Improved Incremental Non-dominated Sorting for Steady-State Evolutionary Multiobjective Optimization

Ilya Yakupov
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia 197101
iyakupov93@gmail.com

Maxim Buzdalov
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia 197101
mbuzdalov@gmail.com

ABSTRACT

We present an algorithm for incremental non-dominated sorting, a procedure to use with steady-state multiobjective algorithms, with the complexity of $O(N(\log N)^{M-2})$ for a single insertion, where N is the number of points and M is the number of objectives. This result generalizes the previously known $O(N)$ algorithm designed for two objectives.

Our experimental performance study showed that our algorithm demonstrates a superior performance compared to the competitors, including various modifications of the divide-and-conquer non-dominated sorting algorithm (which significantly improve the performance on their own), and the state-of-the-art Efficient Non-domination Level Update algorithm. Only for $M = 2$ the specialized algorithm for two dimensions outperforms the new algorithm.

CCS CONCEPTS

•**Theory of computation → Online algorithms;** *Sorting and searching;*

KEYWORDS

Non-dominated sorting, steady-state algorithms, divide-and-conquer.

ACM Reference format:
Ilya Yakupov and Maxim Buzdalov. 2017. Improved Incremental Non-dominated Sorting for Steady-State Evolutionary Multiobjective Optimization. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071307

1 INTRODUCTION

Many real-world optimization problems are multiobjective, that is, they require maximizing or minimizing several objectives, which are often conflicting. These problems most often do not have a single solution, but instead feature many incomparable solutions, which trade one objective for another. It is often not known *a priori* which solution will be chosen, as decisions of this sort are often recommended to be made late, as the decision maker can learn more about the problem [1]. This encourages finding a set of diverse incomparable solutions, which is a problem often approached by multiobjective evolutionary algorithms.

In the realm of scaling-independent preference-less, and thus general-purpose, evolutionary multiobjective algorithms, three paradigms currently seem to prevail [1]: Pareto-based, indicator-based, and decomposition-based approaches. Although there exist well-known decomposition-based [26] and indicator-based [30, 32, 33] algorithms, the majority of modern algorithms are Pareto-based [5–7, 31].

Most Pareto-based algorithms belong to one of big groups according to how solutions are selected or ranked: the algorithms which maintain non-dominated solutions [4, 5, 15], perform non-dominated sorting [6, 7, 9], use domination count [11], or domination strength [31]. In this research we concentrate on non-dominated sorting, as some popular algorithms make use of it [6, 7].

Non-dominated sorting assigns ranks to solutions in the following way: the non-dominated solutions get rank 0, and the solutions which are dominated only by solutions of rank at most i get rank $i + 1$. In the original work [23], this procedure was performed in $O(N^3M)$, where N is the population size and M is the number of objectives. This was later improved to be $O(N^2M)$ in [7].

As the quadratic complexity is still quite large, both from theoretical and practical points of view, many researchers concentrated on improving practical running times [10, 13, 19, 22, 24, 27, 28], however, without improving the worst-case $O(N^2M)$ complexity. Jensen was the first to adapt the earlier result of Kung et at. [16], who solved the problem of finding non-dominated solutions in $O(N(\log N)^{\max(1,M-2)})$, to non-dominated sorting. This algorithm has the worst-case complexity of $O(N(\log N)^{M-1})$. However, this algorithm could not handle coinciding objective values, which was later corrected in subsequent works [2, 12]. A more efficient algorithm for non-dominated sorting, or finding *layers of maxima*, exists for three dimensions [21], whose complexity is $O(N(\log\log N)^2)$ with the use of randomized data structures, or $O(N(\log\log N)^3)$ for deterministic ones. However, whether this algorithm is useful in practice is still an open question.

When implementing a steady-state multiobjective evolutionary algorithm which uses non-dominated sorting, one can in theory apply non-dominated sorting on each point insertion (and deletion, in case it removes a point from a level which is not the last one). However, this would increase the cost of insertion by a factor proportional to the population size. Indeed, in generational algorithms, non-dominated sorting is typically applied once per iteration, that is, when a large set of offspring is evaluated, the cost of sorting is amortized, which is not the case of steady-state algorithms. A study on a steady-state variation of the NSGA-II algorithm [20]

GECCO '17, Berlin, Germany

978-1-4503-4920-8/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3071178.3071307

showed a slowdown of more than 10 times compared to the original generational NSGA-II [7], taking into account the time needed for fitness evaluations and genetic operations, which have a bigger unit cost and which have the same impact on both algorithm variations.

To date, only two approaches for incremental non-dominated sorting exist: the general-purpose Efficient Non-domination Level Update method [17, 18], or ENLU, which has the worst-case insertion complexity of $O(N^2M)$ but works well in practice, and the binary search tree based data structure [25] which works only for two objectives, but has the worst-case insertion complexity of $O(N)$ and good practical performance as well [3].

The main result of this paper generalizes the recent results for incremental non-dominated sorting. We present an algorithm which has the complexity of $O(N(\log N)^{M-2})$ and typically performs better than ENLU on small to moderate numbers of objectives. The rest of the paper is structured as follows. Section 2 describes the notation, definitions and, in rather deep detail, the algorithms which we build on or which we compare to. Section 3 delivers the proposed algorithm, which recombines the divide-and-conquer algorithm from [2] and the known incremental algorithms [17, 25]. Section 4 presents the experimental results, where we compare the proposed algorithm with its predecessors, as well as discuss the impact of various components of the algorithm. Finally, Section 5 concludes.

2 PRELIMINARIES

In this section, we give the necessary definitions of Pareto dominance and non-dominated sorting, including the incremental version of the problem. We also overview the following algorithms:

- the divide-and-conquer algorithm for non-dominated sorting, as the proposed algorithm uses its internals directly;
- the efficient non-dominating level update approach, or ENLU [17], as it is currently the most efficient approach, in practice, for general-purpose incremental non-dominated sorting;
- the algorithm for incremental non-dominated sorting tailored for two dimensions [3, 25], as it is currently the most efficient approach in its class, both in practice and in theory, and its ideas are directly used in the proposed algorithm.

