Efficient Removal of Points with Smallest Crowding Distance in Two-dimensional Incremental Non-dominated Sorting

Niyaz Nigmatullin
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
nigmatullin@rain.ifmo.ru

Maxim Buzdalov
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
buzdalov@rain.ifmo.ru

Andrey Stankevich
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
stankev@rain.ifmo.ru

ABSTRACT

Many evolutionary multi-objective algorithms rely heavily on non-dominated sorting, the procedure of assigning ranks to individuals according to Pareto domination relation. The steady-state versions of these algorithms need efficient implementations of incremental non-dominated sorting, an algorithm or data structure which supports efficient addition of a new individual and deletion of the worst individual.

Recent research brought new advanced algorithms, but none of them can be cheaply adapted to sublinear location of the point having the smallest crowding distance, a measure used in the NSGA-II algorithm. In this paper we address this issue by reducing it to a series of extreme point queries to certain convex polygons. We present theoretical estimation of the worst-case running time, as well as experimental results which show that the proposed modifications reduce the running time significantly on benchmark problems for large population sizes.

Keywords

Multi-objective optimization, crowding distance, performance evaluation, non-dominated sorting, steady-state algorithms.

1. INTRODUCTION

Many well-known and widely used evolutionary multi-objective algorithms use the procedure of *non-dominated sorting* for ranking solutions. Such algorithms include the famous NSGA-II [7], NSGA-III [6], DM1 [1], MOPSO [5] and many more. The time complexity of these algorithms is often dominated by the time complexity of non-dominated sorting, which is especially true for large generation sizes [10]. One can often make these evolutionary algorithms work faster without changing the quality of results by using a faster implementation of non-dominated sorting.

GECCO'16 Companion July 20-24, 2016, Denver, CO, USA

ACM ISBN 978-1-4503-4323-7/16/07.

DOI: http://dx.doi.org/10.1145/2908961.2931685

Evolutionary algorithms are typically thought to be efficient when running on parallel computers. However, most of then are *generational*, which means that they typically have to wait for fitness evaluation of the entire population in order to perform the next task. In turn, this reduces the efficiency of using a parallel computer or cluster. To utilize the full power of such a machine, asynchronous algorithms seem to be better. *Steady-state* evolutionary algorithms are closer to being asynchronous, since they are designed to support insertion, deletion and querying of single individuals.

Steady-state versions of some evolutionary multi-objective algorithms are known. For instance, a steady-state version of the NSGA-II algorithm [17] is reported to have a good convergence rate and high quality of Pareto front approximation on benchmark problems. However, it runs the non-dominated sorting procedure each time a new individual is added, which increases the running time from $O(N^2K)$ for a population of $\Theta(N)$ individuals and K objectives, to $O(N^2K)$ for a single individual, which is $\Theta(N)$ times slower. These running times hold for fast non-dominated sorting [7] and many sequential algorithms for non-dominated sorting [15, 20, 22]; for certain algorithms based on the divide-and-conquer approach [3, 9, 12], the corresponding bound is $O(N(\log N)^{K-1})$, but it nevertheless becomes $\Theta(N)$ times slower for the steady-state algorithms. Thus, there is a need for an efficient method of updating the state of non-dominated sorting each time a new individual arrives or one existing individual, typically from the last layer, is deleted. Such a method would allow many possible steady-state variations of evolutionary algorithms to be implemented and analyzed, which may open a door for better algorithms [2].

The first algorithm for efficient handling of incremental insertions and deletions, the "Efficient Non-dominated Level Update" (or ENLU for short), was proposed in 2014 in a technical report by Ke Li et al [13], see also the later version [14]. Its main idea is to maintain the set of points which are changing their layer from k to $k+1$ and, based on this set, construct the next set from points which change their layer from $k+1$ to $k+2$. Although its worst-case time complexity is $\Theta(N^2K)$ for a single insertion, its average time complexity is typically much smaller. The paper [16] suggests some improvements on the ideas of ENLU by using binary search trees in appropriate places, but without an asymptotic improvement in the worst case.

A slightly different approach is used in the papers by Buzdalov et al. [4, 21]. This approach currently covers only

the two-dimensional case, but it brings a remarkable $O(N)$ worst-case complexity of a single insertion or a single deletion. The key idea of this approach is that in two dimensions the pieces of non-dominated layers are manipulated in contiguous fragments, which results in $O(1)$ operations per layer. These operations can be handled efficiently by using binary search trees augmented with some appropriate information in the nodes. As [4] mentions, the implementation is quite efficient even for $N \approx 100$ – the running times are approximately 50% smaller than those of ENLU for these N. What is more, the conditions for $\Theta(N)$ insertion time are quite rare; for example, for the constant number of fronts, the insertion time can be proven to be $O(\log N)$.

In NSGA-II, a measure called the *crowding distance* is used to choose the worst point(s) to remove from the last Pareto layer. To evaluate the worst point in the steady-state setting for the two-dimensional case, a straightforward algorithm with $\Theta(N_{\text{last}}) = O(N)$ time is used, where N_{last} is the size of the last layer, assuming that the data structure keeps the last layer sorted by one of the coordinates. This $O(N)$ looks the same as the $O(N)$ insertion time. However, the worst cases happen in radically different conditions: $\Theta(N)$ insertions happen in rare conditions when there are $\Theta(N)$ specifically aligned layers, while $\Theta(N)$ removals happen when the last layer has the $\Theta(N)$ size, which is a usual condition in multi-objective optimization, especially when some noticeable optimization time has been spent. This leads to the idea that the overall running time may be reduced by speeding up the removal part, probably even at a slight expense of slowing down the insertion.

This paper is dedicated to reduction of the running time of the removal part. Note that the removal part consists of *location* of the worst point and of actual *deletion* of that point. While deletion is trivial and can be done in $O(\log N)$, location is significantly harder. The crowding distance of a point is computed in the two-dimensional case as $\frac{dx}{DX} + \frac{dy}{DY}$, where dx and dy are the coordinate-wise distances between the point's neighbors, while DX and DY are the coordinate-wise spans of the whole layer. While dx and dy, being local properties, can be easily tracked for each point during updates, DX and DY are global properties and may also change. We reduce the number of points which have to be checked for being the worst point using convex hulls. More precisely, we associate with every point in the last layer another *derivative* point $(dx; dy)$ with dx and dy defined as above. The points inside the convex hull of any point set will never correspond to the worst points. What is more, the location of the worst point in the convex hull can be done in $O(\log L)$, where L is the convex hull size, as it reduces to minimization of the crowding distance, which is a linear function, in a convex polygon.

The rest of the paper is structured as follows. Section 2 recalls the necessary definitions, briefly describes the incremental non-dominated sorting algorithm from [4, 21], or INDS for short, which is used as the base algorithm, and explains the idea of the algorithm for finding extreme points in a convex polygon, which is the same as minimizing a linear function. In Section 3, the algorithm is given which augments INDS in order to maintain the necessary data. Section 4 contains the results of experiments on certain benchmark problems and their discussion. Finally, Section 5 concludes.

