the execution of the process. The new population is obtained after calculated the pareto fronts and then it is returned.

|  |
| --- |
| public abstract class AbstractSuperPositionGAPSOCombinator**<**S **extends** Solution**<?>>** **implements** Runnable  **{**  protected AbstractSuperPositionGA**<**S**,** List**<**S**>>** algorithm1**;**  protected AbstractSuperPositionPSO**<**S**,** List**<**S**>>** algorithm2**;**  protected boolean bothAlgorithmsFinished**;**  protected List**<**S**>** resultPopulation**;**  public IAction onNewGenerationDone**;**  public IAction onSuperPositionDone**;**  public AbstractSuperPositionGAPSOCombinator**(**AbstractSuperPositionGA**<**S**,** List**<**S**>>** alg1**,**  AbstractSuperPositionPSO**<**S**,** List**<**S**>>** alg2**)**  **{**  algorithm1 **=** alg1**;**  algorithm2 **=** alg2**;**  bothAlgorithmsFinished **=** **false;**  **}**  protected abstract List**<**S**>** ApplySuperPosition**();**  public void start**()**  **{**  bothAlgorithmsFinished **=** **false;**  **}**  public List**<**S**>** getResult**()**  **{**  **return** resultPopulation**;**  **}**  public void run**()**  **{**  **while(!**bothAlgorithmsFinished**)**  **{**  **if(**algorithm1**.**isWaiting **&&** algorithm2**.**isWaiting**)**  **{**  **if(**algorithm1**.**isStoppingConditionReached**())**  **{**  **if(!**algorithm2**.**isStoppingConditionReached**())**  **{**  algorithm2**.**forceStoppingCondition**();**  **}**  bothAlgorithmsFinished **=** **true;**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  resultPopulation **=** ApplySuperPosition**();**  onNewGenerationDone**.**invoke**();**  onSuperPositionDone**.**invoke**();**  **}**  **else** **if(**algorithm2**.**isStoppingConditionReached**())**  **{**  algorithm1**.**forceStoppingCondition**();**  bothAlgorithmsFinished **=** **true;**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  resultPopulation **=** ApplySuperPosition**();**  onSuperPositionDone**.**invoke**();**  **}**  **else**  **{**  resultPopulation **=** ApplySuperPosition**();**  onNewGenerationDone**.**invoke**();**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  **}**  **}**  **}**  **}**  // 2. RANKING AND TAKING THE BEST FRONT - we make this step to take the best 100 individuals from the combined population  protected Ranking**<**S**>** computeRanking**(**List**<**S**>** solutionList**)**  **{**  Ranking**<**S**>** ranking **=** **new** FastNonDominatedSortRanking**<>();**  ranking**.**compute**(**solutionList**)** **;**  **return** ranking **;**  **}**  // 3. GENERATE NEW POPULATION - here we set the 100 best as the new generation  protected List**<**S**>** take**(**Ranking**<**S**>** fronts**,** int populationCount**)**  **{**  List**<**S**>** population **=** **new** ArrayList**<>();**  int noOfSelectedIndividuals **=** 0**;**  int noOfFronts **=** fronts**.**getNumberOfSubFronts**();**  **for** **(**int frontIndex **=** 0**;** frontIndex **<=** noOfFronts **-** 1**;** frontIndex**++)**  **{**  List**<**S**>** currentFront **=** fronts**.**getSubFront**(**frontIndex**);**  int currentFrontSize **=** currentFront**.**size**();**  int currentIndividualIndex **=** 0**;**  **while** **(**noOfSelectedIndividuals **<** populationCount **&&** currentIndividualIndex **<** currentFrontSize**)**  **{**  population**.**add**(**currentFront**.**get**(**currentIndividualIndex**));**  noOfSelectedIndividuals**++;**  currentIndividualIndex**++;**  **}**  **}**  **return** population**;** |

