

Evolutionary Computing

Final Project:

An adaptive multi-objective evolutionary algorithm

based on grid subspaces

Supervisor:

Dr. Ali Hamzeh

Atefe Rajabi

Winter 2024

Content

Algorithm

Implementation

Recombination

Subspace

Problem Evaluator

AGMOEA

Result

WFG-AGMOEA

Algorithm:

The article presents an adaptive multi-objective evolutionary algorithm based on grid subspaces, termed AGMOEA. It focuses on enhancing the performance of multi-objective optimization by dynamically allocating evolutionary opportunities across different subspaces within a grid system. This approach aims to balance convergence and diversity efficiently. The methodology section introduces concepts fundamental to multi-objective optimization and the unique subspace system developed for this algorithm. The core of AGMOEA involves adaptive selection strategies, an evolutionary scheme utilizing multiple operators, and mechanisms for capacity management and external archive extension. Experimental results demonstrate AGMOEA's competitiveness against existing algorithms across various benchmark problems. The article concludes with a summary of its contributions and potential for future research in evolutionary computation.

The main steps of the proposed AGMOEA algorithm can be outlined as follows, based on the article's description:

1. \*\*Objective Space Division into Subspaces:\*\* The initial step involves dividing the objective space into multiple subspaces using a grid approach. This division is fundamental to AGMOEA, as it allows the algorithm to manage and navigate the solution space more effectively.

2. \*\*Initial Population Generation and Evaluation:\*\* An initial population of solutions is randomly generated. Each solution is then evaluated to determine its objective values, which are necessary for subsequent steps such as subspace assignment and ranking.

3. \*\*Subspace Assignment:\*\* Based on the objective values, each solution is assigned to a specific subspace. This assignment is crucial for organizing the population according to the grid structure and facilitating targeted evolutionary operations within each subspace.

4. \*\*Adaptive Selection Strategy:\*\* AGMOEA implements an adaptive selection strategy that dynamically selects representative parent solutions from the subspaces. This selection is based on criteria such as subspace ranking and the subspace dominance relationship. The goal is to maintain a balance between convergence (quality of solutions) and diversity (variety of solutions) across the population.

5. \*\*Evolutionary Scheme with Multiple Operators:\*\* The algorithm applies a variety of evolutionary operators (e.g., mutation, crossover) to generate offspring from the selected parent solutions. The use of multiple operators and local search techniques helps accelerate convergence towards optimal solutions while exploring the solution space efficiently.

The article utilizes a combination of five crossover operators and a specific mutation operator for generating offspring in the evolutionary process. The crossover operators include:

1. \*\*Simulated Binary Crossover (SBX):\*\* A recombination operator that simulates the single-point crossover on binary strings but is designed for real-valued representations. It tends to produce offspring close to the parents, allowing for fine-grained search near the parent solutions.

2. \*\*Parent Centric Crossover (PCX):\*\* A crossover technique that generates offspring around a centroid of selected parents, emphasizing the exploitation of the current population's fitness landscape.

3. \*\*Simplex Crossover (SPX):\*\* A method that generalizes the concept of midpoint crossover to multiple parents, creating offspring that lie within a simplex formed by parent solutions. It is useful for maintaining diversity in the population.

4. \*\*Blend Crossover (BLX-α):\*\* This operator generates offspring by blending the genes of the parents, with the blending range controlled by a parameter α. It allows exploration of the search space between and around the parent solutions.

5. \*\*Differential Evolution (DE/rand/1):\*\* A strategy within the Differential Evolution framework that perturbs a target vector with the scaled difference between two randomly selected vectors from the population. This operator is known for its robustness in exploring the search space effectively.

After applying one of these crossover operators, the \*\*polynomial mutation operator\*\* is used on the offspring. This mutation operator introduces small changes in the offspring's genes to maintain genetic diversity within the population. The mutation probability is set to 1/n, where n is the number of decision variables, which is a common setting to ensure that mutation occurs at a reasonable rate.

These operators are chosen for their ability to balance exploration and exploitation in the search space effectively. By using multiple operators, the algorithm can adapt to different problem landscapes, enhancing its robustness and ability to find high-quality solutions across a variety of multi-objective optimization problems (MOPs).