2. PRELIMINARIES

This section introduces the necessary definitions and recalls the algorithms which are used in this paper.

2.1 Pareto Domination and Non-dominated Sorting

In the K-dimensional space, a point $A = (a_1, \ldots, a_K)$ is said to *dominate* a point $B = (b_1, \ldots, b_K)$ (denoted as $A \prec B$) when for all $1 \le i \le K$ it holds that $a_i \le b_i$ and there exists j such that $a_j < b_j$. *Non-dominated sorting* of points in the K-dimensional space is a procedure of marking all points which are not dominated by any other point with the *rank* of 0, all points which are dominated by at least one point of the rank of 0 are marked with the rank of 1, all points which are dominated by at least one point of the rank of $i-1$ are marked with the rank of i. In this paper, we call a set of all points having the rank of i the *i-th layer*.

In most evolutionary multi-objective algorithms based on non-dominated sorting, only *offline* non-dominated sorting is used, where all points are known a priori. For steady-state algorithms, *online* non-dominated sorting is needed. The problem of online non-dominated sorting can be formulated as maintaining a data structure which supports (efficiently) at least the following operations:

- insert an arbitrary point;
- query the k-th point (in certain arbitrary but fixed order, for example, lexicographical) along with its rank;
- remove a point from the last layer (typically the worst point according to certain criterion).

These operations (the last one refined to the point with the smallest crowding distance) are already enough to implement NSGA-II [7], or its steady-state version [17]. However, for other algorithms the following operations may be necessary as well:

- return the total number of layers;
- iterate over all points from the given layer;
- return the minimum or the maximum from all points from the given layer by the given coordinate;
- return the rank of a given point without inserting it;
- evaluate a certain measure (for example, crowding distance) for a given point;
- remove a given point (if it exists).

2.2 Two-dimensional Incremental Non-dominated Sorting

In [21], a data structure for two-dimensional incremental non-dominated sorting was proposed. The main idea of this data structure is as follows: when a new point is inserted, the layers exchange possibly large but contiguous fragments (see Fig. 1 for an example). If an appropriate data structure is used, layers can be cut in pieces and glued together for just $O(\log N)$ per single operation. The paper [21] suggests using Cartesian trees for storing points in each layer, and another Cartesian tree is used for storing layers themselves (consult Fig. 2 for the illustration of the idea).

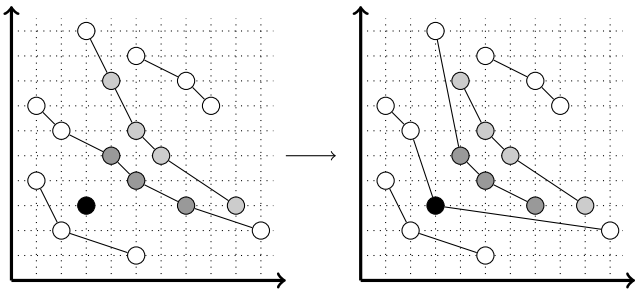

Figure 1: In two dimensions, insertion of a point makes layers exchange contiguous fragments.

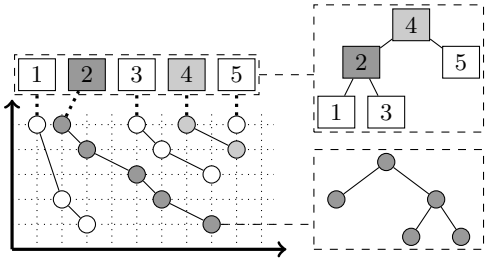

Figure 2: The tree-of-trees data structure from the paper [21].

Let there be N points contained in M layers. When a new point is inserted, the worst-case time for locating the layer for this point is:

$$O\left(\log M \cdot \log \frac{N}{\log M}\right).$$

The time for the actual point insertion can be bounded in the worst case as follows:

$$O\left(M \cdot \left(1 + \log \frac{N}{M}\right)\right).$$

These bounds are not worse than $O((\log N)^2)$ and $O(N)$, correspondingly [21]. A query of the k-th point in the predefined order – namely, first the points are compared by rank, then by first coordinate, finally by second coordinate – takes $O(\log N)$ worst-case time [4].

2.3 Convex Hulls and Extreme Points of a Convex Polygon

A *convex hull* of the set of points S is the smallest subset $H \subseteq S$ such that the convex closure of H contains all points from S. Many efficient algorithms are known which find the convex hull of the set of N two- and three-dimensional points in $O(N \log N)$ [18]. In these dimensions, a convex hull of two already built convex hulls can be built in $O(N)$ time [18].

Assume a linear function $F(s) = F(\{s_1, \ldots, s_K\}) = a_0 + a_1 s_1 + \ldots + a_K s_K$ is given. For any given set of points S, F reaches its minimum on its convex hull: $\min_{s \in S} F(s) = \min_{s \in H(S)} F(s)$. Indeed, for every point p inside the convex hull, there are $t > 1$ points $P_1, \ldots, P_t$ from the convex hull such that $p = \alpha_1 P_1 + \ldots + \alpha_t P_t$ where $\alpha_i \geq 0$ and $\sum_{i=1}^{t} \alpha_i = 1$ (the fact known as Carathéodory's theorem). The same will hold for the linear function F: $F(p) = \alpha_1 F(P_1) + \ldots + \alpha_t F(P_t)$, which means that $F(p) \geq \min_{s \in H(S)} F(s)$. Thus, we may skip the points inside the convex hull when we search for a point which minimizes any linear function.

For two-dimensional spaces, the algorithm exists which finds the extreme point (a point with the minimum value of a linear function) in $O(\log |H(S)|) = O(\log |S|)$ time [19, Section 7.9]. The idea of the algorithm is to perform binary search on the points of the convex hull ordered counterclockwise: assuming that the extreme point is contained between the indices l and r of the convex hull, we choose $m = \lfloor (l + r)/2 \rfloor$ and determine (by analyzing the six possible cases of relative locations of points P_l, P_m and the vector (α_1, α_2)) which of the segments, $[l; m]$ or $[m; r]$, contains the extreme point. The algorithm starts with $l = 1$, $r = |H(S)|$.

3. ALGORITHM

In this section, the proposed ways to modify the INDS algorithm from the paper [21] are explained. First, we explain how the search of the point with the smallest crowding distance can be reduced to the search of extreme point in a convex polygon. Second, we describe a technique which builds the convex hulls only when necessary and maintains their sizes in desired limits, so that the better running times are experienced and can be proven.