A screenshot of a computer

Description automatically generated

Fig. 31 AbstractSuperPositionGAPSOCombinator1 UML diagram

## Superposition usage

In the context of this application, to run the superposition on different problems, in this case for DTZL1 problem and GAP simulator, both NSGAII and SMPSO (13) algorithms needs to be configured with different values. In order to run the superposition, a *GAPRunner* class is created that inherits the AbstractAlgorithmRunner class from jMetal library which helps in printing the population results of the algorithm. In the main() method, the problem, DTLZ1 or GAP is initialized as the problem variable:

*AbstractDoubleProblem problem = new DTLZ1(3, 2);*

*AbstractDoubleProblem problem = new GapProblem(6, 2);*

For GAPProblem class, detailed information is described in the next subchapter 4.5.

After defining the problem, the NSGAII algorithm needs to be configured as follows:

A screen shot of a computer program

Description automatically generated

Fig. 32 NSGAII configuration

The explanation of the used variables is the following:

* crossoverProbability1: represents the crossover probability which is used by the crossover operator. It can be configured in the application
* crossoverDistributionIndex1: influences the distribution of offspring solutions around the parents and is used to calculate the crossover operator. It can be configured in the application
* crossover1: the variable is initialized with the SBXCrossover (Simulated Binary Crossover operator), which creates offspring that are similar to their parents while maintaining a certain degree of diversity in the population.
* mutationProbability1: represents the mutation probability which is used to calculate the mutation operator
* mutationDistributionIndex1: used in polynomial mutation to control the distribution of mutated values around the parent solution
* mutation1: initialized by the PolynomialMutation operator, which produces offsprings that are near the parent solution, but with a controlled degree of variation
* builder: initialized by the NSGAII builder contructor with the needed parameters
* problem: the problem to be ran, DTLZ1 or GAP
* populationSize: can be every number
* setMaxEvaluations: can be any number but is better to be higher than the size of the population.
* setMatingPoolSize: it is the individual’s proportion which are used by mutation and crossover operators.

The algorithm is ran by using the builder.build() method on the final code line.

SMPSO algorithm is configured as follows:

A screen shot of a computer program

Description automatically generated

Fig. 33 SMPSO configuration

The explanation of the used variables is the following:

* archive: is used to store non-dominated solutions found during the search process.
* mutationProbability: is the same as for NSGAII, but the values are set differently and helps in preserving the best solutions that have been discovered and ensures that the algorithm converges towards the Pareto front.
* mutationProbabilityIndex: is the same as for NSGAII, but the values are set differently
* mutation: is the same as for NSGAII
* problem: the problem to be ran, DTLZ1 or GAP
* setMaxIterations: can be every number, but is better to be higher than the population size.
* setSwarmSize: sets the number of swarm that is evaluated; can be every number
* setSolutionListEvaluator: calls the evaluate() method used to evaluate the SMPSO generations

The algorithm is ran by calling the build() method.

After the configuration of both algorithms, the superposition method is applied. Firstly, the two wrappers are initialized: NSGAIIWrapper and SMPSOWrapper and then, for each algorithm, the superposition is applied, so in the final, the combination of both can be used as the final result.

A screen shot of a computer program

Description automatically generated

Fig. 34 Superposition call

Each algorithm is run on a separate thread and in the end of the simulation, the result of the population is displayed in the console. The result is evaluate by two metrics, Hypervolume and SetCoverage, which will be detailed more in the next subchapter.

This *evaluateMetrics(result);* method creates the referenceParetoFront and the paretoFront to initialize the hypervolume and coverage metrics in order to generate results of the population displayed by the superposition.

A screen shot of a computer program

Description automatically generated

Fig. 35 evaluateMetrics() method

## Integration of GAP simulator

The Grid ALU Processor (GAP) represents a processor architecture designed to optimize sequential instruction streams executed through a reconfigurable grid of Arithmetic Logic Units (ALUs). (17)

Regarding its architecture, GAP has the following advantages:

* integrates a superscalar pipeline front-end, which handles the in-order execution of instructions.
* includes a configuration unit that dynamically issues both dependent and independent instructions to the ALUs.
* the ALUs are arranged in a grid, enabling parallel execution of instructions that were originally written for sequential execution.