In the following, we assume that all points are different, which enables us to name any unordered collection of points a *set*. This is not true in general, however, all equal points will receive the same rank, so implementations are free, depending on their need, to either discard a point if there is an equal one, or to keep all equal points in a same entity and run algorithms on these entities instead, or to work directly with equal points with more care. None of these precautions change the worst-case algorithmic complexity.

2.1 Definitions

We use capital Latin letters to denote sets of points, as well as the global constants N (the number of points) and M (the number of objectives), while small Latin letters are used for single points, standalone objectives and rank values, and small Greek letters are used for mappings. The value of the i-th objective of a point p is denoted as p_i.

In the rest of the paper we assume, without losing generality, that we solve a multiobjective minimization problem with the number of objectives equal to M. In this case, the *Pareto dominance relation* is determined on two points in the objective space as follows:

$$a \prec b \leftrightarrow \forall i \in [1; M]\ a_i \le b_i \text{ and } \exists i \in [1; M]\ a_i < b_i$$
$$a \preceq b \leftrightarrow \forall i \in [1; M]\ a_i \le b_i$$

where $a \prec b$ is called the *strict dominance* and $a \preceq b$ is the *weak dominance*.

Non-dominated sorting is a procedure which, for a given set P of N points in the M-dimensional objective space, assigns each point $p \in P$ an integer *rank* $\tau(p)$, such that:

$$\tau(p) = \max\{0\} \cup \{1 + \tau(q) \mid q \in P, q \prec p\}.$$

In other words, a rank of a point which is not dominated by any other point is zero, and a rank of any other point is one plus the maximum rank among the points which dominate it.

Following the convention from [17], we call the set of all points with the given rank r a *non-domination level* L_r:

$$L_r = \{p \in P \mid \tau(p) = r\}.$$

Incremental non-dominated sorting is an extension to the ordinary non-dominated sorting which supports incremental changes, namely, insertion of a single point and removal of a single point. More precisely, incremental non-dominated sorting requires a data structure which stores a set of points P and supports at least the following operations:

- insert an arbitrary point p into the set P;
- remove one of the points from the last non-domination level of P, where the exact point of choice is guided by an algorithm-dependent condition (such as taking the point with the smallest crowding distance in NSGA-II [7]);
- pick the i-th point p ($1 \le i \le |P|$) from P, according to some arbitrary but predefined order on points, together with its rank $\tau(p)$.

We explicitly note here that, to enable faster operations, we do not require to store all points explicitly in their non-domination levels (although some implementations may choose to do so), and we do not fix the order of points for the third operation.

2.2 The Divide-and-Conquer Approach

The divide-and-conquer approach dates back to 1975, when Kung et al. proposed a multidimensional divide-and-conquer algorithm for finding the maxima of a set of vectors [16], which, in the realm of evolutionary computation, corresponds to the set of non-dominated points, or to points with rank zero. The complexity of this algorithm is $O(\min(N^2, N(\log N)^{\max(1,M-2)}))$, which we shorten to $O(N(\log N)^{M-2})$ for clarity.

This algorithm can be used to implement non-dominated sorting in the following manner: first we determine the points with rank zero, then we remove these points and run the algorithm again on the remaining points (which yields points with rank one), then we repeat it until no points left. However, the worst-case complexity of this approach is $O(N^2(\log N)^{M-2})$. In contrast, fast non-dominated sorting, shipped with the original NSGA-II of Deb et at. [7], has a better $O(N^2M)$ complexity.

The divide-and-conquer approach has been generalized to perform non-dominated sorting by Jensen [14], shortly afterwards the NSGA-II arrived. The algorithm from [14] solves the problem in

$O(N(\log N)^{M-1})$, which is much faster for small values of K, as well as for large values of N, than fast non-dominated sorting. However, this algorithm was designed with an assumption that no two points have equal objectives, which is often not the case, especially in discrete optimization, and is known to produce wrong results when this assumption is violated. This problem was overcome by Fortin et al. [12], who proposed modifications of this algorithm to always produce correct results. The average complexity was proven to be the same, but the worst-case complexity was left at $O(N^2M)$. Finally, Buzdalov et al. [2] introduced further modifications to achieve the worst-case time complexity of $O(N(\log N)^{M-1})$.

We shall now briefly illustrate the working principles of this approach. At any moment of time, the algorithm maintains, for every point p, a lower bound on its rank $r'(p)$, which are initially set to zero. The reason for this lower bound can be explained as follows: at any moment of time, we have performed a subset of necessary objective comparisons, which impose approximations of ranks of the affected points. These approximations are of course lower bounds of the real ranks.

To ease the notation, in the following we do not use the term "lower bound of the rank", as well as the r' symbol. Instead, we will say "current rank" for the current state of the lower bound of a certain point, which possibly coincides with the real rank, and "final rank" when we know that the lower bound coincides with the real rank.

One of the main properties of the algorithm is that whenever a comparison of p_m and q_m is performed for the first time, where p and q are points and $1 \le m \le M$ is the objective, then the following holds:

- for all objectives m' such that $m < m' \le M$, it holds that $p_{m'} \le q_{m'}$, that is, p weakly dominates q in objectives $[m'; M]$;
- the rank of p is known and final, that is, all comparisons necessary to determine the rank of p have already been done.

The top-level concept is the procedure HelperA(S, m), which takes a set of points S sorted lexicographically (where non-zero lower bounds are possibly known for some of the points from S) and makes sure all necessary comparisons between the objectives $[1; m]$ of these points are performed. This procedure is called only when all necessary comparisons of points p and q, such that $q \in S$ and $p \notin S$, have already been performed. To perform non-dominated sorting of a set P with M objectives, one should run HelperA(P, M).