3.1 Reduction of Worst Point to Extreme Point of a Convex Polygon

Recall the definition of the crowding distance in two dimensions, assuming that we have a layer P sorted in increasing order of the x-coordinates, which is the same as decreasing order of the y-coordinate:

$$D(P_i) = \frac{X_{P_{i+1}} - X_{P_{i-1}}}{X_{P_{|P|}} - X_{P_1}} + \frac{Y_{P_{i-1}} - Y_{P_{i+1}}}{Y_{P_1} - Y_{P_{|P|}}}.$$

We denote for brevity the point-dependent values $dx(P_i) = X_{P_{i+1}} - X_{P_{i-1}}$ and $dy(P_i) = Y_{P_{i-1}} - Y_{P_{i+1}}$, as well as the front-depending values $DX = X_{P_{|P|}} - X_{P_1}$ and $DY = Y_{P_1} - Y_{P_{|P|}}$. In this notation we get that

$$D(P_i) = \frac{dx(P_i)}{DX} + \frac{dy(P_i)}{DY}.$$

In the data structure of INDS, points belonging to each layer are stored in a single binary search tree, for example, Cartesian tree. We augment each node of such tree with pointers to the next-in-order and the previous-in-order nodes, such that these pointers are updated when the tree is rebuilt without disrupting the $O(\log N)$ complexity of basic operations. With these pointers, we may evaluate $dx(P_i)$ and $dy(P_i)$ in $O(1)$ time. However, DX and DY, being front-depending values, cannot be easily reconstructed during tree updates, especially splits and merges, so tracking $D(P_i)$ for all points during updates seems infeasible.

However, when one needs to find the point with the smallest crowding distance, the values of DX and DY can be obtained in two queries to the tree of the last layer, each having $O(\log N)$ time complexity. With these known DX and DY, crowding distance becomes a linear function defined on $dx(P_i)$ and $dy(P_i)$. Thus if we store for each point P_i the *derivative point* $D_i = (dx(P_i), dy(P_i))$, and maintain a convex hull for a certain subset L of the layer P, the point with the smallest crowding distance can be found in this subset L in $O(\log |L|)$ time. Note that D_i can be easily maintained for all the points stored in the data structure. Consult Fig. 3 for an illustrative example of this approach.

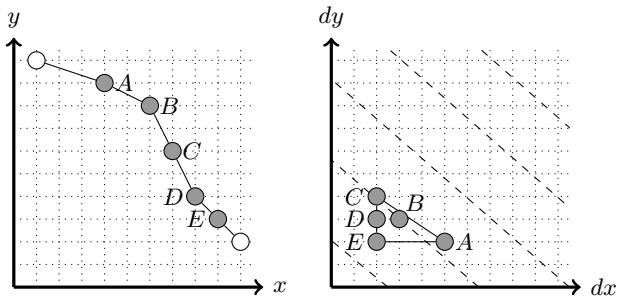

Figure 3: Illustration to searching for the point with the smallest crowding distance using convex hulls of derivative points. The current layer is given on the left plot. The boundary points have the coordinates of $(1; 10)$ and $(10; 2)$, so the crowding distance has the form of $D(P_i) = \frac{dx(P_i)}{9} + \frac{dy(P_i)}{8}$. We seek the smallest crowding distance among the gray points only. Their derivative points are given on the right plot. The convex hull of these derivative points is $\{A, C, E\}$. The slanted dashed lines on the right plot indicate lines with equal values of the crowding distance function, which indicates that the point with the minimum crowding distance is E.

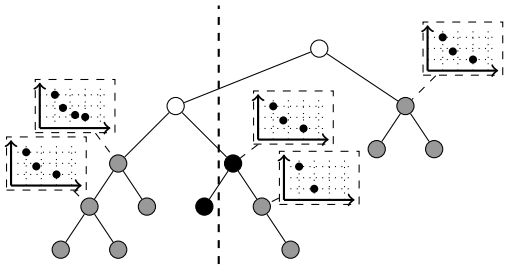

Figure 4: Convex hulls for tree vertices. The tree represents a single layer. White color is for the vertices without hulls. The maximum hull size is $L = 5$, so hulls are not built for the white vertices. Black color is for the vertices whose hulls are invalidated if the tree is split along the vertical line. All vertices colored gray preserve their hulls after the split.

3.2 Algorithm for Maintaining Hulls

If one maintains a convex hull for all the points for the last layer, finding the point with the smallest crowding distance (the "worst point" for simplicity) will require as few as $O(\log N)$ operations. Unfortunately, maintaining such a hull may require $O(N \log N)$ operations every time the contents of the last layer change, which renders the whole approach unusable. Indeed, on every insertion an arbitrary big set of points from the next-to-last layer may come to the last layer. This may change some of the crowding distance components of the existing points and subsequently may force to rebuild the convex hull. What is more, the last layer may be exhausted, and the next-to-last layer may become the new last layer – but only for a couple of steps. So we need to keep the balance between the complexity of convex hull maintaining and the ease of worst point query.

Our algorithm selects a number L and splits the last layer into several point groups of size at most L (but mostly close to L). This way, the running time of the worst point query would be $O(\frac{N_{\text{last}}}{L} \log L)$, where N_{last} is the size of the last layer. As we already have the points in a layer contained in a tree, we augment every node which has its subtree size of at most L with the convex hull of derivative points from its subtree (see Fig. 4).

When a tree is split or merged, the derivative points of some points change because either their neighbors change. Because of this, some hulls are typically invalidated on split and merge events. Fig. 4 illustrates this fact on the example of tree splitting.

Splitting and merging may happen not only in the last layer, but in all other layers too. In fact, if there is a decent number of layers, most convex hulls are built only to be subsequently invalidated. Our solution to this problem is to build convex hulls lazily: a hull is constructed only the first time it is accessed during search of the worst point. However, all hulls in the subtree of the corresponding tree vertex are built too. More precisely, we use the earlier mentioned fact that, for two dimensions, the convex hull of two convex hulls can be built in $O(N)$. This allows us to build the hulls in the entire subtree in $O(N \log N)$ time.

Due to this design choice, it is possible to prove that in each split or merge event the total expected size of invalidated hulls is bounded by $O(L)$. Indeed, for every tree node with changed derivative point the maximum containing hull has the size of $H = O(L)$. The corresponding node has only one "broken" child, because the other subtree is not affected. As the tree is binary and balanced, the average size of the hull in the "broken" child is at most $H/2$. When descending towards the deepest "broken" node, the hull sizes are cut in half for each layer, which brings the total expected size of "broken" hulls to be at most $H + H/2 + H/4 + \ldots < 2H = O(L)$. In our implementation we use Cartesian trees, but a similar bound with possibly different constant factors holds for any other binary balanced search tree.

It is clear that the proposed modification does not increase the running time during insertion, as convex hulls are never built in this phase. As the total number of points in invalidated hulls is at most $O(L)$ for each layer, and the subsequent work required to restore these broken hulls, if all of them are needed, is $O(L)$ as well, the average cost of building hulls at each deletion can be roughly estimated as $O(LM)$, where M is the number of layers. This cost grows as L grows. However, we actually expect this cost to be $O(L)$ in average, as when the number of layers M is big, most invalidated hulls were not built and will never be used.