For the purpose of this project, GAP was used to run the three MiBench benchmarks in order to provide results of the superposition of NSGAII and SMPSO.

In the following image, the interaction with GAP is explained. The idea starts with the executable SimALU.exe and the command line that starts the simulation with the given paramters. The benchmarks are ran by the simulator and the results are displayed in the result files.

A diagram of a benchmark gap simulator

Description automatically generated

Fig. 36 Interaction with GAP

Detailed implementation and configuration about GAP are explain in the following subchapters.

### Benchmarks

A benchmark is a standardized test or set of tests used to analyze the performance of a system, component, or software. In the context of computing, benchmarks are designed to measure various attributes such as speed, efficiency, and capability by running specific tasks and workloads, providing a consistent basis for comparison across different systems or architectures.

All the benchmarks used for the purpose of this project are part of the MiBench suite which is designed specifically for assessing the effectiveness of embedded systems. This was developed to represent a wide range of applications and workloads that are typical in the embedded computing domain. MiBench is divided into six distinct categories, each representing a different type of application commonly used in embedded systems; three of them were used for the GAP simulations: (18)

1. Automotive and Industial Control: qsort - simulate the computational tasks required in automotive and industrial control applications, such as signal processing and control algorithms.
2. Office Automation: stringsearch - simulate tasks found in office environments, such as text-to-speech conversion, searching for substrings within text, and encryption/decryption.
3. Networking: dijkstra - evaluate the performance of algorithms used in networking applications, such as routing algorithms, packet classification, and error checking.

### GAP integration

To use GAP simulator in the application, a folder containing the following files needs to be added in the project folder:

* SimALU.exe: which is the executable file of the simulator
* gap\_dumps folder: which is the folder needed by the simulator to run the MiBench available benchmarks; the folder contains data formatted in Intel HEX used by the GAP simulator. Each HEX file corresponds to different parts of the processor's state, such as the memory segments, instruction sets, or specific registers.
* miBenchTiny folder: which is the folder that contains the MakeFile file, needed to compile source code into executable binaries

After setting these files, the simulator can be called from the application.

### GAP parameters

In order to be run from an application, the simulator needs to be configured as follows:

* A command line with the following parameters needs to be created:

*SimALU.exe gap\_dump\_1234\_benchmark\_name n\_lines n\_colomns n\_rows memory\_latency chunk\_size chunk\_lines chunk\_sets*

* SimALU.exe (the executable file of the simulator)
* gap\_dump\_1234\_benchmark\_name(the path to the benchmark that will be executed)
* n\_lines (the number of rows in a grid where each row contains several ALU units)
* n\_colomns (the number of columns in the grid of ALU units)
* n\_rows (the number of rows in a multi-dimensional array or grid structure within the processor)
* memory\_latency (the number of cycles it takes to access data from memory)
* cache chunk\_size (the size of data chunks processed by each ALU unit)
* cache chunk\_lines (indicates the number of lines (or rows) within a chunk)
* cache chunk\_sets (the number of sets of chunks; refer to sets of processing elements that work together)

### GAP implementation

In the context of application implementation, the GAP simulator is used as a problem, as described in the 4.4.2. subchapter, to run the superposition of NSGAII (19) and SMPSO algorithms (13).

The implementation of GAP simulator starts with the *GAPProblem* class, which has the following structure:

* Create the command line
* Wait for the command line to be executed
* Read the results of the executed command line
* Set the objectives for the algorithm, in order to analyze the resulted population; in this case, the objectives are CPI (cycles per instructions) and HC (hardware complexity). The objectives are calculated by two methods implemented in the application code

This GAPProblem class is like a conector and has the role of triggering a simulation through GAP and then, take its results to set the objectives for the superposition algorithm.

