Fast Implementation of the Steady-State NSGA-II Algorithm for Two Dimensions Based on Incremental Non-Dominated Sorting

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

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

Andrey Stankevich
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
stankev@gmail.com

ABSTRACT

Genetic algorithms (GAs) are widely used in multi-objective optimization for solving complex problems. There are two distinct approaches for GA design: generational and steady-state algorithms. Most of the current state-of-the-art GAs are generational, although there is an increasing interest to steady-state algorithms as well. However, for algorithms based on non-dominated sorting, most of steady-state implementations have higher computation complexity than their generational counterparts, which limits their applicability.

We present a fast implementation of a steady-state version of the NSGA-II algorithm for two dimensions. This implementation is based on a data structure which has $O(N)$ complexity for single solution insertion and deletion in the worst case. The experimental results show that our implementation works noticeably faster than steady-state NSGA-II implementations which use fast non-dominated sorting.

CCS Concepts

•**Theory of computation** → **Mathematical optimization;** *Data structures design and analysis;*

Keywords

steady-state, multi-objective, NSGA-II, non-dominated sorting, incremental updates

1. INTRODUCTION

In the K-dimensional space, a point $A = (a_1, \ldots, a_K)$ is said to *dominate* a point $B = (b_1, \ldots, b_K)$ 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 $i-1$ are marked with the rank of i.

Many well-known and widely used multi-objective evolutionary algorithms use the procedure of non-dominated sorting, or the procedure of determining the non-dominated solutions, which can be reduced to non-dominated sorting. These algorithms include NSGA-II [5], PESA [4], PESA-II [3], SPEA2 [18], PAES [10], PDE [1], and many more. The time complexity of a single iteration of these algorithms is often dominated by the complexity of a non-dominated sorting algorithm, so optimization of the latter makes such multi-objective evolutionary algorithms faster.

In Kung et al [11], the algorithm for determining the non-dominated solutions is proposed with the running time complexity of $O(N \log^{K-1} N)$, where N is the number of points and K is the dimension of the space. It is possible to use this algorithm to perform non-dominated sorting: first, the non-dominated solutions are found and assigned the rank of 0. Then, these solutions are removed, the non-dominated solutions from the remaining ones are found and assigned the rank of 1. The process repeats until all the solutions are removed. This leads to the complexity of $O(N^2 \log^{K-1} N)$ in the worst case, if the maximum rank of a point in the result is $O(N)$.

Jensen [9] was the first to propose an algorithm for non-dominated sorting with the complexity of $O(N \log^{K-1} N)$. However, his algorithm was developed for the assumption that no two points share a common value for any objective, and the complexity was proven for the same assumption. The first attempt to fix this issue belongs, to the best of the authors' knowledge, to Fortin et al [7]. The corrected (or, as in [7], "generalized") algorithm works in all cases, and for the general case the performance is still $O(N \log^{K-1} N)$, but the only upper bound that was proven for the worst case is $O(N^2 K)$. Finally, Buzdalov et al in [2] proposed several modifications to the algorithm of Fortin et al to make the $O(N \log^{K-1} N)$ bound provable as well.

Evolutionary algorithms have a big advantage due to their great degree of parallelism, however, synchronous variants (which wait for evaluation of all individuals, then recompute their internal state) have only a limited applicability for distributed systems. Even on multicore computers an algorithm may have a poor performance if it spends big periods of time between fitness evaluations without using most of computer resources. To overcome these limitations, steady-state algorithms are developed, often with an intention to

GECCO '15 July 11-15, 2015, Madrid, Spain
© 2015 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3472-3/15/07.
DOI: http://dx.doi.org/10.1145/2739480.2754728

become asynchronous. Particularly, a steady-state version of the NSGA-II algorithm was developed [13] which showed good convergence rate and high quality of Pareto front approximation. However, the running time of it is poor.

It is possible to perform incremental non-dominated sorting by doing a complete non-dominated sorting from scratch every time an element is added. However, running times become very high: $O(KN^2)$ for a single insertion when the fast non-dominated sorting [5] is used, or $O(N \log^{K-1} N)$ when the sorting from [2] is used. Thus, it is needed to develop new algorithms and data structures to handle incremental non-dominated sorting efficiently.

However, almost no such algorithms and data structures have been developed so far. To our best knowledge, the only paper which addresses this issue is a technical report by Li et al. [12]. In that report, a procedure called "Efficient Non-domination Level Update" is introduced, which has the complexity of $O(NK\sqrt{N})$ for a single insertion when solutions are spread evenly over layers. This procedure was shown experimentally to be quite efficient, however, the worst-case complexity for a single insertion is still $O(N^2K)$.

