NON-DOMINATED SORTING METHODS FOR MULTI-OBJECTIVE OPTIMIZATION: REVIEW AND NUMERICAL COMPARISON

Qiang Long and Xue Wu
School of science
Southwest University of Science and Technology, Mianyang 621010, China

Changzhi Wu∗
School of Management, Guangzhou University, Guangzhou 510006, China

(Communicated by Kok Lay Teo)

ABSTRACT. In multi-objective evolutionary algorithms (MOEAs), non-dominated sorting is one of the critical steps to locate efficient solutions. A large percentage of computational cost of MOEAs is on non-dominated sorting for it involves numerous comparisons. By now, there are more than ten different non-dominated sorting algorithms, but their numerical performance comparing with each other is not clear yet. It is necessary to investigate the advantage and disadvantage of these algorithms and consequently give suggestions to specific users and algorithm designers. Therefore, a comprehensively numerical study of non-dominated sorting algorithms is presented in this paper. Firstly, we design a population generator. This generator can generate populations with specific features, such as population size, number of Pareto fronts and number of points in each Pareto front. Then non-dominated sorting algorithms were tested using populations generated in certain structures, and results were compared with respect to number of comparisons and time consumption. Furthermore, In order to compare the performance of sorting algorithms in MOEAs, we embed them into a specific MOEA, dynamic sorting genetic algorithm (DSGA), and use these variations of DSGA to solve some multi-objective benchmarks. Results show that dominance degree sorting outperforms the other methods, fast non-dominance sorting performs the worst and the other sorting algorithms performs equally.

1. Introduction. Multi-objective optimization problem (MOP) has extensive applications in engineering and management [1, 24]. Most optimization problems in real-world applications have multiple objectives, which can be modelled as MOPs. However, due to the theoretical and computational challenges, it is not easy to solve MOPs[27]. Therefore, MOP attracts a wide range of research over the last few decades.

One of the most popular methods for solving MOPs is multi-objective evolutionary algorithm (MOEA). MOEAs solve MOPs by directly exploring the objective function value space (the image set of multi-objective values) with gradually
selecting elitist solutions through operators such as mutation, crossover, reproduction and selection. Since MOEAs directly work in the objective function value space, we generally classified them as direct method whose counterpart is indirect method, such as weighted sum method. The reason for evolutionary algorithms being used to solve MOPs is that their iterations are population-based, so a single run could achieve many efficient solutions. This feature of evolutionary algorithms properly fits the characteristic of MOP whose Pareto solution is not unique but many (even infinitely many). In the last two decades, various MOEAs were developed, such as NSGA[16], NSGAII[5] and NSGAIII[4, 9] which applied genetic algorithm; SPEA[29] and SPEA2[30] which applied evolutionary algorithm. Most MOEAs use Pareto domination to lead the search and return a set of non-dominated solutions as results. So these algorithms carry on some kinds of vector ranking schemes which detects and indicates the elitism of each solution in a population, for example, non-dominated sorting. However, non-dominated sorting is computationally expensive for it involves numerous comparisons. It is reasonable to say that non-dominated sorting in a large extent decides the performance of a MOEA. There are more than ten different non-dominated sorting algorithms have been proposed in the last decades, but their numerical performance comparing with each other is still not clear yet. In this paper, we try to fill this gap by presenting a comprehensively numerical study of non-dominated sorting algorithms. We first design a population generator, this generator can generate populations with specific features, such as population size, number of Pareto fronts and number of points in each Pareto front. Then, we test non-dominated sorting algorithms using populations generated in certain structures and compare results with respect to number of comparisons and time consumption. Furthermore, in order to compare the performance of sorting algorithms in MOEAs, we embed them into a specific MOEA, dynamic sorting genetic algorithm (DSGA), and use these variations of DSGA to solve some multi-objective benchmarks.

The rest of this paper is arranged as follows: In Section 2, we give a brief introduction of MOP. In Section 3, we review a theoretical analysis of non-dominated sorting algorithms and briefly describe the existing non-dominated sorting algorithms. In section 4, we design the population generator. Section 5 presents numerical experiments and comparisons. Section 6 concludes the paper.

2. Non-dominated sorting in MOP. Before going to the detail of non-dominated sorting, let us first recall some definition of MOP. The general mathematical model of MOP is

\[
\begin{align*}
\text{(MOP)} & \quad \left\{ \begin{array}{l}
\text{Minimize} \quad F(x) \\
\text{Subject to} \quad x \in X.
\end{array} \right.
\end{align*}
\]

Here \( F : \mathbb{R}^n \mapsto \mathbb{R}^m \) is a vector function. \( X \in \mathbb{R}^n \) is the feasible set; it could include only box constraint or include both box and functional constraints. In the following contents, we also call \( X \) the decision variable space, and its image set \( F(X) = \{ F(x) | x \in X \} \) the objective function value space.

Given two vectors \( y = (y_1, y_2, \cdots, y_m)^T \) and \( z = (z_1, z_2, \cdots, z_m)^T \in \mathbb{R}^m \), then

- \( y = z \Leftrightarrow y_i = z_i \) for all \( i = 1, 2, \cdots, m \);
- \( y \prec z \Leftrightarrow y_i < z_i \) for all \( i = 1, 2, \cdots, m \);
- \( y \preceq z \Leftrightarrow y_i \leq z_i \) for all \( i = 1, 2, \cdots, m \), and \( y \neq z \).
“>” and “⪯” can be defined similarly. In this paper, if \( y \preceq z \), we say \( y \) dominates \( z \) or \( z \) is dominated by \( y \), if \( y \not\preceq z \) and \( z \not\preceq y \), we say \( y \) and \( z \) are non-dominated with each other.

**Definition 2.1.** [22] Let \( Y \subseteq \mathbb{R}^m \) and \( y^* \in Y \). If there is no \( y \in Y \) such that

\[
y \preceq y^* \text{ (or } y \prec y^*)\,
\]

then \( y^* \) is called an efficient point (or weakly efficient point) of \( Y \).

**Definition 2.2.** [22] Suppose that \( x^* \in X \). If there is no \( x \in X \) such that \( F(x) \preceq F(x^*) \) (or \( F(x) \prec F(x^*) \)), i.e. \( F(x^*) \) is an efficient point (or weakly efficient point) of the objective function value space \( F(X) \), then \( x^* \) is called an efficient solution (or weakly efficient solution) of Problem (1).