The above structure starts with the constructor of the *GAPProblem* class, where the *numberOfVariables* and *numberOfObjectives* variables are set with their lower and upper limits. In order to be ran, the simulator needs to be initialized with a number of parameters, which is 6 in this case, because of the parameters from the received command line, and a number of objectives, which is 2, because we set *CPI* and *HC* at the end of the implementation.

To create the command line, the *getCommandLine*() method is created, where the solution is given as parameter and the arguments are created from the solution variables in the meantime of execution of simulator and saved in an *arguments* string. The benchmark path is set here and also the “*/lb*” parameter, which represents the loop optimization module that will be run by the GAP simulator. The returned command line will contain the executable of the simulator created by the *getMySimulator*() method, the set benchmark path together with the *arguments* string.

*getMySimulator*() method will create a copy of SimALU.exe with a random id for every simulation, to avoid the concurrent behavior of threads.

To wait for command line to execute, a try catch block is implemented where on try block, the process is executed with the created command line and a waitFor() set with 50000 miliseconds, while on catch block, the exceptions are caught.

For getting the results from the simulator output, the *parseIPCFromFile(string filePath)* method is implemented. The parameter is the results file created by the simulator after an execution.

After getting the IPC value from the results file, the CPI is calculated by the following formula:

CPI = 1/IPC

This value is then set as the first objective for the superposition algorithm.

The second objective, which is the hardware complexity, is calculate by the *getHardwareComplexity()* method, which receives as paramteres the values of:

* nLines
* nColomns
* nRows
* cache\_chunk\_size
* cache\_lines\_per\_set
* cache\_sets

To calculate this objective, firstly, the cost of the FU (functional unit of GAP) and secondly, the cost of the cache layers. For the cost of FUs, some constants are set before calculating the result:

* COST\_PER\_ALU = 1; A cost of 1 is assigned to each ALU to represent its relative complexity or resource requirement.
* COST\_PER\_LSU = 3.5; The cost of 3.5 indicates that an LSU (Load/Store unit) is considered to be 3.5 times more complex or resource-intensive than a single ALU.
* COST\_PER\_TOP\_REGISTER = 0.02; The cost of 0.02 reflects the relative simplicity and low resource requirement of a single register compared to an ALU.

COST\_PER\_LAYER\_CELL constant, which is set to 0.02, is used to calculate the cost of the cache layers.

After calculating and setting these two objectives to the solution, the solution is returned by the *evaluate(DoubleSolution solution)* method of the *GAPProblem* class.

## Simulations without GAP simulator

In the following subchapter, simulations of superposition of NSGAII and SMPSO algorithms are represented, using the DTLZ1 problem available in jMetal library. The results of the simulations are generated by two objectives obj1 and obj2. (7) These two objectives have the responsibility to minimize an xOy function. The solution provided contains a group of Pareto optimal solutions.

The algorithm was analyzed based on the Hypervolume and SetCoverage metrics, modifying the population and swarm size between 10 and 200 individuals. There were 5 simulations executed using DTZL1 problem.

### 4.6.1. NSGAII simulation

The results provided from the execution of NSGAII algorithm using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.41 | 0.5 |
| 50 | 0.99 | 0.5 |
| 100 | 1.00 | 0.5 |
| 150 | 1.00 | 0.5 |
| 200 | 1.00 | 0.5 |

Fig. 37 HV result of NSGAII

Analyzing the result of the hypervolume metric, we can see that as the population size increases, the hypervolume will maintain its value.

Fig. 38 Coverage result of NSGAII

The coverage result is linear, as it has almost the same value for every population size of the algorithm, except the first simulation, where the population size is 10 individuals.

### 4.6.2. SMPSO simulation

The results provided from the execution of SMPSO algorithm using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.5 | 0.5 |
| 50 | 0.5 | 0.5 |
| 100 | 0.5 | 0.5 |
| 150 | 0.5 | 0.5 |
| 200 | 0.5 | 0.5 |

Fig. 39 HV result of SMPSO

Fig. 40 Coverage result of SMPSO

Regarding the SMPSO algorithm, the results of coverage and hypervolume metrics were the same for all the simulations. The algorithm maintained a trend of 0.5 value for coverage and hypervolume too.