This paper briefly describes the incremental non-dominated sorting algorithm for the two-dimensional case ($K = 2$) which has worst-case insertion and deletion complexity of $O(N)$ and worst-case query complexity of $O(\log N)$. It is described in our previous paper [16] in more detail. We additionally modified it to support evaluation of the *crowding distance*, a density measure employed by the NSGA-II algorithm. After that, the steady-state version of the NSGA-II algorithm is presented. Various implementation details are discussed, and results of experiments are presented.

2. INCREMENTAL NON-DOMINATED SORTING ALGORITHM

In this section, we briefly describe the proposed incremental non-dominated sorting algorithm and the data structure which supports it. They are described in detail in [16] along with necessary proofs.

The choice of the Cartesian tree as an underlying balanced search tree is discussed in Section 2.1. The data structure design is described in Section 2.2. The procedure of solution lookup (finding which layer a solution belongs to) is described in Section 2.3. The procedure of solution insertion is described in Section 2.4. The procedure of querying the k-th solution (in some predefined order) together with its layer number and crowding distance is described in Section 2.5. The worst solution deletion is described in Section 2.6.

When discussing the runtime analysis, we denote by N the total number of solutions stored in the data structure and by M the current number of non-domination layers. For the sake of brevity, non-domination layers are called just "layers" in the rest of the paper.

2.1 Cartesian Trees

To implement the algorithm, we need to have a data structure for container of elements which performs the following operations in $O(\log N)$:

- search of an element in the container;
- split of the container by key into two parts (the elements less than the key and the elements not less than the key);

```
 1: structure SOLUTION
 2:     – a solution to the optimization problem
 3:     $X$ : REAL – the first objective
 4:     $Y$ : REAL – the second objective
 5: end structure
 6: structure LLNODE
 7:     – a node of a low-level tree
 8:     $L$ : LLNODE – the left child
 9:     $R$ : LLNODE – the right child
10:     $V$ : SOLUTION – the node key
11:     $P$ : LLNODE – the previous-in-order node
12:     $N$ : LLNODE – the next-in-order node
13:     $S$ : INTEGER – the subtree size
14: end structure
15: structure HLNODE
16:     – a node of a high-level tree
17:     $L$ : HLNODE – the left child
18:     $R$ : HLNODE – the right child
19:     $P$ : HLNODE – the previous-in-order node
20:     $N$ : HLNODE – the next-in-order node
21:     $V$ : LLNODE – the node key
22:     $S$ : INTEGER – the subtree size
23:     $W$ : INTEGER – the sum of key sizes in the subtree
24: end structure
```

Figure 1: A pseudocode for the data structure

- merge of two containers C_1 and C_2 (every element from C_1 is not greater than every element from C_2).

We will call data structures which fulfil these requirements "split-merge balanced search trees". There are several such data structures, including Cartesian Tree [15] and Splay Tree [14]. In the case of Cartesian Tree, the $O(\log N)$ bound holds *with high probability*, while Splay Tree has *amortized* $O(\log N)$ bounds. From the mentioned data structures, Cartesian Tree generally performs slightly better in practice, so we use it in an implementation of our algorithm.

2.2 Data Structure

The idea of the data structure is to arrange layers in a binary search tree (each tree node corresponds to a layer) in the increasing order of their numbers. Each layer, in turn, is represented by a binary search tree itself, where solutions are sorted in the increasing order of their first objective. Since for two different solutions a and b from the same layer it holds that either $a_X > b_X$ and $a_Y < b_Y$ or $a_X < b_X$ and $a_Y > b_Y$, solutions in each layer are effectively sorted in the decreasing order of their second objective as well.

For the sake of brevity, we denote the tree of layers as the "high-level" tree and every tree containing layer elements as a "low-level" tree. The pseudocode for the data structure is given in Fig. 1. The resulting data structure is presented in Fig. 2.

Note that the high-level tree can be an ordinary balanced tree, while every low-level tree should be a split-merge tree. However, to evaluate the number of a certain layer in $O(\log M)$, one needs to store the number of tree elements in a subtree in each node of the high-level tree. Additionally, to move between adjacent layers in $O(1)$, nodes should be augmented with pointers to the previous-in-order and the next-in-order nodes (which can be done without affecting $O(\log N)$ performance of basic operations).