Efficient and weakly efficient solutions can also be defined using cone or partial order [22]. Another name of the efficient solution is Pareto solution, which was introduced by T. C. Koopmans in 1951 [12]. The meaning of Pareto solution is that, if \( x^* \) is a Pareto solution, then there is no feasible solution \( x \in X \), such that \( f_i(x) \leq f_i(x^*) \) for any \( i \in \{1, 2, \cdots, m\} \) and there is at least one \( i_0 \in \{1, 2, \cdots, m\} \) such that \( f_{i_0}(x) < f_{i_0}(x^*) \). In other words, \( x^* \) is the best solution in the sense of "\( \preceq \)". Another intuitive interpretation of Pareto solution is that it cannot be improved with respect to any objective without worsening at least one of the others. Weakly efficient solution was introduced by S. Karlin in 1960 [18]. It means that if \( x^* \) is a weakly efficient solution, then there is no feasible solution \( x \in X \), such that any \( f_i(x) \) of \( F(x) \) is strictly better than that of \( F(x^*) \). In other words, \( x^* \) is the best solution in the sense of "\( \prec \)". Obviously, efficient solution is also a weakly efficient solution. The set of Pareto solutions is denoted by \( \mathcal{P}^* \), its image set \( F(\mathcal{P}^*) \) is called the Pareto front, denoted by \( \mathcal{PF}^* \).

The aim of MOEAs is to find the Pareto solutions \( \mathcal{P}^* \) and Pareto front \( \mathcal{PF}^* \) of MOP by searching its objective function value space. Non-dominated sorting is a critical phase in searching the space, we will briefly describe it in the next two paragraphs. It is worth to note that, in the following contents, the term "solutions" may both mean points in the decision variable space and objective function value space. We do not distinguish them since they are related, readers can identify them without any difficulty.

Non-dominated sorting is a procedure to assign solutions in a population to different Pareto fronts based on their dominance relationships. Taking a population \( P \) as an example, firstly, all non-dominated solutions in population \( P \) are assigned to the first Pareto front, denoted as \( \mathcal{F}_1 \); then, all non-dominated solutions in the rest population \( P - \mathcal{F}_1 \) are assigned to the second Pareto front, denoted as \( \mathcal{F}_2 \); this procedure repeats until all solutions in the population \( P \) are assigned. Without loss of generality, assume that there are \( k \) Pareto fronts, so we have \( P = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \cdots \cup \mathcal{F}_k \).

Furthermore, solutions belonging to Pareto front \( \mathcal{F}_j \) are dominated by at least one solution belonging to Pareto front \( \mathcal{F}_i \), if \( i < j \), \( i, j = 1, 2, \cdots, k \). Fig. 1 demonstrates an illustrative example of population with 17 solutions and 4 Pareto fronts.

Non-dominated sorting is computationally intensive, in particular when the population size is large. The principal computational cost is spent on dominance comparison, which is essential to detect dominating relationship between solutions. It is reasonable to say that the number of dominance comparisons determines efficiency of a non-dominated sorting algorithm; therefore, most non-dominated sorting algorithms focus on reducing unnecessary and redundant dominance comparisons. In
this paper, we numerically study more than ten different existing non-dominated sorting algorithms.

3. Methods.

3.1. Analysis of non-dominated sorting methods. In order to reduce the number of unnecessary and redundant dominance comparisons, we have to identify them at the first place. Zhang, et al., [25] presented a close investigation of dominance comparison, which categorized all dominance comparisons into four cases. Assume dominance comparison happens between \( y_1, y_2 \in P \), then one of following cases must fit their dominance relationship,

- **case 1:** \( y_1 \preceq y_2 \) or \( y_2 \preceq y_1 \);
- **case 2:** \( y_1 \) and \( y_2 \) are non-dominated, i.e., \( y_1 \npreceq y_2 \) and \( y_2 \npreceq y_1 \), and \( y_1 \) and \( y_2 \) belong to the same front \( F_i \), where \( F_i \) is the current front (i.e., the front which the solutions are being assigned to);
- **case 3:** \( y_1 \) and \( y_2 \) are non-dominated, i.e., \( y_1 \npreceq y_2 \) and \( y_2 \npreceq y_1 \), and \( y_1 \) and \( y_2 \) belong to the same front \( F_i \), where \( F_i \) is not the current front;
- **case 4:** \( y_1 \) and \( y_2 \) are non-dominated, i.e., \( y_1 \npreceq y_2 \) and \( y_2 \npreceq y_1 \), but they belong to different fronts.

Take solutions in Fig. 1 as an example and suppose we are now detecting the first Pareto front \( F_1 \): In case 1, \( y_2 \preceq y_3 \), \( y_3 \) must not belong to the current front \( F_1 \) no matter \( y_2 \) belongs to \( F_1 \) or not, so there is no need to consider any dominance comparison involving \( y_3 \) any more. In case 2, \( y_1 \) and \( y_2 \) are non-dominated and belong to the current front \( F_1 \); the dominance comparison between \( y_1 \) and \( y_2 \) has to be done to identify their non-dominated relationship. In case 3, \( y_4 \) and \( y_5 \) are non-dominated and belong to \( F_3 \) which is not the current front, so there is no need to do dominance comparison between \( y_4 \) and \( y_5 \) since their dominance relationship will not affect detecting of \( F_1 \). In case 4, \( y_3 \) and \( y_5 \) are non-dominated but not in the same front; there is no need to do dominance comparison between \( y_3 \) and \( y_5 \) because at least one of them will be exclude using case 1. In fact, here both \( y_3 \) and \( y_5 \) are excluded when comparing with \( y_2 \) which belongs to case 1.

Based on the above analysis, for non-dominated sorting, dominance comparisons in case 1 and 2 cannot be avoided, which are termed *necessary comparisons*; whereas dominance comparison in case 3 and 4 can be avoided, which are termed *unnecessary comparisons*. Zhang, et al., [25] presented formulas to calculate the theoretical number of comparisons in case 1 and case 2. Suppose population \( P \) has \( k \) Pareto
fron\(ts \mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_k\); numbers of solutions in each front are \(N_1, N_2, \cdots, N_k\); so the total number of necessary comparison in case 1 is
\[
C_1 = \sum_{i=1}^{k} (i - 1)N_i;
\]
the total number of necessary comparison in case 2 is
\[
C_2 = \sum_{i=1}^{k} p_i N_i (N - i - 1) / 2.
\]
Therefore, the theoretical lower bound of number of dominance comparisons for any non-dominated sorting approach is the sum of \(C_1\) and \(C_2\), i.e., \(C_1 + C_2\). Unfortunately, absolutely avoiding dominance comparisons in case 3 and 4 is hard. In recent years, researchers developed various methods to reduce the number of dominance comparisons in case 3 and 4 as much as possible.

The existing non-dominated sorting algorithms can be classified into three groups according to their strategy of organizing comparisons. The first group directly detects dominance relationships between solutions using dominance comparisons, such as the fast non-dominated sorting [5], arena’s principle sorting [21], deductive sorting [11], climbing sorting [11], corner sorting [23] and dynamic sorting [14]. The second group involves sorting for a single objective. For example, the efficient non-dominated sorting [25] sorts the first objective ascending before applying dominance comparison, which dramatically reduces the number of dominance comparisons in case 3 and 4; the dominance degree matrix sorting [28] sorts every objective separately and use dominance degree matrix to identify fronts, which does not need any dominance comparison. The third group applies the idea of sorting a real number sequence. For example, the series of Jense-Fortin non-dominated sorting [13, 10, 8, 15] applies the idea of divide-and-conquer in quick sorting; the dominate tree sorting [7] applies the idea of tree sorting. Recently, some new non-dominated sorting algorithms alternatively for many-objective optimization are presented, such as M-front[6], T-ENS[26] and ENS-NDT[17].