### 4.6.3. SuperPosition of NSGAII and SMPSO simulation

The results provided from the execution of superposition of NSGAII and SMPSO algorithms using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.41 | 0.5 |
| 50 | 0.99 | 0.5 |
| 100 | 1.00 | 0.5 |
| 150 | 1.00 | 0.5 |
| 200 | 1.00 | 0.5 |

Fig. 41 HV result of NSGAII and SMPSO superposition

Analyzing the result of the hypervolume metric, we can see that as the population size increases, the hypervolume will maintain its value. For the hypervolume metric, we can see a slight difference for every simulation, comparing to the results obtained at chapters 4.5.1 and 4.5.2. The superposition of the NSGAII and SMPSO generated better results for the hypervolume metric while analyzing its individuals.

Fig. 42 Coverage result of NSGAII and SMPSO superposition

The coverage result is linear, as it has the same value for every population size of the algorithm.

As a conclusion of these simulations, we can specify the followings:

* as the population size increases, the algorithms may converge more effectively towards the true Pareto front or a close approximation of it. Increasing the population size might not significantly affect the hypervolume value once convergence is reached.
* with a larger population size, the algorithms explore a bigger portion of the search space, potentially covering more areas of the Pareto front. This increased sampling density can lead to a more accurate approximation of the front Pareto, concluding in a stable hypervolume value as the population size grows.
* while the algorithms use random population initialization, increasing the population size can lead to a broader initial exploration of the search space. This initial exploration may result in a mixed group of solutions that cover separated regions of the Pareto front, contributing to a stable hypervolume value as the population size increases.

## Simulations with GAP simulator

In the following subchapter, simulations of superposition of NSGAII and SMPSO algorithms are represented, using the GAP simulator. The results of the simulations are generated by two objectives CPI (cycles per instructions) and HC (hardware complexity). These two objectives are used by the superposition algorithm to generate individuals. The solution provided contains a group of Pareto optimal solutions.

The algorithm was analyzed based on the Hypervolume and SetCoverage metrics, modifying the population and swarm size between 5 and 50 individuals. There were 4 simulations executed using GAP problem.

For each simulation, the following configurations were set:

* For NSGAII:

|  |  |  |
| --- | --- | --- |
| Nr crt | Parameter Name | Parameter Value |
| 1 | Crossover probability | 0.9 |
| 2 | Crossover distribution index | 20.0 |
| 3 | Crossover operator | SBXCrossover |
| 4 | Mutation probability | 1.0/6 (6 = nr of variables) |
| 5 | Mutation distribution index | 20.0 |
| 6 | Mutation operator | PolynomialMutation |
| 7 | Population size | Varies between 5-50 |
| 8 | Max evaluations | Varies between 10-100; it depends on the population size; for the simulations, the max evaluations nr was double of population size |
| 9 | Mating pool size | 2 |
| 10 | Selection operator | BinaryTournamentSelection |
| 11 | Evaluator | SequentialSolutionListEvaluator |
| 12 | Dominance comparator | DefaultDominanceComparator |

* For SMPSO:

|  |  |  |
| --- | --- | --- |
| Nr crt | Parameter Name | Parameter Value |
| 1 | Mutation probability | 1.0/6 (6 = nr of variables) |
| 2 | Mutation distribution index | 20.0 |
| 3 | Mutation operator | PolynomialMutation |
| 4 | Swarm size | Varies between 5-50 |
| 5 | Max evaluations | Varies between 10-100; it depends on the population size; for the simulations, the max evaluations nr was double of population size |
| 6 | Archive | 2 |
| 7 | Evaluator | SequentialSolutionListEvaluator |
| 8 | Dominance comparator | DefaultDominanceComparator |

### NSGAII and SMPSO superposition ran on Dijkstra benchmark

The results provided from the execution of NSGAII and SMPSO superposition using the GAP problem and Dijkstra benchmark are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 5 | 916,79 | 0,25 |
| 10 | 987,68 | 0,66 |
| 30 | 1528,07 | 1 |
| 50 | 1973,89 | 1 |

Fig. 43 HV result of superposition running on Dijkstra benchmark

This increasing HV values indicate that the hybrid algorithm performs better with larger population sizes. Higher HV values imply better performance of the optimization algorithm, as it indicates a larger volume covered by the obtained Pareto front in the objective space.