The following observation motivates our expectation. If the problem is encoded sufficiently well, new individuals mostly appear in smaller layers. Thus, a typical flow of a solution is from smaller layers to larger ones, and the migration in the reverse direction can occur for only a constant number of layers. As hulls are only build at the last layer, most alive hulls reside in a constant number of last layers. Unfortunately, we do not yet have a proof for this.

The complexity of the worst point query is $O(\frac{N_{\text{last}}}{L} \log L)$, and it decreases as L grows. The most natural way to achieve a trade-off, in our opinion, is to equate the running times of the two phases by choosing an appropriate L: $O(L) = O(\frac{N_{\text{last}}}{L} \log L)$. In the algorithm we track $N_{\max}$, the maximum seen value of N_{last}, and dynamically adjust L to be the maximum value which satisfies $L^2 / \log L \le N_{\max}$.

Table 1: Experiment results. For each benchmark problem, the table has two rows. For each combination of computational budget and generation size, the first row presents the running times, in seconds, of the basic INDS algorithm and of the proposed convex hull based algorithm. The second row presents the hypervolume at the end of the run and the ratio of the running time of INDS to the time of the hull based algorithm. Light gray cells correspond to situations when running times were similar (interquartile ranges overlap), dark gray cells signify that the hull based algorithm was faster, white cells signify that INDS was faster.