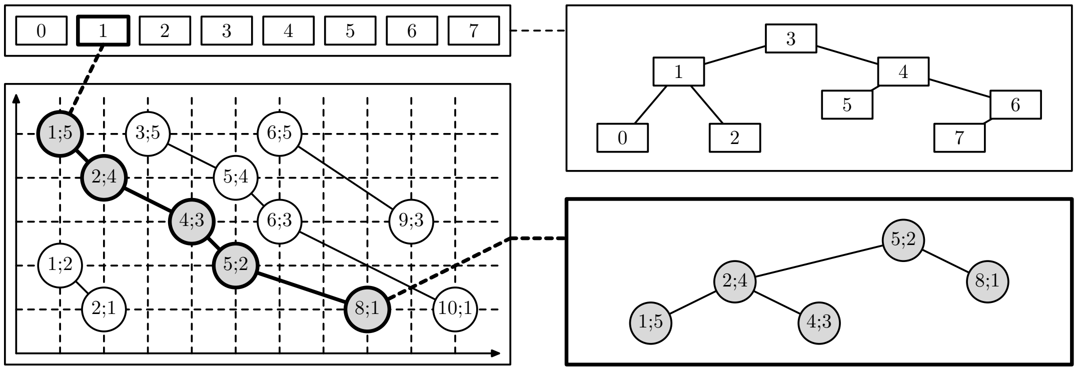

Figure 2: The data structure for the algorithm – the "tree of trees". Nodes of the "high-level" tree correspond to the layers. Each layer is, in turn, represented by a "low-level" tree, where nodes are sorted by the first objective. Note that layer numbers are not stored in nodes explicitly, they are just shown for convenience.

2.3 Lookup

Given a low-level tree T and a solution s, it is possible to find if s is dominated by at least one solution from T in $O(\log |T|)$. To do it, one needs to find a solution u from T such that $u_X \le s_X$ and u_X is maximum possible, which can be done by traversing the tree T from its root. If u is found and dominates s, then a dominating solution from T is found, otherwise, no solution from T dominates s.

Using this algorithm, one can traverse the high-level tree and find a layer with the minimum number which does not dominate a certain solution s. The algorithm is presented in Fig. 3.

A rough estimation of the running time is $O(\log M \log N)$, where $O(\log M)$ is an estimation of the height of the high-level tree, and $O(\log N)$ is an estimation of heights of all low-level trees.

However, one can perform a better estimation using the following idea. There are $k = O(\log M)$ layers which were tested for domination. Let their sizes be $L_1 \dots L_k$, and $L_1 + \dots + L_k \le N$. The running time for a layer of size L_i can be expressed as $O(1 + \log L_i)$ (we add extra 1 to handle a condition of $\log L_i = o(1)$). The total running time of a single lookup is:

$$O\left(k + \sum_{i=1}^{k} \log L_i\right).$$

Due to Cauchy's inequality, $\sum_{i=1}^{k} \log L_i \le k \log(N/k)$, which finally gives the following complexity of a lookup operation:

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

which, due to the fact that $M \le N$ and $\log(N/\log M)$ is $\omega(1)$, can be simplified to:

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

When N is fixed and M varies, this expression reaches its maximum at $M = \Theta(N)$, yielding $O((\log N)^2)$ worst-case running time.

```
1: function LowLevelDominates($T$, $s$)
2:     – returns whether any solution from $T$ dominates $s$
3:     $T$ : LLNode – the root node of the low-level tree
4:     $s$ : Solution – the solution to test for domination
5:     $B \leftarrow$ null – the best node so far
6:     while $T \ne$ null do
7:         if $T.V.X \le s.X$ then
8:             $B \leftarrow T$
9:             $T \leftarrow T.R$
10:        else
11:            $T \leftarrow T.L$
12:        end if
13:    end while
14:    if $B =$ null then
15:        return false
16:    end if
17:    return $B.Y < s.Y$ or $B.Y = s.Y$ and $B.X < s.X$
18: end function
19: function SmallestNonDominatingLayer($H$, $s$)
20:    – returns the layer with the smallest index from $H$
21:    – which does not dominate $s$
22:    $H$ : HLNode – the root node of the high-level tree
23:    $s$ : Solution – the solution to find a layer for
24:    $I \leftarrow 0$ – the number of dominating layers so far
25:    $B \leftarrow$ null – the best node so far
26:    while $H \ne$ null do
27:        if LowLevelDominates($H.V, s$) then
28:            $I \leftarrow I + H.S$
29:            $H \leftarrow H.R$
30:            if $H \ne$ null then
31:                $I \leftarrow I - H.S$
32:            end if
33:        else
34:            $B \leftarrow H$
35:            $H \leftarrow H.L$
36:        end if
37:    end while
38:    return $(B, I)$
39: end function
```