6. \*\*External Archive and Local Search:\*\* AGMOEA utilizes an external archive to store high-quality solutions found during the evolutionary process. A grid-based local search is performed on this archive to refine the solutions further and guide the search process.

7. \*\*Update Mechanism:\*\* After each generation, the algorithm updates the objective space division (subspaces), re-evaluates the solutions, and repeats the selection and evolutionary processes. This iterative mechanism ensures that AGMOEA adapts to changes in the solution landscape and progressively improves the solution quality over time.

8. \*\*Termination:\*\* The algorithm terminates after a predefined number of generations or when another stopping criterion is met. The final output is a set of non-dominated solutions representing the Pareto-optimal front for the given multi-objective optimization problem.

These steps illustrate AGMOEA's comprehensive approach to tackling multi-objective optimization problems by leveraging the structure provided by grid-based subspaces, adaptive selection, and evolutionary operations to balance convergence and diversity effectively.

Implementation

Recombination:

This `Recombination` class is designed to implement various crossover and mutation strategies for evolutionary algorithms.

### `\_\_init\_\_(self, parents, parameters, crossover\_probability=0.9)`

The constructor initializes the `Recombination` class with:

- `parents`: A list of parent solutions, which are typically represented as numpy arrays.

- `parameters`: A dictionary containing parameters specific to each crossover operator. This allows for flexible configuration of each operator's behavior.

- `crossover\_probability`: The probability with which a crossover operation will be performed.

### `sbx(self, \*\*kwargs)`

Simulated Binary Crossover (SBX) is designed for real-valued genetic representations. It generates two offspring from two parents by simulating the binary crossover behavior on real numbers. The `eta` parameter controls the distribution of the offspring around the parents; a higher `eta` results in offspring closer to the parents, promoting exploitation.

### `pcx(self, \*\*kwargs)`

Parent Centric Crossover (PCX) generates a single offspring from a selected parent and a centroid of other parents. It uses Gaussian distributions controlled by `sigma\_s` and `sigma\_eta` to introduce variation. This method is useful for maintaining diversity while exploiting the current search area around the parent.

### `spx(self, \*\*kwargs)`

Simplex Crossover (SPX) creates offspring by linearly combining parents' genes using random weights from a Dirichlet distribution. The `epsilon` parameter controls the simplex's size, allowing for exploration within and around the parents' space.

### `blx\_alpha(self, \*\*kwargs)`

Blend Crossover (BLX-α) generates a single offspring whose genes are a blend of the corresponding genes from two parents, within a range expanded by `alpha`. This operator promotes diversity by exploring the space between and around the parents.

### `de\_rand\_1(self, \*\*kwargs)`

Differential Evolution (DE/rand/1) strategy perturbs one target vector with the weighted difference between two other randomly selected vectors. The `cr` and `f` parameters control the crossover rate and the differential weight, respectively, facilitating a robust search mechanism.

### `select\_crossover\_operator(self, operator\_probabilities)`

This function selects a crossover operator based on the provided probabilities. It allows the algorithm to adaptively choose between different crossover strategies, promoting a dynamic balance between exploration and exploitation.

### `execute\_crossover(self, crossover\_name)`

Executes the selected crossover operator if the random chance falls below the `crossover\_probability`. This function acts as the main entry point to perform crossover operations, ensuring that the algorithm can control when and how crossovers occur based on the defined probability.

Overall, this class encapsulates various crossover mechanisms within a single, adaptable framework, allowing for efficient exploration and exploitation of the solution space in evolutionary algorithms. Each function is carefully designed to contribute to the diversity and convergence of the population towards optimal solutions.

Subspace:

The `Subspace` class is designed to manage a subspace within the context of a grid-based approach to multi-objective optimization, particularly within algorithms like AGMOEA that leverage grid subspaces for better search space exploration and management. Each subspace represents a distinct region in the objective space, with its own set of solutions and characteristics. Here's a detailed explanation of each function:

### `\_\_init\_\_(self, coordinates, ideal\_point, grid\_intervals)`

The constructor initializes a `Subspace` object with:

- `coordinates`: The position of the subspace within the grid, indicating its relative location in the objective space.

- `ideal\_point`: A reference point in the objective space, typically used to guide the search process towards desirable areas.