For $m = 2$, it calls a sweep line based algorithm SweepA(S), which runs in $O(|S| \log |S|)$, which we will cover later. If there are at most two points in S, it performs their direct comparisons and updates the rank of the second point if necessary. If all values of the objective m are the same in the entire S, it directly calls HelperA$(S, m-1)$. Otherwise, it divides S into three parts using the objective m: the S_L part with lower values, the S_M part with median values, and the S_H part with higher values.

It is clear that ranks of points in S_L do not depend on ranks of points in neither S_M nor S_H, and S_M also does not depend on S_H. The algorithm first calls HelperA(S_L, m), which results in finding the exact ranks in S_L, because all necessary comparisons with points from S_L on the right side and other points on the left side have been performed before this call.

Next comes the set S_M, but the ranks of these points still need to be updated using the set S_L (and nothing more). To do this, the algorithm calls another procedure, HelperB$(S_L, S_M, m-1)$, whose meaning is to update the ranks of points from the second argument using the first argument and objectives in $[1; m-1]$. Then it calls HelperA$(S_M, m-1)$, as all other necessary comparisons have been done, and all values for the objective m are equal in S_M. It then proceeds with HelperB$(S_L \cup S_M, S_H, m-1)$ and finishes with HelperA(S_H, m).

The HelperB(L, H, m) procedure, as follows from the short description above, shall perform all the necessary comparisons between points $p \in L$ on the left and $q \in H$ on the right, provided that in objectives $[m+1; M]$ it holds that $p \prec q$, and all ranks in L are final. For $m = 2$, it, again, runs a sweep line procedure SweepB(L, H). If $|L| = 1$ or $|H| = 1$, a straightforward pairwise comparison is performed. If the maximum value of the objective m in L does not exceed the minimum value in H, it calls HelperB$(L, H, m-1)$. Otherwise, it chooses a median of the objective m in $L \cup H$ and then, similarly to HelperA, splits L into L_L, L_M and L_H, and also splits H into H_L, H_M and H_H. Following the same logic as in HelperA, it performs the following recursive calls:

- HelperB(L_L, H_L, m);
- HelperB$(L_L, H_M, m-1)$;
- HelperB$(L_M, H_M, m-1)$;
- HelperB$(L_L \cup L_M, H_H, m-1)$;
- HelperB(L_H, H_H, m).

The remaining parts to explain are SweepA(S) and SweepB(L, H). The SweepA procedure utilizes a sweep line approach. Points from the set S are processed in lexicographical order using first two objectives. In the same time, the procedure maintains a binary search tree which contains the last seen representative points for each non-domination level. When the next point is processed, this binary search tree is traversed to determine the biggest number of the level which still dominates the point in question, and then the rank of this point is updated correspondingly. After that, this point is inserted in the tree: it becomes the last representative of its non-domination level and possibly throws out some of the other representatives, which have no more chance to determine rank of any point on their own. An example is shown in Fig. 1.

The SweepB procedure works in a similar way. The sweep line goes over the union of sets, $L \cup H$, however, the tree is built of the points from L only, and rank updates are performed with points from H only.

The running times of SweepA and SweepB are $O(|S| \log |S|)$ and $O((|L| + |H|) \log |L|)$, respectively. From the well-known theory of solving recursive relations, and from the strategies of creating subproblems, it follows that the running time of HelperB(L, H, m) is $O((|L| + |H|) \cdot (\log(|L| + |H|))^{m-1})$, and of HelperA$(S, m)$ it is $O(|S| \cdot (\log |S|)^{m-1})$.

2.3 Efficient Non-domination Level Update

The Efficient Non-domination Level Update, or ENLU, was proposed in [17], see also [18] for the journal version. In a sense, it is based on the fast non-dominated sorting [7], however, it performs only those comparisons which have a point, which is going to change

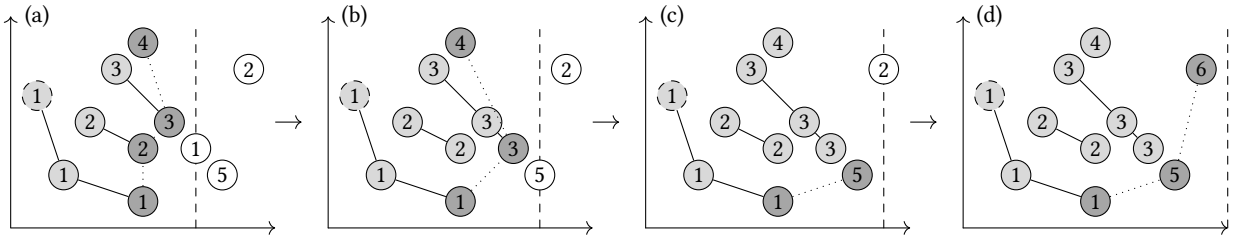

Figure 1: Example iterations of the SweepA procedure. Gray points are those whose rank is determined, the darker ones constitute a binary search tree and thus connected with dotted lines. The numbers in white points are the lower bounds on ranks. The vertical dashed line is the sweep line. In (b), the representer of level 2 is removed as all possible remaining points dominated by that point are also dominated by the representer of level 3; similar thing happens in (c).

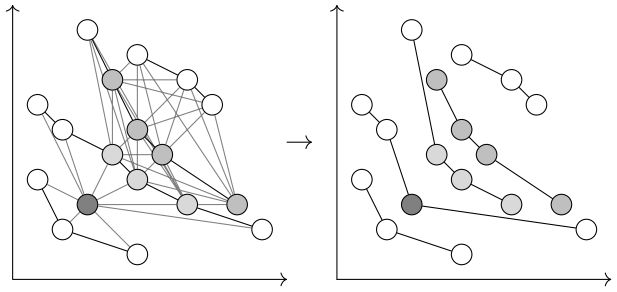

Figure 2: Example of an ENLU behavior. The darkest point is the newly inserted point. All other gray points are the points which change their rank. The gray lines indicate the comparisons which ENLU performs. The initial levels are shown on the left, the resulting levels on the right.

its rank, at either side. Although it does not change the worst-case complexity, which is $O(N^2M)$ and which is reachable on some synthetic data [25], it significantly improves performance in an average case and on the real-world data [18].