Figure 3: A pseudocode for determining the smallest layer which doesn't dominate the given solution

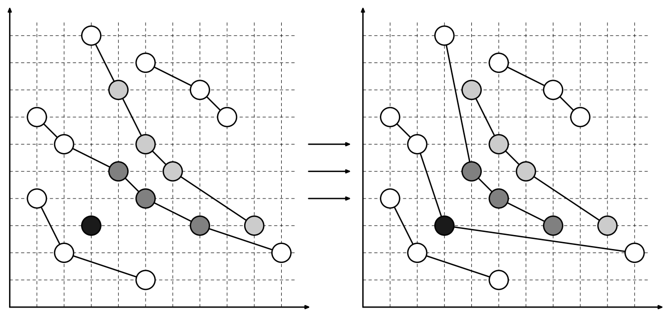

Figure 4: An example of the insertion process. Solutions which don't change their layer nodes (not numbers!) during the insertion process are white. A solution which is being inserted is black. Two clusters of solutions which together change their layer node are dark-gray and light-gray, correspondingly.

2.4 Insertion

Given a high-level tree H and a solution s, the insertion procedure updates H so that s is included in one of its low-level trees.

A key idea of fast implementation of insertion procedure is the fact that solutions who change their layers form contiguous pieces in their original layers and remain contiguous in their new layers as well. Fig. 4 illustrates an example of the insertion process.

The algorithm for a solution insertion is given in Fig. 5. It maintains a low-level tree which, at each stage, contains the solutions which needs to be inserted to the next layer. Initially it consists of the single solution which needs to be inserted. The layer to insert is initially found using the performing the SMALLESTNONDOMINATINGLAYER operation from Fig. 3.

The insertion algorithm works in iterations, each iteration pushes solutions to the layer that is immediately dominated by the layer of the previous iteration. On each iteration, the following operations are performed:

- The low-level tree of the current layer is split in three parts using the current pushed set of solutions C in the following way:
  - the "left part" T_L consists of all solutions from the current layer whose X coordinates are less than the smallest X coordinate of a solution from C;
  - the "middle part" T_M consists of all solutions from the current layer which are dominated by at least one solution from C;
  - the "right part" T_R consists of all solutions from the current layer whose Y coordinates are less than the smallest Y coordinate of a solution from C.
- The current layer is built by merging the trees T_L, C and T_R.
- If both T_L and T_R are empty, this means that the entire level was dominated by solutions from C. In turn, this means that a new layer consisting entirely of T_M should be inserted just after the current level. All remaining layers will effectively have their index increased by one. The insertion procedure stops here.

```
1: function SPLITX($T$, $s$)
2:     – splits a tree $T$ into two trees $L$, $R$
3:     – such that for all $l \in L$ holds $l.X < s.X$
4:     – and for all $r \in R$ holds $r.X \geq s.X$
5:     $T$ : LLNODE
6:     $s$ : SOLUTION
7: end function
8: function SPLITY($T$, $s$)
9:     – splits a tree $T$ into two trees $L$, $R$
10:    – such that for all $l \in L$ holds $l.Y \geq s.Y$
11:    – and for all $r \in R$ holds $r.Y < s.Y$
12:    $T$ : LLNODE
13:    $s$ : SOLUTION
14: end function
15: function MERGE($L$, $R$)
16:    – merges two trees $L$ and $R$ into a single one
17:    – given for any $l \in L$ and $r \in R$ holds $l.X < r.X$
18:    $L$ : LLNODE
19:    $R$ : LLNODE
20: end function
21: function INSERT($H$, $s$)
22:    – inserts a solution $s$ into a high-level tree $H$
23:    $H$ : HLNODE
24:    $s$ : SOLUTION
25:    $C \leftarrow$ NEW LLNODE($s$)
26:    $(G, i) \leftarrow$ SMALLESTNONDOMINATINGLAYER($H, s$)
27:    while $G \neq$ NULL do
28:        $C_{\min} \leftarrow$ a solution with minimum $x$ from $C$
29:        $C_{\max} \leftarrow$ a solution with minimum $y$ from $C$
30:        $(T_L, T_i) \leftarrow$ SPLITX($G.V, C_{\min}$)
31:        $(T_M, T_R) \leftarrow$ SPLITY($T_i, C_{\max}$)
32:        $G.V \leftarrow$ MERGE($T_L$, MERGE($C, T_R$))
33:        if $T_M =$ NULL then
34:            return – no more solutions to push down
35:        end if
36:        if $T_L =$ NULL and $T_R =$ NULL then
37:            – the current layer is dominated in whole
38:            – just insert pushed solutions as a new layer
39:            Insert NEW HLNODE($T_M$) after $G$
40:            return
41:        end if
42:        $C \leftarrow T_M$
43:        $G \leftarrow G.N$
44:    end while
45:    Insert NEW HLNODE($C$) after last node of $H$
46: end function
```