- `grid\_intervals`: The size of each grid cell in each objective dimension, defining the granularity of the grid.

- `solutions`: A list to store solutions (e.g., decision vectors) that belong to this subspace.

### `calculate\_scp()`

Calculates the specific coordinate point (`z\_scp`) for the subspace, which is derived by scaling the subspace's coordinates by the grid intervals and adding the ideal point. This operation effectively maps the subspace's grid coordinates to the actual objective space.

### `calculate\_sdv(epsilon=1e-6)`

Computes the subspace density value (`sdv`), which inversely relates to the subspace's coordinates. It introduces an `epsilon` to avoid division by zero, ensuring numerical stability. This function provides a measure of how densely packed a subspace is, which can guide the selection of subspaces for exploration.

### `calculate\_sws(x, epsilon=1e-6)`

Calculates the solution within subspace score (`sws\_value`) for a given solution `x`. It uses the solution's objective values, the subspace's density value (`sdv`), and the specific coordinate point (`z\_scp`) to compute how well the solution fits within the subspace. This score can be used to select representative solutions from the subspace.

### `subspace\_dominance(other)`

Determines if the current subspace dominates another subspace (`other`) based on their grid coordinates. A subspace dominates another if it is better (has lower coordinate values) in at least one dimension without being worse in any other. This function supports the notion of Pareto dominance at the subspace level.

### `strong\_subspace\_dominance(other)`

Checks for strong dominance between the current subspace and another subspace. Strong dominance occurs when the current subspace has strictly lower coordinates in all dimensions compared to the other subspace. This is a stricter criterion than simple dominance.

### `weak\_subspace\_dominance(other)`

Evaluates weak dominance, where the current subspace is considered to dominate another if it is at least as good in all dimensions and better in at least one dimension, with equality allowed. This function allows for a more inclusive form of dominance, useful in scenarios where distinguishing between closely related subspaces is necessary.

### `select\_representative()`

Selects a representative solution from the subspace based on the solution within subspace scores (`sws`). It calculates `sws` for all solutions in the subspace and selects the one with the lowest score, indicating the best fit within the subspace's context. This representative can be used in various algorithmic processes, such as guiding the search or maintaining diversity.

Overall, the `Subspace` class encapsulates the functionality necessary for managing a section of the objective space within a grid-based multi-objective optimization framework. It provides mechanisms for evaluating and comparing subspaces, selecting representative solutions, and guiding the evolutionary search process.

Problem Evaluators:

These classes are designed as evaluators for different types of optimization problems within the framework of multi-objective optimization. Each class is tailored to evaluate a specific category of problems: `WFGEvaluator` for WFG problems, `ZDTEvaluator` for ZDT problems, and `DTLZEvaluator` for DTLZ problems.

### `WFGEvaluator`

This class is designed to evaluate solutions for WFG (Walking Fish Group) benchmark problems, which are a suite of test problems for multi-objective optimization. The class is initialized with the problem name, number of variables (`n\_var`), and number of objectives (`M`). The `evaluate` method allows evaluating a given solution `x`, returning its objective values. The class also provides methods to retrieve the true Pareto front, the problem's bounds, the ideal point, and the nadir point, offering comprehensive information necessary for analysis and comparison of optimization algorithms.

### `ZDTEvaluator`

Similar to `WFGEvaluator`, this class focuses on ZDT (Zitzler-Deb-Thiele) benchmark problems, which are another set of test problems used extensively in multi-objective optimization research. The ZDT problems are known for their simplicity and are useful for testing and comparing evolutionary algorithms. The class initialization and methods mirror those of `WFGEvaluator`, adjusted for the ZDT problem specifications. It simplifies the initialization for ZDT problems by defaulting to two objectives (`M=2`), as all ZDT problems are bi-objective.

### `DTLZEvaluator`

This class is tailored for evaluating DTLZ (Deb-Thiele-Laumanns-Zitzler) benchmark problems, which are designed to test the performance of multi-objective evolutionary algorithms (MOEAs) in handling many-objective problems. The `DTLZEvaluator` class allows for more flexibility in terms of the number of objectives (`M`) and provides an additional parameter for the number of partitions (`n\_partitions`), which is used to generate reference directions for certain DTLZ problems that require them. The `get\_dtlz\_problem` method specifically configures the DTLZ problem, taking into account the reference directions, making it suitable for problems where a well-distributed set of solutions is desired.

