Multi-objective optimization

Jovan Vukićević

May 2024

Contents

1 Introduction

In mathematics and computer science, optimization problems consist of finding the optimal values of given objective functions within their domains, optionally constrained by certain rules. Generally, these problems are problems of minimizing or maximizing the value of a real function, while adhering to the constraints of the problem.

Many real problems require the optimization of multiple objective functions simultaneously. An issue that arises here is that, more often than not, these objective functions directly conflict with each other. Optimizing the "quality" of one function can lead to a reduction of "quality" in another. **Multi-objective optimization** is an area of optimization that specializes in solving these types of problems. One solution that Multi-objective optimization presents for these kinds of problems is giving each objective function a weight and creating one aggregated function out of them, reducing the problem to the optimization of a single objective function. However, in most cases, this solution is not ideal. A better solution is the use of **Pareto fronts**.

Given multiple objective functions, solution X1 dominates solution X2 if there doesn't exist an evaluation of one of the objective functions in which the value of the function for solution X2 is more optimal than the value of the function for solution X1. A solution is **Pareto-optimal** if there doesn't exist a solution that dominates it. The set of all Pareto-optimal solutions of a certain problem is called a **Pareto front**. Solutions belonging to the Pareto front are optimal in the sense that none of their objective functions can be optimized further without deoptimizing other ones. Once a Pareto front of a problem has been established, it is left up to the expert in the field of the problem to pick a solution from the front based on their prior knowledge.

There are many approaches to finding Pareto fronts. In this study, we will mainly be focusing on the use of evolutionary algorithms for this task. In the forthcoming pages, we will go over the core **brute force algorithm** for Pareto front finding, then a **"simple" genetic algorithm**, and finally the optimized **Non-dominated sorting genetic algorithm (NSGA-II)**.

2 Solutions

2.1 Brute force

To begin with, calling this algorithm the "brute force" algorithm might be a bit misleading. While it does do what a brute force algorithm would do in theory, in practice it is a simple algorithm for finding the Pareto front of the given set of functions and solutions. Furthermore, this algorithm is used in both of the coming genetic algorithms.

The algorithm works by comparing every solution with every other solution. Each solution that is not dominated by any other solution is added to the Pareto front. At the end of the algorithm, every all solutions have been compared and we are left with the Pareto front of the current set of solutions.

2.2 Simple genetic algorithm

The reason this algorithm is called the "simple" genetic algorithm is the fact that it uses the most basic concepts of genetic algorithms to solve the problem of multi-objective optimization, without using domain-specific efficient methods. In this section, we will go over the core genetic algorithm concepts and how they are implemented in this algorithm.

Firstly, a single individual of the population used in the genetic algorithm contains a single solution to the multi-objective problem. Instead of the usual fitness function, objects of the class Individual contain a function evaluating their solution in all objective functions.

The **selection** operator used in this algorithm is a tournament selection modification, where instead of calculating the most fit individual of the selected tournament, we calculate the pareto front of the tournament and choose a random individual belonging to it.

As the **crossover** operator, whole arithmetic recombination is used. Given two parents and the parameter α which takes value between 0 and 1, the children's i-th vector-solution element is calculated by the formula:
$child1.code[i] = parent1.code[i] * \alpha + parent2.code[i] * (1 - \alpha)$,
$child2.code[i] = parent2.code[i] * \alpha + parent1.code[i] * (1 - \alpha)$

The **mutation** used is uniform mutation. If an individuals i-th vector-solution element is chosen for mutation, its mutated value is chosen uniformly from that element's domain.

Lastly, the **elitism** operator used in this algorithm consists of keeping the previous generations' Pareto front elements in the next generation.

With all the operators explained, the entire genetic algorithm works by generating a population of individuals with random solution values, and then, until the maximum number of generations has been reached, using the aforementioned operators to generate new "better" populations. At the end of the algorithm, the Pareto front of the last generation is calculated and returned.

2.3 Non-dominated sorting genetic algorithm (NSGA-II)

Non-dominated sorting genetic algorithm, or NSGA-II for short, is a genetic algorithm optimized for multi-objective optimization which bases itself on the concept of sorting the population by non-dominance and Euclidean distance.

The main concept of **non-dominated sorting** in this algorithm works by finding the Pareto front of the population and giving them all members of the front a ranking, then finding the pareto front of the remaining population and giving it a rank lower than the one before, and so on until there are no unranked elements. After that, the individuals belonging to the same fronts are ranked by their Euclidean distance from one another, making more distant elements better ranked, giving the opportunity for more diverse final Pareto fronts.

Like in the previous algorithm, the **selection** operator used is tournament selection, but this time the selected individual from the tournament is the highest ranked individual by non-dominance, and if more of one of the highest ranks exists in the tournament, then the one with the highest **crowding distance** is selected.

The **crossover** operator used in NSGA-II is simulated binary crossover. Given two parents and the parameter η, a random value u from 0 to 1 is selected, and depending on its value, β is calculated by the formula:
$\beta = (2 * u)^{\frac{1}{\eta+1}}, u \leq 0.5$
$\beta = (\frac{1}{2*(1-u)})^{\frac{1}{\eta+1}}, else$
After β is calculated, the children's i-th vector-solution element is calculated by the formula:
$child1.code[i] = 0.5 * ((1 - \beta) * parent1.code[i] + (1 + \beta) * parent2.code[i]),$
$child2.code[i] = 0.5 * ((1 + \beta) * parent1.code[i] + (1 - \beta) * parent2.code[i])$

The **mutation** operator used in NSGA-II is polynomial mutation. If an individuals i-th vector-solution element is chosen for mutation, its mutated value is calculated by the formula:
$indiv.code[i] = indiv.code[i] + (indiv.ub[i] - indiv.lb[i]) * \delta$
where lb and ub are that elements lower and upper bounds and δ is calculated by once again picking a random value r between 0 and 1, and then, based on the parameter η, using one of the following formula:
$\delta = (2 * r)^{\frac{1}{1+\eta}}, r \leq 0.5$
$\delta = 1 - (2 * (1 - r))^{\frac{1}{1+\eta}}, else$

Values of the parameters η of both crossover and mutation operators have empirically been proven to be best between 10 and 30, noting that mutations η should be higher than the crossovers.

Lastly, **elitism** is done implicitly. The new population is mixed with the previous generations population and the combined population is sorted by non-dominance and crowding distance. After that, the "population_size" number of best elements of the joined population are chosen to survive in the next generation.

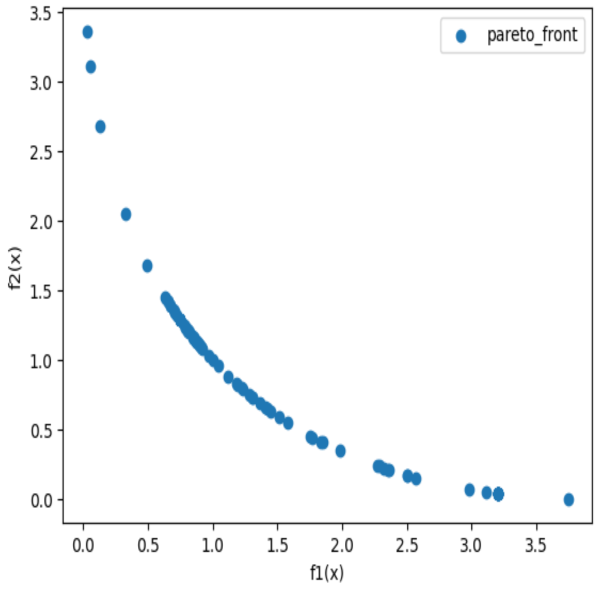
Just like in the previous algorithm, NSGA-II uses the aforementioned operators until the maximum number of generations is reached, and then returns the Pareto front of the final generation.

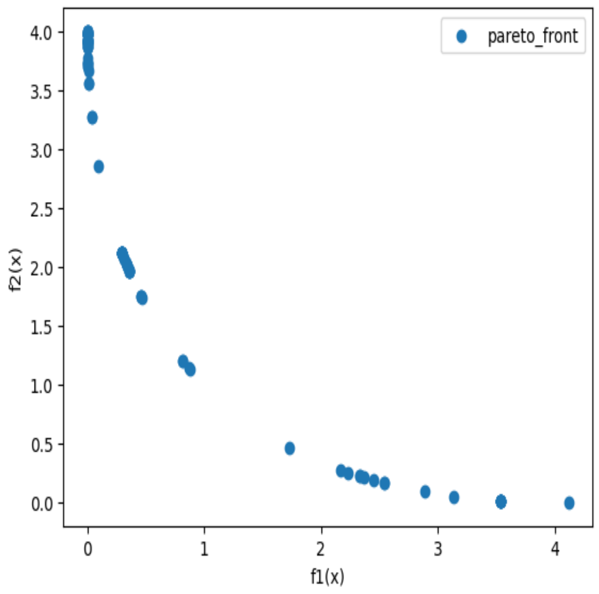
3 Test results

In this section, we will compare the previously explained genetic algorithms on the same set of commonly used multi-objective optimization test functions. For testing purposes, both genetic algorithms use the same parameter values when possible.

3.1 Schaffer function N.1

$$\text{Minimize} = \begin{cases} f_1(x) = x^2 \\ f_2(x) = (x-2)^2 \end{cases}, \text{ where } -A \leq x \leq A, A \in [10, 10^5]$$



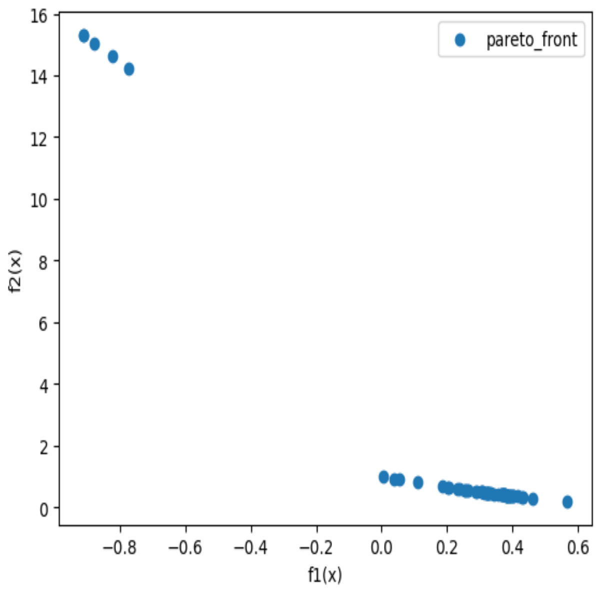


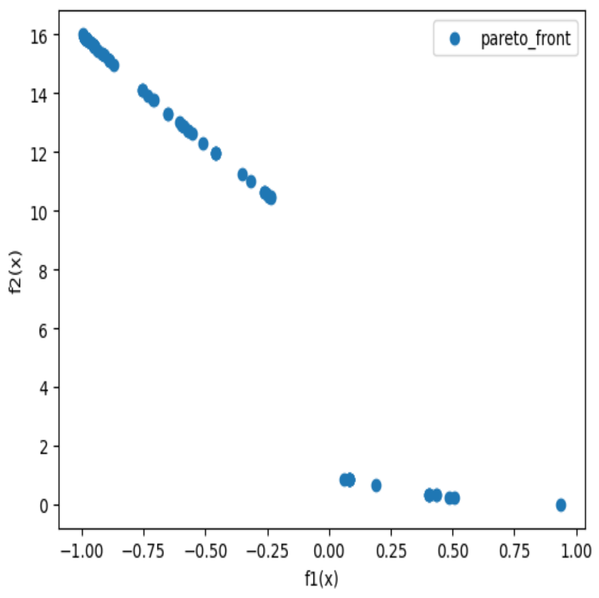
Algorithm	Time	Pareto elements found
GA	0.6868066787719727 s	90/100
NSGA-II	7.995919227600098 s	100/100

3.2 Schaffer function N.2

$$\text{Minimize} = \begin{cases} f_1(x) = \begin{cases} -x, & \text{if } x \leq 1 \\ x-2, & \text{if } 1 < x \leq 3 \\ 4-x, & \text{if } 3 < x \leq 4 \\ x-4, & \text{if } x > 4 \end{cases} \\ f_2(x) = (x-5)^2 \end{cases}$$

, where $-5 \leq x \leq 10$

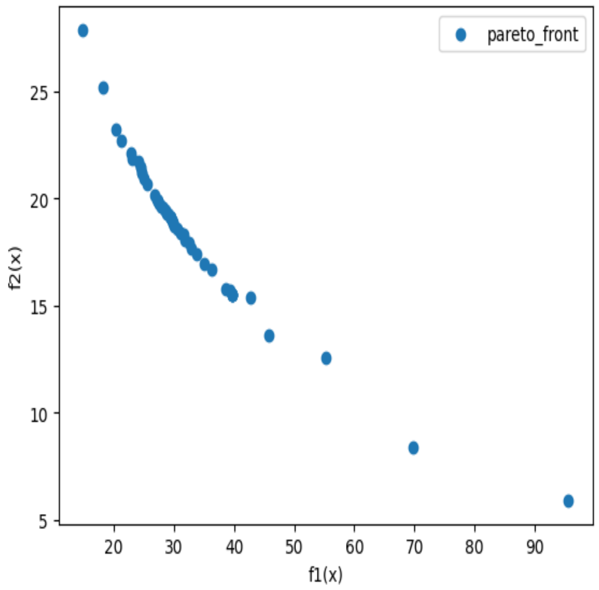


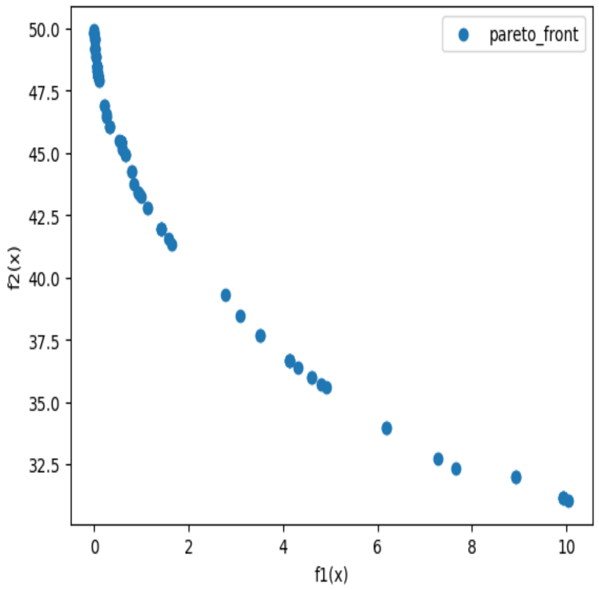


Algorithm	Time	Pareto elements found
GA	0.6683416366577148 s	62/100
NSGA-II	8.870128631591797 s	100/100

3.3 Bihn and Korn function

$$\text{Minimize} = \begin{cases} f_1(x,y) = 4x^2 + 4y^2 \\ f_2(x,y) = (x-5)^2 + (y-5)^2 \end{cases}, \text{ where } 0 \le x \le 5, 0 \le y \le 3$$

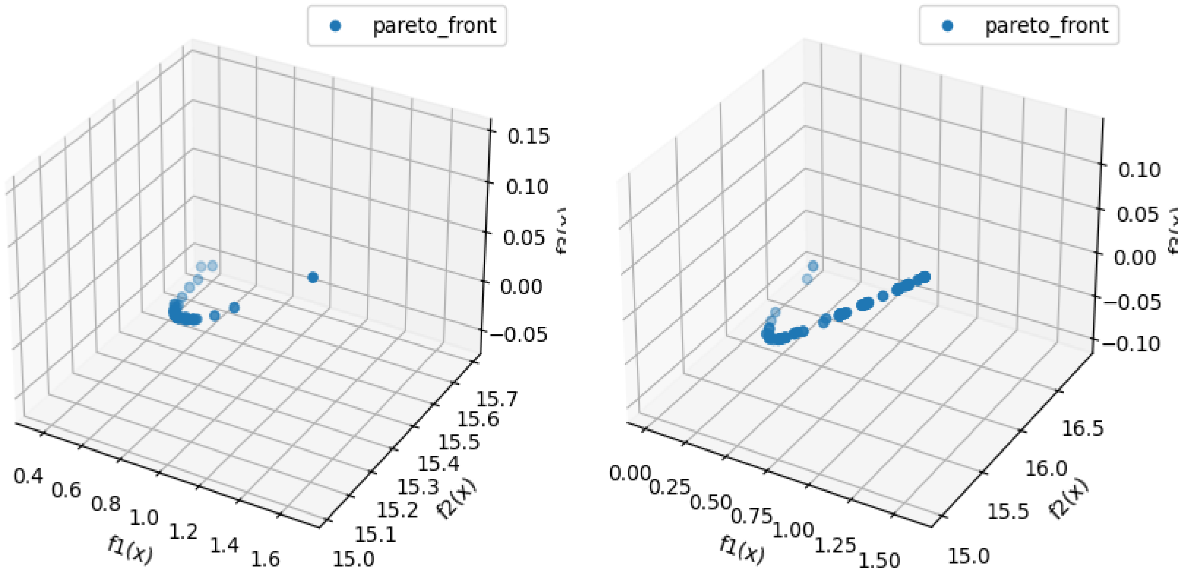




Algorithm	Time	Pareto elements found
GA	0.7873814105987549 s	56/100
NSGA-II	7.852182149887085 s	100/100

3.4 Viennet function

$$\text{Minimize} = \begin{cases} f_1(x,y) = 0.5\left(x^2+y^2\right) + \sin\left(x^2+y^2\right) \\ f_2(x,y) = \frac{(3x-2y+4)^2}{8} + \frac{(x-y+1)^2}{27} + 15 \\ f_3(x,y) = \frac{1}{x^2+y^2+1} - 1.1\exp\left(-\left(x^2+y^2\right)\right) \end{cases}, \text{ where } -3 \leq x, y \leq 3$$



Algorithm	Time	Pareto elements found
GA	0.995173454284668 s	55/100
NSGA-II	8.870128631591797 s	100/100

4 Conclusion

NSGA-II is a complex algorithm that trades off computation time for higher-quality results. As the problems we test become more complex, we can see that simple genetic algorithms can start to fail, while NSGA-II still stands strong. Even though the computation time of NSGA-II is higher than that of the simple genetic algorithm, the former in general produces more diverse and higher-quality Pareto fronts than the latter. For all the test functions given in this study, NSGA-II would find 100/100 Pareto front elements in

10 times fewer generations and similar time as to that of the simple genetic algorithm, but a higher number of generations was left to give the algorithm more chances to diversify the solutions and possibly find higher-quality Pareto fronts. The complexity of NSGA-II also reveals itself in its many adjustable parameters. It is incredibly hard to find the "right" combination of parameters for a given problem. Some parameter configurations that work well for a certain problem might not work as well for another problem.

All in all, multi-objective optimization is a very difficult problem with many real-life applications in domains such as economics or finances. The NSGA-II presented in this study is just one of many different solutions to this problem.

References

[1] Materials from the Computational Intelligence course, Faculty of Mathematics, University of Belgrade

[2] A FAST ELITIST MULTIOBJECTIVE GENETIC ALGORITHM: NSGA-II, Aravind Seshadri

[3] Test functions for optimization, Wikipedia