Evolutionary Multi-Objective Optimization

CSC_42021_EP Project

Contact: Martin (martin.krejca@polytechnique.edu)

General remarks. The source code for this project, the report, as well as the final defense need to be in **English.** You can use **any programming language** as long as the source code is formatted *and annotated* such that it is evident what it does even if one is not familiar with the language.

In general, you can **use any libraries** that you want. However, since some languages might provide shortcuts for certain tasks deemed important in this project, **you need to implement certain algorithms and data structures yourself.** This is stated explicitly in this document. When in doubt, reach out to me.

Please implement *efficient* and *legible* algorithms. It should be easy to understand what you are doing, but do not simplify to a degree where the performance starts to suffer.

If you do not know what programming language to use, consider D or Nim, not because they are best suited for this task, but because they are fun.

1 Context

Many real-world problems aim to optimize multiple, typically conflicting, objectives at once. Think, for example, of finding an option to get from one location to another *quickly* but also *cheaply*. In such a case, we need to deal with *incomparable* solutions, that is, solutions that are better in one objective but worse in another. We call each best incomparable solution a *Pareto optimum* and the set of all Pareto optima the *Pareto front* of the problem. We are interested in finding as many Pareto optima as possible, ideally the entire front.

A common method to approach multi-objective optimization problems are multi-objective evolutionary algorithms (MOEAs). MOEAs usually maintain a population of good solutions, which they aim to improve iteratively. This procedure maps very well to multi-objective problems, and thus a variety of MOEAs exist. In this project, we implement the *non-dominated sorting genetic algorithm II*[1] [DPA+02] (NSGA-II), which is with more than 50 000 citations on Google Scholar the most famous MOEA and also heavily used in practice.

1 As the saying goes, there are three major challenges in computer science: naming things and off-by-one errors.

2 Terminology and Benchmark Function

Before we introduce the NSGA-II, we discuss the framework surrounding the algorithm. As is common in the scientific community of MOEAs, we only regard *pseudo-Boolean* multi-objective functions, that is, functions that map bit strings to (floating-point) vectors. Formally, for problem size $n \in \mathbb{N}_{\geq 1}$ and number of objectives $m \in \mathbb{N}_{\geq 2}$, we consider functions $f\colon \{0,1\}^n \to \mathbb{R}^m$. We call each $\boldsymbol{x} \in \{0,1\}^n$ an *individual*, we call $f(\boldsymbol{x})$ the *objective value* of $\boldsymbol{x}$, and, for all $k \in [1,m] \cap \mathbb{N} =: [m]$, we define $f_k(\boldsymbol{x}) \coloneqq f(\boldsymbol{x})_k$ to refer to the value of $\boldsymbol{x}$ in objective k.

We compare objective values $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^m$ via a partial order called *domination*. We say that $\boldsymbol{u}$ *weakly dominates* $\boldsymbol{v}$ (denoted by $\boldsymbol{u} \succeq \boldsymbol{v}$) if and only if for all $k \in [m]$ holds that $\boldsymbol{u}_k \geq \boldsymbol{v}_k$. If and only if (at least) one of these inequalities is strict, we say that $\boldsymbol{u}$ *strictly dominates* $\boldsymbol{v}$ (denoted by $\boldsymbol{u} \succ \boldsymbol{v}$). Last, if and only if neither $\boldsymbol{u} \succeq \boldsymbol{v}$ nor $\boldsymbol{v} \succeq \boldsymbol{u}$, we say that $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *are incomparable*. We extend this terminology to individuals, where it applies to their objective value.

Given a pseudo-Boolean multi-objective function f as above and an individual $\boldsymbol{x} \in \{0,1\}^n$, we say that $f(\boldsymbol{x})$ is a *Pareto optimum* if and only if there is no individual in $\{0,1\}^n$ that strictly dominates $\boldsymbol{x}$. We call the set of all Pareto optima of f the *Pareto front* (of f), and we call the pre-image of all Pareto optima the *Pareto set* (of f).

Task 1. Implement individuals and objective values in a way that allows you to work properly with them. Especially, implement a comparison for objective values that compares them via domination as defined above. Decide which return values make sense for this procedure. Use this comparison to define the comparison among individuals.