When a point is inserted, ENLU first determines which level this point should belong to. This is done by performing comparisons for domination of the inserted point with all points in level 0, 1, ..., until the level is found where no point dominates the current one. Assume this level has the number i, then the current point belongs to this level. It can dominate some points from the previous state of the level i, which then have to belong to the next level, $i + 1$. These next points need, in turn, to be compared with all points of level $i + 1$ to determine which rank-$(i + 1)$ points have to be promoted further. This process continues, until, at a certain point of time, the set of promoted points becomes empty, or these points shall form a new level (possibly not the last one – it can be the case that *every* point of the next level is dominated by some promoted point). An example of how ENLU works is presented on Fig. 2.

2.4 Incremental Non-dominated Sorting for Two Dimensions

If one looks on Fig. 2, one may immediately notice that the points which change their ranks appear both in their initial levels and their final levels as contiguous ranges. This is indeed true for two dimensions, which enables a particularly efficient algorithm for incremental non-dominated sorting presented in [25]. This algorithm represents every level as a binary search tree which supports split and merge operations in $O(\log n)$ time, where n is the size of the tree.

Using this representation, it is possible to do the entire update of a single level with two splits and one merge, while the set of promoted points – which is, again, a tree – is moved between the levels as a whole. To enable fast insertion of a level between two levels, which can happen, just as in ENLU, when the set of promoted points dominates the entire next level, the levels themselves are stored in a tree. This algorithm has the worst-case performance of $O(N)$, which is a tight bound, and the average performance of $O(\log N)$ in realistic optimization conditions (e.g. when the number of levels is constant). A full implementation of a steady-state version of the NSGA-II algorithm, based on this algorithm, is analyzed in [3], where it outperformed ENLU on realistic population sizes.

Unfortunately, not all these ideas generalize to three or more dimensions. One of these generalizations could be formulated as follows: assume we are promoting a set of points P to a level i, then all points in the level i which are dominated by at least one point in P are also dominated by the maximum point which dominates all the points in P. This is true for two dimensions, but for three dimensions there exists a counter-example. Assume level 0 has points $p_1 = (0, 2, 2)$, $p_2 = (1, 1, 4)$, $p_3 = (3, 4, 1)$, level 1 has a point $p_4 = (2, 3, 3)$ which is dominated by p_1. Assume we insert the point $p_X = (1, 1, 1)$, which dominates p_2 and p_3. When trying to push them to level 1, we form the maximum point $\min(p_2, p_3) = (1, 1, 1)$, which dominates p_4, and thus p_4, according to our generalization, should be assigned rank 2. However, neither p_2 nor p_3 dominates p_4, which means that p_4 remains with rank 1. This difficulty is, to some extent, overcome in this paper.

3 ADAPTING THE DIVIDE-AND-CONQUER ALGORITHM TO INCREMENTAL SORTING

In this section we first describe simple adaptations, which can be applied to the divide-and-conquer algorithm to speed up incremental non-dominated sorting. After that, we describe our main algorithm, which unites these adaptations with the ideas proposed in both ENLU and the algorithm for two-dimensional incremental sorting.

3.1 Finding the Rank of the New Point

The simplest way to determine the rank of the newly inserted point p is to test every existing point q for whether it dominates p using the straightforward $O(M)$ domination test, and to adjust the rank of p accordingly. This will be $O(N \cdot M)$ in the worst case. If, however, the points are stored in groups (that is, non-domination levels) by their ranks, this search can be somewhat optimized using binary search in levels. Depending on how points are distributed in levels, this may degenerate to $O(N \cdot M)$ as well. If the points in each level are ordered lexicographically, the search time in each level can be additionally reduced in practice by stopping the search whenever the abscissa of the current level point exceeds the abscissa of p. Again, this improvement does not influence the worst case.

A substantial improvement can be achieved if certain data structures are used for storing points in each level. For instance, Gustavsson and Syberfeldt used *k-d* trees for this purpose, which reduced the search time to $O((\log N)^2)$ [13]. As maintaining such trees while doing other operations is not a trivial task, we do not use them in this work, however, this can be a topic of the future work. In incremental sorting, this idea has been implemented for two dimensions in [25] with the same $O((\log N)^2)$ worst-case complexity, however, it is unclear how to generalize it to higher dimensions.

3.2 Reducing the Set of Points to Work With

The following simple lemma enables dropping many points when running non-dominated sorting after insertion of a new point.

Lemma 3.1. *If a point p is inserted into a set of points P with correctly assigned ranks, then ranks can be changed only for the points from the subset $P' = \{p' \in P \mid p \prec p'\}$.*

Proof. Assume there exists a point q, such that p does not dominate q, which originally had a rank r and changed its rank when p was inserted. Consider such point with the smallest original rank. This means that either $r = 0$ or all points with rank $r - 1$ remained as such after insertion of p. Thus, the only point which has a rank of at least r, which in the same time dominates q, is p. However, by the assumption, p does not dominate q. □

The corresponding algorithm for incremental non-dominated sorting first looks up the maximum level i which dominates the inserted point p, then determines the set of points P_p which are dominated by p, then calls HelperB($\{p\}, P_p, M$) to perform all necessary explicit comparisons with the point p, then updates the ranks of P_p by calling HelperA(P_p, M). In fact, HelperB($\{p\}, P_p, M$) immediately reduces to the pairwise domination comparison of p and every point in P_p, which runs in $O(M \cdot |P_p|)$. The complexity of the HelperA(P_p, M) call is $O(|P_p| \cdot (\log |P_p|)^{M-1})$, which, in the worst case, is the same complexity as the whole divide-and-conquer algorithm. However, in practice, the size of P_p is often much smaller than N, which leads to the running time improvement.

