Rank-based Non-dominated Sorting

Bogdan Burlacu¹,²,³

¹University of Applied Sciences Upper Austria
²Heuristic and Evolutionary Algorithms Laboratory
³Josef Ressel Centre for Symbolic Regression

March 29, 2022

Abstract

Non-dominated sorting is a computational bottleneck in Pareto-based multi-objective evolutionary algorithms (MOEAs) due to the runtime-intensive comparison operations involved in establishing dominance relationships between solution candidates. In this paper we introduce Rank Sort, a non-dominated sorting approach exploiting sorting stability and ordinal information to avoid expensive dominance comparisons in the rank assignment phase. Two algorithmic variants are proposed: the first one, *RankOrdinal* (RO), uses ordinal rank comparisons in order to determine dominance and requires $O(N)$ space; the second one, *RankIntersect* (RS), uses set intersections and bit-level parallelism and requires $O(N^2)$ space. We demonstrate the efficiency of the proposed methods in comparison with other state of the art algorithms in empirical simulations using the NSGA2 algorithm as well as synthetic benchmarks. The *RankIntersect* algorithm is able to significantly outperform the current state of the art offering up to 30% speed-up for many objectives. C++ implementations are provided for all algorithms.

1 Introduction

Multiobjective optimization problems (MOP) consist of finding the best compromise or trade-off between a collection of competing objectives. Formally, a MOP can be defined as:

$$\text{Minimize } F(X) \text{ subject to } x \in X$$

where $F : X \rightarrow Y$, $X \in \mathbb{R}^n$ is the decision variable space (also called a feasible set) and $Y \in \mathbb{R}^m$ is the objective function value space.

Pareto optimality is an established formalism used to approach this type of problem, based on the concept of dominance. Given two vectors $y, z \in \mathbb{R}^m$, it defines the following dominance relationships:

- equality: $y = z \iff y_i = z_i, \forall i = 1, ..., m$ (2)
- strong dominance: $y < z \iff y_i < z_i, \forall i = 1, ..., m$ (3)
- weak dominance: $y \preceq z \iff y_i \leq z_i, \forall i = 1, ..., m$ and $y \neq z$ (4)

Thus, if $y \preceq z$ then we say that $y$ dominates $z$ ($z$ is dominated by $y$). If $y \ngeq z$ and $z \ngeq y$ then $y$ and $z$ are mutually non-dominated. A Pareto front represents a set of mutually non-dominated solutions.

Note that $<$ is a strict partial order and $\preceq$ is a partial order on the objective space $\mathbb{R}^m$. This allows to more generally describe Pareto relationships in terms of "predecessors" and "successors" as will be needed later on when Rank Sort is introduced.

When optimizing multiple objectives, it is often the case that a solution cannot be further optimized with respect to any objective without worsening at least one of the others. The concept of Pareto optimality represents an intuitive interpretation of this situation. If $x^*$ is a Pareto optimal solution, then there is no other solution $x \in X$ such that $x \preceq x^*$.

Evolutionary algorithms (EA) are one of the most popular methods for solving multi-objective optimization problems (MOP). In Pareto-based MOEAs, non-dominated sorting (NDS) refers to the process of dividing the population into Pareto fronts which are subsequently used to guide parent selection.

In this paper we present an approach to non-dominated sorting which can substantially reduce the runtime costs of the parent selection step. This is motivated by the fact that, due to its transactional nature and the need to carry
The remainder of the paper is organized as follows: Section 2 gives an overview of existing algorithms from the literature. Section 3 discusses the proposed approach, Section 4 provides experimental results and Section 5 is dedicated to conclusions.

2 Related work

Improvements in the asymptotic complexity of non-dominated sorting algorithms are mainly driven by new techniques to reduce (or eliminate altogether) the number of Pareto dominance calculations necessary to divide the population into Pareto fronts.

We distinguish between two main categories of approaches: those that perform direct dominance comparisons and those that perform sorting in order to determine the Pareto ranking of the population.

2.1 Inference-based approaches

Inference-based methods avoid unnecessary dominance comparisons by exploiting the transitive property of dominance relationships (e.g., if $a \prec b$ and $b \prec c$, then $a \prec c$) in order to improve the asymptotic complexity and runtime performance.

*Deductive sort (DS)* [McClymont and Keedwell, 2012] iterates over the solutions in natural order and maintains a strict order of assessment and comparison to ensure that all dominated solutions are discarded and not incorrectly added to the wrong front. Once a front has been filled, all solutions assigned to that front are ignored and the process is repeated until all solutions are assigned to a front.

*Efficient non-dominated sorting* [Zhang et al., 2015] avoids unnecessary dominance comparisons by comparing a solution that needs to be assigned to a front only with the other solutions that have already been assigned. Two algorithmic variants are proposed, differing in the search strategy used to find if any solution in a given front dominates the current solution. Two search strategies are proposed: sequential search (ENS-SS) and binary search (ENS-BS).

*Hierarchical Non-dominated Sorting (HS)* [Bao et al., 2017] sorts all solutions lexicographically and then uses successive rounds of comparison between the first solution and the succeeding solutions to establish dominance relationships. Dominated solutions are discarded in the current round of comparison so they will not be compared again with the first solution or other solutions which are non-dominated with the first solution. Once a solution is ranked it is removed from the comparison and added to its respective front.

2.2 Sort-based approaches

In contrast with inference-based approaches where dominance is explicitly considered, sort-based approaches rely on ordinal ranking information (in some form or another) in order to determine dominance relationships between solutions. This typically involves a more expensive preprocessing step where the solutions are sorted w.r.t. every objective.

The most notable examples of sort-based approaches are Best Order Sort (BOS) [Roy et al., 2016] and Merge Non-dominated Sort (MNDS) [Moreno et al., 2020]. Both approaches share the same principle of using a solution’s dominance set (i.e., the set of solutions dominating the current solution) to update its rank. The dominance set is obtained as the intersection between the sets of solutions ranked better than the current solution according to each objective. The ranking is obtained by stable sorting – a key characteristic which is also employed by our algorithm.

A key difference between BOS and MNDS is given by the actual implementation of the set intersection procedure for obtaining the dominance set. BOS uses a linked list to store a solution’s dominance set while MNSE uses a bitset. Both algorithms are deeply integrated with merge sort which they use in the sorting step and employ various other auxiliary data structures to keep track of already ranked solutions or duplicates (MNDS).

*Best Order Sort (BOS)* [Roy et al., 2016] sorts the population according to each objective preserving lexicographical order. The resulting ranking is used to compute the dominance set for each solution. Finally, a solution’s rank becomes the rank of the worst-ranked solution in the dominance set plus one.

---

1Implementation available at https://github.com/proteekroy/Best-Order-Sort
Table 1: Summary of algorithm time and space complexity. For DS, the worst case is $O(MN^3)$ cf. Mishra and Buzdalov Mishra and Buzdalov [2020].

| Algorithm | Best | Worst | Space |
|-----------|------|-------|-------|
| DS        | $O(MN^2)$ | $O(MN^3)$ | $O(N)$ |
| HS        | $O(MN\sqrt{N})$ | $O(MN^2)$ | $O(N)$ |
| MNDS      | $O(MN \log N)$ | $O(MN^2)$ | $O(N^2)$ |
| ENS-SS    | $O(MN\sqrt{N})$ | $O(MN^2)$ | $O(1)$ |
| ENS-BS    | $O(MN \log N)$ | $O(MN^2)$ | $O(1)$ |
| RO        | $O(MN \log N)$ | $O(MN^2)$ | $O(N)$ |
| RS        | $O(MN \log N)$ | $O(MN^2)$ | $O(N^2)$ |

Merge Non-dominated Sort (MNDS) [Moreno et al., 2020]. Employs a stable-sorting algorithm (merge sort) to rank solutions according to each objective and uses the ranking to construct a solution’s dominance set as the set of solutions that dominate the current solution in every objective. Then, the rank of the current solution is updated according to the worst rank in the dominance set. Prior to rank assignment, the algorithm removes duplicate solutions from the population and keeps track of them in a separate list. Finally, the ranks of the solutions are obtained based on the dominance sets and duplicates are inserted again with their corresponding rank.

A summary of the runtime complexity and space requirements of the described algorithms is given in Table 1.

3 Rank-based Non-dominated Sorting

Similar to MNDS and BOS, the Rank Sort algorithm uses the concept of a “dominance set” to compute solution ranks. However, while MNDS and BOS define the dominance set as the set of predecessors of the current solution, Rank Sort defines the dominance set as the set of successors of the current solution. Here, the notions of “predecessor” and “successor” correspond to the $\preceq$ and $\succeq$ Pareto relationships, respectively.

Considering the dominance set as the set of successors brings an important advantage: during rank update, it is sufficient to consider equally ranked successors of the current solution (considering provisional ranks iteratively computed by the algorithm). Consequently, the rank update becomes a simple increment operation of the successor’s rank. This insight simplifies the rank update phase and reduces the number of operations such that the total number of rank updates will be equal to the sum of ranks in the final Pareto front assignment.

Due to this rank update rule, the algorithm’s behavior can be interpreted as establishing each new Pareto front by moving dominated individuals from the current Pareto front into the new one.

We present two variants that differ in their space complexity:

- **RankOrdinal** (RO) requires $O(N)$ space by avoiding set intersections. Instead, it takes the smallest set of successors w.r.t. any objective $k = 1, ..., M$ as the dominance set:
  \[
  D_{\text{ordinal}}(S_i) = D_k(S_i), k = \arg \min_l |D_l(S_i)|
  \]  

- **RankIntersect** (RS) requires $O(N^2)$ space for storage of dominance sets and employs bit-level parallelism to speed up the computation of set intersections. The dominance set is defined as:
  \[
  D_{\text{intersect}}(S_i) = \bigcap_k D_k(S_i), k = 1, ..., M
  \]

Here, $D_k(S_i)$ represents the set of successors w.r.t. objective $k$:

\[
\text{neurips}D_k(S_i) = \{ S_j | \text{rank}(S_i) = \text{rank}(S_j) \text{ and } S_j \succeq S_i \}
\]  

Node that Equation 7 should be taken in the context of iterative rank assignment within the algorithm’s inner loop, where the ranks are not yet final.

**RankIntersect** is the faster algorithm overall however its space requirements might make it less suitable for resource-constrained environments, where **RankOrdinal** might be preferable.

\(^2\)Available in the open-source framework JMetal [https://github.com/jMetal/jMetal]
3.1 RankOrdinal Algorithm

*RankOrdinal* (RO) uses the comparison of ordinal ranks to establish dominance. Let \( i \in 1, ..., N \) be an index over the solutions \( S_i \), let \( k \in 1, ..., M \) be an index over the objectives, and let \( F_k(i) \) be a function returning the value of the \( k \)-th objective for solution \( S_i \).

For every objective \( k \), we compute the permutation vector \( p_k \) and ordinal rank vector \( r_k \):

\[
p_k = \left[ p_k(1) \ldots p_k(N) \right]^T \quad \text{such that} \quad F_k(i) \leq F_k(i+1) \tag{8}
\]

\[
r_k = \left[ r_k(1) \ldots r_k(N) \right]^T \quad \text{such that} \quad r_k(p_k(i)) = i \tag{9}
\]

Let \( P = [p_k^T] \in \mathbb{N}^{N \times M} \) and \( R = [r_k^T] \in \mathbb{N}^{N \times M} \) be the permutation matrix and ordinal rank matrix, where \( k = 1, ..., M \).

It is clear from Eq. 8 that for every \( p_k \in P \), the leftmost solution \( S_{p_k(1)} \) is always non-dominated and in general, \( S_{p_k(i)} \leq S_{p_k(j)} \) for every \( i < j \). Conversely, a solution \( S_{p_k(i)} \) can only dominate other solutions that come after it in the permutation. Therefore, it will be efficient to examine solutions left-to-right in the partial permutation where \( S_{p_k(i)} \) is farthest from the left, such that there remain fewer successors to be examined.

By virtue of Eqs. 8, 9, the matrix \( R \) helps infer a dominance relationship between two solutions:

\[
S_i \text{ non-dominated with } S_j, \quad \text{if } r_k(i) > r_k(j), \quad \forall k = 1, ..., M \tag{10}
\]

\[
S_i \leq S_j, \quad \text{if } r_k(i) < r_k(j), \quad \forall k = 1, ..., M \tag{11}
\]

Non-dominated sorting algorithms assign solutions to their corresponding Pareto fronts based on the *domination rank* value (see for example [Deb et al., 2002]). Testing for dominance in permutation order gives our approach the advantage of a natural rank update mechanism.

\[
S_i \leq S_j \text{ and rank}(i) = \text{rank}(j) \implies \text{rank}(j) \leftarrow \text{rank}(i) + 1 \tag{12}
\]

When the rank of \( S_j \) is updated, the rank of \( S_i \) will have already been updated at a previous iteration. The entire algorithm is described in pseudocode in Algorithm 1. The outer loop iterates over solution indices given by \( p_1 \) (the first column of \( P \)). The inner loop of the algorithm will iterate over the smallest set of successors, as defined by \( 5 \). The corresponding column of \( P \) and the start index for the partial permutation are identified at line 6 in Algorithm 1. The rank assignment procedure is performed at lines 9–11. The comparison of ranks \( r_k \) is actually an element-wise comparison between the columns of matrix \( R \) corresponding to the two solutions (fast in practice due to vectorization and the fact that the elements are integers).

### 3.1.1 Solution equality

Note that due to its reliance on sorting, ordinal ranking is unable to detect the situation when two solutions are equal in all objectives. To handle this case it is necessary to extend for example Eqs. 11 and 12 to explicitly check for equality. This can be cheaply implemented by e.g. hashing objective values in the preprocessing phase. However, it is a better design choice to separate duplicates handling logic from non-dominated sorting.

### 3.1.2 Correctness of the algorithm

The permutations of solutions w.r.t. every objective are generated by calling a sorting procedure. It is important to note that the algorithm is correct only when the sorting procedure is stable, as otherwise Eqs. 10, 11 would not be reliable. This issue is apparent in certain extended definitions of dominance such as \( \epsilon \)-dominance [Laumanns et al., 2002] where in the absence of stability there can be no guarantee about the order in which \( \epsilon \)-dominated solutions are added to their respective fronts.

Many non-dominated sorting algorithms explicitly make use of sorting in order to avoid the cost of dominance comparisons in at least one dimension [Zhang et al., 2015, Roy et al., 2016, Bao et al., 2017, Zhou et al., 2017, Xue et al., 2020, Moreno et al., 2020]. However, only a few explicitly consider stability [Roy et al., 2016, Moreno et al., 2020]. While this is almost never a problem in empirical runs, it can be a potential source of inconsistency in MOEAs, for example if two equal solution candidates are placed on the Pareto front(s) in different relative order due to sorting instability. Therefore, in some particular cases, use of an unstable stable sorting procedure may introduce an undesirable dependency to an implementation detail that can cause unexplained behavior or an inability to reproduce results (i.e., runs with a fixed seed). This was also observed by [Buzdalov, 2018] who called this phenomenon “bug-compatible” non-dominated sorting.
Algorithm 1: RankOrdinal

**Input:** \(S_1, ..., S_N\)

**Output:** Pareto fronts \(F_1, F_2, ...\)

1. **Function** RankOrdinal\((S_1, ..., S_N)\):
   2. \(P \leftarrow\) StableSort\((S_1, ..., S_N)\);
   3. \(R \leftarrow\) calculate using Eq. 9;
   4. \(\text{rank} \leftarrow\) zero-initialized array of size \(N\);
   5. **foreach** \(S_i \in p_1\) **do**
      6. \(k \leftarrow\) arg max \(l \leq r \leq 1 \rightarrow i\);
      7. **if** \(r_k(i) = N\) **then**
         8. **continue**;  // \(S_i\) cannot dominate any other
      9. **foreach** \(S_j \in p_k\) **do**
         10. **if** rank\((i) = \text{rank}(j)\) **and** \(S_i \leq S_j\) cf. Eq 11 **then**
             11. \(\text{rank}(j) \leftarrow\) rank\((j) + 1\);
      12. \(F \leftarrow\) list of fronts of size max\(i\) rank\((i) + 1\);
      13. **foreach** \(i\) from 1 to \(N\) **do**
          14. **append** \(i\) to front \(F_{\text{rank}(i)}\);
      15. **return** \(F\);

### 3.1.3 Computational complexity

Since the preprocessing part has a fixed cost (one lexicographic sorting step followed by \(M - 1\) regular sorting steps, its complexity is \(O(MN \log N)\). For the rank assignment part of the algorithm, we first consider the total number of iterations performed. Regardless how little work is done per iteration (e.g., when the conditions allow to skip over an already ranked solution), this quantity is important in determining the final complexity of the algorithm.

As shown in Algorithm 1, the rank assignment procedure consists of two nested loops: the outer loop iterates over all the solutions \(S_i\) in the order given by \(p_1\), while the inner loop iterates over the successors of \(S_i\) in the permutation \(p_k\) where \(k = \text{arg max}_l r_l(i)\).

**Worst case**

The worst case occurs when the input is degenerate, namely when the points to be sorted either belong to a single front or belong to individual fronts.

- When the solutions are non-dominated and belong to a single Pareto front, this corresponds to the unique situation where matrix \(P\) has the form:

\[
P = \begin{bmatrix}
1 & 2 & \ldots & N \\
N & N-1 & \ldots & 1
\end{bmatrix}^T
\]  

(13)

In this case, the size of dominance set \(D_{\text{ordinal}}(S_i)\) will be:

\[
|D_{\text{ordinal}}(S_i)| = \min(i, N - i)
\]  

(14)

Therefore, the total number of iterations will be:

\[
\text{total iterations} = \sum_{i=1}^{N/2} i = 2 \cdot \frac{N(N+1)}{2} = \frac{N^2}{4}
\]  

(15)

- When each solution \(S_i\) belongs to its own unique front, this corresponds to the unique situation where matrix \(P\) has the form:

\[
P = \begin{bmatrix}
1 & 2 & \ldots & N \\
1 & 2 & \ldots & N
\end{bmatrix}^T
\]  

(16)

In this case, the size of dominance set \(D_{\text{ordinal}}(S_i)\) will be:

\[
|D_{\text{ordinal}}(S_i)| = N - i
\]  

(17)
The total number of iterations in the worst case will be the sum:
\[
\text{total iterations} = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2} \quad (18)
\]

Note that the results above hold for \( M > 2 \) since adding more columns to \( P \) would not influence the resulting fronts because of the degenerate configuration of the first two columns. Therefore, \( \text{RankOrdinal} \) has the worst-case complexity of \( O(MN^2) \).

**Best case**

In the following we make the argument that the best-case complexity of \( \text{RankOrdinal} \) is \( O(MN \log N) \). We consider the problem of finding the Pareto set of randomly selected points in the unit hypercube.

- A full dominance check is always precluded by the condition \( \text{rank}(i) = \text{rank}(j) \) in Algorithm 1 line 10. Since the algorithm works by incrementing the ranks of dominated solutions within the current Pareto front (in order to “move” them to a new front), this condition will be true a finite number of times, and this number is proportional to the size of the Pareto fronts.

- Pareto front size is bounded in expectation by generalized harmonic numbers and thus scales in proportion to the natural logarithm [Yukish et al., 2004], [Köppen et al., 2005]. Therefore, the number of time the condition \( \text{rank}(i) = \text{rank}(j) \) is true will also be proportional to the logarithm of \( N \), bounding the dominance checks.

Additionally, it can be shown that the expected minimum of \( M \) uniform i.i.d. random variables is
\[
\mathbb{E}[\min(X_i)] = \frac{1}{M+1} \quad (19)
\]

For random objective values sampled from the unit hypercube, the total number of iterations (regardless of the amount of work done inside the inner loop and without considering the rank equality condition) will be \( \frac{N^2}{M+1} \).

Figure 1, determined empirically by taking 1000-run averages of inner loop iterations and dominance checks of \( \text{RankOrdinal} \), shows that the number of dominance comparisons grows much slower than the total number of iterations.

**Figure 1:** RankOrdinal total number of iterations and total number of rank comparisons

![Graph showing the relationship between number of points and total iterations/comparisons for M = 2, 3, and 5.]

3.1.4 Example

We consider the case \( N = 10 \), \( M = 2 \) with the following points:

\[
\begin{align*}
S_1 &= \begin{bmatrix} 0.79 & 0.35 \end{bmatrix} & S_2 &= \begin{bmatrix} 0.40 & 0.71 \end{bmatrix} & S_3 &= \begin{bmatrix} 0.15 & 0.014 \end{bmatrix} \\
S_4 &= \begin{bmatrix} 0.46 & 0.82 \end{bmatrix} & S_5 &= \begin{bmatrix} 0.28 & 0.98 \end{bmatrix} & S_6 &= \begin{bmatrix} 0.31 & 0.74 \end{bmatrix} \\
S_7 &= \begin{bmatrix} 0.82 & 0.52 \end{bmatrix} & S_8 &= \begin{bmatrix} 0.84 & 0.19 \end{bmatrix} & S_9 &= \begin{bmatrix} 0.85 & 0.78 \end{bmatrix} \\
S_{10} &= \begin{bmatrix} 0.96 & 0.83 \end{bmatrix}
\end{align*}
\]

The following permutations are obtained by stable-sorting according to the two objectives:

\[
\begin{align*}
p_1 &= [3 \ 5 \ 6 \ 2 \ 4 \ 1 \ 7 \ 8 \ 9 \ 10]^T \\
p_2 &= [3 \ 8 \ 1 \ 7 \ 2 \ 6 \ 9 \ 4 \ 10 \ 5]^T
\end{align*}
\]
Figure 2: Step-by-step operation of the RankOrdinal and RankIntersect algorithms

| #  | Perm. index | Rank | Domination set | Rank set | Domination ranks |
|----|-------------|------|----------------|----------|-----------------|
| 1  | p_2(1) = 3  | rank(S_3) = 4 | D(S_3) = {1, 2, 4, 5, 6, 7, 8, 9, 10} | r_1 = {3} | 2 2 1 2 2 2 2 2 2 2 |
| 2  | p_2(2) = 8  | rank(S_8) = 2 | D(S_8) = {9, 10} | r_2 = {1, 2, 4, 5, 6, 7, 9, 10} | 2 2 1 2 2 2 2 2 3 3 |
| 3  | p_2(3) = 1  | rank(S_1) = 2 | D(S_1) = {7} | r_3 = {7} | 2 2 1 2 2 2 3 2 3 3 |
| 4  | p_2(4) = 7  | rank(S_7) = 3 | D(S_7) = {9, 10} | r_4 = {1, 2, 4, 5, 6, 8} | 2 2 1 2 2 3 2 3 4 4 |
| 5  | p_2(5) = 2  | rank(S_2) = 2 | D(S_2) = {4} | r_5 = {1, 2, 5, 6, 8} | 2 2 1 2 2 3 2 3 4 4 |
| 6  | p_2(6) = 6  | rank(S_6) = 2 | D(S_6) = {9} | r_6 = {1, 2, 5, 6, 8} | 2 2 1 2 2 3 2 3 4 4 |
| 7  | p_2(7) = 9  | rank(S_9) = 4 | D(S_9) = {10} | r_7 = {9} | 2 2 1 2 2 3 2 3 2 4 5 |
| 8  | p_2(8) = 4  | rank(S_4) = 3 | D(S_4) = {9} | r_8 = {4, 7} | 2 2 1 2 2 3 2 3 2 4 5 |
| 9  | p_2(9) = 10 | rank(S_10) = 5 | D(S_5) = {9} | r_9 = {10} | 2 2 1 2 2 3 2 3 2 4 5 |
| 10 | p_2(10) = 5 | rank(S_5) = 2 | D(S_5) = {4, 7} | r_10 = {9, 10} | 2 2 1 2 2 3 2 3 2 4 5 |

(a) RankOrdinal Example

The corresponding matrices P and R are:

\[
P = \begin{bmatrix}
3 & 5 & 6 & 2 & 4 & 1 & 7 & 8 & 9 & 10 \\
3 & 8 & 1 & 7 & 2 & 6 & 9 & 4 & 10 & 5
\end{bmatrix}^T
\]

\[
R = \begin{bmatrix}
6 & 4 & 1 & 5 & 2 & 3 & 7 & 8 & 9 & 10 \\
3 & 5 & 1 & 8 & 10 & 6 & 4 & 2 & 7 & 9
\end{bmatrix}^T
\]

The progress of the algorithm is illustrated in Figure 2a. The resulting fronts are: F_1 = {3}, F_2 = {1, 2, 5, 6, 8}, F_3 = {4, 7}, F_4 = {9}, F_5 = {10}.
3.2 RankIntersect Algorithm

The RankIntersect (RS) algorithm takes a solution’s dominance set as the intersection of its objective-wise dominance sets:

\[
D(S_i) = \bigcap_k D_k(S_i)
\]  \hspace{1cm} (20)

The intersections are efficiently computed by exploiting bit-level parallelism. In a bitset, integer values are encoded as positions of the set bits. This has the advantage of reducing set intersections to simple logical \(\land\) operations that are optimized in the hardware.

The bitset approach is also employed by the MNDS Moreno et al. [2020], however MNDS considers a dominance set as the set of solutions (or equivalently, their permutation indices) that dominate the solution under consideration and updates the considered solution’s rank using the maximum rank in the dominance set (the same rank update mechanism is also shared by best order sort Roy et al. [2016]). This leads to inefficiencies due to the overlap in dominance sets and the need to compute maximum ranks.

According to Equation 7, for each solution \(S_i\), RankIntersect updates the ranks of successors \(S_j\) where the current rank of \(S_j\) equals the rank of \(S_i\) and \(S_j \succeq S_i\).

To implement this idea, RankIntersect maintains a separate collection of sets (bitsets) corresponding to each dominance rank value (from 1 to \(\ldots\)), such that each set contains the permutation indices of individuals with that rank. This rank set is updated along with the rank updates and used to reduce each individual’s dominance set according to Equation 7. The resulting algorithm is illustrated in Algorithm 2:

- The first part of the algorithm up to line 11 is responsible for preprocessing and initialization of data structures. A work bitset \(b\) is used to initialize dominance sets \(B(i)\) for each individual \(i\).
- Line 12 marks the beginning of the algorithmic loop where objective-wise dominance sets are intersected. When the last objective is reached, the algorithm proceeds to rank the rank assignment phase.
- Lines 19–20 perform the bitset intersections including the additional intersection with the corresponding rank set.
- The dominance set consisting of equally ranked successors is obtained at line 21. These successors will have their rank incremented at line 25. Therefore, they are removed from the current rank set (line 22) and added to the next one (line 23).

The resulting algorithm is functionally equivalent with RankOrdinal whose correctness has already been discussed. Relying on bitset operations requires extra storage space for the bitsets but offers a speed advantage as bit-level parallelism can accelerate intersection operations with a factor equal to the basic data block size (i.e., a 64-bit integer used as a basic data block can store 64 permutation indices), which can be further increased by vectorization.

3.2.1 Computational complexity

Since RankIntersect does not perform any dominance checks we discuss computational complexity in terms of the costs of computing the necessary set intersections.

Worst case

Logical operations \(\land, \lor\) over bitsets are performed by iterating over \(\lceil \frac{N}{64} \rceil\) basic data blocks which are internally stored in a list by the bitset. This is due to the fact that a data block is represented by a 64-bit unsigned integer. The outer loop at line 14 performs \(N\) iterations, while the inner loop performs \(\lceil \frac{N}{64} \rceil\) iterations to intersect the bitsets. Therefore the total number of iterations will be in the order of \(N^2\) which leads to \(O(MN^2)\) worst-case complexity.

Best case

Since Algorithm 2 successively intersects bitsets going from one objective to the next (lines 12, 17), it becomes possible to memorize the regions at the beginning and end of the list where the data blocks have become zero after the intersection and then skip them in the next iteration. This creates a short-circuiting mechanism for early stopping in the case of non-dominance. When all solutions are non-dominated, as all the dominance sets will be empty, the remaining complexity is \(O(MN \log N)\) given by the sorting phase.
Algorithm 2: RankIntersect

Input: $S_1, ..., S_N$

Output: Pareto fronts $F_1, F_2, ...$

Function RankIntersect($S_1, ..., S_N$):

1. $P ←$ StableSort($S_1, ..., S_N$);
2. $B ←$ list of bitsets associated to each $S_i$;
3. $K ←$ list of bitsets associated to each dominance rank $r$;
4. $K(1) ←$ new bitset;
5. set all bits in $K(1)$;
6. $b ←$ auxiliary bitset used to compute the set intersections;
7. set all bits in $b$;
8. $\textbf{foreach } i \textbf{ from 1 to } N \textbf{ do}$
   9. $B(i) ← b$;
10. $\textbf{foreach } k \textbf{ from 2 to } M \textbf{ do}$
11. $\textbf{set all bits in } b$;
12. $\textbf{foreach } i \textbf{ in } p_k \textbf{ do}$
13. $\textbf{reset bit } i \textbf{ in } b$;
14. $\textbf{if } k < M \textbf{ then}$
15. $B(i) ← B(i) \land b$;
16. $\textbf{else}$
17. $r ← K(\text{rank}(i))$;
18. $s ← K(\text{rank}(i + 1))$;
19. $v ← B(i) \land b \land r$;
20. $r ← r \land \neg v$;
21. $s ← s \lor v$;
22. $\textbf{foreach } j \textbf{ in } v \textbf{ do}$
23. $\text{rank}(j) = \text{rank}(j) + 1$;
24. $\textbf{end foreach}$
25. $\textbf{end else}$
26. $\textbf{end foreach}$
27. $\textbf{end foreach}$
28. $F ←$ list of fronts of size $\max_i \text{rank}(i)+1$;
29. $\textbf{foreach } i \textbf{ from 1 to } N \textbf{ do}$
30. $\text{append } i \textbf{ to front } F_{\text{rank}(i)}$;
31. $\textbf{end foreach}$
32. $\textbf{return } F$;
We conclude that the overall complexity of \textit{RankIntersect} is $O(MN^2)$ in the worst case and $O(MN \log N)$ in the best case. However, the advantage of \textit{RankSort} comes from the fact that its operations are very basic and can exploit the characteristics of modern processors. For $M > 2$, most runtime effort is spent performing objective-wise dominance set intersections at line 17. However in practice, a relatively modern CPU can intersect four basic data blocks at a time using AVX2 instructions (since a “wide” SIMD type can hold four 64-bit unsigned integers). For large $N$, the runtime effort shifts towards the sorting phase, suggesting that additional runtime benefits might be achieved by parallelizing the sorting.

3.2.2 Example

We consider the same input from Section 3.1.4 consisting of $N = 10$ points with $M = 2$. The matrix $P$ was:

$$P = \begin{bmatrix}
3 & 5 & 6 & 2 & 4 & 1 & 7 & 8 & 9 & 10 \\
3 & 8 & 1 & 7 & 2 & 6 & 9 & 4 & 10 & 5
\end{bmatrix}^T$$

The algorithm will iterate over $p_2$ (the last column of $P$) in the order: 3, 8, 1, 7, 2, 6, 9, 4, 10, 5. Since initially all individuals start at rank one, the first rank set $r_1$ will contain all the permutation indices. Each subsequent rank set is initialized with $\emptyset$. Then, the dominance set and rank sets are updated as shown in Figure 2b. As their ranks get updated, the permutation indices of the respective solutions are removed from the current rank set and inserted into the next one. Note that the dominance sets and rank sets shown in Figure 2b represent the final values resulting after the assignments at lines 19 and 20, respectively, in Algorithm 2.

4 Benchmark Results

The effectiveness of the proposed algorithms is measured both synthetically and in a practical optimization setting with the help of the \textit{Pagmo} framework [Biscani and Izzo, 2020], a C++ library for massively parallel optimization that provides a unified interface to optimization algorithms and problems.