Benchmark Function

We consider the LeadingOnesTrailingZeros [LTZ04] (LOTZ) benchmark, which is a simple bi-objective pseudo-Boolean function, mainly of interest for theoretical analyses. Its long name is very telling of how the function works. The first objective returns the number of consecutive 1s, starting from the first position of the given individual. Analogously, the second objective returns the number of consecutive 0s, starting from the last position. Formally, using for all $i, j \in \mathbb{N}$ the notation $[i..j] \coloneqq [i,j] \cap \mathbb{N}$, we have for all $k \in [2]$ that

$$\mathrm{LOTZ}_k\colon \boldsymbol{x} \mapsto \begin{cases} \max\{i \in [0..n] \mid \forall j \in [i]\colon \boldsymbol{x}_j = 1\} & \text{if } k = 1,\\ \max\{i \in [0..n] \mid \forall j \in [i]\colon \boldsymbol{x}_{n+1-i} = 0\} & \text{else.} \end{cases}$$

We extend this function to versions with $m \in \mathbb{N}_{\geq 2}$ objectives for even m such that $\frac{m}{2}$ divides n. We call the resulting function mLOTZ. Roughly, mLOTZ chunks each individual into $n' \coloneqq \frac{2n}{m}$ consecutive disjoint pieces and applies to each of these pieces the bi-objective LOTZ function (of problem size n'). To ease notation, for all $i, j \in [n]$ with $i \leq j$, let $\boldsymbol{x}_{[i..j]} \coloneqq (\boldsymbol{x}_\ell)_{\ell \in [i..j]}$ denote the substring of $\boldsymbol{x}$ from i to j. For all $k \in [m]$, we have

$$m\mathrm{LOTZ}_k\colon \boldsymbol{x} \mapsto \begin{cases} \mathrm{LOTZ}_1(\boldsymbol{x}_{[n'(k-1)/2+1..n'(k+1)/2]}) & \text{if } k \text{ is odd},\\ \mathrm{LOTZ}_2(\boldsymbol{x}_{[n'(k/2-1)+1..n'k/2]}) & \text{else.} \end{cases}$$

Note that for all $\ell \in [\frac{m}{2}]$, the objectives $2\ell - 1$ and 2ℓ make use of the same substring. Moreover, note that 2LOTZ is just LOTZ.

LOTZ and mLOTZ are both *maximization* problems. That is, larger values in an objective are preferably over smaller values.

Task 2. Let $m \in \mathbb{N}_{\geq 2}$ be even for this task. Implement mLOTZ.

In the report, write down the Pareto set and Pareto front of mLOTZ. Determine the cardinality of both sets. Argue why those solutions are Pareto optima.

3 NSGA-II

The NSGA-II (Algorithm 1) aims at optimizing a given pseudo-Boolean m-objective function f by maintaining a multi-set P (called a *population*) of $N \in \mathbb{N}_{\geq 1}$ individuals. Initially, P contains individuals drawn independently and uniformly at random. Afterward, the NSGA-II creates N new individuals each iteration (via *mutation*) and selects among the N previous individuals and the N new individuals the N overall best individuals (via *selection*). Due to the multi-objective nature of f, *best* is not necessarily straightforward to define. Roughly, the NSGA-II prefers individuals that are strictly dominated by as few individuals as possible. It breaks ties in a process that consists of three stages. In the following, we detail the different operations.

Mutation

The NSGA-II creates N new individuals by performing the following steps N times independently. First, it picks an individual $\boldsymbol{x}$ uniformly at random from the current population P. Then, it creates a copy $\boldsymbol{y}$ of $\boldsymbol{x}$ and inverts each bit of $\boldsymbol{y}$ independently with probability $\frac{1}{n}$. That is, for all $i \in [n]$, it holds that $\Pr[\boldsymbol{y}_i = \boldsymbol{x}_i] = 1 - \frac{1}{n}$ and $\Pr[\boldsymbol{y}_i = 1 - \boldsymbol{x}_i] = \frac{1}{n}$.

We note that the NSGA-II usually also creates new individuals by combining multiple individuals via a process known as *crossover*. However, we only consider mutation in this project.

Selection

As described above, selection aims at determining the N best individuals among the current population P and the N new individuals created by mutation (the *offspring population*). Let Q denote the union of the current and the offspring population. We sketch the entire selection process first before we explain the more complicated operators separately.

The selection process picks N individuals in a greedy manner from Q, making use of at most three different operations in order to break all potential ties. First, it selects those individuals in Q that are not strictly dominated by other individuals in Q and then continues to choose individuals that are only strictly dominated by the ones already chosen. The order in which to choose these individuals is determined by a process known as *non-dominated sorting* (which is from where the NSGA-II gets its name). Non-dominated sorting gives each individual a (not necessarily unique) *rank*. The NSGA-II selects all individuals of a rank, starting from the lowest rank, until it selected at least N individuals. If it selects exactly N individuals, we are done.