Figure 5: A pseudocode for insertion of a solution into a high-level tree

- If T_M is empty, the remaining layers should remain unchanged. The insertion procedure stops here.
- Otherwise, $C \leftarrow T_M$, and the insertion procedure continues with the next iteration.

If after the last iterations there are some solutions which were not inserted, a new layer is formed from them and is added as the last layer into the high-level tree.

We omit a long and rigorous proof of correctness of this algorithm for the sake of brevity. The running time of the insertion algorithm sums up from the running time of the lookup algorithm (which is $O(\log M \log(N/\log M))$) and from the total time spent in iterations. Assume that $P \leq M$ iterations were performed. Without losing generality, assume that the layers of sizes $L_1 \dots L_P$ were split in these iterations. Denote the sizes of their pairs after splits to be $L_1^L, L_1^M, L_1^R, \dots, L_P^L, L_P^M, L_P^R$. The value $L_0^M = 1$ corresponds to the initial set C consisting of the solution which is to be inserted. In i-th iteration, the following operations with $\omega(1)$ complexity were performed:

- finding minimum and maximum of C in $O(1+\log L_{i-1}^M)$;
- SplitX in $O(1 + \log(L_i^L + L_i^M + L_i^R))$;
- SplitY in $O(1 + \log(L_i^M + L_i^R))$;
- inner Merge in $O(1 + \log(L_{i-1}^M + L_i^R))$;
- outer Merge in $O(1 + \log(L_i^L + L_{i-1}^M + L_i^R))$.

In total, the sum of all numbers under logarithms does not exceed $4\sum_{i=1}^{P} L_i$, and hence is $O(N)$. By Cauchy's inequality, the sum of all running times for all iterations is $O(P(1 + \log(N/P)))$. For a fixed N, this function has a maximum when $P = \Theta(N)$, which both gives us that $O(P(1+\log(N/P))) = O(M(1+\log(N/M)))$ and the worst-case running time of $O(N)$. The layer insertion operations which can happen at the end of the algorithm cost only $O(\log M)$ and thus don't change the estimations.

The total running time complexity for the insertion algorithm is:

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

2.5 Querying the k-th Solution

The NSGA-II algorithm features a selection operator based on binary tournament which considers every solution exactly twice. It does this by maintaining a permutation of indices $1 \dots N$, where N is the number of solutions in the population. When a solution is to be taken from the population, it takes an element E of this permutation which was not yet considered and takes a solution which is located at the position of E. When all elements of the permutation are considered, a new (random) permutation is generated instead.

To support this selection operator, we need to support an operation of querying the k-th solution in some predefined order ($1 \leq k \leq N$). Note that querying a random solution can be implemented using this operation (k is chosen randomly uniformly from the range $[1; N]$). Each query must return a solution, the number of layer it resides at, and the crowding distance of this solution in its layer.

To achieve this aim, we utilize node fields containing subtree sizes, and a field which stores the total size of all keys in the subtree of a high-level tree node. We use the lexicographical order on the solutions: first, the layer number is used to order the solutions, second, the abscissa is used to break ties.