In the rest of this section, we briefly describe the existing non-dominated sorting algorithms. We mainly recall their key ideas, algorithms and numerical complexities. For the sake of convenience, we use the following notations: \(P\) stands for a population, \(N\) is the number of solutions in \(P\), \(m\) is the number of objective functions. \(\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_k\) are Pareto fronts in population \(P\), \(N_1, N_2, \cdots, N_k\) are number of solutions in each Pareto front, \(k\) is the number of Pareto fronts.

### 3.2. Fast non-dominated sorting

The fast non-dominated sorting [5] is one of the earliest non-dominated sorting algorithms; it has been continuously attracting a wide range of attention since it was developed. In the naive approach [16], in order to find if a solution is dominated, the solution has to do dominance comparisons with all the other solutions, which makes the naive approach a numerical complexity of \(O(mN^3)\) in the worst case. The drawback of naive approach is that it does not record any dominance comparison result, which in return causes a lot of repeated computation.

The fast non-dominated sorting improves the naive approach by calculating two entities for each solution \(p\): (i) domination count \(n_p\), the number of solutions which dominate the solution \(p\); and (ii) dominate set \(S_p\), a set of solutions that the solution
p dominates. Then we use the following steps to sequentially detect Pareto fronts from 1 to k:

**step 1:** the point $p$ with $n_p = 0$ belongs to the current Pareto front;

**step 2:** for each $p$ belonging to the current Pareto front, reduce dominance count of these point in $S_p$ by 1, i.e., $n_q := n_q - 1$ for any $q \in S_p$;

**step 3:** repeat step 1 and 2 until all solutions in $P$ are assigned.

The calculation of $n_p$ and $S_p$ for population $P$ requires $O(mN^2)$ comparisons, step 1 and 2 repeat at most $O(N)$ times, so the overall complexity of fast non-dominated sorting is $O(mN^2)$. But comparing to the naive approach, the storage requirement of the fast non-dominated sorting increases to $O(N^2)$.

### 3.3. Climbing sorting.

The idea of climbing sorting [11] is similar to the bubble sorting for real number sequence. It follows dominating relationships between solutions and climbing the graph toward the Pareto front. In order to locate a non-dominated solution on the current front, the candidate is shifted from dominated solution to the dominating one. For example, $y_i$, the current candidate, compares with $y_j$, if $y_j \preceq y_i$, then the candidature shifts from $y_i$ to $y_j$, this process repeats until all currently available solutions in the population are visited, then the final candidate must locate on the current Pareto front.

The climbing sorting reduces redundant dominance comparisons through marking and discarding dominated solutions and assigned solutions. Discarding dominated solutions actually reduces dominance comparisons in case 3. In order to speed the process, the climbing sorting chooses the last mutually non-dominating solution as the candidate at the beginning of a new round, in the assumption that this solution is also likely to occupy the current front. This strategy actually reduces dominance comparisons in case 3 and 4.

The climbing sorting takes advantage of dominating relationships between solutions. As it is claimed in the work of Kent, et al. [11], the algorithm performs better on populations containing a large number of Pareto fronts. In contrast, for single front populations, the advantage is lost. The climbing sorting does not record dominating relationships between solutions, so it can only apply dominating relationship for the first degree. Therefore, improvement can be made on the climbing sorting by recording the dominating relationships. This reduces dominance comparisons in case 1 but increases space complexity.

The numerical complexity of the climbing sorting highly depends on structure of the population. In the worst case, when solutions are all non-dominated, no dominated solution can be marked and discarded. Consequently, in each round, the candidate has to compare with the rest $N - 1$ solutions, which makes the total comparison complexity $O(mN^2)$.

### 3.4. Deductive sorting.

The key idea of Climbing sorting is to shift the candidature from dominated solution to dominating one, which makes the candidate solution always be non-dominated. Therefore, in each loop of the climbing sorting, a non-dominated solution has to be identified. Comparing with the climbing sorting, the deductive sorting [11] applies a different mechanism. It terminates the loop whenever the current candidate is identified to be dominated, and starts a new loop with the next candidate. So a non-dominated solution can only be found when a loop continues until the last solution in the population.

More specifically, the deductive sorting evaluates each solution following the fixed natural order of the population; each candidate only has dominance comparisons
with its following solutions in the order. The process of deductive sorting consists of two loop levels: outer loop and inner loop. Outer loops identify Pareto fronts from $F_1$ to $F_k$, and inner loops scan unassigned solutions one by one according to the fixed natural order of the population. Each inner loop has a candidate solution which ends up with one of the following situations: whenever the candidate solution is found to be dominated, terminates the inner loop and start a new one with the next candidate solution; if the candidate solution is always non-dominated until the end of inner loop, then this candidate solution should be assigned to the current Pareto front; any solution found to be dominated should not be considered in the following inner loops which reduces dominance comparisons in case 4. In each outer loop, the assigned solutions should not be considered anymore, and all the dominated solutions are back to active again.

The deductive sorting is more or less like a filter; it sieves dominated solutions and keeps non-dominated ones. In the extreme case of each Pareto front has only one solution, the deductive sorting needs $N(N - 1)/2$ dominance comparisons. In the other extreme case of all solutions are non-dominated, it needs $N(N - 1)/2$ dominance comparisons too. In the best case where $N$ solutions of the population evenly occupy each Pareto front, i.e., $N_i = N/k$ (in the best case $N = k^2$), $i = 1, 2, \cdots, k$, where $k$ is the number of Pareto fronts, the deductive sorting needs

$$\frac{Nk - 1}{2} + \frac{N/k(N/k - 1)}{2}$$

dominance comparisons.

3.5. **Corner sorting.** The basic idea of saving comparisons in the corner sorting [23] is to use non-dominated solutions to discard solutions that they dominate. The corner sorting sufficiently applies the fact that a solution with the minimal value of one objective must be a non-dominated solution. The principal phases of corner sorting are: firstly, choose an objective, say $f_j$, and find the solution $p$ with the minimal $f_j$ (so solution $p$ must be non-dominated, i.e., belongs to the current Pareto front); secondly, mark and discard all solutions which are dominated by $p$ ($p$ should be marked too); thirdly, run the same process on the left unmarked solutions until all solution on the current Pareto front are assigned.

The flag objective $f_j$ can be fixed all the time, or be chosen alternatively. However, there is a bug if some solutions identical in objective $f_j$. According to corner sorting, if some solutions are identical in objective $f_j$, any one of them could be chosen as the “non-dominated” solution, which may not be right. For example, suppose $p_1, p_2$ are identical in $f_1$ but different in $f_2$, i.e., $p_1^{f_1} = p_2^{f_1}$, $p_1^{f_2} < p_2^{f_2}$; we may choose $p_2$ as the “non-dominated” one, which is wrong because $p_1$ is the real non-dominated solution. The way to repair this bug is to sort the population in alphabet order before applying corner sorting. This idea is implemented in the efficient non-dominated sorting [25] which will be introduced in the next subsection.