Algorithm 1: The non-dominated sorting genetic algorithm II (NSGA-II). Given the population size $N \in \mathbb{N}_{\geq 1}$, the problem size $n \in \mathbb{N}_{\geq 1}$, the number of objectives $m \in \mathbb{N}_{\geq 2}$, and the pseudo-Boolean multi-objective function $f\colon \{0,1\}^n \to \mathbb{R}^m$, the algorithm generates better individuals with respect to f over time. It stops after a user-defined termination criterion. Sets in the following always denote multi-sets.

```
1  $t \leftarrow 0$;
2  $P^{(t)} \leftarrow N$ individuals, each chosen independently uniformly at random from $\{0,1\}^n$;
3  while termination criterion not met do
4      for $i \in [N]$ do
5          $\boldsymbol{x}^{(t,i)} \leftarrow$ individual chosen uniformly at random from $P^{(t)}$;
6          $\boldsymbol{y}^{(t,i)} \leftarrow$ copy of $\boldsymbol{x}^{(t,i)}$, flipping each bit independently with probability $1/n$;
7      $(F_i)_{i \in [I]} \leftarrow$ fronts returned by non-dominated sorting of $P^{(t)} \cup \{\boldsymbol{y}^{(t,i)} \mid i \in [N]\}$;
8      $i^* \leftarrow \min\{i \in I \mid |\bigcup_{j \in [i]} F_j| \geq N\}$;
9      $C^{(t)} \leftarrow N - |\bigcup_{i \in [i^*-1]} F_i|$ individuals from $F_{i^*}$ with the largest crowding distance
         (in $F_{i^*}$), breaking ties uniformly at random;
10     $P^{(t+1)} \leftarrow C^{(t)} \cup \bigcup_{i \in [i^*-1]} F_i$;
11     $t \leftarrow t + 1$;
```

Otherwise, we break ties via a process known as *crowding distance*. To this end, the algorithm only needs to break ties in the largest *(critical)* rank i^* among all currently selected individuals.

The crowding distance gives each of the individuals among which we break ties a value (the crowding distance), based on where each individual lies within the competing ones. Roughly speaking, the crowding distance penalizes individuals that are close to other individuals. Like the ranks from non-dominated sorting, the crowding distance of each individual does not need to be unique. The NSGA-II greedily selects individuals, starting with those with the largest crowding distance, until it selected N individuals overall this iteration.

Since breaking ties via the crowding distance can still lead to ties, the NSGA-II makes use of a final tie breaker. This time, it breaks ties uniformly at random.

Non-Dominated Sorting Non-dominated sorting applied to a population Q assigns each individual in Q a rank and clusters individuals into equivalence classes of the same rank, called *fronts*. We assume that there are in total $I \in \mathbb{N}_{\geq 1}$ fronts in Q.

The fronts are defined inductively, with the first front F_1 denoting all non-dominated individuals in Q, and all other fronts denoting the non-dominated individuals among the remaining ones. Formally, for all $i \in [I]$ holds $F_i \coloneqq \{\boldsymbol{x} \in Q \setminus \bigcup_{j \in [i-1]} F_j \mid \forall \boldsymbol{y} \in Q \setminus \bigcup_{j \in [i-1]} F_j \colon f(\boldsymbol{y}) \not\succ f(\boldsymbol{x})\}$.

Task 3. Implement non-dominated sorting. When given a population of size $N \in \mathbb{N}_{\geq 1}$, the run time of your algorithm should be $O(N^2)$, assuming constant run time per comparison. Explain why your implementation satisfies this bound.

The *rank* of an individual is the index of its front. We recall that the *critical rank* is the smallest rank such that the union of the first i^* fronts contains at least N individuals. If these are more than N individuals, we need to break ties in F_{i^*} via the crowding distance.

Crowding Distance Crowding distance assigns each individual in a given population F a non-negative number or the value (positive) infinity. The crowding distance of an individual is defined as the sum of the *crowding distance of each objective.*

Let $k \in [m]$. The crowding distance of $\boldsymbol{x}$ for objective k is based on the relative fitness difference of the neighbors of $\boldsymbol{x}$ when sorting F with respect to objective k. That is, let $\{\boldsymbol{x}^{(i)}\}_{i \in [|F|]}$ denote F after sorting it based on f_k in ascending order.[2] We assign the two border elements $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(|F|)}$ each a crowding distance (in objective k) of positive infinity. For all $i \in [2..|F|]$, we assign $\boldsymbol{x}^{(i)}$ a crowding distance (in objective k) of

$$\frac{f_k(\boldsymbol{x}^{(i+1)}) - f_k(\boldsymbol{x}^{(i-1)})}{f_k(\boldsymbol{x}^{(|F|)}) - f_k(\boldsymbol{x}^{(1)})}.$$

Do not forget that the final crowding distance of an individual is the sum of the crowding distance over all objectives!

Task 4. Implement the crowding distance. When given a population F of size $N \in \mathbb{N}_{\geq 1}$, the run time of your algorithm should be $O(mN \log N)$, assuming a constant run time for evaluating f.

The NSGA-II breaks ties in F_{i^*} with respect to the crowding distance, preferring larger values. If there are still ties among the individuals that need to be selected, those are broken uniformly at random.