Given k, the number of a solution in the lexicographical order, the first part of the query algorithm finds the layer in which this solution resides. This is done by traversing the high-level tree from the root and tracking the total size of keys in subtrees. The second part uses the same idea to find the solution inside the layer. The third part evaluates the crowding distance and constructs the query result. The algorithm is outlined on Fig. 6. The running time of this algorithm is straightforwardly $O(\log N)$.

```
structure QueryResult
    – the result of a query
    $S$ : Solution – the solution
    $L$ : Integer – the layer number
    $C$ : Real – the crowding distance
end structure
function QueryLow($T, k, W, H, L$)
    – finds $k$-th solution in a low-level tree $T$
    $T$ : LLNode
    $k$ : Integer
    $W$ : Real – the abscissa difference
    $H$ : Real – the ordinate difference
    $L$ : Integer – the layer number
    if $T.L \neq$ null then
        if $k \leq T.L.S$ then
            return QueryLow($T.L, k, W, H, L$)
        else
            $k \leftarrow k - T.L.S$
        end if
    end if
    if $k = 1$ then
        $R \leftarrow$ new QueryResult
        $R.S \leftarrow T.V$, $R.L \leftarrow L$
        if $T.P =$ null or $T.N =$ null then
            $R.C \leftarrow \infty$
        else
            $R.C \leftarrow \frac{|T.N.V.X - T.P.V.X|}{W} + \frac{|T.N.V.Y - T.P.V.Y|}{H}$
        end if
        return $R$
    else
        $k \leftarrow k - 1$
    end if
    return QueryLow($T.R, k, W, H, L$)
end function
function Query($T, k$)
    – returns the $k$-th solution in a high-level tree $T$
    $T$ : HLNode
    $k$ : Integer
    $L$ : Integer $\leftarrow 0$ – the layer number
    loop
        if $T.L \neq$ null then
            if $k \leq T.L.W$ then
                $T \leftarrow T.L$
                continue
            else
                $k \leftarrow k - T.L.W$, $L \leftarrow L + T.L.S$
            end if
        end if
        if $k \leq T.V.S$ then
            $LL \leftarrow$ leftmost child of $T.V$
            $RR \leftarrow$ rightmost child of $T.V$
            $W \leftarrow |RR.V.X - LL.V.X|$
            $H \leftarrow |RR.V.Y - LL.V.Y|$
            return QueryLow($T.V, k, W, H, L$)
        end if
        $k \leftarrow k - T.V.S$, $L \leftarrow L + 1$, $T \leftarrow T.R$
    end loop
end function
```

Figure 6: A pseudocode for querying the k-th solution in lexicographical order

2.6 Deletion of the Worst Solution

In the NSGA-II algorithm, when the number of solutions exceeds the required population size, solutions are deleted from the population starting from the last layer. When one needs to retain some solutions in the last layer, they are deleted in the decreasing order of their crowding distances.

We have not invented an efficient data structure which locates the solution with the smallest crowding distance in a layer. So, to delete a single solution with the worst crowding distance from the last layer, we need to compute crowding distances for every solution in the layer and then delete the one with the smallest crowding distance. This can be done in $O(N)$.

3. EXPERIMENTS

To assess the performance of the proposed implementation, we performed experiments which are largely based on the experiments from [13]. The following benchmark problems are considered: ZDT1–ZDT4 and ZDT6 [17], DTLZ1–DTLZ7 [6], WFG1–WFG9 [8]. The parameters of these problems are the same as in the corresponding papers, except that we considered only two-dimensional versions of these problems.

The NSGA-II algorithm was implemented in as much compatible way to [13] as possible. Specifically, their implementation of the selection operator uses dominance comparison in the first step, unlike classical definition [5], which uses layer number comparison instead. A significant difference is that WFG problems in [13] were implemented using single-precision floating-point numbers, while we used double-precision numbers consistently.

We tested both generational and steady-state variants of the NSGA-II algorithm. Each of these variants was run using the proposed implementation, the ENLU approach [12] and the fast non-dominated sorting from [5]. For each combination, 100 runs were performed, each run had the computational budget of 25 000 evaluations as in [13]. In each run, the random seed for all operations related to evolutionary operators and to individual selection was derived from the number of the run. For other operations which require a random number generator, such as operations with Cartesian trees, a separate random number generator was used.

To assess quality of the results, we evaluated the hypervolume indicator [19] at the end of each run. To assess performance, we evaluated the wall-clock running time and the number of objective comparisons. Notably, each crowding distance evaluation was thought to use four objective comparisons. As the wall-clock time includes not only the time of non-dominated sorting, but the time of evolutionary operators and fitness function evaluations as well, we estimate the latter time by separately running evolutionary operators and fitness functions on randomly generated data and subtracting this time from the wall-clock time. For each mentioned measure we track the median and the interquartile range (IQR).

The source code for experiment reproduction is available both as supplementary materials to the paper and at GitHub[1]. We ran this code on a server with two Intel® Xeon® E5606 CPUs clocked at 2.13 GHz, having eight cores in total.

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

For each problem, medians and interquartile ranges for hypervolumes were correspondingly equal for all generational algorithms, and the same was true for all steady-state algorithms, so we are not presenting the results for hypervolumes. All other results of experiments are presented in Table 1.