3.6. **Efficient non-dominated sorting.** The efficient non-dominated sorting [25] first sorts the $N$ solutions in population $P$ in an ascending order according to the first objective value. If the first objective value of two solution are the same, they are sorted according to the second objective value, and so forth. If two solutions have the same value in all objectives, their order can be arbitrary. This sorted population has a very important feature: a solution $p_m$ will never be dominated by a solutions $p_n$ if $m < n$, since there exists at least one objective in $p_m$ whose value is smaller than that of the same objective in $p_n$. Therefore, there are only
two dominating relationships between \( p_m \) and \( p_n \): either \( p_m \) dominates \( p_n \) or \( p_m \) and \( p_n \) are non-dominated.

Given the feature of pre-sorted population, the efficient non-dominated sorting assigns solutions to Pareto fronts one by one following the sorted order. To assign the current candidate \( p \), \( p \) only needs to compare with the assigned solutions, i.e., these solutions who are before \( p \) in the pre-sorted order; there is no need to consider the subsequence solutions. If solution \( p \) is assigned to front \( F_i \); then \( F_i \) must satisfy the following two conditions:

(i): there exists at least one solution in each front \( F_j \) that has been assigned and dominates \( p \), for \( 1 \leq j \leq i - 1 \);

(ii): there exists no solution in any of the assigned fronts \( F_k \) that dominates \( p \), for \( k \geq i \).

In this way, the front to which a solution belongs can be determined by finding the front which satisfies the above two conditions. Zhang, et al., [25] provided two strategies to implement these two conditions: the sequential search and binary search strategies. They are different in search directions and perform equally well in numerical experiments. We will include the sequential search strategy in our numerical study.

As shown in Table (III) in paper [25], numerical complexity of the efficient non-dominated sorting is \( O(mN^2) \) in the worse case; while in the best case, the efficient non-dominated sorting using the sequential searching strategy is \( O(mN\sqrt{N}) \) and using the binary searching strategy is \( O(mN \log N) \).

3.7. Arena’s principle sorting. The arena’s principle sorting [21] is the same as the efficient non-dominated sorting [25] in the first phase; that is sorting the population with respect to the first objective. In the second phase, the efficient non-dominated sorting assigns solutions to Pareto fronts one by one; in contrast, the arena’s principle sorting detects Pareto fronts one by one. Therefore, the efficient non-dominated sorting takes unassigned solution as its outer loop, detected Pareto fronts as its inner loop; whereas arena’s principle sorting takes Pareto fronts as its outer loop, unassigned solutions as its inner loop. However, the efficient non-dominated sorting and arena’s principle sorting both benefit from the same feature of the pre-sorted population: the later solution will never dominate the former one in the pre-sorted order. So when assigning a new solution \( p \), \( p \) does not need to do dominance comparisons with its successors in the pre-sorted order. In other words, \( p \) only need to do dominance comparison with its precursors, i.e., these who have already been assigned.

The arena’s principle sorting supposes to have the same numerical complexity with the efficient non-dominated sorting. Depending on the distribution of the population, there might be some slightly different in the second phase, which will be investigated in the section of numerical experiments.

3.8. Dynamic non-nominated sorting. The original dynamic non-dominated sorting [14] is only slightly improved from the naive sorting [20]. In the naive sorting, results of dominance comparisons are not recorded, so there are numerous repeated comparisons. In order to record the results of dominance comparisons, the dynamic non-dominated sorting computes a \( N \times N \) dynamic matrix \( D \) whose
elements is
\[
d_{ij} = \begin{cases} 
1 & \text{if } p_i \text{ dominates } p_j; \\
-1 & \text{if } p_i \text{ is dominated by } p_j; \\
0 & \text{if } p_i \text{ and } p_j \text{ are non-dominated.}
\end{cases}
\]

Pareto fronts can be detected from the dynamic matrix \( D \). More specifically, in the current dynamic matrix, find rows without \(-1\), i.e.,
\[
I = \{ i | d_{ij} \neq -1, \quad i = 1, 2, \ldots, N \};
\]
then solutions corresponding to set \( I \) compose the current Pareto front. The next Pareto front can be located in the same way using the dynamic matrix with discarding the \( i \)th row and column, where \( i \in I \). All the Pareto fronts can be found when the dynamic matrix becomes empty.

It is obvious that the dynamic matrix is antisymmetric; calculating it needs \( N(N-1)/2 \) dominance comparisons. So the numerical complexity of dynamic non-dominating sorting is \( O(mN^2) \) no matter how the distribution of population is.

The dynamic non-dominated sorting can be improved by taking into account dominating relationship of solutions. For example, if \( d_{ij} = 1, \ d_{jk} = 1 \) (which means \( p_i \) dominates \( p_j \), \( p_j \) dominates \( p_k \)); then we can directly have \( d_{ik} = 1 \) (which means \( p_i \) dominates \( p_k \)) based on the transitivity of dominating relationship, which means that there is no need to have dominance comparison between \( p_i \) and \( p_k \). In numerical experiments, we use recursion to implement this strategy.

3.9. Dominance degree sorting. So far, all the non-dominated sorting algorithms discussed above are based on the dominance comparison. Although the efficient non-dominated sorting and arena’s principle sorting use real number sequence sorting method to pre-sort the population, dominance comparison is still applied when assign each solution. Unlike these algorithms, the dominance degree sorting [28] dose not use any dominance comparison, and only use real number sequence sorting. It identifies Pareto fronts using a dominance degree matrix which is constructed based on the properties of Pareto domination. The dominance degree matrix converts the dominance comparison into counting the number of special element pairs.

The process of calculating the dominance degree matrix is as follows: Firstly, for each objective, a \( N \times N \) comparison matrix is constructed. For example, for the first objective, we have vector \( w^{f_1} = (p_{f_1}^1, p_{f_1}^2, \ldots, p_{f_1}^N) \), the entry of \( C^{f_1} \) is
\[
C^{f_1}_{ij} = \begin{cases} 
1, & \text{if } p_{f_1}^i \leq p_{f_1}^j, \\
0, & \text{otherwise.}
\end{cases}
\]

Secondly, the dominance degree matrix can be obtained by summing comparison matrixes for all objectives, i.e.,
\[
D = C^{f_1} + C^{f_2} + \cdots + C^{f_N}.
\]

Thirdly, to eliminate the effect of these solutions with identical values for all objectives, we set the corresponding element of \( D \) to be zero. For example, if \( p_i \) and \( p_j \) are identical respect to all objectives, we set \( D_{ij} = 0 \). Obviously, according to this rule, \( D_{ii} = 0 \) for all \( i = 1, 2, \ldots, N \). Fourthly, the elements of \( D \) stands for the dominance counting between solutions; that is, for any \( p_i \neq p_j \in P \), \( p_i \) dominates \( p_j \) if and only if \( D_{ij} = m \). Now, maximize matrix \( D \) with respect to column to get a row vector \( D_{\text{max}} \). Then, based on the feature of \( D \), if any element of \( D_{\text{max}} \) equals to \( m \), the corresponding solution is dominated; otherwise, it is non-dominated and
belongs to the current Pareto front. Lastly, discard rows and columns corresponding to the last Pareto front from matrix $D$ and run the same process to identify the next Pareto front until matrix $D$ becomes empty.