3.3 Taking Advantage of Known Ranks

One more improvement idea follows from another simple lemma.

Lemma 3.2. *If a point p is inserted into a set of points P with correctly assigned ranks, then after insertion these ranks never decrease, and they increase by at most one.*

Proof. Assume there exist points whose rank decreased once p is inserted, and consider a point q with the smallest original rank $r > 0$. This means that the point with rank $r - 1$ which dominated q had disappeared, which contradicts the lemma statement, as no point was removed.

Assume there exist points whose rank increased by at least two once p is inserted, and consider such point q with the smallest original rank r. If $r = 0$, this implies that at least two points were inserted, one dominating another, which contradicts the lemma statement. Otherwise, as all points with rank $r - 1$ have their rank increased by at most one (since q is the point with the minimum rank r among those whose rank increased by at least two), there is no point, except for the inserted one, which is dominated by at least one point of original rank $r - 1$ and in the same time dominates q. However, the mentioned point of rank $r - 1$ dominates the inserted point p and has its rank changed, which contradicts Lemma 3.1, whose preconditions are the same as of the current lemma. □

This lemma enables re-implementing SweepA and SweepB specially for incremental non-dominated sorting to work in linear time. Instead of maintaining a binary search tree, and querying it to determine the closest point which dominates the point in question, one can maintain and query a hash table, which maps from the rank to the point with smallest ordinate which has this rank. When a point p needs to be processed, the hash table is queried for its current rank, and if the query result exists and dominates p, the rank of p is incremented.

The running time of this modification, along with the heuristic described in Section 3.2, is $O(|P_p| \cdot (\log |P_p|)^{M-2})$, where $P_p = \{p' \in P \mid p \prec p'\}$ is the set of points in P which are dominated by the inserted point p. Thus, Lemma 3.2 improves the worst-case running time of the incremental non-dominated sorting, compared to the full-scale divide-and-conquer algorithm, by a factor of $\log N$.

We should note that the heuristic from Section 3.2 can alone demonstrate the $O(|P_p| \cdot (\log |P_p|)^{M-2})$ behavior when the number of levels in the affected point set P_p is constant. Under this condition, the tree size in either SweepA or SweepB is constant as well, as there is at most one entry per active level, which renders their running time to be $O(|P_p|)$. This condition affects even the original divide-and-conquer algorithm, which makes it run faster towards the end of optimization, when all individuals tend to be non-dominated. However, this behavior is output-dependent, while the linear sweep version always demonstrates an asymptotically improved performance.

3.4 Is-Anything-Changed Heuristic

The last idea which improves the running time of incremental divide-and-conquer modifications is based on the following observation. Suppose such algorithm calls HelperB($X, Y, m - 1$) and then HelperA(Y, m) or HelperA($Y, m - 1$). Assume further that the HelperB call has not changed rank of any point in the set Y, which can often happen, as large parts of the dominated region of the inserted point p can still remain untouched by this insertion. In this case, there is no need to call HelperA, because the existing ranks are already correct. Implementation of this heuristic imposes only a negligible overhead, so even if it never removes an extra HelperA call, it does not influence the running time.

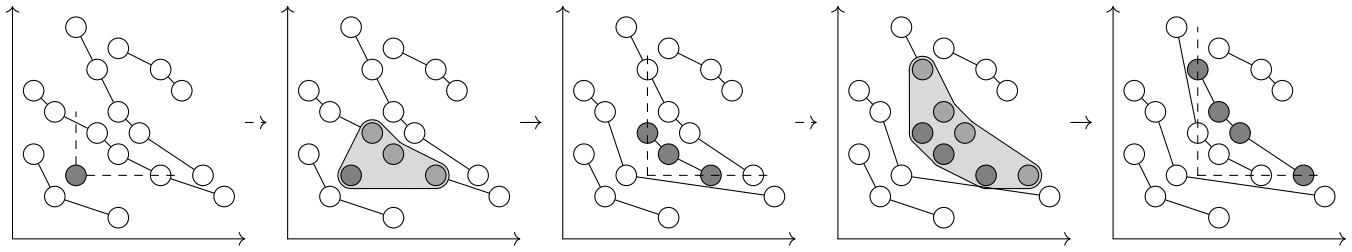

Figure 3: An example of behavior of the proposed algorithm. Gray areas denote the areas where HelperB is called, where dark gray points belong to the first argument, and light gray ones to the second argument. In pictures without gray areas, promoted points are marked gray.

3.5 Putting Everything Together

Now we merge the ideas discussed above, as well as the ideas from both ENLU and the two-dimensional incremental algorithm, to create the algorithm for incremental non-dominated sorting with both the best known theoretically worst-case complexity and very competitive performance.

A very short description of the algorithm is as follows:

- the rank of the point p being inserted is found using binary search in levels, where testing whether a level L dominates p is done using domination comparisons in $O(|L| \cdot M)$;
- as in ENLU and the algorithm for two dimensions, the set of promoted points Q is maintained, where initially $Q = \{p\}$;
- on each iteration, the set Q is pushed into the next level L_i using the following procedure:
  - the maximum dominating point q is created from the set Q by taking, for each objective, the minimum over the entire Q;
  - the affected subset of the level is formed: $L'_i = \{p' \in L_i \mid q \prec p'\}$;
  - the ranks of points in L'_i are updated by calling first HelperB(Q, L'_i, M), where the procedure SweepB is implemented as in Section 3.3;
  - the points in L'_i which have retained the rank i remain in the new level L_i along with the entire set Q;
  - the new set Q is formed from the points in L'_i which have changed their rank.
- whenever Q is empty, or dominates the next level completely, the appropriate actions are taken as in ENLU and the algorithm for two dimensions.