### Common Functionality

All three classes share a common set of functionalities:

- `evaluate(x)`: Evaluates the given solution `x` and returns its objective values.

- `get\_true\_pareto()`: Returns the true Pareto front for the problem, useful for performance evaluation and comparison.

- `get\_bounds()`: Retrieves the lower and upper bounds for the decision variables, important for ensuring solutions remain within valid ranges.

- `ideal\_point()`: Returns the ideal point for the problem, representing the best possible values for each objective.

- `nadir\_point()`: Provides the nadir point, indicating the worst values for each objective found in the Pareto front.

AGMOEA:

The `AGMOEA` class represents an implementation of an Adaptive Grid-based Multi-Objective Evolutionary Algorithm. It's designed for solving multi-objective optimization problems by effectively exploring and exploiting the search space using a grid-based approach.

### `\_\_init\_\_(self, ...)`

Initializes an instance of the AGMOEA algorithm with various parameters including population size, grid intervals, subspace and external archive capacities, mutation probability, decision variable and objective numbers, along with an evaluator for problem-specific operations. It sets up structures for storing solutions and tracking algorithm progress.

### `initialize\_population(self)`

Generates the initial population of solutions randomly within the defined bounds. Each solution's objective values are evaluated using the provided evaluator.

### `construct\_subspaces(self, solutions)`

Organizes solutions into subspaces based on their objective values. This method divides the objective space into a grid and assigns each solution to a subspace, aiding in diversity maintenance and focused search.

### `generate\_grid\_coordinates(self)`

Creates a list of all possible grid coordinates based on the number of intervals and objectives. This is used for initializing and managing subspaces.

### `polynomial\_mutation(self, chromosome, eta\_m=20)`

Applies polynomial mutation to a given chromosome. This method introduces small, random changes to the decision variables, helping to explore the search space around current solutions.

The parameter `eta\_m` in the polynomial mutation function plays a crucial role in controlling the distribution of the mutation. Specifically, it determines the mutation's distribution index, which affects the perturbation's spread and how far mutated values can deviate from the original value.

- \*\*Role of `eta\_m`:\*\* It is a parameter that shapes the mutation operation's probability distribution. A higher value of `eta\_m` makes the distribution more peaked, meaning mutations are likely to produce offspring closer to the parent. Conversely, a lower value of `eta\_m` leads to a flatter distribution, allowing for more significant deviations from the parent's gene value.

- \*\*Effect of `eta\_m`:\*\* The choice of `eta\_m` impacts the explorative and exploitative capabilities of the genetic algorithm. A higher `eta\_m` value encourages fine-grained search around the parent solutions, enhancing exploitation of the current search space. In contrast, a lower `eta\_m` fosters exploration by enabling larger jumps in the solution space, potentially escaping local optima but risking the stability of convergence.

The polynomial mutation function, through `eta\_m`, therefore, provides a mechanism to balance between exploration of new areas in the search space and exploitation of known good areas, directly influencing the genetic algorithm's ability to find optimal or near-optimal solutions.

### `improve\_EXA(self)`

Improves the External Archive (EXA) by generating new offsprings from selected subspaces, followed by updating the EXA with these offsprings. This method aims to enhance the quality of solutions in the archive.

### `select\_subspace\_EXA(self)`

Selects a subspace from the external archive for generating offsprings, based on the subspace density and diversity. It aims to balance the exploration and exploitation by focusing on less crowded subspaces.

### `SR(self, subspace)`

Calculates the sum of coordinates for a given subspace, used in subspace selection processes to evaluate subspace density.

### `select\_subspace(self)`

Chooses a subspace from the current population for offspring generation, similar to `select\_subspace\_EXA` but focuses on the main population's subspaces.

### `update\_degraded\_subspaces(self, selected\_subspace)`

Updates the set of degraded subspaces based on the dominance relationship with the selected subspace, to avoid repetitive exploration of dominated or less promising regions.