Fig. 44 Coverage result of superposition running on Dijkstra benchmark

As the population size increases from 10 to 30, the coverage also increases, indicating that the algorithm is able to cover a larger portion of the search space or Pareto front.

Beyond a population size of 30, the coverage stabilizes at 1. This suggests that a population size of 30 is sufficient for the algorithm to achieve full coverage in this benchmark, and increasing the population size further does not provide additional benefits in terms of coverage.

The coverage results, similar to the HV results, show that increasing the population size improves the performance of the algorithm up to a certain point. In this case, a population size of 30 is optimal for achieving full coverage, indicating that the algorithm can fully explore the search space or Pareto front with this population size.

### NSGAII and SMPSO superposition ran on qSort benchmark

The results provided from the execution of NSGAII and SMPSO superposition using the GAP problem and qsort benchmark are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 5 | 3207,81 | 0,25 |
| 10 | 2926,36 | 0,66 |
| 30 | 3475,26 | 0,67 |
| 50 | 4784,62 | 0,85 |

Fig. 45 HV result of superposition running on qSort benchmark

The HV results for the qsort benchmark indicate that while there might be fluctuations at smaller population sizes, increasing the population size consistently improves the performance of the NSGAII and SMPSO superposition. This suggests that larger populations are more effective in exploring the solution space and optimizing the hypervolume metric for this particular benchmark.

Fig. 46 Coverage result of superposition running on qSort benchmark

The sharp increase in coverage from population size 10 to 20 suggests that the algorithm benefits from having more individuals, which helps in better exploration of the solution space.

The coverage results for the qsort benchmark demonstrate that increasing the population size significantly enhances the performance of the NSGAII and SMPSO superposition up to a certain point. A population size of 20 provides a substantial improvement in coverage, while further increases up to 50 continue to improve coverage but at a diminishing rate. This suggests that larger population sizes are beneficial for ensuring comprehensive exploration and optimization of the solution space in the qsort benchmark.

### NSGAII and SMPSO superposition ran on stringSearch benchmark

The results provided from the execution of NSGAII and SMPSO superposition using the GAP problem and stringSearch benchmark are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 5 | 1435,11 | 0 |
| 10 | 1734,45 | 0,28 |
| 30 | 1457,99 | 1 |
| 50 | 1993,92 | 1 |

Fig. 47 HV result of superposition running on stringSearch benchmark

A population size of 20 shows a significant improvement in HV, while a decrease in performance is observed at population size 30.

The highest HV value is achieved at a population size of 50, indicating that larger populations tend to produce higher quality solutions in the string search benchmark.

This analysis suggests that the superposition of NSGAII and SMPSO benefits from larger population sizes, though there may be certain intermediate sizes where performance does not improve as expected.

Fig. 48 Coverage result of superposition running on stringSearch benchmark

The significant increase in coverage from population size 10 to 30 indicates that the algorithm performs better with larger populations, rapidly improving its ability to cover the solution space.

The coverage value reaching 1 at population size 30 and remaining constant up to 50 suggests that the algorithm achieves full coverage at a population size of 30. Increasing the population size beyond 30 does not provide additional benefits in terms of coverage.