The main computational effort of the dominance degree sorting is on constructing comparison matrix. Intuitively, for each objective, function values are compared with each other, so $N(N - 1)/2$ real number comparisons are needed, which makes the numerical complexity of dominance degree sorting $O(mN^2)$. However, the process to construct comparison matrix can be improved by apply real number sequence sorting method, such as quick sorting [28], which reduces its numerical complexity to $O(mN \log N)$.

3.10. Jensen-Fortin non-dominated sorting. The Jensen-Fortin non-dominated sorting originates from the algorithm developed by Kung, et al. [13] in 1975. Kung’s algorithm presents a simple strategy for locating the non-dominated set of a collection of points in $\mathbb{R}^2$. The key idea is that, firstly, we sort the points respect to the first dimension, and then identify the non-dominated ones only by comparing the second dimension. The process is the same as the efficient non-dominated sorting [25]. Since most of the comparisons are made for sorting the first dimension, its numerical complexity is $O(N \log N)$. The algorithm presented by Kung, et al. [13] only extracts the first non-dominated front $F_1$, but it is not difficult to modify it to extract all fronts.

In the year of 2003, Jensen, et al. [10] extended Kung’s algorithm to non-dominated sorting with respect to two aspects: from two dimensions to three and more than three dimensions, and from identifying only $F_1$ to all Pareto fronts. Jensen’s extension applies the strategy of divide-and-conquer on general real number sequence sorting with a guarantee that all the Pareto fronts can be identified in a single run. However, the extension is based on an assumption that no solutions share identical values for any coordinates, which is unreasonable but difficult to remove.

In the year of 2013, Fortin, et al. [8] improved Jensen’s algorithm by removing the limitation of no two solutions can share identical values for any objective. It is worth to note that the improvement can be achieved without affecting its general complexity which is $O(N \log^{m-1} N)$. In 2014, Maxim, et al. [15] presented a slight modification of the Jensen-Fortin algorithm with a proof that its complexity is $O(N \log^{m-1} N)$ in the worst case.

3.11. Dominance tree sorting. A critical approach to reduce redundant comparisons is to take into account transitivity of the dominance relationship. For example, if solution $p_i$ dominates solution $p_j$, all solutions dominated by solution $p_j$ do not need to compare with solution $p_i$. Many methods try to employ this feature in different ways, such the deductive sorting [11] uses flag sets to record the transitive relationship, the dynamic non-dominated sorting [19] uses recursive programming to search the transitive relationship among solutions. The dominance tree sorting uses a hierarchical data structure, that is, a tree structure, to save the dominance information. The dominance tree is constructed by nodes which stand for solutions and links which represent the dominance relationship between solutions. There are two different kinds of links: non-dominance and dominance links. Solutions connected by a non-dominance link are called siblings; while solutions connected by a dominance link are called parent and child. The divide-and-conquer mechanism is adopted to generate the dominance tree.
After the dominance tree is generated for the entire population, we can obtain the non-dominated fronts by merging the children of all siblings at the same levels. This process starts from the root of the dominance tree and propagates until no more merge needs to be performed.

As aforementioned, the dominance tree sorting improves efficiency of the non-dominated sorting process by reducing redundant comparisons, which only exists when there are dominated solutions. So in the worst case, when all solutions are non-dominated, full comparisons are required and there will not be any redundancy. In this case, the numerical complexity of dominance tree sorting is $O(mN^2)$. The lower-bound complexity is the same as that of the divide-and-conquer algorithm, which is $O(mN \log N)$.

4. Test population generator. In order to specifically test and compare the numerical performance of various non-dominated sorting algorithms, we need to use test populations with certain features, such as having prefixed number of points, number of Pareto fronts and number of points in each Pareto front. In this section, we design a population generator to generate populations with these features setting in advance. We first describe the idea of the generator, then present the algorithm.

In the population generating process, Pareto fronts are generated sequently from $F_1$ to $F_k$. Points belonging to each Pareto front lay on a line segment. In the process, we have to guarantee that any point in the later front has to be dominated by at least one point in the previous front. In the following, we take front $F_1$ and $F_2$ as an example; the rest fronts can be generated in the same way.

For the sake of demonstration, we restrict the example in $\mathbb{R}^2$. Initially, points belonging to the first Pareto front $F_1$ are randomly selected on the line segment $AB$, where $A = (0,1)$, $B = (1,0)$ (see Fig. 2). We can easily control the number of points being generated. Next, we explain how to use some points in $F_1$ to generate a point in $F_2$.

Firstly, locate the vertex point $v$ of $F_1$. Vertex point is the one with the smallest value in every dimension among points in $F_1$. That is, element of $v$ is the minimal one respect to the corresponding element among points in $F_1$. Mathematically,

$$v_i = \min\{p_i | p \in F_1\}, \ i = 1, 2. \quad (2)$$

Secondly, calculate a bridge point $b$. Bridge point is a convex combination of some randomly selected points in $F_1$, i.e.,

$$b = \sum_{j=1}^{l} \lambda_j p_j^{i}, \quad (3)$$

where $\lambda_j \geq 0$, $j = 1, 2, \cdots, l$, $\sum_{j=1}^{l} \lambda_j = 1$ are convex combination coefficients, $p_1, p_2, \cdots, p_l$ are $l$ randomly selected points from Pareto front $F_1$.

Thirdly, generate a point $p$ belonging to Pareto front $F_2$. The mechanism is that, based on bridge point $b$, mapping vertex point $v$ to line $l_2$ where Pareto front $F_2$ locates (see Fig. 2). Mathematically,

$$p = v + \left(2 - \sum_{i=1}^{2} v_i\right) \frac{b - v}{\sum_{i=1}^{2} (b_i - v_i)}. \quad (4)$$

In this way, we generate a point in Pareto front $F_2$. It is easy to generate more point on $F_2$ through repeating the process. Furthermore, by calculating the vertex
1. **Numerical study.** In this section, we do the numerical study on the non-dominated sorting algorithms introduced previously. The numerical experiments consists of two parts: (i) numerical comparisons among non-dominated sorting algorithms using populations generated by Algorithm 1 with fixed features; (ii) comparing non-dominated sorting algorithms by embedding them into a multi-objective evolutionary algorithm. Aims of these two parts are different: by using populations with pre-defined structures in part (i), we test specific properties of algorithms; by embedding algorithms into a MOP solver, we investigate performance of algorithm with respect to different practical problems.

We use three metrics to measure the numerical performance of non-dominated sorting algorithms: time consumption $T$, number of comparisons $C$ and comparison...
efficiency $E$. Time consumption is the CPU time that an algorithm needs to sort a specific pre-defined population. Number of comparisons means the number of real number comparisons that an algorithm needs to sort a population. The reason for using the real number comparison instead of the dominance comparison is that some algorithms, such as dominance degree sorting, do not have dominance comparison. Comparison efficiency is defined as the average CPU time per real number comparison, mathematically,

$$E = \frac{\text{time comparison}}{\text{number of comparison}} = \frac{T}{C}.$$  

Comparison efficiency is designed to measure efficiency of an algorithm; a smaller comparison efficiency means that the algorithm is more efficient with respect to the real number comparison.

For the sake of convenience, we use the following abbreviations for these algorithms:

- Fast non-dominated sorting — FNS
- Climbing sorting — CLS
- Deductive sorting — DES
- Corner sorting — COS
- Efficient non-dominated sorting — ENS
- Arena’s principle sorting — APS
- Dynamic non-dominated sorting — DNS
- Dominance degree sorting — DDS

All the numerical experiments are implemented in an environment of MATLAB(2010a) installed on an ACER ASPIRE 4730Z laptop with a 2G RAM and a 2.16GB CPU.

5.1. Numerical tests using populations with fixed features. In this subsection, we test non-dominated sorting algorithms using populations with different fixed features. According to the parameters of population generation (the number of objectives $m$, number of Pareto fronts $k$, number of points in each Pareto front $N$), we generate six different series of populations whose parameters are illustrated in Table 1.
Table 1. Five series of populations

| Series No. | Description | $m$ | $k$ | $N$                        |
|------------|-------------|-----|-----|---------------------------|
| Series (i) | fixed $m$   | 3   | 1   | $N = (200)$               |
|            | various $k$ | 3   | 2   | $N = (100, 100)$          |
|            | $\sum N = 200$ | 3   | 3   | $N = (70, 70, 60)$        |
|            |             | 3   | 4   | $N = (50, 50, 50, 50)$    |
|            |             | 3   | 5   | $N = (40, 40, 40, 40, 40)$|
|            |             | 3   | 6   | $N = (33, 33, 33, 33, 33, 33)$|
| Series (ii)| fixed $m$   | 3   | 5   | $N = (10, 10, 10, 10, 10)$|
|            | fixed $k$   | 3   | 5   | $N = (20, 20, 20, 20, 20)$|
|            | various $N$ | 3   | 5   | $N = (30, 30, 30, 30, 30)$|
|            |             | 3   | 5   | $N = (40, 40, 40, 40, 40)$|
|            |             | 3   | 5   | $N = (50, 50, 50, 50, 50)$|
|            |             | 3   | 5   | $N = (60, 60, 60, 60, 60)$|
| Series (iii)| various $m$ | 2   | 6   | $N = (20, 20, 20, 20, 20)$|
|            | fixed $k$   | 3   | 5   | $N = (20, 20, 20, 20, 20)$|
|            | fixed $N$   | 4   | 5   | $N = (20, 20, 20, 20, 20)$|
|            |             | 5   | 5   | $N = (20, 20, 20, 20, 20)$|
|            |             | 6   | 5   | $N = (20, 20, 20, 20, 20)$|
|            |             | 7   | 5   | $N = (20, 20, 20, 20, 20)$|
| Series (iv)| fixed $m$   | 3   | 1   | $N = 50$                  |
|            | fixed $k$   | 3   | 1   | $N = 100$                 |
|            | various $N$ | 3   | 1   | $N = 150$                 |
|            |             | 3   | 1   | $N = 200$                 |
|            |             | 3   | 1   | $N = 250$                 |
|            |             | 3   | 1   | $N = 300$                 |
| Series (v) | fixed $m$   | 3   | 10  | $N_i = 1$, $i = 1, \ldots, k$|
|            | various $k$ | 3   | 20  | $N_i = 1$, $i = 1, \ldots, k$|
|            | various $N$ | 3   | 30  | $N_i = 1$, $i = 1, \ldots, k$|
|            |             | 3   | 40  | $N_i = 1$, $i = 1, \ldots, k$|
|            |             | 3   | 50  | $N_i = 1$, $i = 1, \ldots, k$|
|            |             | 3   | 60  | $N_i = 1$, $i = 1, \ldots, k$|
| Series (vi)| fixed $m$   | 3   | 5   | $N_i$ is a random integer |
|            | fixed $k$   | 3   | 5   | between 1 and 50          |
|            | various $N$ | 3   | 5   | between 1 and 50          |

Populations in series (i) are fixed in number of objectives and the total number of points, while the number of fronts ranges from 1 to 6 and points in each front are distributed as evenly as possible. Fig. 4 and Fig. 5 illustrate the time consumption and number of comparisons for series (i), respectively. It can be seen from the figures that, when the total number of points is fixed, the time consumption and number of comparisons both decrease as the increasing of the number of fronts. However, FNS is an exception with its number of comparisons staying as a constant and time consumption even rising slightly, which reveals that the complexity of FNS mainly depends on the total number of points in the population. The performance of DDS is surprisingly better than other algorithms.

Populations in series (ii) are fixed in the number of objectives and number of fronts; every front has the same number of points in a population. Populations are various in the number of points for fronts, ranging from 10 to 60 in an increment of 10. From Fig. 6 and Fig. 7, we can observe that FNS consumes far more time and comparisons than other algorithms, and its expense increases quadratically as the increasing number of points. DDS still outperforms the other algorithms, its time consumption and number of comparisons increase also but in a very slight extent. Performances of other algorithms are close, END, APS and DNS are slightly better than CLS, DES and ENS.

Populations in series (iii) are fixed in the number of fronts and number of points in each front; the number of objectives increases from 2 to 7 in a step of 1. Fig. 8 and Fig. 9 demonstrate the time consumption and number of comparisons for
this series. It can be seen from the figures that numerical complexity is increase linearly as the rising of the number of objectives. Among them, FNS increases the fastest, DDS increases the slowest and the other algorithms have the same slope of increasing.

In series (iv) and (v), we investigated two extreme situations for populations. Populations in series (iv) are all non-dominated, so there is only one Pareto front in these population. On the contrary, populations in series (v) have only one point in each front. The results for series (iv) are illustrated in Fig. 10 and 11; for series (v) are illustrated in Fig. 12 and 13. From Fig. 11, for populations with all
points non-dominated, FNS, CLS, DES and COS perform equally in the number of comparisons, slightly better than ENS, APS and DNS who are even as well. However, from Fig. 10, we can see that FNS spends slightly more time than CLS, DES, and COS, which reveals that FNS is not less efficient than the other three. Fig. 12 and 13 demonstrate the same trend: FNS, CLS, DES and COS are slightly better than the ENS, APS and DNS both in the time consumption and number of comparisons. Still, DDS outperforms all other algorithms in both situations.