The pipeline of the proposed algorithm is very similar to the one in the two-dimensional incremental non-dominated sorting, except that we have now to create the affected subset in linear time, and we also have to filter it using the set of promoted points using the algorithm from Section 3.3. What is more, during this filtration we do not have to call HelperA(L'_i, M), as in this case the dominated set, L'_i, is a piece of a non-domination layer L_i, thus no two points from L'_i ever dominate each other, and no comparisons between them are necessary. This means that the heuristic from Section 3.4 never fires in this algorithm. An example of the behavior of this algorithm can be seen in Fig. 3.

The worst-case running time of this algorithm is $O(N(\log N)^{M-2})$, as every point is used at most twice in a series of runs of HelperB, where every run takes $O(L(\log L)^{M-2})$, L being the size of the set of promoted points plus the size of the affected set. As in Section 3.3, one can express the worst-case bound also as $O(|P_p| \cdot (\log |P_p|)^{M-2})$, where P_p is the subset of all points dominated by the inserted point p. Note that the real set of affected points can be smaller than P_p, as the algorithm, if data allow, narrows down in each iteration the conditions for a point to become affected.

4 EXPERIMENTS

For experimental comparison, we consider the following algorithms (with the notations used further in the text and in figures):

- *DC*: the divide-and-conquer algorithm as in [2];
- *DC1*: the algorithm based on Sections 3.2 and 3.4;
- *DC2*: the algorithm based on Sections 3.2, 3.3 and 3.4;
- *Level*: the main algorithm of this paper as in Section 3.5;
- ENLU: the ENLU algorithm from [18];
- *2D*: the incremental algorithm from [25] for $M = 2$.

All these algorithms were implemented[1] in Java (the runtime version 1.8.0_121), and their performance was evaluated on an eight-core machine with Intel® Xeon® E5606 CPUs clocked at 2.13GHz running 64-bit Linux (kernel version 4.4.39).

We evaluated the algorithms on the well-known benchmark problems DTLZ1–DTLZ4 and DTLZ7 [8], as well as ZDT1–ZDT4 and ZDT6 [29]. Our datasets were created as follows. First, we simulated first 90 generations of the generational[2] NSGA-II with population size 10000, while recording the state of the population at the beginning of generations 0, 10, ..., 90, where 0 is the initial random population. Next, for every such state we synthesized a random individual to insert into population using the selection mechanisms and genetic operators of NSGA-II, thus creating an insertion task. Finally, for every algorithm and every insertion task we performed measurements of the point insertion time using the Java Microbenchmark Harness (JMH) suite[3] using 30 warm-up iterations and six measuring iterations, each for two forks of the Java virtual machine.

The measurement results are shown on Fig. 4 for two-dimensional ZDT problems, Fig. 5 for three-dimensional DTLZ problems, Fig. 6 for DTLZ1 with dimensions $\{3, 4, 6, 8, 10\}$, and Fig. 7 for DTLZ2

[1] The implementation is available at GitHub: https://github.com/yakupov/nds2016

[2] The idea here is to generate realistic data reflecting various optimization stages, e.g. the more iterations we did, the closer the population is to the Pareto-optimal set. From this point of view, generational and steady-state NSGA-II algorithms behave the same.

[3] http://openjdk.java.net/projects/code-tools/jmh

with the same dimensions. On these figures, the error bars denote the minimum and maximum observed running times.

It can be seen that the running time of *DC* has a tendency to decrease with more generations. The biggest difference is for the two-dimensional ZDT problems, where after 90 generations it decreases almost by 10 times compared to the initial population. This happens because, as the population converges to the Pareto front, the number of levels reaches some constant, thus removing a factor of $O(\log N)$ in SweepA and SweepB.

However, all incremental algorithms, including *DC1* and *DC2*, demonstrate a much better performance on all datasets. In the case of *DC1*, which has precisely the same worst-case asymptotic behavior as *DC*, this can be related only to ignoring points not dominated by the inserted point p. *DC2*, except for few cases, demonstrates only a marginally better running time compared to *DC1*. Apart from two-dimensional cases, ENLU and *Level* are clear winners by a large margin. On certain occasions, however, ENLU performs even worse than *DC1* and *DC2*, and under these conditions *Level* performs the best. Finally, for two dimensions the *2D* algorithm shows much better results than any other algorithm, indicating that there is enough room for improving the general-purpose algorithms.

For all 180 different configurations of optimization problem, number of objectives and simulated NSGA-II generations, presented in Fig. 4–7, we conducted the Wilcoxon rank sum tests for the results of *Level* and ENLU, with both one-sided alternative hypotheses, and applied the Bonferroni correction (using the multiple 360). Out of these cases, *Level* was statistically better than ENLU in 119 cases, worse in 31 cases and indistinguishable in 30 cases, all with the threshold p-value of 0.01. This suggests that the proposed algorithm typically performs better than ENLU.

5 CONCLUSION

We presented an algorithm for incremental non-dominated sorting, to be used together with steady-state evolutionary multiobjective algorithms, which demonstrates the best known worst-case complexity, $O(N(\log N)^{M-2})$, and shows the practical performance on par or better than its closest competitor to date, the Efficient Non-domination Level Update algorithm. This algorithm works well with coinciding point coordinates arising in discrete settings.

While we admit that the amount of experimental work is only a tiny fraction of what is necessary for this topic, results of these experiments suggest that our algorithm is competitive and may be an adequate choice for practitioners and for researchers who develop steady-state evolutionary algorithms.

This research was financially supported by the Government of Russian Federation, Grant 074-U01.

REFERENCES

[1] Dimo Brockhoff and Tobias Wagner. 2016. GECCO 2016 Tutorial on Evolutionary Multiobjective Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*. 201–227.