One can see that the running times and the numbers of comparisons differ significantly. In all cases except one the proposed implementation (called INDS in the tables and further on) performs noticeably faster (in terms of wall-clock time) both with generational and with steady-state versions.

As the running time of Deb's fast non-dominated sorting is not input-sensitive, as follows from the algorithm, the number of comparisons as well as wall-clock times are almost equal for all problems, considering separately generational and steady-state NSGA-II variants. Two other approaches are clearly input-sensitive (for the steady-state case, the biggest time (0.12) is almost three times higher than the smallest time (0.042) in the case of ENLU).

The average operation complexity for INDS is expected to be higher than of the ENLU (due to usage of the tree-based data structure), and ENLU seems to be more complex than the fast non-dominated sorting as well. This made us interested in estimating the overhead imposed by using this or that algorithm, measured as the average wall-clock time **per objective comparison**. From the data presented in Table 1, we computed this ratio for all three algorithms separately for generational and steady-state variants. It appeared to be almost problem-independent. We give medians and interquartile ranges for these values (notation is the same as in Table 1):

- INDS(gen): median $4.07 \cdot 10^{-8}$, IQR $8.40 \cdot 10^{-10}$;
- ENLU(gen): median $1.39 \cdot 10^{-8}$, IQR $1.90 \cdot 10^{-10}$;
- debNDS(gen): median $8.97 \cdot 10^{-9}$, IQR $2.31 \cdot 10^{-11}$;
- INDS(ss): median $1.36 \cdot 10^{-8}$, IQR $1.87 \cdot 10^{-10}$;
- ENLU(ss): median $1.08 \cdot 10^{-8}$, IQR $2.00 \cdot 10^{-11}$;
- debNDS(ss): median $5.88 \cdot 10^{-9}$, IQR $1.58 \cdot 10^{-11}$.

We account the difference between generational and steady-state variants for different individual removal patterns (bulk removal in the first case and single element removal in the second case). According to the median values of average overhead, INDS is indeed the most expensive (approximately five times more work per single comparison than in fast non-dominated sorting), but it is not significantly worse than ENLU in this aspect. All differences are much smaller for the steady-state variant.

Finally, running times for the same algorithm on the same problem, but for two different NSGA-II variants (generational and steady-state) are very similar in the case of INDS and of ENLU (steady-state times never exceed generational times multiplied by two), but fast non-dominated sorting is slower by more than 10 times in the steady-state case. This removes a big complication of using steady-state multiobjective evolutionary algorithms in practice: while often being superior in terms of quality of results, they are no more slower in terms of running times – at least for biobjective problems.

Table 1: Results of experiments. Notation: "time" stands for the running time (in seconds), "cmp" for the number of comparisons, "med" for the median, "IQR" for the interquartile range. "INDS" is the proposed implementation, "ENLU" is the ENLU-based implementation, "debNDS" is the standard implementation which uses the Deb's fast non-dominated sorting, "gen" corresponds to the generational variant of the NSGA-II algorithm and "ss" to the steady-state variant. Gray cells denote minima of the running time and of the number of comparisons (for generational and steady-state variants separately).