Series (vi) is designed to test the average numerical performances of algorithms. The number of objectives and number of fronts are fixed; and in all populations,
number of points in each front is a random integer between 1 and 50. In order to evaluate the average efficiency, we generate 50 populations in these features. Fig. 14, 15 and 16 illustrate the average time consumption, average number of comparisons and average comparison efficiency for algorithms, respectively. From these figures, we can observe that FNS performs the worst, DDS performs the best, and the other algorithms are slightly different.

To be summarized from this subsection, we can have a rough ranking about numerical performance (according to Fig. 16) for sorting algorithms, with using ≥
as “better” and \( \approx \) as “approximately equal”:

\[
\text{DDS} \gg \text{DNS} \approx \text{APS} \gg \text{ENS} \gg \text{CLS} \gg \text{DES} \gg \text{COS} \gg \text{FNS}.
\]  

(5)

5.2. **Numerical tests using MOPs.** In this subsection, we investigate numerical performance of sorting algorithms by embedding them into a direct method called dynamic sorting genetic algorithm (DSGA) [14]. The DSGA is a MOP solver based on the genetic algorithm with using DNS (see Section 3.8) to do non-dominated ranking. In this experiment, we amend the DSGA by substituting DNS using other sorting algorithms and keep other parts the same. When solving a MOP using those
Figure 16. Average Comparison efficiency for algorithms

Table 2. Multi-objective test problems

| Pro. | n  | Variable bounds | Objective functions                        |
|------|----|-----------------|--------------------------------------------|
| SCH  | 1  | [-5, 10]        | \( f_1(x) = x^2 \)                        |
|      |    |                 | \( f_2(x) = (x - 2)^2 \)                  |
| FON  | 3  | [-4, 4]         | \( f_1(x) = 1 - \exp\left(-\sum_{i=1}^{3}(x_i - \frac{1}{\sqrt{3}})^2\right) \) |
|      |    |                 | \( f_2(x) = 1 - \exp\left(-\sum_{i=1}^{3}(x_i + \frac{1}{\sqrt{3}})^2\right) \) |
| KUR  | 3  | [-5, 5]         | \( f_1(x) = \sum_{i=1}^{n-1}\left(-10\exp\left(-0.2\sqrt{x_i^2 + x_{i+1}^2}\right)\right) \) |
|      |    |                 | \( f_2(x) = \sum_{i=1}^{n}\left(|x_i|^{0.8} + 5\sin^3(x_i)\right) \) |

DSGAs, we use the same parameters for different variation of DSGAs; that is to say, the only difference is the non-dominated sorting algorithms. In this way, we test the numerical performance of sorting algorithms when solving MOPs.

Three MOP benchmarks [5] (SCH, FON and KUR) are used in this experiments with details illustrating in Table 2. Problem SCH (see Fig. 17(a)) is a one dimension, two objectives problem. Its objective function value space is a curve, so in non-dominated sorting, the situation of all points are non-dominated will happen. Problem FON (see Fig. 17(b)) is three dimensions, two objectives problems. Its objective function value space is an area, so randomly distribution populations will be ranked in non-dominated sorting. Problem KUR (see Fig. 17(c)) is also a three dimensions, two objectives problem, but its Pareto front is disconnected.

Figure 17. Objective function value space

We use three metrics for numerical performance: (i) the total time consumption, (ii) the percentage of time spent on sorting, and (ii) the total number of comparisons. Since the non-dominated sorting algorithm is now embedded in DSGA, the total time consumption not only includes the time spent in non-dominated sorting,
but also includes time spent by other operations of DSGA. Therefore, we calculate the percentage of time spent by non-dominated sorting over the total time consumption. The total number of comparison is now the summary of comparison in all generations. In all experiments, we fixed parameters of DSGA as the crossover rate $\alpha_c = 0.7$, mutation rate $\alpha_m = 0.1$, population size $popu\_size = 100$, and maximum generation $max\_gene = 20$.

Fig. 18 illustrates the numerical performance on the problem SCH. Fig. 18(a) shows that DSGA with DDS spends much less time than the others in solving the problem SCH, whereas DSGA with FNS spends more time than the others. Fig. 18(b) shows that percentage of time spent on sorting has the same trend with the total time consumption. From Fig. 18(b), we can still observe that, except DDS (only occupies about 20% of the total time consumption), the non-dominated sorting operation took most of the time resource (about 80%~90%) of DSGA. It is not surprise that, in Fig. 18(c), the trend of the total number of comparisons is also agree with the total time consumption.

![Figure 18. Numerical performance on SCH](image)

Fig. 19 illustrates numerical performance on the problem FON. From Fig. 19(a) and 19(c), we can see that the total time consumption and number of comparisons are closely coordinated. DDS still performs the best, continuing with ENS, while FNS the worst. APS and DNS, CLS and DES perform the same, respectively. From Fig. 19(b) all variations of DSGA spend more than 90% of time consumption on non-dominated sorting except DSGA with DDS.

![Figure 19. Numerical performance on FON](image)

Fig. 20 illustrates numerical performance on the problem KUR. The same as in FON, the total time consumption and number of comparisons are closely coordinated. DDS still performs the best, continuing with ENS, while FNS the worst. APS and DNS, CLS and DES perform the same, respectively. For problem KUR, DDS spends a little bit more time on sorting (about 55%) than the other two problems.

To be summarized from Fig. 18, 19 and 20, we can obtain the following results:
• Non-dominated sorting occupies most of time consumption of DSGAs;
• The total time consumption and total number of comparisons are closely coordinated;
• Broadly, we can have the following ranking for numerical performance of sorting algorithms embedding in DSGA:

$$\text{DDS} \succ \text{ENS} \succ \text{CLS} \approx \text{DES} \succ \text{COS} \approx \text{APS} \approx \text{DNS} \succ \text{FNS}.$$ \hspace{1cm} (6)

Concluding from ranking (5) and (6), it is no doubt that DDS performs the best, while FNS performs the worst in both experiments. CLS and DES, APS and DNS performs even, respectively.

6. Conclusion. Non-dominated sorting is one of the principal steps to locate elitist efficient solutions in multi-objective evolutionary algorithms (MOEAs). A large percentage of computational cost of MOEAs is on non-dominated sorting for it involves numerous comparisons. In the last few decades, more than ten different non-dominated sorting algorithms were proposed, but, to our best knowledge, there is no research on the numerical performance comparison of these sorting methods. It is obligatory to investigate the excellence and weakness of these methods and consequently give suggestions to specific users and algorithm designers. In order to fill this gap, this paper presented a comprehensively numerical study of non-dominated sorting algorithms. In order to specifically test non-dominated sorting algorithms, we designed a population generator; this generator can generate populations with specific features, such as the population size, number of Pareto fronts, and number of points in each Pareto front. Then non-dominated sorting algorithms were tested using populations generated in certain structures, and algorithms were compared with respect to the number of comparisons and time consumption. In order to investigate efficiency of sorting methods when applying in MOEAs, we also embed them into a specific MOEA, dynamic sorting genetic algorithm (DSGA), to compare their numerical performance on some multi-objective benchmarks.