### `adaptive\_selection\_probability(self)`

Calculates the probability of selecting parents from the external archive versus the subspaces, adapting over generations to balance exploration and exploitation.

### `parent\_selection(self, selected\_subspace)`

Selects parents for crossover operations, potentially mixing solutions from the external archive and the selected subspace to generate diverse offsprings.

### `update\_operator\_probabilities(self)`

Updates the probabilities of selecting each crossover operator based on their past usage and success, aiming to dynamically adapt the search strategy over time.

### `generate\_offspring(self, selected\_subspace)`

Generates offspring solutions from selected parents using the chosen crossover operator, followed by mutation. This is a core part of the search and optimization process.

### `evaluate\_individual(self, chromosome)`

Evaluates the objective values of a given solution using the problem-specific evaluator provided during initialization.

### `fast\_non\_dominated\_sort(self, population)`

Sorts the given population into fronts based on non-domination levels, used for environmental selection and archive updating.

### `crowding\_distance(self, front)`

Calculates the crowding distance for solutions within a front, used for diversity preservation during environmental selection.

### `manage\_exa\_capacity(self)`

Ensures the external archive does not exceed its maximum capacity by removing less promising solutions based on crowding distance.

### `environmental\_selection(self, population)`

Performs environmental selection to form the next generation, using non-dominated sorting and crowding distance to maintain both convergence and diversity.

### `termination\_criterion(self)`

Checks whether the termination condition of the algorithm has been met, typically based on the maximum number of function evaluations or generations.

### `agmoea\_algorithm(self)`

The main method that encapsulates the AGMOEA algorithm's logic, orchestrating initialization, generation updates, and convergence towards the Pareto-optimal front.

### `calculate\_hypervolume(self, pareto\_objectives)`

Computes the hypervolume of the set of solutions provided, offering a measure of the quality and diversity of the Pareto front approximation.

### `calculate\_igd(self, pareto\_objectives)`

Calculates the Inverted Generational Distance (IGD) metric for the set of solutions, assessing the algorithm's performance in terms of convergence and diversity.

### `correct\_pareto\_front(self)`

Ensures the true Pareto front is correctly represented for comparison purposes, potentially correcting or updating it based on the problem evaluator.

### `create\_chromosome(self, objectives)`

Utility method for creating a chromosome instance from objective values, used in processing or preparing the true Pareto front.

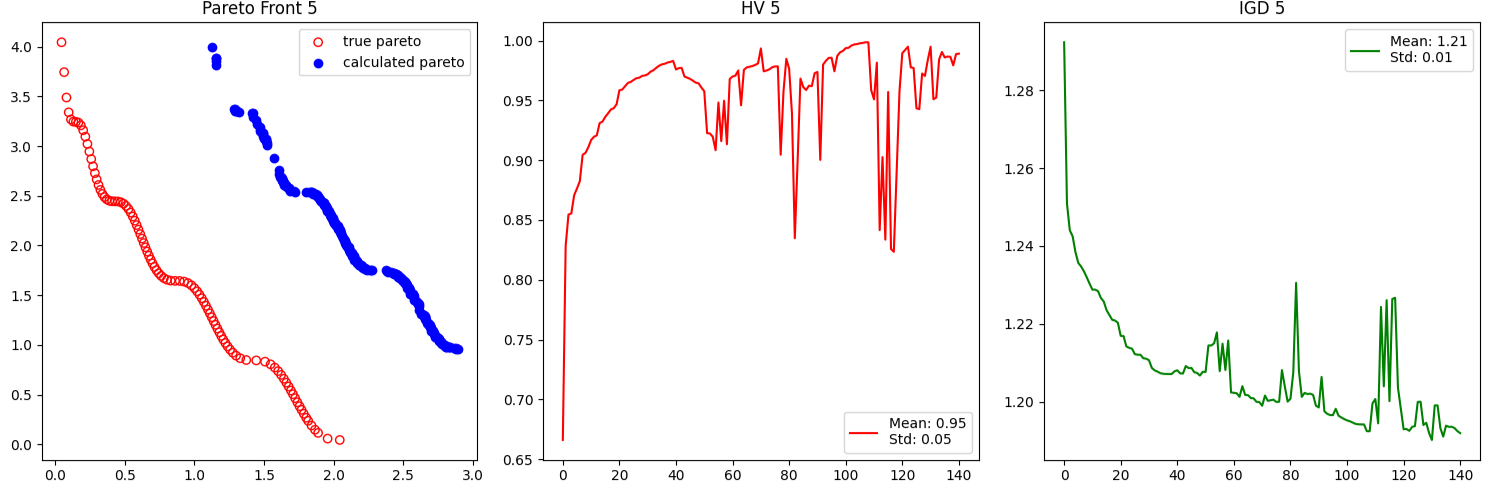
Each method within the `AGMOEA` class plays a specific role in the adaptive, grid-based multi-objective evolutionary optimization process, from initializing the population and managing subspaces to selecting parents for recombination and updating solution archives, all while dynamically adjusting the search strategy based on the evolving state of the solution population.