4 Performance Analysis

We are interested in how well the NSGA-II performs on mLOTZ. That is, we are interested in how long it takes until the algorithm's population P contains at least one individual for each Pareto optimum of mLOTZ. We then say that P *covers* the Pareto front. We stop the NSGA-II once its population covers the Pareto front of mLOTZ. Moreover, we specify further below a maximum number of iterations after which we stop the algorithm as well.

Task 5. Implement the NSGA-II with the above-specified termination criterion.

Let M denote the size of the Pareto front of mLOTZ, determined in Task 2. In your experiments, please always choose $N = 4M$, which theoretically guarantees success for LOTZ after a sufficient number of iterations [ZD23]. As maximum number of iterations, choose $9n^2$ (which is a big

2 Note that we may also have ties when sorting. We do not specify how those ties are broken.

number). When running your experiments, please use deterministic seeds for your random-number generators to make your results reproducible. This is good scientific practice.

Task 6. Run the NSGA-II on LOTZ and 4LOTZ with the setting explained above. Choose at least two different values for n, and perform multiple runs per value of n. Explain why you chose these values.

Report in a way that seems useful to you how much of the Pareto front the algorithm covers over the course of the run. How long does it take to cover the Pareto front? Is it always successful?

A Modification to the NSGA-II

A major flaw in the NSGA-II is that the crowding distance is not updated whenever an individual is selected. This way, the algorithm does not account for the change observed in a population when sequentially selecting individuals. An easy way to fix this is by re-computing the crowding distance every time we make a decision. Since we update the crowding distance repeatedly, the order of operations matters. Here, in contrast to what line 9 says, we do not take the $N - |\bigcup_{i \in [i^*-1]} F_i|$ individuals with the largest crowding distance but remove instead the $|F_{i^*}| - (N - |\bigcup_{i \in [i^*-1]} F_i|)$ individuals with the smallest crowding distance (and choose eventually all remaining individuals for the next population). Every time we remove an individual in F_{i^*}, we re-compute the crowding distance of all remaining individuals in F_{i^*}. When determining which individual has the smallest crowding distance, we break ties uniformly at random. Note that if we do not re-compute the crowding distance, then this results logically in the original NSGA-II.

The intention of this operator is to represent the crowding distance truthfully whenever individuals are removed, an idea that was co-suggested from Ecole Polytechnique [ZD24].

Re-computing may seem costly at first, but since an individual affects the crowding distance of at most $2m$ other individuals when it is removed, we do not have to re-compute *all* crowding distances but just those of the affected individuals. By using a priority queue (and some extra neighborhood information per individual), the crowding distance of each neighbor can be updated efficiently, and selection can be performed easily based on the queue.

Task 7. Implement a modified version of the NSGA-II that re-computes the crowding distance every time an individual with critical rank is removed. To this end, implement a priority queue as a binary heap, and make sure to support the operation *decrease key*. Your data structure should follow the performance guarantees mentioned by Wikipedia *(Binary)*. Your modified computation of the crowding distance should still have a run time of $O(mN \log N)$. Explain why your implementation satisfies this.

Run the modified version of the NSGA-II for the same setting as in Task 6. Report your results and observations.

What to Hand In?

Please hand in your (sufficiently well documented) code files as well as a PDF report. The report should complement your code. Ideally, it explains the most important details about how you solved each task. It should focus on the high-level ideas.

Note that you are sometimes also explicitly asked to justify certain choices or claims. This is always done in the report. When justifying run times, a formal proof is not necessary, but please be precise enough so that it is easy to follow why the claimed bound should hold.

When reporting on the results of the experiments, please also try to *interpret* them. That is, do not just write *what* you see, but also provide possible explanations for *why* you see it.

In case of any doubts, please send me an e-mail.

References

[DPA+02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017 (see page 1).

[LTZ04] Marco Laumanns, Lothar Thiele, and Eckart Zitzler. “Running time analysis of multiobjective evolutionary algorithms on pseudo-Boolean functions”. In: *IEEE Transactions on Evolutionary Computation* 8 (2004), pp. 170–182. DOI: 10.1109/TEVC.2004.823470 (see page 2).

[ZD23] Weijie Zheng and Benjamin Doerr. “Mathematical runtime analysis for the non-dominated sorting genetic algorithm II (NSGA-II)”. In: *Artificial Intelligence* 325 (2023), p. 104016. DOI: 10.1016/j.artint.2023.104016 (see page 5).

[ZD24] Weijie Zheng and Benjamin Doerr. “Overcome the difficulties of NSGA-II via truthful crowding distance with theoretical guarantees”. In: *CoRR* abs/2407.17687 (2024). DOI: 10.48550/ARXIV.2407.17687 (see page 6).