[2] Maxim Buzdalov and Anatoly Shalyto. 2014. A Provably Asymptotically Fast Version of the Generalized Jensen Algorithm for Non-Dominated Sorting. In *Parallel Problem Solving from Nature – PPSN XIII*. Number 8672 in Lecture Notes in Computer Science. Springer, 528–537.

[3] Maxim Buzdalov, Ilya Yakupov, and Andrew Stankevich. 2015. Fast Implementation of the Steady-State NSGA-II Algorithm for Two Dimensions Based on Incremental Non-Dominated Sorting. In *Proceedings of Genetic and Evolutionary Computation Conference*. 647–654.

[4] C.A. Coello Coello and G. Toscano Pulido. 2001. A Micro-Genetic Algorithm for Multiobjective Optimization. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*. Number 1993 in Lecture Notes in Computer Science. 126–140.

[5] David W. Corne, Nick R. Jerram, Joshua D. Knowles, and Martin J. Oates. 2001. PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 283–290.

[6] Kalyanmoy Deb and Himanshu Jain. 2013. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2013), 577–601.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[8] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. 2005. Scalable Test Problems for Evolutionary Multiobjective Optimization. In *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*. Springer, 105–145.

[9] M. Erickson, A. Mayer, and J. Horn. 2001. The Niched Pareto Genetic Algorithm 2 Applied to the Design of Groundwater Remediation Systems. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*. Number 1993 in Lecture Notes in Computer Science. 681–695.

[10] Hongbing Fang, Qian Wang, Yi-Cheng Tu, and Mark F. Horstemeyer. 2008. An Efficient Non-dominated Sorting Method for Evolutionary Algorithms. *Evolutionary Computation* 16, 3 (2008), 355–384.

[11] C. M. Fonseca and P. J. Fleming. 1996. Nonlinear System Identification with Multiobjective Genetic Algorithm. In *Proceedings of the World Congress of the International Federation of Automatic Control*. 187–192.

[12] Félix-Antoine Fortin, Simon Grenier, and Marc Parizeau. 2013. Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting. In *Proceedings of Genetic and Evolutionary Computation Conference*. ACM, 615–622.

[13] Patrik Gustavsson and Anna Syberfeldt. 2017. A New Algorithm Using the Non-dominated Tree to improve Non-dominated Sorting. *Evolutionary Computation* (Jan. 2017). Just Accepted publication.

[14] M. T. Jensen. 2003. Reducing the Run-time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms. *IEEE Transactions on Evolutionary Computation* 7, 5 (2003), 503–515.

[15] Joshua D. Knowles and David W. Corne. 2000. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation* 8, 2 (2000), 149–172.

[16] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. 1975. On Finding the Maxima of a Set of Vectors. *Journal of ACM* 22, 4 (1975), 469–476.

[17] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Sam Kwong. 2014. *Efficient Non-domination Level Update Approach for Steady-State Evolutionary Multiobjective Optimization*. Technical Report COIN 2014014. Michigan State University. www.egr.msu.edu/~kdeb/papers/c2014014.pdf

[18] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Qiang Zhang. 2016. Efficient Nondomination Level Update Method for Steady-State Evolutionary Multiobjective Optimization. *IEEE Transactions on Cybernetics* (2016), 1–12. DOI: http://dx.doi.org/10.1109/TCYB.2016.2621008 accepted for publication.

[19] Kent McClymont and Ed Keedwell. 2012. Deductive Sort and Climbing Sort: New Methods for Non-Dominated Sorting. *Evolutionary computation* 20, 1 (2012), 1–26.

[20] Antonio J. Nebro and Juan J. Durillo. 2009. On the Effect of Applying a Steady-State Selection Scheme in the Multi-Objective Genetic Algorithm NSGA-II. In *Nature-Inspired Algorithms for Optimisation*. Number 193 in Studies in Computational Intelligence. Springer Berlin Heidelberg, 435–456.

[21] Yakov Nekrich. 2011. A Fast Algorithm for Three-Dimensional Layers of Maxima Problem. In *Algorithms and Data Structures*. Number 6844 in Lecture Notes in Computer Science. 607–618.

[22] Proteek Chandan Roy, Md. Monirul Islam, and Kalyanmoy Deb. 2016. Best Order Sort: A New Algorithm to Non-dominated Sorting for Evolutionary Multi-objective Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*. 1113–1120.

[23] N. Srinivas and Kalyanmoy Deb. 1994. Multiobjective Optimization Using Non-dominated Sorting in Genetic Algorithms. *Evolutionary Computation* 2, 3 (1994), 221–248.

[24] Handing Wang and Xin Yao. 2014. Corner Sort for Pareto-Based Many-Objective Optimization. *IEEE Transactions on Cybernetics* 44, 1 (2014), 92–102.

[25] Ilya Yakupov and Maxim Buzdalov. 2015. Incremental Non-Dominated Sorting with $O(N)$ Insertion for the Two-Dimensional Case. In *Proceedings of IEEE Congress on Evolutionary Computation*. 1853–1860.

[26] Quingfu Zhang and Hui Li. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation* 11, 6 (2007), 712–731.

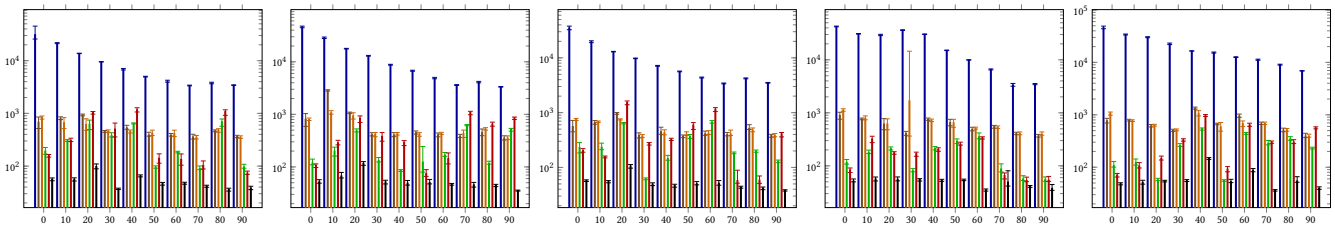

Figure 4: Running times of single point insertions, in microseconds, of the algorithms on two-dimensional ZDT1–ZDT4 and ZDT6. Abscissa marks are the number of generations simulated. The algorithms, left to right: *DC*, *DC1*, *DC2*, *Level*, ENLU, *2D*.