WFG-Evaluator Result:

WFG1:

Article:



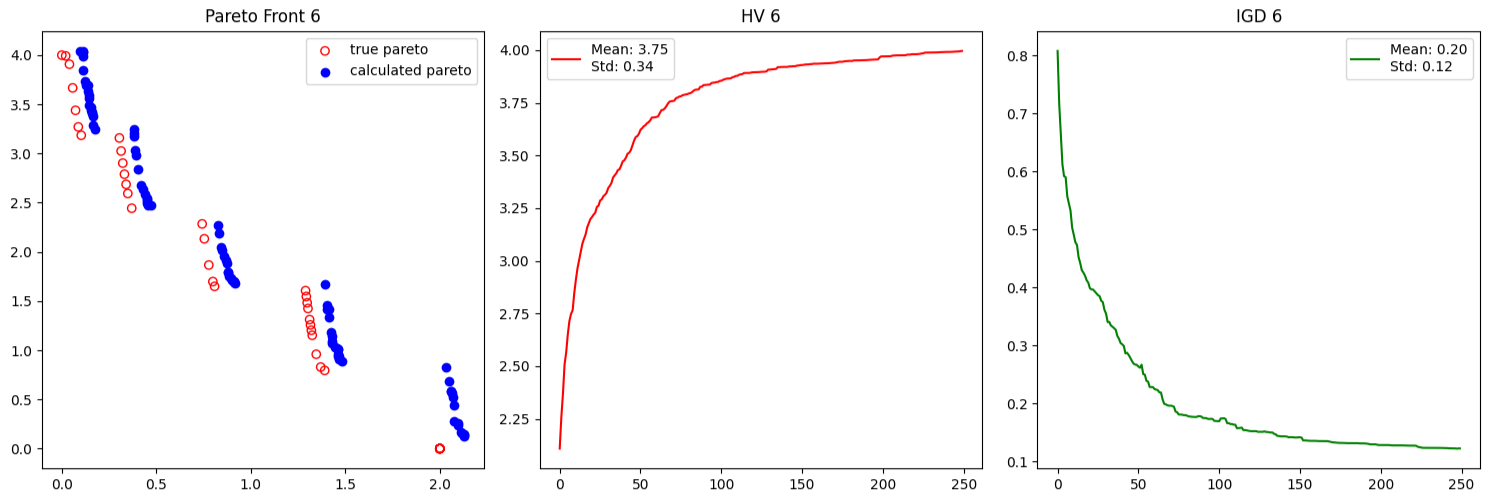
Current Implementation:

WFG2:

Article:



Current Implementation:

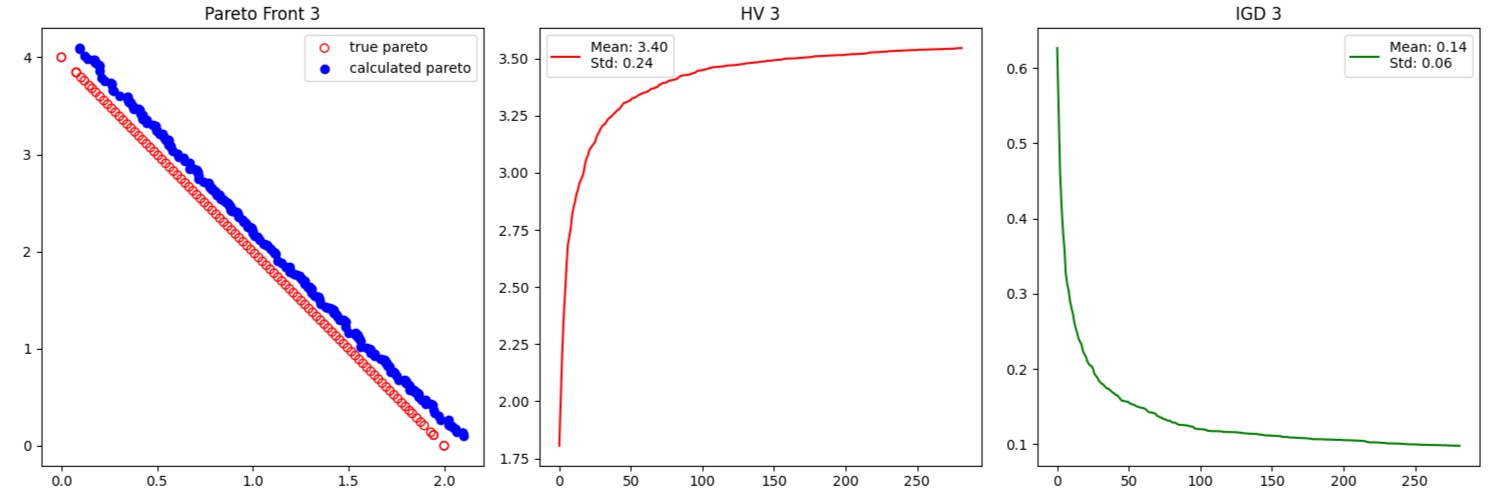


WFG3:

Article:



Current Implementation:

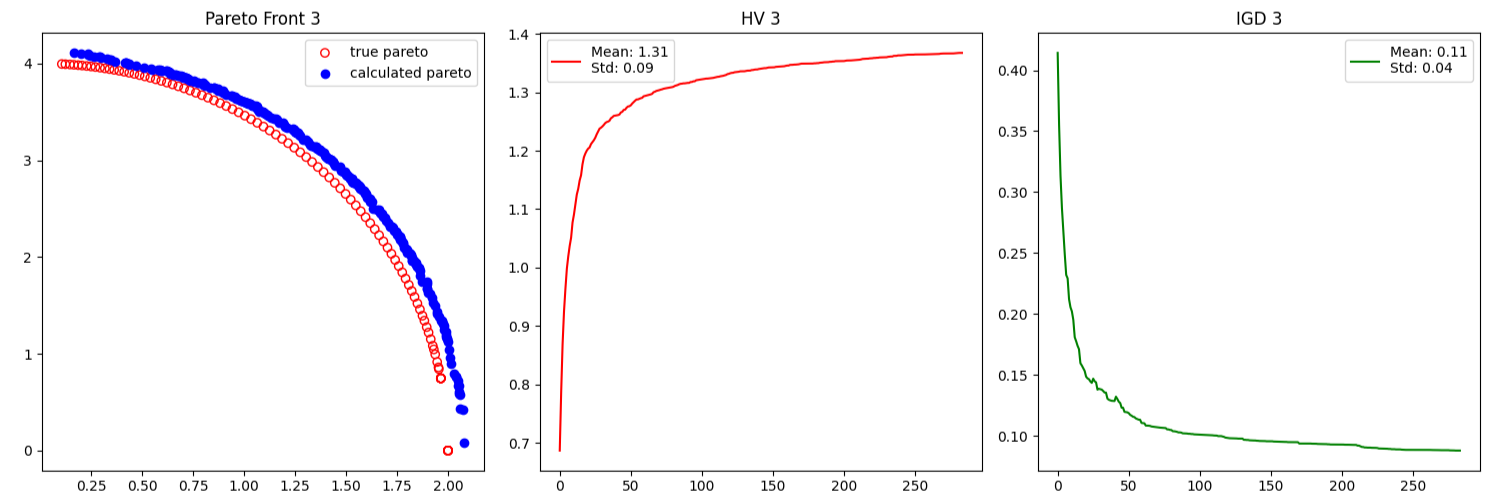


WFG4:

Article:



Current Implementation:



WFG5:

Article: