COMPUTER SCIENCE PROJECT

Evolutionary Multi-Objective Optimization

16 janvier 2026

Bidias Cabrel TIOTSOP NGUEGUIM, François Alexandre MOUSSINGA NDOUMBE

ÉCOLE POLYTECHNIQUE
INSTITUT POLYTECHNIQUE DE PARIS
TYPOGRAPHIX

TABLE DES MATIÈRES

ABSTRACT

Many real-world problems aim to optimize completely antagonistic functions. For instance, one might try to obtain the most number of items from a shop while spending the least amount of money. The first thing to notice is that not all possible pairs of values are comparable. Economists have developed a partial order called preference, which helps users make economic decisions in such cases, after maximizing the function associated with the problem. The question here is : how can we maximize the objective function ? The Non-Dominated Sorting Algorithm (NSGA) is a random algorithm that provides optima for a benchmark function. It is not purely deterministic, but according to the results in the article [1], it yields sufficiently good results on the benchmark mLOTZ (Leading Ones Trailing Zeros). The aim of this project is to implement the NSGA II algorithm and analyze the results on the mLOTZ function.

1
TASK 1 : IMPLEMENTATION OF INDIVIDUALS AND OBJECTIVE CLASSES

We choose to implement an individual as both an array of integer values and a String which's characters are either 0 or 1. For sure it doubles the storage complexity, but storing in string makes it easier to read and create examples and storing in arrays makes it easier to manipulate the individual, especially in the task 5, where we need to randomly flip the bits of the individuals.(For more details, For more details, see Appendix, Figure 1)

We choose to implement an ObjectiveValue as an array of floats, for the simple reason we don't know the limit of the values an objective value can take. After a little peek through the exercise, we could just use integers, since the values of the mLOTZ are integers. (Fore more details, see 2)

2
TASK 2 : IMPLEMENTATION OF MLOTZ AND ITS PARETO SET AND PARETO FRONTS

Let m be an even integer, mLOTZ takes an individual of size n such that $\frac{m}{2}$ divides n and returns and objective value in $\mathbb{N}^m$.
For m=2 it returns a couple of integers where the first coordinate is the number of consecutive 1 starting from the first position while the second coordinate id the number of consecutive 0 starting from the las position. It is easy to see that for an individual I1 of size n(even) LOTZ returns a couple of the form (a, b) with a, $b \in \mathbb{N}$ and $a + b \leq n$. We can already see that $a + b < n \Rightarrow b < n - a$. Therefore an individual whose objective value is $(a, n - a)$ is stricly superior to I1. It suffices to take a table with a consecutive 1 and $n - a$ consecutive 0 starting from the last position. It is clear that its objective value is $(a, n-a)$. In addition to that let u a couple of the form $(c, n - c)$ with $c \in [n]$ and $v = (v1, v2)$; $v1, v2 \in \mathbb{N}$ such that $u < v$. Either $c < v1$ and $n - c \leq v2$ or $n - c < v2$ and $c \leq v1$, Either way $n < v1 + v2$ so v can not be an objective value of the LOTZ function. We have therefore proven that the Pareto Front is exactly $P_n = \{(a, n - a); a \in [n]\}$. And by the same construction as before we see that the Pareto Set is $P'_n = \{(1, ..., 1, \underbrace{0, ..., 0}_{atimes\ n-atimes});\ a \in \mathbb{N}$İt is direct that $card(P_n) = card(P'_n) = n + 1$. It is easy to see that the run time compexity is O(n) because we have two traversals of the array with one starting from the first position and the other from the last and will meet one at at most one cell which will end both traversals as one is searching for consecutive ones while the other for consecutive zeroes meaning that neither traversal can continue.
Now let m be an even integer, n an integer such that $\frac{m}{2}$ divides n and $k = \frac{m}{2}$. mLOTZ is obtained by dividing an array(representation of an individual) of size n in k subarrays of sizes $\frac{2n}{m}$ and by applying the previously seen LOTZ on each subarray. The objective value results of the element in $\mathbb{N}$ obtained by merging the k couples in order. According to the preceeding paragraph we already know what are the Pareto optima for each subarray. We prove now that for an objective value to be a Pareto optimum of mLOTZ, all its k subarrays must be Pareto optima of LOTZ. It is actually direct because if an indivdual I1 represented by an array A has let say of its first subarray A1 which is not a Pareto optimum then by taking an individual I2 which is represented by an B array equal to A except from the first subarray B1 which is a Pareto optimum strictly greater than A1(Such subarray exists by what preceeds), We have an individual I2 strictly greater than I1. In other words we have the Pareto Fronts $P_{n,m} = P^k_{\frac{2n}{m}}$ and the Pareto Set $P'_{n,m} = P'^k_{\frac{2n}{m}}$. $card(P_{n,m}) = card(P'_{n,m}) = (\frac{2n}{m} + 1)^k$. It is clear that the run time is still O(n).

3
TASK 3 : IMPLEMENTATION OF THE NON DOMINATED SORTING

For the non-dominated sorting, we use the fast non-dominated sorting algorithm presented in the article [1] (see page 4, Algorithm 1)

Algorithm 1 Fast-non-dominated-sort(S)

Require: $S = \{S_1, \ldots, S_{|S|}\}$: the set of individuals
Ensure: $F_1, F_2, \ldots, F_I$
for $i = 1, \ldots, |S|$ **do**
  $ND(S_i) = 0$ ▷ number of individuals strictly dominating S_i
  $SD(S_i) = \emptyset$ ▷ set of individuals strictly dominated by S_i
  $OVals(S_i) = f(S_i)$ ▷ Objective values the indivdual S_i
end for
for $i = 1, \ldots, |S|$ **do** ▷ compute ND and SD
  for $j = 1, \ldots, |S|$ **do**
    if $Ovals(S_i) \prec Ovals(S_j)$ **then**
      $ND(S_i) = ND(S_i) + 1$
      $SD(S_j) = SD(S_j) \cup \{S_i\}$
    end if
  end for
end for
$F_1 = \{S_i \,|\, ND(S_i) = 0,\ i = 1, 2, \ldots, |S|\}$
$k = 1$
while $F_k \neq \emptyset$ **do**
  $F_{k+1} = \emptyset$
  for any $s \in F_k$ **do** ▷ discount F_k from ND and SD
    for any $s' \in SD(s)$ **do**
      $ND(s') = ND(s') - 1$
      if $ND(s') = 0$ **then**
        $F_{k+1} = F_{k+1} \cup \{s'\}$
      end if
    end for
  end for
  $k = k + 1$
end while

The main difficulty here was the data structures we needed to use. We consider the original S as a HashSet, so that it's elements are distinct, and thus, adding and retrieving elements will be in time complexity O(1). The classified set of the different $(F_i)_{i\in[I]}$ is considered as a Vector of HashSets.
We will also have a prepared HashMap with the objective values of all the individuals, to avoid having an extra N factor in the complexity.
ND will be represented as a HashMap, the key being the individual and the value being the number of individuals strictly dominating the individual S_i. ad SD will be a HashMap where the keys are the individuals and the values will be HashSets of individuals strictly dominated by S_i

3.1 Analysys of time and complexity

Let N be the number of individuals.

3.1.1 • Analysis of the time complexity

— The first loop (line 1-5) trivially excecutes in $O(N^2)$, since we added the computation of the objective values of each individual.
— The second loop (line 6 - 13) excecutes in $O(N^2)$ steps, since adding elements to $ND(S_i)$ and $ND(S_i)$ is in $O(1)$, but the comparison between $f(S_i)$ and $f(S_j)$ is supposed to be made in $O(1)$.
— The last loop excecutes in $O(N^2)$ because a set F_i has at most N elements and each element is strictly dominated by at most $N-1$ elements.

Thus the time complexity is $O(N^2)$. Though we should note that the complexity is clearly pseudo polynomial, but in our case, we'll not use large enough values of M.

3.1.2 • Analysis of space complexity

The space complexity is $O(N^2 + Nm)$, since $SD(S_i)$ has at most N elements, and $Ovals(S_i)$ is stored in an array of m elements.

4
TASK 4 : IMPLEMENTATION OF THE CROWDING DISTANCE

The algorithm proposed for the crowding distance was quite straightforwad. Once again, finding the most suitable datastructure was the central part of the problem. We used an 2-D Array to store Individuals in the form of their respective hashcode and each coordinate of their resulting objective values. This was made so as to sort them by rows respective to each coordinate of the objectivevalues. We returned for implementation purposes an array of pairs containing individuals and crowding distance.

The pseudo code for the algorithm is described in algorithm 2

5
TASK 5 : IMPLEMENTATION OF THE NSGA II

We had many doubts on why our algorithms didn't perform very well, for we didn't have lots of elements in the final Pareto Front. For $n = 20$ and $m = 4$, we had around 90% of the Pareto front. We were expecting to have a larger approximation of the front. That led us to push a little bit the number of iterations.

According to theorem 7 in the article [1], if we note T the number of iterations before we obtain the Pareto front, for $m = 2$, we have

$$Pr\left(T \geq \frac{2e^2}{e-1}(1+\delta)\right) \leq e^{-\frac{\delta^2}{2(1+\delta)}(2n-1)}$$

Algorithm 2 crowdingDistance(F)

Require: $F = \{S_1, \ldots, S_{|F|}\}$: the set of individuals to compute the crowding distance
Ensure: Array T of pairs of individuals with their crowding distance

data = Array containing pairs of individuals and their objective value
H := Hashmap containing Individuals as keys and crowding ditance as value
T := Array of Pairs of Individuals and crowding distances
for $i = 1, \ldots, n$ **do**
    data(i) = Pair(S_i, $f(S_i)$)
    H.add(S_i,0)
end for
for $i = 1, \ldots, n$ **do**
    sort(data, with respect to f_i)
    data =[Pair($S_{i(1)}$,$f(S_{i(1)})$),. . .,Pair($S_{i(|F|)}$,$f(S_{i(|F|)})$)]
    H.add($S_{i(1)}, +\infty$)
    H.add($S_{i(|F|)}, +\infty$)
    for $j = 2, \ldots, n-1$ **do**
        H.add($S_{i(j)}$,H.get($S_{i(j)}$) + $\frac{f_i(S_{i(j)}^{(j+1)}) - f_i(S_{i(j)}^{(j-1)})}{f_i(S_{i(j)}^{|F|}) - f_i(S_{i(j)}^{(1)})}$)
    end for
end for
for $l = 1, \ldots, |S|$ **do**
    T(l) = Pair(S_l, H.get(S_l))
end for
return T

When we take $\delta = 0.04$, we obtain $Pr(T \geq 9n^2) \leq 0.96$ for $n = 2$. Thus $9n^2$ isn't a good boundary for the number of iterations. That's why we prefered $18n^2$, which corresponds to the value $\delta = 1$, and leads to a better exponential factor.

We would like to point out one last thing.When breaking ties uniformly after having broken ties with the crowding distance. We chose to store the "new critical set" in an array, shuffle it randomly and then choose its last k elements. We did it this way because in java removing the last element of an ArrayList is done in constant time

6
TASK 6 : TESTS OF THE NSGA II

For $m = 4$, we tested for the values $n = 10$ and $n = 20$. The images of the tests are in the appendix (see 3, 5,??.

- For the value $n = 10$, we always obtain the complete pareto front.
- For the value $n = 20$, we obtained in average 95% of the Pareto front after 10 tests, which does not really correspond to the theory. We expected to have the complete pareto front with a probability of at least 99.99993%.
- It is sure that for values of $n \geq 24$, we would never obtain a larger proportion of the pareto front, since the number of iterations are not sufficient to visit all the set of individuals, even in the best of cases. Though it was not explicitely asked, we tested it for fun.

7
TASK 7 : MODIFIED VERSION OF NSGA II

7.1 Discussion on the algorithm

For this task we needed to implement a priority queue and use it for the modified version of the crowding distance.

Regarding the priority queue we implemented it as an array based heap. For its purpose in the modified crowding distance we chose a min-heap. We would like to point that out although we were asked to implement a decreaseKey method we rather chose the increaseKey method. We will explain why later.

For the last task, we had to compute the crowding distance of the elemnt in the critical Set, store it in our priority queue and then remove the individual with the smallest crowding distance while modyfing its neighbours. Those neighboors are the ones who preceed and follow the said individual on the sorted array respective to each coordinate($\mathbb{R}^n$). This means that for each individual we must store its lists of 2 neighbours by coordinates. We stored this information in an 2-D Array of n rows and 2 columns and in order to access it in constant time we used an HashMap with the Key being the individual and the value being the 2-D Array.
Regarding the modification we have to do when removing the minimum we clearly see that for each neighbours of the removed element, we have an increase of the difference in the index (in the sorted array) of the individuals used to compute the new crowding distance. Therefore the new value is always greater or equal to the last. For example let $F = x_1, x2, ..., x_s$ be our critical Set sorted (in ascending order) respective to an arbitrary coordinate i. Let's suppose x_2 is the Individual with the smallest crowding Distance. The new value of the crowding distance of x_3 would become $\frac{f_i(x_4)-f(x_1)}{f_i(x_s)-f(x_1}$ which is an increse of $\frac{f_i(x_2)-f(x_1)}{f_i(x_s)-f(x_1}$. We can similarly see that it is the same fpr the precceding neighbour. This is the observation that led us to implement an increaseKey method instead of a decreaseKey
One last remark is that we stored to additionnal informations so as to assure the runtime complexity. One was the objective Value of each individual so as not to recompute it again and for the second, the value $\frac{f_i(x_2)-f(x_1)}{f_i(x_s)-f(x_1)}$ used as the denominator when computing the crowding distance. We can see that it never changes because the first and last elemnet are the last to leave (their value is infinite).

7.2 Time Complexity

We can see that to the algorithm compute the crowding Distance like previously which is done in O(mNlogN), iterate through the critical set and increaseKey(O(log(size of the heap)) for the 2m neighbours of the removed elemnt. This leads to a runtime complextity of 2mlogN per iterations. This guarantees the O(mNlogN) complexity for our algorithm.

7.3 Spacial Complexity

For the spacial complexity as discussed before we have an Hashmap which stores the 2m neighbours which are essentially arrays of length m(O(m^2), an Hashmap which stores the respective objective value (O(m)) and an array which for each coordinate stores its unchanged denominator in the calculation of the crowding diastance When summing these for each individuals we have a spacial complexity of O(nm^2)

7.4 Comments

After running the NGSA2, with the original crowding distance algorithm, with its modified version and without any implemented crowding distance just by breaking ties randomly ; We obtained the best results when using the modified version but the difference is quite small and only for values of $n \geq 20$. It is clear nevertheless that breaking ties uniformly provides the worst results. The runtime complexity stays pretty much the same

RÉFÉRENCES

[1] Weijie Zheng and Benjamin Doerr. “Mathematical runtime analysis for the nondominated sorting genetic algorithm II (NSGA-II)”. In : Artificial Intelligence 325 (2023), p. 104016. doi :

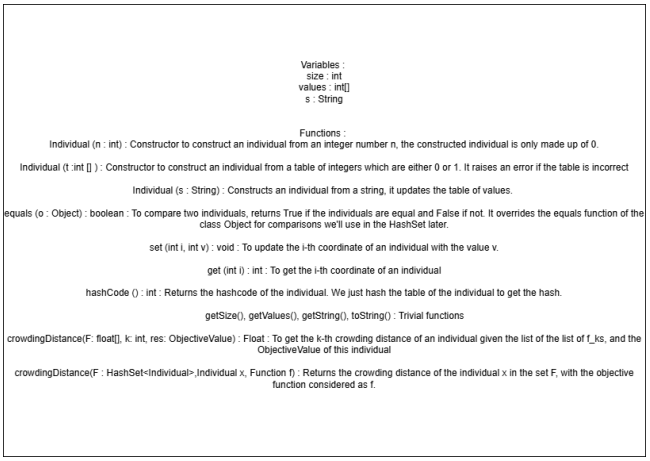

Figure 1 – Details of class Individual

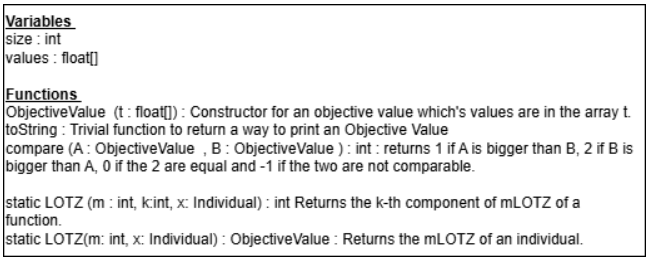

Figure 2 – Details of class ObjectiveValue

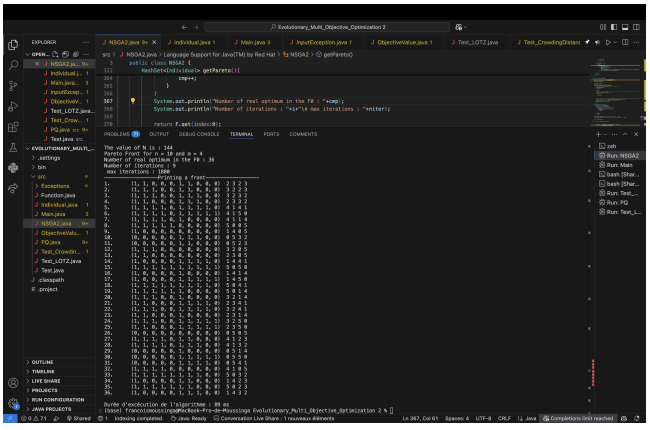

Figure 3 – Screenshot of the test for $n = 10$ and $m = 4$

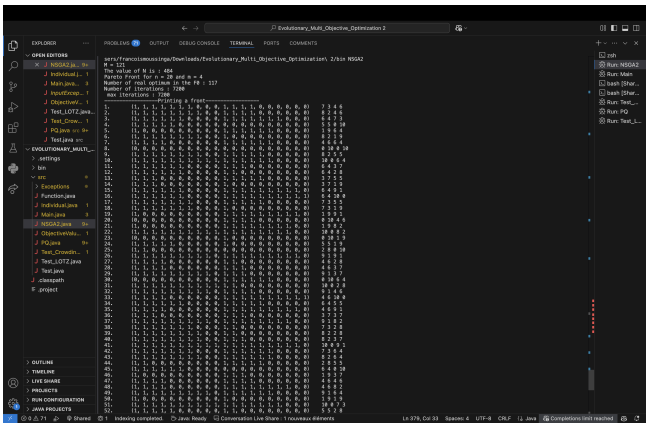

Figure 4 – First screenshot of the test for $n = 20$ and $m = 4$

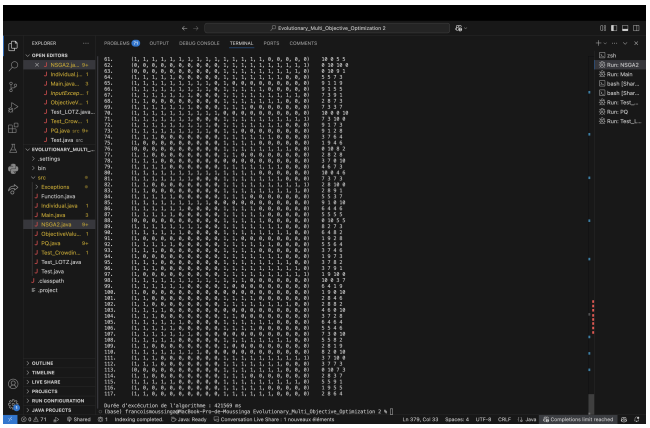

Figure 5 – Second screenshot of the test for $n = 20$ and $m = 4$

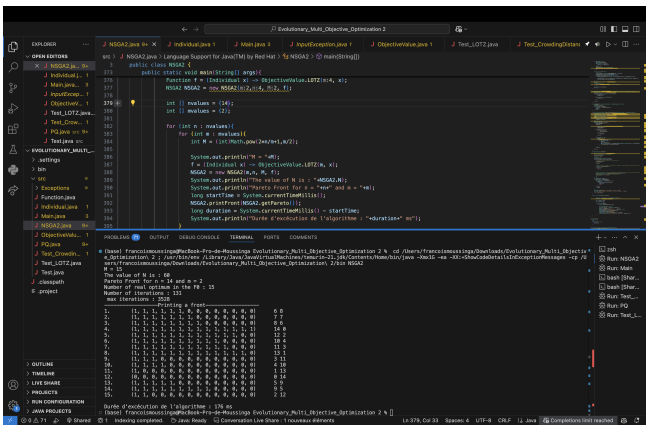

Figure 6 – screenshot of the test for $n = 14$ and $m = 2$

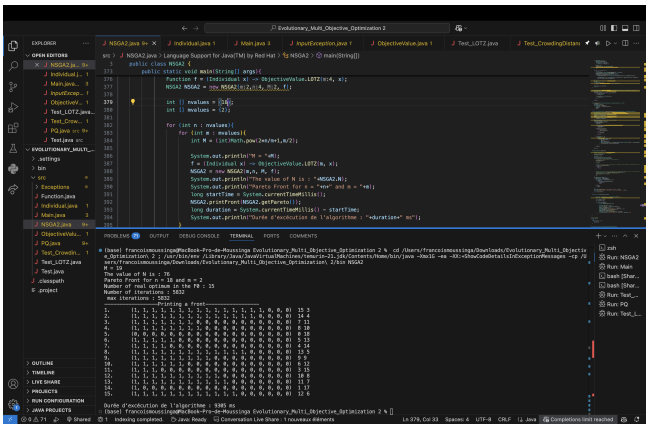

Figure 7 – screenshot of the test for $n = 18$ and $m = 2$