# Chapter 5. Conclusions and future work

The conclusions of this dissertation project are the following:

* I studied the behavior of genetic algorithms (NSGAII), as well as the inspired by nature algorithms (SMPSO).
* I created a separate project where the jMetal 6.0 library was integrate.
* The superposition of NSGAII and SMPSO algorithms was integrate in this project. It can be run using the problems implemented in jMetal, but also using a simulator.
* For the purpose of this project, the GAP simulator was used as a problem for superposition algorithm to run on different benchmarks.
* NSGAII shows variability in performance with population size, but it achieves higher HV values (~1.0) compared to SMPSO when population sizes are optimized.
* SMPSO demonstrates stable performance but with lower HV values (~0.5) that do not improve with increasing population size.
* The results of hypervolume in the context of superposition show a clear and steady increase in HV with increasing population size. Highest HV observed at the largest population size (50), significantly higher than single algorithm results.
* The superposition of NSGAII and SMPSO consistently yields higher HV values compared to either algorithm alone.
* The superposition of NSGAII and SMPSO benefits significantly from increasing population size, achieving the highest HV of ~1973.89 at a population size of 50 (for Dijkstra benchmark). This is markedly higher than the HV values achieved by NSGAII (~1.0) and SMPSO (~0.5) individually at their respective optimal population sizes.
* The superposition approach scales better with population size, showing consistent improvements in HV values as the population size increases.
* Both NSGAII and SMPSO reach a stable coverage value of 0.5 at a population size of 50, with no further improvement as the population size increases. This indicates that both algorithms are limited in their ability to improve coverage with larger populations when used individually.
* The superposition of NSGAII and SMPSO demonstrates significant improvements in coverage with increasing population size. Full coverage (value of 1) is achieved at a population size of 30 (for stringSearch benchmark), which is not reached by either NSGAII or SMPSO alone.
* Using NSGAII or SMPSO alone is less effective in terms of coverage, with both algorithms reaching only half of the maximum possible coverage value.

Future work that can be implemented in this project:

* Integrate the superposition methods in FADSE framework.
* Analyze the superposition methods with other metrics like TSHD, Spread, Error Ratio, Epsilon, etc.
* Create a more dynamic structure for GAP simulator to execute more benchmarks in the same time and process their results.

# Chapter 6. References

1. *A novel solver for multi-objective optimization: dynamic non-dominated sorting genetic algorithm (DNSGA).* Long, Q., Li, G. & Jiang, L. 2022, Vol. Soft Comput 26, pp. 725–747.

2. Coello, Carlos A. Coello, Arturo Hernández Aguirre, and Eckart Zitzler. *Evolutionary multi-criterion optimization.* Verlag Berlin/Heidelberg : Springer, 2005.

3. Calborean, H. *Multi-objective optimization of advanced computer architectures using domain-knowledge.* Sibiu : s.n., 2011.

4. *SMPSO: A new PSO-based metaheuristic for multi-objective optimization.* Nebro, A.J., Durillo, J.J., Garcia-Nieto, J., Coello, C.C., Luna, F. and Alba, E. 2009, Symposium on computational intelligence in multi-criteria decision-making (MCDM), pp. 66-73.

5. SourceTree for Windows 2.0 is now in beta. [Online] [Cited: 06 20, 2024.] https://blog.sourcetreeapp.com/2017/01/27/sourcetree-for-windows-2-0-is-now-in-beta/.

6. statista. Most used programming languages among developers worldwide as of 2023. [Online] [Cited: 06 20, 2024.] https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

7. jMetal. [Online] [Cited: 06 20, 2024.] https://jmetal.readthedocs.io/en/latest/problems.html .

8. Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J.J. and Schwefel, H.P. eds. *Parallel Problem Solving from Nature-PPSN VI: 6th International Conference.* Paris, France : Springer, September 18-20 2000.

9. *Measuring the convergence and diversity of CDAS multi-objective particle swarm optimization algorithms: a study of many-objective problems.* De Carvalho, A.B. and Pozo, A. 2012, Neurocomputing, pp. 43-51.

10. Deb, Kalyanmoy. Multi-Objective Optimization. *Introductory Tutorials in Optimization and Decision Support Techniques.* Harvard : Springer, 2005.

11. wikipedia. Pareto front. [Online] [Cited: 06 20, 2024.] https://en.wikipedia.org/wiki/Pareto\_front.

12. *Multi-objective optimization of advanced computer architectures using domain-knowledge.* Calborean, H. 2011.

13. Couceiro, M., Ghamisi, P., Couceiro, M. and Ghamisi, P. *Particle swarm optimization.* s.l. : Springer International Publishing, 2016.

14. Followchart-of-the-SMPSO-algorithm. [Online] [Cited: 06 20, 2024.] https://www.researchgate.net/figure/Followchart-of-the-SMPSO-algorithm\_fig3\_349661785.

15. *Many objective particle swarm optimization.* Figueiredo, E.M., Ludermir, T.B. and Bastos-Filho, C.J. 2016, Information Sciences, pp. 115-134.

16. *A fast hypervolume driven selection mechanism for many-objective optimisation problems.* Rostami, S. and Neri. 2017, Swarm and evolutionary computation, Vol. 34, pp. 50-67.

17. *MULTI OBJECTIVE DESIGN SPACE EXPLORATION FOR GRID ALU PROCESSOR.* Asgari, R., Ramaki, A.A., Atani, R.E. and Shahbahrami, A.

18. *MiBench: A free, commercially representative embedded benchmark suite.* Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and Brown, R.B. 2001, Proceedings of the fourth annual IEEE international workshop on workload characterization, pp. 3-14.

19. *Multi-objective optimization using evolutionary algorithms.* Deb, K. Vol. 16.

20. Osaba, E. and Yang, X.S. eds. *Applied Optimization and Swarm Intelligence.* Singapore : Springer, 2021.

21. *Calculating and aggregating direct trust and reputation in organic computing systems.* Kiefhaber, R. 2013.

22. *Multi‐objective PSO/GA optimization control strategies for energy efficient PMDC motor drives.* Sharaf, A.M. and El‐Gammal, A.A. 2011, European Transactions on Electrical Power, Vol. 21(8), pp. 2080-2097.

23. *MPI-based parallel synchronous vector evaluated particle swarm optimization for multi-objective design optimization of composite structures.* Omkar, S.N., Venkatesh, A. and Mudigere, M. 2012, Engineering Applications of Artificial Intelligence, pp. 1611-1627.

24. *Application of particle swarm optimization to the design of multi-band antennas with lumped loads.* Mims, M.E. and Kelley, D.F. 2006, In 2006 IEEE Antennas and Propagation Society International Symposium, Vol. IEE, pp. 3547-3550.

25. *jMetal: A java framework for developing multi-objective optimization metaheuristics.* Durillo, J.J., Nebro, A.J., Luna, F., Dorronsoro, B. and Alba, E. Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, ETSI Informática, Campus de Teatinos, Tech. Rep. ITI-2006-10.

26. Metaheuristics included in jMetal. [Online] [Cited: 06 20, 2024.] https://jmetal.sourceforge.net/algorithms.html.

27. Engineering:Pareto front. [Online] [Cited: 06 20, 2024.] https://handwiki.org/wiki/Engineering:Pareto\_front.

28. *An automatic design space exploration framework for multicore architecture optimizations.* Calborean, H. and Vinţan, L. 2010, In 9th RoEduNet IEEE International Conference, pp. 202-207.

29. *A Comparison of MultiObjective Algorithms for the Automatic Design Space Exploration of a Superscalar System, Advances in Intelligent Control Systems and Computer Science, Advances in Intelligent Systems and Computing.* Calborean, H., Jahr, R., Ungerer, T. and Vintan, L. 2013, pp. 489-502.

30. Coello, C.A.C. *Evolutionary algorithms for solving multi-objective problems.* s.l. : springer. com, 2007.

31. Jahr, R., Calborean, H., Vintan, L. and Ungerer, T. Finding near‐perfect parameters for hardware and code optimizations with automatic multi‐objective design space explorations. s.l. : John Wiley & Sons, 2012, pp. 2196-2214.