Prob.	Budget: 25000				Budget: 250000				Budget: 2500000			
	Gen. size: 100		Gen. size: 1000		Gen. size: 10000		Gen. size: 100000					
	INDS/HV	Hull/Ratio	INDS/HV	Hull/Ratio	INDS/HV	Hull/Ratio	INDS/HV	Hull/Ratio	INDS/HV	Hull/Ratio	INDS/HV	Hull/Ratio
DTLZ1	$1.01 \cdot 10^{-1}$	$1.19 \cdot 10^{-1}$	$1.80 \cdot 10^{-1}$	$2.23 \cdot 10^{-1}$	$3.27 \cdot 10^{0}$	$1.74 \cdot 10^{0}$	$2.82 \cdot 10^{0}$	$3.09 \cdot 10^{0}$	$4.03 \cdot 10^{2}$	$2.19 \cdot 10^{1}$	$9.29 \cdot 10^{1}$	$1.12 \cdot 10^{2}$
	$4.92 \cdot 10^{-1}$	$8.53 \cdot 10^{-1}$	$0.00 \cdot 10^{0}$	$8.08 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$1.88 \cdot 10^{0}$	$3.29 \cdot 10^{-3}$	$9.12 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$1.84 \cdot 10^{1}$	$1.60 \cdot 10^{-1}$	$8.33 \cdot 10^{-1}$
DTLZ2	$1.03 \cdot 10^{-1}$	$2.85 \cdot 10^{-1}$	$2.23 \cdot 10^{-1}$	$2.94 \cdot 10^{-1}$	$4.32 \cdot 10^{0}$	$1.45 \cdot 10^{0}$	$6.23 \cdot 10^{0}$	$3.91 \cdot 10^{0}$	$4.56 \cdot 10^{2}$	$2.49 \cdot 10^{1}$	$2.95 \cdot 10^{2}$	$1.17 \cdot 10^{2}$
	$2.11 \cdot 10^{-1}$	$3.62 \cdot 10^{-1}$	$2.13 \cdot 10^{-1}$	$7.58 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$2.99 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$1.59 \cdot 10^{0}$	$2.15 \cdot 10^{-1}$	$1.83 \cdot 10^{1}$	$2.15 \cdot 10^{-1}$	$2.51 \cdot 10^{0}$
DTLZ3	$9.40 \cdot 10^{-2}$	$1.18 \cdot 10^{-1}$	$1.65 \cdot 10^{-1}$	$2.75 \cdot 10^{-1}$	$2.20 \cdot 10^{0}$	$1.43 \cdot 10^{0}$	$3.13 \cdot 10^{0}$	$3.28 \cdot 10^{0}$	$2.60 \cdot 10^{2}$	$2.50 \cdot 10^{1}$	$1.02 \cdot 10^{2}$	$1.18 \cdot 10^{2}$
	$1.51 \cdot 10^{-1}$	$7.99 \cdot 10^{-1}$	$0.00 \cdot 10^{0}$	$6.01 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$1.54 \cdot 10^{0}$	$0.00 \cdot 10^{0}$	$9.54 \cdot 10^{-1}$	$2.15 \cdot 10^{-1}$	$1.04 \cdot 10^{1}$	$0.00 \cdot 10^{0}$	$8.66 \cdot 10^{-1}$
DTLZ4	$1.45 \cdot 10^{-1}$	$1.88 \cdot 10^{-1}$	$2.26 \cdot 10^{-1}$	$2.95 \cdot 10^{-1}$	$4.20 \cdot 10^{0}$	$1.70 \cdot 10^{0}$	$5.46 \cdot 10^{0}$	$3.72 \cdot 10^{0}$	$4.37 \cdot 10^{2}$	$2.31 \cdot 10^{1}$	$2.21 \cdot 10^{2}$	$1.05 \cdot 10^{2}$
	$2.11 \cdot 10^{-1}$	$7.74 \cdot 10^{-1}$	$2.13 \cdot 10^{-1}$	$7.64 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$2.47 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$1.47 \cdot 10^{0}$	$2.15 \cdot 10^{-1}$	$1.89 \cdot 10^{1}$	$2.15 \cdot 10^{-1}$	$2.10 \cdot 10^{0}$
DTLZ5	$1.39 \cdot 10^{-1}$	$2.15 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$	$2.86 \cdot 10^{-1}$	$4.32 \cdot 10^{0}$	$1.96 \cdot 10^{0}$	$6.16 \cdot 10^{0}$	$3.89 \cdot 10^{0}$	$4.55 \cdot 10^{2}$	$2.33 \cdot 10^{1}$	$2.87 \cdot 10^{2}$	$1.18 \cdot 10^{2}$
	$2.11 \cdot 10^{-1}$	$6.45 \cdot 10^{-1}$	$2.13 \cdot 10^{-1}$	$7.92 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$2.20 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$1.58 \cdot 10^{0}$	$2.15 \cdot 10^{-1}$	$1.96 \cdot 10^{1}$	$2.15 \cdot 10^{-1}$	$2.44 \cdot 10^{0}$
DTLZ6	$7.91 \cdot 10^{-2}$	$2.21 \cdot 10^{-1}$	$2.02 \cdot 10^{-1}$	$2.85 \cdot 10^{-1}$	$5.76 \cdot 10^{0}$	$1.71 \cdot 10^{0}$	$4.14 \cdot 10^{0}$	$4.71 \cdot 10^{0}$	$1.15 \cdot 10^{3}$	$5.43 \cdot 10^{1}$	$1.37 \cdot 10^{2}$	$1.61 \cdot 10^{2}$
	$2.11 \cdot 10^{-1}$	$3.58 \cdot 10^{-1}$	$0.00 \cdot 10^{0}$	$7.09 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$3.37 \cdot 10^{0}$	$0.00 \cdot 10^{0}$	$8.79 \cdot 10^{-1}$	$2.15 \cdot 10^{-1}$	$2.11 \cdot 10^{1}$	$0.00 \cdot 10^{0}$	$8.50 \cdot 10^{-1}$
DTLZ7	$1.05 \cdot 10^{-1}$	$1.69 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$	$3.18 \cdot 10^{-1}$	$4.54 \cdot 10^{0}$	$1.92 \cdot 10^{0}$	$4.75 \cdot 10^{0}$	$5.18 \cdot 10^{0}$	$5.05 \cdot 10^{2}$	$3.03 \cdot 10^{1}$	$1.62 \cdot 10^{2}$	$1.78 \cdot 10^{2}$
	$3.09 \cdot 10^{-1}$	$6.22 \cdot 10^{-1}$	$2.94 \cdot 10^{-1}$	$7.13 \cdot 10^{-1}$	$3.09 \cdot 10^{-1}$	$2.36 \cdot 10^{0}$	$3.02 \cdot 10^{-1}$	$9.16 \cdot 10^{-1}$	$3.09 \cdot 10^{-1}$	$1.67 \cdot 10^{1}$	$3.04 \cdot 10^{-1}$	$9.14 \cdot 10^{-1}$
WFG1	$1.28 \cdot 10^{-1}$	$2.69 \cdot 10^{-1}$	$2.01 \cdot 10^{-1}$	$2.44 \cdot 10^{-1}$	$4.94 \cdot 10^{0}$	$1.98 \cdot 10^{0}$	$3.83 \cdot 10^{0}$	$3.76 \cdot 10^{0}$	$8.45 \cdot 10^{2}$	$3.77 \cdot 10^{1}$	$1.28 \cdot 10^{2}$	$1.39 \cdot 10^{2}$
	$2.16 \cdot 10^{-1}$	$4.75 \cdot 10^{-1}$	$4.72 \cdot 10^{-2}$	$8.22 \cdot 10^{-1}$	$4.61 \cdot 10^{-1}$	$2.50 \cdot 10^{0}$	$8.90 \cdot 10^{-2}$	$1.02 \cdot 10^{0}$	$4.66 \cdot 10^{-1}$	$2.24 \cdot 10^{1}$	$1.53 \cdot 10^{-1}$	$9.18 \cdot 10^{-1}$
WFG2	$1.79 \cdot 10^{-1}$	$2.54 \cdot 10^{-1}$	$6.44 \cdot 10^{-1}$	$2.52 \cdot 10^{-1}$	$9.88 \cdot 10^{0}$	$3.03 \cdot 10^{0}$	$7.86 \cdot 10^{1}$	$4.68 \cdot 10^{0}$	$1.52 \cdot 10^{3}$	$3.71 \cdot 10^{1}$	$3.01 \cdot 10^{4}$	$1.64 \cdot 10^{2}$
	$5.58 \cdot 10^{-1}$	$7.06 \cdot 10^{-1}$	$5.59 \cdot 10^{-1}$	$2.56 \cdot 10^{0}$	$5.59 \cdot 10^{-1}$	$3.26 \cdot 10^{0}$	$5.59 \cdot 10^{-1}$	$1.68 \cdot 10^{1}$	$5.59 \cdot 10^{-1}$	$4.09 \cdot 10^{1}$	$5.59 \cdot 10^{-1}$	$1.83 \cdot 10^{2}$
WFG3	$2.28 \cdot 10^{-1}$	$2.51 \cdot 10^{-1}$	$2.23 \cdot 10^{-1}$	$2.01 \cdot 10^{-1}$	$5.82 \cdot 10^{0}$	$2.03 \cdot 10^{0}$	$5.93 \cdot 10^{0}$	$4.39 \cdot 10^{0}$	$8.30 \cdot 10^{2}$	$3.06 \cdot 10^{1}$	$2.94 \cdot 10^{2}$	$1.20 \cdot 10^{2}$
	$4.42 \cdot 10^{-1}$	$9.07 \cdot 10^{-1}$	$4.43 \cdot 10^{-1}$	$1.11 \cdot 10^{0}$	$4.44 \cdot 10^{-1}$	$2.86 \cdot 10^{0}$	$4.44 \cdot 10^{-1}$	$1.35 \cdot 10^{0}$	$4.44 \cdot 10^{-1}$	$2.71 \cdot 10^{1}$	$4.44 \cdot 10^{-1}$	$2.45 \cdot 10^{0}$
WFG4	$1.27 \cdot 10^{-1}$	$2.63 \cdot 10^{-1}$	$2.06 \cdot 10^{-1}$	$2.55 \cdot 10^{-1}$	$5.76 \cdot 10^{0}$	$1.95 \cdot 10^{0}$	$5.22 \cdot 10^{0}$	$3.62 \cdot 10^{0}$	$8.65 \cdot 10^{2}$	$3.28 \cdot 10^{1}$	$2.38 \cdot 10^{2}$	$9.37 \cdot 10^{1}$
	$2.11 \cdot 10^{-1}$	$4.83 \cdot 10^{-1}$	$2.13 \cdot 10^{-1}$	$8.07 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$2.95 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$1.44 \cdot 10^{0}$	$2.15 \cdot 10^{-1}$	$2.64 \cdot 10^{1}$	$2.15 \cdot 10^{-1}$	$2.54 \cdot 10^{0}$
WFG5	$1.80 \cdot 10^{-1}$	$4.60 \cdot 10^{-1}$	$3.41 \cdot 10^{-1}$	$2.77 \cdot 10^{-1}$	$8.58 \cdot 10^{0}$	$3.08 \cdot 10^{0}$	$2.42 \cdot 10^{1}$	$3.60 \cdot 10^{0}$	$1.26 \cdot 10^{3}$	$4.25 \cdot 10^{1}$	$8.32 \cdot 10^{2}$	$1.04 \cdot 10^{2}$
	$1.79 \cdot 10^{-1}$	$3.92 \cdot 10^{-1}$	$1.82 \cdot 10^{-1}$	$1.23 \cdot 10^{0}$	$1.82 \cdot 10^{-1}$	$2.79 \cdot 10^{0}$	$1.82 \cdot 10^{-1}$	$6.73 \cdot 10^{0}$	$1.82 \cdot 10^{-1}$	$2.97 \cdot 10^{1}$	$1.88 \cdot 10^{-1}$	$7.97 \cdot 10^{0}$
WFG6	$1.49 \cdot 10^{-1}$	$3.55 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$	$3.20 \cdot 10^{-1}$	$5.02 \cdot 10^{0}$	$2.26 \cdot 10^{0}$	$4.50 \cdot 10^{0}$	$3.61 \cdot 10^{0}$	$5.65 \cdot 10^{2}$	$3.31 \cdot 10^{1}$	$1.61 \cdot 10^{2}$	$1.14 \cdot 10^{2}$
	$1.91 \cdot 10^{-1}$	$4.19 \cdot 10^{-1}$	$2.09 \cdot 10^{-1}$	$6.40 \cdot 10^{-1}$	$2.13 \cdot 10^{-1}$	$2.22 \cdot 10^{0}$	$2.12 \cdot 10^{-1}$	$1.24 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$1.71 \cdot 10^{1}$	$2.13 \cdot 10^{-1}$	$1.41 \cdot 10^{0}$
WFG7	$1.71 \cdot 10^{-1}$	$3.30 \cdot 10^{-1}$	$2.61 \cdot 10^{-1}$	$2.43 \cdot 10^{-1}$	$6.45 \cdot 10^{0}$	$2.65 \cdot 10^{0}$	$8.33 \cdot 10^{0}$	$3.26 \cdot 10^{0}$	$9.87 \cdot 10^{2}$	$3.38 \cdot 10^{1}$	$7.99 \cdot 10^{2}$	$1.13 \cdot 10^{2}$
	$2.11 \cdot 10^{-1}$	$5.18 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$1.07 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$2.44 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$2.56 \cdot 10^{0}$	$2.15 \cdot 10^{-1}$	$2.93 \cdot 10^{1}$	$2.15 \cdot 10^{-1}$	$7.05 \cdot 10^{0}$
WFG8	$6.61 \cdot 10^{-2}$	$1.45 \cdot 10^{-1}$	$1.44 \cdot 10^{-1}$	$1.93 \cdot 10^{-1}$	$1.25 \cdot 10^{0}$	$1.11 \cdot 10^{0}$	$2.71 \cdot 10^{0}$	$2.84 \cdot 10^{0}$	$3.86 \cdot 10^{1}$	$1.63 \cdot 10^{1}$	$8.86 \cdot 10^{1}$	$8.60 \cdot 10^{1}$
	$1.49 \cdot 10^{-1}$	$4.56 \cdot 10^{-1}$	$1.45 \cdot 10^{-1}$	$7.45 \cdot 10^{-1}$	$1.55 \cdot 10^{-1}$	$1.13 \cdot 10^{0}$	$1.74 \cdot 10^{-1}$	$9.56 \cdot 10^{-1}$	$2.10 \cdot 10^{-1}$	$2.36 \cdot 10^{0}$	$1.93 \cdot 10^{-1}$	$1.03 \cdot 10^{0}$
WFG9	$2.05 \cdot 10^{-1}$	$2.19 \cdot 10^{-1}$	$1.95 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$	$5.70 \cdot 10^{0}$	$1.87 \cdot 10^{0}$	$4.28 \cdot 10^{0}$	$2.87 \cdot 10^{0}$	$9.39 \cdot 10^{2}$	$3.16 \cdot 10^{1}$	$1.42 \cdot 10^{2}$	$9.06 \cdot 10^{1}$
	$2.09 \cdot 10^{-1}$	$9.36 \cdot 10^{-1}$	$2.11 \cdot 10^{-1}$	$8.57 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$	$3.05 \cdot 10^{0}$	$2.13 \cdot 10^{-1}$	$1.49 \cdot 10^{0}$	$2.14 \cdot 10^{-1}$	$2.97 \cdot 10^{1}$	$2.14 \cdot 10^{-1}$	$1.56 \cdot 10^{0}$
ZDT1	$1.01 \cdot 10^{-1}$	$1.47 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$4.07 \cdot 10^{0}$	$2.02 \cdot 10^{0}$	$4.61 \cdot 10^{0}$	$4.87 \cdot 10^{0}$	$4.78 \cdot 10^{2}$	$3.19 \cdot 10^{1}$	$1.69 \cdot 10^{2}$	$1.72 \cdot 10^{2}$
	$6.62 \cdot 10^{-1}$	$6.91 \cdot 10^{-1}$	$6.13 \cdot 10^{-1}$	$6.65 \cdot 10^{-1}$	$6.66 \cdot 10^{-1}$	$2.02 \cdot 10^{0}$	$6.41 \cdot 10^{-1}$	$9.47 \cdot 10^{-1}$	$6.67 \cdot 10^{-1}$	$1.50 \cdot 10^{1}$	$6.47 \cdot 10^{-1}$	$9.83 \cdot 10^{-1}$
ZDT2	$1.17 \cdot 10^{-1}$	$1.41 \cdot 10^{-1}$	$2.15 \cdot 10^{-1}$	$2.72 \cdot 10^{-1}$	$4.44 \cdot 10^{0}$	$1.82 \cdot 10^{0}$	$4.79 \cdot 10^{0}$	$5.19 \cdot 10^{0}$	$4.36 \cdot 10^{2}$	$3.01 \cdot 10^{1}$	$1.74 \cdot 10^{2}$	$1.96 \cdot 10^{2}$
	$3.28 \cdot 10^{-1}$	$8.31 \cdot 10^{-1}$	$2.19 \cdot 10^{-1}$	$7.92 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$2.44 \cdot 10^{0}$	$2.72 \cdot 10^{-1}$	$9.23 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$1.45 \cdot 10^{1}$	$2.82 \cdot 10^{-1}$	$8.88 \cdot 10^{-1}$
ZDT3	$1.09 \cdot 10^{-1}$	$1.39 \cdot 10^{-1}$	$2.15 \cdot 10^{-1}$	$2.91 \cdot 10^{-1}$	$4.10 \cdot 10^{0}$	$1.83 \cdot 10^{0}$	$4.95 \cdot 10^{0}$	$4.94 \cdot 10^{0}$	$3.82 \cdot 10^{2}$	$2.82 \cdot 10^{1}$	$1.62 \cdot 10^{2}$	$1.64 \cdot 10^{2}$
	$5.16 \cdot 10^{-1}$	$7.84 \cdot 10^{-1}$	$4.78 \cdot 10^{-1}$	$7.39 \cdot 10^{-1}$	$5.17 \cdot 10^{-1}$	$2.23 \cdot 10^{0}$	$4.98 \cdot 10^{-1}$	$1.00 \cdot 10^{0}$	$5.17 \cdot 10^{-1}$	$1.36 \cdot 10^{1}$	$5.03 \cdot 10^{-1}$	$9.90 \cdot 10^{-1}$
ZDT4	$8.56 \cdot 10^{-2}$	$1.29 \cdot 10^{-1}$	$1.83 \cdot 10^{-1}$	$2.57 \cdot 10^{-1}$	$2.82 \cdot 10^{0}$	$1.20 \cdot 10^{0}$	$3.64 \cdot 10^{0}$	$3.82 \cdot 10^{0}$	$3.16 \cdot 10^{2}$	$2.22 \cdot 10^{1}$	$1.30 \cdot 10^{2}$	$1.51 \cdot 10^{2}$
	$6.59 \cdot 10^{-1}$	$6.65 \cdot 10^{-1}$	$0.00 \cdot 10^{0}$	$7.14 \cdot 10^{-1}$	$6.66 \cdot 10^{-1}$	$2.35 \cdot 10^{0}$	$0.00 \cdot 10^{0}$	$9.53 \cdot 10^{-1}$	$6.67 \cdot 10^{-1}$	$1.42 \cdot 10^{1}$	$0.00 \cdot 10^{0}$	$8.62 \cdot 10^{-1}$
ZDT6	$1.35 \cdot 10^{-1}$	$1.97 \cdot 10^{-1}$	$2.30 \cdot 10^{-1}$	$3.21 \cdot 10^{-1}$	$3.90 \cdot 10^{0}$	$2.75 \cdot 10^{0}$	$4.39 \cdot 10^{0}$	$4.49 \cdot 10^{0}$	$7.52 \cdot 10^{2}$	$3.09 \cdot 10^{1}$	$1.62 \cdot 10^{2}$	$1.91 \cdot 10^{2}$
	$3.97 \cdot 10^{-1}$	$6.85 \cdot 10^{-1}$	$0.00 \cdot 10^{0}$	$7.15 \cdot 10^{-1}$	$4.06 \cdot 10^{-1}$	$1.42 \cdot 10^{0}$	$0.00 \cdot 10^{0}$	$9.78 \cdot 10^{-1}$	$4.06 \cdot 10^{-1}$	$2.44 \cdot 10^{1}$	$0.00 \cdot 10^{0}$	$8.47 \cdot 10^{-1}$