It can be summarized from numerical experiments that the dominance degree sorting (DDS) outperforms all the other sorting algorithms, whereas the fast non-dominated sorting (FNS) performs the worst among these algorithms. The climbing sorting (CLS) and deductive sorting (DES) performs even to each other, so are the arena’s principle sorting (APS) and dynamic non-dominated sorting (DNS). The reason for DDS performs the best is that it only use real number sequence sorting, which is much faster than the dominance comparison. FNS performs the worst most likely because it does not discard any unnecessary and redundant dominance comparisons. The other sorting algorithms perform slightly different; their ranking may change from case to case, but not in a big extent.
It is worth to note that the selected non-dominated sorting methods in this paper only works in multi-objective evolutionary algorithms, i.e., multi-objective optimization problems whose number of objectives are less than or equal to three. For many-objective optimization problems whose number of objectives are more than four, these non-dominated sorting methods cannot work smoothly or even fail to work. In many-objective optimization problems, because the number of objectives is large, most of the points in a population will become nondominated. This means that most of the points are in the first Pareto front, which makes the non-dominated sorting meaningless. Therefore, in many-objective optimization, the non-dominated sorting should be totally different. This is an important and difficult issue in many-objective optimization, which arises some new research interests in this field.

**Funding.** This work is supported by the National Natural Science Foundation of China (Grants No. 11671062 and 11871128).

**REFERENCES**

[1] A. Cheng and L. Cheng-Chew, Optimizing System-On-Chip verifications with multi-objective genetic evolutionary algorithms, *Journal of Industrial and Management Optimization*, 10 (2014), 383–396.

[2] D. W. Corne, J. D. Knowles and M. J. Oates, The pareto envelope-based selection algorithm for multiobjective optimization, *Parallel Problem Solving from Nature PPSN VI. Springer*, (2000), 839–848.

[3] D. W. Corne, N. R. Jerram, J. D. Knowles and M. J. Oates, PESA-II: Region-based selection in evolutionary multi-objective optimization, *Genetic and Evolutionary Computation Conference*, (2001), 283–290.

[4] K. Deb and J. Himanshu, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints, *IEEE Transactions Evolutionary Computation*, 18 (2014), 577–601.

[5] K. Deb, A. Pratap, S. Agarwal and T. A. M. T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Transactions on Evolutionary Computation*, 6 (2002), 182–197.

[6] M. Drozdik, A. Youhei, A. Hernan and T. Kiyoshi, Computational cost reduction of non-dominated sorting using the M-front, *IEEE Transactions on Evolutionary Computation*, 19 (2015), 659–678.

[7] H. B. Fang, Q. Wang, Y. C. Tu and M. F. Horstemeyer, An efficient non-dominated sorting method for evolutionary algorithms, *Evolutionary Computation*, 16 (2008), 355–384.

[8] F. A. Fortin, S. Grenier and M. Parizeau, Generalizing the Improved Run-time Complexity Algorithm for Non-Dominated Sorting, Proceedings of the 15th annual conference on Genetic and evolutionary computation, 2013.

[9] J. Himanshu and K. Deb, An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part II: Handling constraints and extending to an adaptive approach, *IEEE Transactions Evolutionary Computation*, 18 (2014), 602–622.

[10] M. T. Jensen, Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms, *IEEE Transactions on Evolutionary Computation*, 7 (2003), 503–515.

[11] M. Kent and E. Keedwell, Deductive sort and climbing sort: New methods for non-dominated sorting, *Evolutionary Computation*, 20 (2012), 1–26.

[12] T. C. Koopmans and others, *Activity Analysis of Production and Allocation*, Wiley New York, 1951.

[13] H. T. Kung, L. Fabrizio and F. P. Franco, On finding the maxima of a set of vectors, *Journal of the ACM (JACM)*, 22 (1975), 469–476.

[14] Q. Long, W. N. Xu and K. Q. Zhao, Dynamic sorting genetic algorithm for multi-objective optimization, *Swarm and Evolutionary Computation*, (2017).
[15] B. Maxim and A. Shalyto, A provably asymptotically fast version of the generalized Jensen algorithm for non-dominated sorting, International Conference on Parallel Problem Solving from Nature, Springer, Cham, 2014.

[16] S. Nidamarthi and K. Deb, Multiobjective optimization using nondominated sorting in genetic algorithms, Evolutionary Computation, 2 (1994), 221–248.

[17] G. Patrik and A. Syberfeldt, A new algorithm using the non-dominated tree to improve non-dominated sorting, Evolutionary Computation, 26 (2018), 89–116.

[18] K. Samuel and J. Gillis, Mathematical methods and theory in games, programming, and economics, Physics Today, 13 (1960), 54.

[19] C. Shi and Z. Y. Yan, Z. Z. Shi and L. Zhang, A fast multi-objective evolutionary algorithm based on a tree structure, Applied Soft Computing, 10 (2010), 468–480.

[20] N. Srinivas and K. Deb, Multiobjective optimization using nondominated sorting in genetic algorithms, Evolutionary Computation, 2 (1994), 221–248.

[21] S. Q. Tang, Z. X. Cai and J. H. Zheng, A fast method of constructing the non-dominated set: Arena’s principle, The Fourth International Conference on Natural Computation, 2008.

[22] C. K. Vira and Y. Y. Haimes, Multiobjective Decision Making: Theory and Methodology, Courier Dover Publications, 2008.

[23] H. D. Wang and Y. Xin, Corner sort for Pareto-based many-objective optimization, IEEE Transactions on Cybernetics, 44 (2014), 92–102.

[24] J. Xiong, Z. B. Zhou, K. Tian, T. J. Liao and J. M. Shi, A multi-objective approach for weapon selection and planning problems in dynamic environments, Journal of Industrial and Management Optimization, 13 (2017), 1189–1211.

[25] X. Y. Zhang, Y. Tian, R. Cheng and Y. C. Jin, An efficient approach to nondominated sorting for evolutionary multiobjective optimization, IEEE Transactions on Evolutionary Computation, 19 (2015), 201–213.

[26] X. Y. Zhang, Y. Tian, R. Cheng and Y. C. Jin, Empirical analysis of a tree-based efficient non-dominated sorting approach for many-objective optimization, Computational Intelligence, IEEE, 2016.

[27] L. Zhang, J. Zhang and Y. Zhang, Second-order optimality conditions for cone constrained multi-objective optimization, Journal of Industrial and Management Optimization, 14 (2018), 1041–1054.

[28] Y. R. Zhou, Z. F. Chen and J. Zhang, Ranking vectors by means of the dominance degree matrix, IEEE Transactions on Evolutionary Computation, 21 (2017), 34–51.

[29] E. Zitzler and L. Thiele, Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach, IEEE Transations Evolutionary Computation, 3 (1999), 257–271.

[30] E. Zitzler, M. Laumanns and L. Thiele, SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization, Fifth Conference on Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems, (2001), 95–100.

Received October 2018; revised September 2019.

E-mail address: longqiang@swust.edu.cn
E-mail address: wuxue@mails.swust.edu.cn
E-mail address: C.Wu@exchange.curtin.edu.au