We extended \textit{Pagmo} with C++ implementations for the tested algorithms and ran experiments with the DTLZ benchmark set [Deb et al., 2005] (problems DTLZ1 to DTLZ5) using the NSGA2 algorithm [Deb et al., 2002].

All tests were performed on a workstation containing an AMD Ryzen™ 5950X CPU with 8Mb of L2 cache and 64Mb of L3 cache, and 64Gb of DDR4 3600Mhz-CL16 memory. The experiments were run on a single thread to minimize effects such as resource contention or CPU pipeline stalls and to be able to acquire accurate profiling results. The code was compiled on GNU/Linux using the GNU C++ compiler version 11.2.0 with compilation flags: -O3 -mavx2 -mfma.

Each tested algorithm was implemented according to descriptions and pseudocode made available in their respective publications. A reasonable amount of effort has been expended for profiling and tuning each implementation. In the case of MNDS, the code has been ported from the original Java implementation in jMetal.

Since not all algorithms handle duplicate solutions in the same way, the procedure for handling duplicates in the population was abstracted away and made common between implementations. This ensures consistency and reproducibility among algorithms such that, regardless of the concrete non-dominated sorting implementation, the same random seed leads to the same results.

Due to runtime and space constraints, we first compare a larger selection of algorithms, consisting of RO, RS, MNDS, ENS-BS, ENS-SS, DS and HS on the DTLZ1 test problem, using 5 repetitions of the NGSA2 algorithm. We report the average elapsed time over the 5 algorithmic runs (Figure 3).

We then focus on a direct comparison between the three fastest algorithms, RO, RS and MNDS, where we report the average elapsed time over all runs (20 repetitions) and all problems (DTLZ1 to DTLZ5).

The NSGA2 algorithm was parameterized as follows:

- Population size: 500, 1000, 2500, 5000, 7500, 10000
- Maximum generations: 100
- Problem dimension: 25
- Crossover rate: 95%
- Crossover distribution rate $\eta_c$: 10
- Mutation rate: 10%
mutations rate \( \eta_m \): 50

The results indicate that bitset-based approaches RS and MNDS are the overall fastest, followed by RO. In the case of two objectives, the Efficient Non-dominated Sort family of algorithms (ENS-SS and ENS-BS) also performs very well, being similarly fast to RS and MNDS and marginally ahead of RO. This is a remarkable result especially considering their simplicity. However, as the number of objectives increases (\( M = 3, M = 5 \)) these two algorithms become slower due to the overhead of direct dominance comparisons within their respective search strategy. The elapsed time for each algorithm with varying number of objectives is shown in Figure 3.

Figure 3: Pagmo DTLZ1 – elapsed time (seconds) averaged over five algorithmic runs

\[
\begin{array}{cccccccc}
\text{N} = 500 & \text{N} = 1000 & \text{N} = 2500 & \text{N} = 5000 & \text{N} = 7500 & \text{N} = 10000 \\
\text{elapsed (s)} & \text{elapsed (s)} & \text{elapsed (s)} & \text{elapsed (s)} & \text{elapsed (s)} & \text{elapsed (s)} \\
2 & 3 & 4 & 5 & 2 & 3 & 4 & 5 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
0.0 & 0.2 & 0.4 & 0.6 & 0.0 & 0.4 & 0.8 & 1.0 \\
\end{array}
\]

We perform a detailed comparison of the fastest algorithms, RO, RS and MNDS, on the DTLZ benchmark set consisting of problems DTLZ1 to DTLZ5. Each algorithm was tested with the same random seeds over 20 repetitions for each configuration. The number of objectives was varied from 1 to 10 with an increment of 1 and from 10 to 20 with an increment of 2.

Figure 4 shows that RankIntersect provides a consistent runtime benefit, finishing ahead of MNDS in all cases. RankOrdinal performs better than MNDS for small populations but becomes slower as the population size increases (\( N > 5000 \)), exhibiting an unexpected performance decrease for \( M < 10 \). This aspect is not yet fully understood, but we suspect some inefficiency at the level of the ordinal rank comparison, also considering the fact that the size of the dominance set is inversely proportional with \( M \).

Next, we tested the performance of all algorithms on a synthetic benchmark where the objective values are uniformly sampled from the unit hypercube. The results shown in Figure 5 again show that RO is slightly faster than MNDS for lower population sizes but becomes slower as population size increases (\( N > 5000 \)). The same slowdown for \( M < 10 \) is observed for RO on synthetic data. The other algorithms are noticeably slower than the trio RO, RS, MNDS but perform similarly well to each other. As the points are random, their respective ranking strategies are dominated by the runtime cost of Pareto dominance checks.

5 Conclusion

In this paper, we introduced two simple and performant algorithms for non-dominated sorting, one of the most runtime-intensive components of Pareto-based MOEAs.

In contrast to similar approaches BOS or MNDS, RankSort defines a dominance set as the set of solutions dominated by the current solution (and not dominating the current solution). The main insight leading to the better performance of the RankSort approach is that a solution’s rank needs only be updated when the solution is found to be dominated by another solution of the same rank. This further reduces the size of the dominance set that needs to be examined.

The first algorithm, RankOrdinal, uses the comparison of ordinal ranks to establish dominance. Instead of performing set intersection operations to compute dominance sets, it simply iterates over the smallest objective-wise dominance set. RankOrdinal is slower than RankIntersect and MNDS but requires only \( O(N) \) space. It is also slightly slower than ENS-SS and ENS-BS in the two-objective case, but outperforms them as the number of objectives increases.

The second algorithm, RankIntersect, uses set intersections to compute dominance sets. These are efficiently implemented using bit-level parallelism, which is the main factor in its performance.
Figure 4: RO, RS, MNDS – Pagmo DTLZ benchmark set – elapsed time (seconds) averaged over twenty runs and five problems (DTLZ1 to DTLZ5)

Figure 5: RO, RS, MNDS – Synthetic benchmark randomly uniform objective values

The two algorithms have $O(MN \log N)$ best-case and $O(MN^2)$ worst-case asymptotic complexity, in line with other non-dominated sorting algorithms. As already apparent from Algorithms 1 and 2, the proposed methods are simple and easy to include in other frameworks. They use only basic data structures and have low constant factors. Nevertheless, C++ implementations are provided for all the methods tested in this paper.

Future work will focus on a better understanding of the computational complexity of the proposed methods and on exploring other algorithmic ideas. For example, performance can likely be improved by hybridizing the two steps of the algorithm: stable sort and rank update into a single hybrid step in order to further minimize the number of operations.

It will also be interesting to profile each algorithm and express its complexity in terms of low-level hardware events (executed instructions, branches, branch misses, cache misses). This will serve to provide a unified framework for the analysis of asymptotic complexity.

We also plan to provide efficient implementations of other state of the art non-dominated sorting algorithms and to publish all the algorithms as a stand-alone open-source library.

References

Chunteng Bao, Lihong Xu, Erik D. Goodman, and Leilei Cao. A novel non-dominated sorting algorithm for evolutionary multi-objective optimization. Journal of Computational Science, 23:31–43, 2017. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2017.09.015. URL https://www.sciencedirect.com/science/article/pii/S1877750317310530.
Francesco Biscani and Dario Izzo. A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53):2338, 2020. doi: 10.21105/joss.02338. URL https://doi.org/10.21105/joss.02338.

Maxim Buzdalov. Generalized offline orthant search: One code for many problems in multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO ’18, page 593–600, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356183. doi: 10.1145/3205455.3205469. URL https://doi.org/10.1145/3205455.3205469.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002.

Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable test problems for evolutionary multiobjective optimization. In *Evolutionary Multiobjective Optimization*, 2005.

Mario Köppen, Raul Vicente-Garcia, and Bertram Nickolay. The pareto-box problem for the modelling of evolutionary multiobjective optimization algorithms. In *Adaptive and Natural Computing Algorithms*, 2005.

Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining Convergence and Diversity in Evolutionary Multiobjective Optimization. *Evolutionary Computation*, 10(3):263–282, 09 2002. ISSN 1063-6560. doi: 10.1162/106365602760234108. URL https://doi.org/10.1162/106365602760234108.

Kent McClymont and Ed Keedwell. Deductive Sort and Climbing Sort: New Methods for Non-Dominated Sorting. *Evolutionary Computation*, 20(1):1–26, 03 2012. ISSN 1063-6560. doi: 10.1162/EVCO_a_00041. URL https://doi.org/10.1162/EVCO_a_00041.

Sumit Mishra and Maxim Buzdalov. If unsure, shuffle: Deductive sort is $o(mn^3)$, but $o(mn^2)$ in expectation over input permutations. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO ’20, page 516–523, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-4323-7. doi: 10.1145/3377930.3390246. URL https://doi.org/10.1145/3377930.3390246.

Javier Moreno, Daniel Rodriguez, Antonio J. Nebro, and Jose A. Lozano. Merge nondominated sorting algorithm for many-objective optimization. *IEEE Transactions on Cybernetics*, page 1–11, 2020. ISSN 2168-2275. doi: 10.1109/TCYB.2020.2968301. URL http://dx.doi.org/10.1109/TCYB.2020.2968301.

Proteek Chandan Roy, Md. Monirul Islam, and Kalyanmoy Deb. Best order sort: A new algorithm to non-dominated sorting for evolutionary multi-objective optimization. In *Proceedings of the 2016 Genetic and Evolutionary Computation Conference Companion*, GECCO ’16 Companion, pages 1113–1120, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4323-7. doi: 10.1145/2908961.2931684. URL http://doi.acm.org/10.1145/2908961.2931684.

Lingling Xue, Peng Zeng, and Haibin Yu. Setnds: A set-based non-dominated sorting algorithm for multi-objective optimization problems. *Applied Sciences*, 10(19), 2020. ISSN 2076-3417. doi: 10.3390/app10196858. URL https://www.mdpi.com/2076-3417/10/19/6858.

Michael A. Yukish, W. Simpson, Mark Traband, Soundar R. T. Kumara, and Richard C. Benson. Algorithms to identify pareto points in multi-dimensional data sets. Technical report, Mechanical Engineering Dept., The Pennsylvania State University, State College, 2004.

Xingyi Zhang, Ye Tian, Ran Cheng, and Yaochu Jin. An efficient approach to nondominated sorting for evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 19(2):201–213, 2015. doi: 10.1109/TEVC.2014.2308305.

Yuren Zhou, Zefeng Chen, and Jun Zhang. Ranking vectors by means of the dominance degree matrix. *IEEE Transactions on Evolutionary Computation*, 21(1):34–51, 2017. doi: 10.1109/TEVC.2016.2567648.