4. EXPERIMENTS

To compare the performance of the proposed algorithm to that of the basic INDS algorithm, we ran a series of experiments similar to the ones in [4], which, in turn, derive from the experiments in [17]. Namely, we use several benchmark problems: DTLZ1–DTLZ7 [8], WFG1–WFG9 [11], and ZDT1–ZDT6 except for ZDT5 [23]. The parameters of these problems are the same as in the corresponding papers, with the one exception: evaluation of fitness functions in WFG problems is done using double-precision floating-point numbers, which influences properties of some of these functions significantly. We considered only two-dimensional versions of these problems.

We use the steady-state version of NSGA-II, described in [4], to optimize the problems, but instead of fast non-dominated sorting we use the INDS algorithm and the proposed hull-based modification. To analyze the performance of the algorithms depending on various generation sizes and various convergence, we considered computational budgets from the set $\{2.5 \cdot 10^4, 2.5 \cdot 10^5, 2.5 \cdot 10^6\}$ individual evaluations and for each budget T two values for generation size: $\{T/250, T/25\}$. These values were chosen based on a somewhat default setting (budget 25000, generation size 100) and were further scaled up while maintaining the average number of improvements per a single individual. We compensated the time spend by NSGA-II in fitness evaluation and genetic operations by performing the same number of these operations, measuring the running time and subtracting it from the overall running time of the full-blown algorithms.

For each problem, each budget/size combination and each algorithm, 25 runs were performed. The median results are presented in Table 1 along with the final hypervolume values.

4.1 Result Analysis

The first thing to note is that the proposed hull-based modification never loses too much. The worst seen case is the DTLZ6 problem, budget 25000, generation size 100, where INDS runs for $7.91 \cdot 10^{-2}$ seconds and the hull-based algorithm for $2.21 \cdot 10^{-1}$ seconds, which is approximately 2.8 times slower. For budget 25000 and generation size 100, the average ratio of the runtime of the hull algorithm to that of INDS is 1.72 and the median one is 1.5. For generation size 1000, the corresponding numbers are 1.24 and 1.31. From these observations we can conclude that the proposed algorithm can rarely be slower than INDS for more than two times.

As computational budget increases, the hull-based algorithm starts to become better. For budgets $2.5 \cdot 10^5$ and $2.5 \cdot 10^6$, it is never worse significantly (except for DTLZ1, budget $2.5 \cdot 10^6$ and generation size 10^5) and is often statistically better. For nearly all runs with budget $2.5 \cdot 10^5$ and generation size 1000 the hull-based algorithm is 1.5–2 times faster than INDS. This trend continues for budget $2.5 \cdot 10^6$ and generation size 10000, where the hull-based algorithm is 15–30 times faster. In one especially noticeable case (WFG2, budget $2.5 \cdot 10^6$ and generation size 10^5) the hull-based algorithm is 180 times (!) faster than INDS. Coincidentally, this was the case of the largest running time of INDS among all tested cases.

A brief look at Table 1 reveals, among two already discussed features (the hull-based algorithm is generally faster for two combinations of budget and generation size), that certain problems seem to give advantage to the hull algorithm while others do not. However, there are exceptions to this rule as well: for example, DTLZ1 seems to be a "simple" problem for the hull algorithm, but the $2.5 \cdot 10^6/10^5$ configuration is different from this trend. This suggests that neither configuration parameters nor the problem are good predictors to the running time of the hull algorithm (related to the running time of INDS).

Our current hypothesis is that the hypervolume – more precisely, how close it is to the theoretically maximum hypervolume of the particular problem – better predicts whether the hull algorithm will be faster than INDS. We did not check this hypothesis statistically, because, in our opinion, more data should be collected, including the running times of the algorithms and the hypervolumes at various computational budgets spend to optimization. However, basic validation can be performed using the existing data.

One can see that the most stable configuration which makes the hull algorithm performs better (budget $2.5 \cdot 10^6$ and generation size 10000) features maximum hypervolume values for each problem among all other configurations (except for WFG5). The failed case for DTLZ1 at budget $2.5 \cdot 10^6$ and generation size 10^5 features small hypervolume (0.16 compared to the maximum of 0.5). The problems that are "simple" to the hull algorithm typically reach the near-to-maximum hypervolume value quite fast (see DTLZ4, WFG2 and WFG3). The difficulty of WFG2 for budget $2.5 \cdot 10^6$ and generation size 10^5 to INDS may be attributed to reaching the maximum hypervolume in the very beginning of optimization. This may result in only one non-dominated layer existing most of the time without even temporal appearance of the second layer, which makes INDS check all the points to find the one with the smallest crowding distance on every iteration.

If the hypervolume is nearly maximal, the next order trends may arise. If one checks configurations of the same budget-to-generation-size ratio for WFG2, WFG3 and possibly some other problems, one can see that the relative efficiency of the hull algorithm grows as the generation size grows. This illustrates the asymptotic difference in the running times of worst point search in INDS and in the hull algorithm for $O(1)$ layers – namely, $\Theta(N)$ for INDS versus $o(\sqrt{N})$ for the hull algorithm.