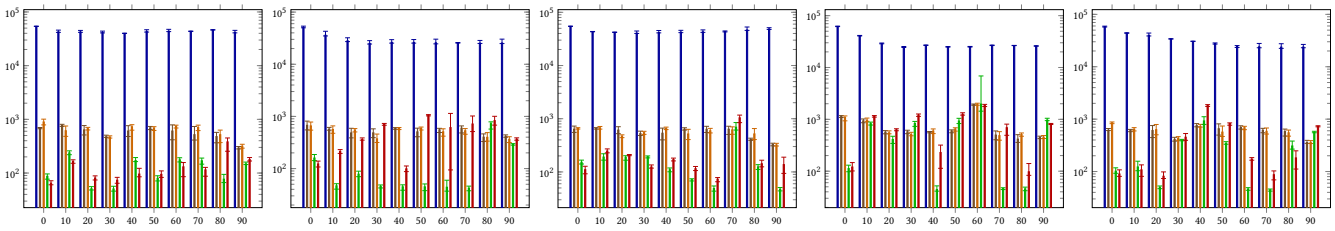

Figure 5: Running times of single point insertions, in microseconds, of the algorithms on three-dimensional DTLZ1–DTLZ4 and DTLZ7. Abscissa marks are the number of generations simulated. The algorithms, left to right: *DC*, *DC1*, *DC2*, *Level*, ENLU.

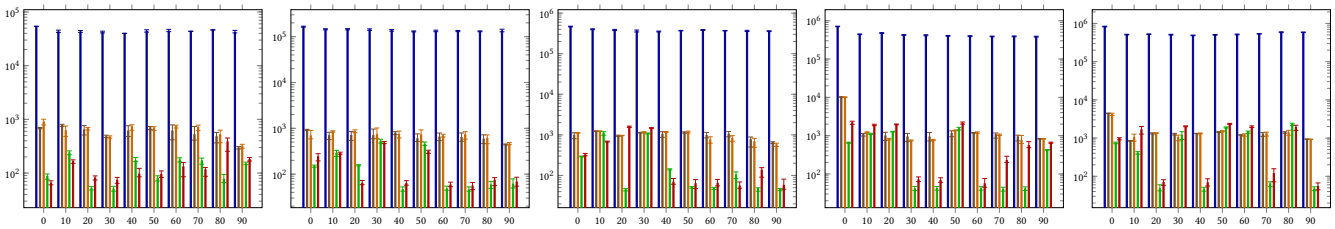

Figure 6: Running times of single point insertions, in microseconds, of the algorithms on DTLZ1 with dimensions 3, 4, 6, 8, 10. Abscissa marks are the number of generations simulated. The algorithms, left to right: *DC*, *DC1*, *DC2*, *Level*, ENLU.

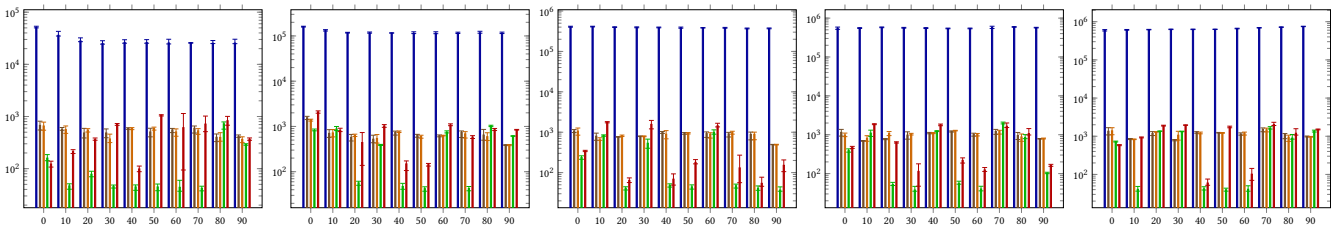

Figure 7: Running times of single point insertions, in microseconds, of the algorithms on DTLZ2 with dimensions 3, 4, 6, 8, 10. Abscissa marks are the number of generations simulated. The algorithms, left to right: *DC*, *DC1*, *DC2*, *Level*, ENLU.

[27] Xingyi Zhang, Ye Tian, Ran Cheng, and Yaochu Jin. 2015. An Efficient Approach to Nondominated Sorting for Evolutionary Multiobjective Optimization. *IEEE Transactions on Evolutionary Computation* 19, 2 (2015), 201–213.

[28] Xingyi Zhang, Ye Tian, Ran Cheng, and Yaochu Jin. 2016. A Decision Variable Clustering-Based Evolutionary Algorithm for Large-scale Many-objective Optimization. *IEEE Transactions on Evolutionary Computation* (2016). DOI: http://dx.doi.org/10.1109/TEVC.2016.2600642

[29] E. Zitzler, K. Deb, and L. Thiele. 2000. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation* 8, 2 (2000), 173–195.

[30] Eckart Zitzler and Simon Künzli. 2004. Indicator-Based Selection in Multiobjective Search. In *Parallel Problem Solving from Nature – PPSN VIII*. Number 3242 in Lecture Notes in Computer Science. 832–842.

[31] E. Zitzler, M. Laumanns, and L. Thiele. 2001. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Proceedings of the EUROGEN'2001 Conference*. 95–100.

[32] E. Zitzler and L. Thiele. 1999. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation* 3, 4 (1999), 257–271.

[33] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. 2003. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation* 7, 2 (2003), 117–132.