4.2 A Note to Possible Users on Reproducibility and Correctness

The source code of the algorithm presented in the paper, along with experimental setup, is available at GitHub [1] along with the algorithms from earlier GECCO-2015 papers.

When one runs the experiments for this paper, one may see that the reported hypervolumes differ for the basic INDS algorithm and the hull algorithm (unlike all other experiments from the same codebase), however, most of times this difference is not significant. This is explained by the fact that, if several points from the last layer have equal crowding distance, a random one should be chosen, and for all algorithms this random choice should be exactly the same throughout the entire experimentation. While this was possible to achieve for all algorithms except for the hull-based one, it was found out to be difficult for the latter.

The key is that an implementation which strictly follows the rules requires to collect all points with the given crowding distance, ordered lexicographically according to the objectives, and to choose a random one from them. To achieve this when using convex hull algorithms, one needs to ensure that:

- all points which produce equal derivative points should be stored together (instead of a random one being selected), which requires storing a list of points where otherwise a single point would be enough;
- an extreme line may touch not one point of the convex hull, but its edge, which, apart from two points of the hull, would require keeping all points belonging to the edges of the hull.

In our opinion, handling of all these requirements would not only complicate the implementation, but reduce its overall speed by a constant factor. On the other hand, most of these conditions are quite rare, except for having a layer of size two, which is handled separately. We decided to keep our implementation fast and less complicated at the price of not repeating the results of other algorithms precisely. We do believe, however, that the distribution of results produced by our algorithm is the same as the ones for other algorithms.

5. CONCLUSION

We presented a modification of the algorithm for incremental non-dominated sorting which speeds up finding a point with the smallest crowding distance on the last front. The idea of this modification is based on reduction to finding an extreme point in the convex hull. Certain work has been done on understanding when to build convex hulls and

[1] https://github.com/mbuzdalov/papers/tree/convex/2015-gecco-nsga-ii-ss

when not to build, resulting in an algorithm lazily constructing convex hulls up to size L, where $L^2 \log L \approx N_{\max}$ and $N_{\max}$ is the maximum observed size of the last layer.

Experimental evaluation shows that the proposed modification is slightly (approximately 1.5–3 times) slower for somewhat standard combinations of computational budget and generation size, but quickly overtakes for larger budgets and generation sizes. While the actual dependency of the speedup on computational budget, generation size and problem features is complicated, one of the most noticeable trends is that the closer the population is to the Pareto optimal set, the bigger is the speedup. Most probably, this makes the last layer size become $\Theta(N)$ in size, which slows down the original INDS algorithm, but the proposed modification does not suffer from this. This observation, especially if validated statistically, hints that the convex hull based modification should be turned on not from the very beginning, but at the last stages of optimization, where it would definitely save time.

There are a couple of known shortcomings of this work. First of all, it is really limited to two dimensions: first, because the base INDS algorithm has not been yet developed to more than two dimensions, and second, because of the simplicity of crowding distance definition, in terms of neighborhood in layers, when the layer itself is one-dimensional. Second, the approach is tailored to the definition of the crowding distance, and is not seen to be easily extendable to other commonly used measures. These two shortcomings may be seen to compensate each other: for two dimensions, NSGA-II is good enough, while many-objective evolutionary algorithms do not use crowding distance, so it is not extremely necessary to seek the way of fast search of the worst crowding distance point in higher dimensions and steady-state setup. However, our algorithm opens a way for much larger generation sizes, which may reintroduce selection pressure for higher dimensions, and thus switch on the necessity of working with crowding distance again.

The real aim of this paper is to demonstrate that advanced algorithmic techniques from classic computer science, such as computation geometry, can really help speeding up various parts of algorithms in evolutionary computation. Although the presented convex hull based modification may hardly be seen in any practitioner's code in probably next five years, it may inspire researchers in evolutionary computation to look at their algorithm implementations from a different point of view, and to improve them, making evolutionary computation more appealing to industry practitioners worldwide.

Acknowledgments

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

6. REFERENCES

[1] S. Adra and P. Fleming. Diversity management in evolutionary many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 15(2):183–195, 2011.

[2] M. Buzdalov and V. Parfenov. Various degrees of steadiness in NSGA-II and their influence on the quality of results. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 749–750, 2015.

[3] M. Buzdalov and A. Shalyto. 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, pages 528–537. Springer, 2014.

[4] M. Buzdalov, I. Yakupov, and A. Stankevich. 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*, pages 647–654, 2015.

[5] C. A. Coello Coello and M. Lechuga. MOPSO: A proposal for multiple objective particle swarm optimization. In *Proceedings of Congress on Evolutionary Computation*, pages 1051–1056, 2002.

[6] K. Deb and H. Jain. 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):577–601, 2013.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[8] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable test problems for evolutionary multiobjective optimization. In *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*, pages 105–145. Springer, 2005.

[9] F.-A. Fortin, S. Grenier, and M. Parizeau. Generalizing the improved run-time complexity algorithm for non-dominated sorting. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 615–622. ACM, 2013.

[10] S. Gupta and G. Tan. A scalable parallel implementation of evolutionary algorithms for multi-objective optimization on GPUs. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 1567–1574, 2015.

[11] S. Huband, P. Hingston, L. Barone, and R. L. While. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation*, 10(5):477–506, 2006.

[12] M. T. Jensen. Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *Transactions on Evolutionary Computation*, 7(5):503–515, 2003.

[13] K. Li, K. Deb, Q. Zhang, and S. Kwong. Efficient non-domination level update approach for steady-state evolutionary multiobjective optimization. Technical Report COIN 2014014, Michigan State University, 2014.

[14] K. Li, K. Deb, Q. Zhang, and Q. Zhang. Efficient non-domination level update method for steady-state evolutionary multi-objective optimization. Technical Report COIN 2015022, Michigan State University, 2015.

[15] K. McClymont and E. Keedwell. Deductive sort and climbing sort: New methods for non-dominated sorting. *Evolutionary computation*, 20(1):1–26, 2012.

[16] S. Mishra, S. Mondal, and S. Saha. Improved solution to the non-domination level update problem. http://arxiv.org/abs/1510.04796.

[17] A. J. Nebro and J. J. Durillo. 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, pages 435–456. Springer Berlin Heidelberg, 2009.

[18] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer New York, 1985.

[19] J. O'Rourke. *Computational Geometry in C, 2nd Edition*. Cambridge Tracts in Theoretical Computer Science, 1998.

[20] H. Wang and X. Yao. Corner sort for pareto-based many-objective optimization. *IEEE Transactions on Cybernetics*, 44(1):92–102, 2014.

[21] I. Yakupov and M. Buzdalov. Incremental non-dominated sorting with $O(N)$ insertion for the two-dimensional case. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 1853–1860, 2015.

[22] X. Zhang, Y. Tian, R. Cheng, and Y. Jin. An efficient approach to nondominated sorting for evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 19(2):201–213, 2015.

[23] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, 2000.