Type	INDS(gen)		ENLU(gen)		debNDS(gen)		INDS(ss)		ENLU(ss)		debNDS(ss)	
	med	IQR	med	IQR	med	IQR	med	IQR	med	IQR	med	IQR
ZDT1												
time	**$5.42 \cdot 10^{-2}$**	**$1.12 \cdot 10^{-2}$**	$1.01 \cdot 10^{-1}$	$4.47 \cdot 10^{-3}$	$1.42 \cdot 10^{-1}$	$2.40 \cdot 10^{-3}$	**$7.49 \cdot 10^{-2}$**	**$2.44 \cdot 10^{-2}$**	$9.46 \cdot 10^{-2}$	$6.11 \cdot 10^{-3}$	$1.66 \cdot 10^{0}$	$2.47 \cdot 10^{-2}$
cmp	**$1.13 \cdot 10^{6}$**	**$2.68 \cdot 10^{4}$**	$6.53 \cdot 10^{6}$	$2.98 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$2.17 \cdot 10^{3}$	**$3.70 \cdot 10^{6}$**	**$7.74 \cdot 10^{4}$**	$7.64 \cdot 10^{6}$	$1.21 \cdot 10^{5}$	$2.58 \cdot 10^{8}$	$8.06 \cdot 10^{4}$
ZDT2												
time	**$5.68 \cdot 10^{-2}$**	**$7.50 \cdot 10^{-3}$**	$9.16 \cdot 10^{-2}$	$9.13 \cdot 10^{-3}$	$1.08 \cdot 10^{-1}$	$1.78 \cdot 10^{-2}$	**$7.09 \cdot 10^{-2}$**	**$2.53 \cdot 10^{-3}$**	$9.44 \cdot 10^{-2}$	$4.04 \cdot 10^{-3}$	$1.65 \cdot 10^{0}$	$4.88 \cdot 10^{-2}$
cmp	**$1.15 \cdot 10^{6}$**	**$2.14 \cdot 10^{4}$**	$6.10 \cdot 10^{6}$	$3.95 \cdot 10^{4}$	$1.07 \cdot 10^{7}$	$4.56 \cdot 10^{3}$	**$3.30 \cdot 10^{6}$**	**$1.14 \cdot 10^{5}$**	$6.90 \cdot 10^{6}$	$1.61 \cdot 10^{5}$	$2.57 \cdot 10^{8}$	$1.20 \cdot 10^{5}$
ZDT3												
time	**$5.46 \cdot 10^{-2}$**	**$8.72 \cdot 10^{-4}$**	$9.63 \cdot 10^{-2}$	$5.89 \cdot 10^{-3}$	$9.80 \cdot 10^{-2}$	$1.92 \cdot 10^{-3}$	**$7.07 \cdot 10^{-2}$**	**$1.80 \cdot 10^{-3}$**	$9.28 \cdot 10^{-2}$	$5.16 \cdot 10^{-3}$	$1.56 \cdot 10^{0}$	$1.31 \cdot 10^{-2}$
cmp	**$1.12 \cdot 10^{6}$**	**$2.31 \cdot 10^{4}$**	$6.32 \cdot 10^{6}$	$2.63 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.76 \cdot 10^{3}$	**$3.38 \cdot 10^{6}$**	**$6.21 \cdot 10^{4}$**	$7.14 \cdot 10^{6}$	$8.97 \cdot 10^{4}$	$2.57 \cdot 10^{8}$	$6.77 \cdot 10^{4}$
ZDT4												
time	**$4.44 \cdot 10^{-2}$**	**$2.89 \cdot 10^{-3}$**	$5.07 \cdot 10^{-2}$	$3.73 \cdot 10^{-3}$	$1.13 \cdot 10^{-1}$	$3.28 \cdot 10^{-3}$	$5.13 \cdot 10^{-2}$	$3.11 \cdot 10^{-3}$	**$4.18 \cdot 10^{-2}$**	**$4.48 \cdot 10^{-3}$**	$1.78 \cdot 10^{0}$	$5.24 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$2.97 \cdot 10^{4}$**	$3.41 \cdot 10^{6}$	$3.81 \cdot 10^{4}$	$1.07 \cdot 10^{7}$	$8.86 \cdot 10^{3}$	**$2.05 \cdot 10^{6}$**	**$1.01 \cdot 10^{5}$**	$3.68 \cdot 10^{6}$	$1.60 \cdot 10^{5}$	$2.56 \cdot 10^{8}$	$1.20 \cdot 10^{5}$
ZDT6												
time	**$5.61 \cdot 10^{-2}$**	**$2.12 \cdot 10^{-3}$**	$6.80 \cdot 10^{-2}$	$6.43 \cdot 10^{-4}$	$1.20 \cdot 10^{-1}$	$2.51 \cdot 10^{-3}$	**$6.65 \cdot 10^{-2}$**	**$2.35 \cdot 10^{-3}$**	$7.19 \cdot 10^{-2}$	$2.91 \cdot 10^{-3}$	$1.71 \cdot 10^{0}$	$3.26 \cdot 10^{-2}$
cmp	**$1.12 \cdot 10^{6}$**	**$1.11 \cdot 10^{4}$**	$4.52 \cdot 10^{6}$	$3.89 \cdot 10^{4}$	$1.07 \cdot 10^{7}$	$4.40 \cdot 10^{3}$	**$2.64 \cdot 10^{6}$**	**$9.39 \cdot 10^{4}$**	$5.29 \cdot 10^{6}$	$1.13 \cdot 10^{5}$	$2.56 \cdot 10^{8}$	$9.17 \cdot 10^{4}$
DTLZ1												
time	**$4.67 \cdot 10^{-2}$**	**$5.04 \cdot 10^{-3}$**	$5.66 \cdot 10^{-2}$	$3.19 \cdot 10^{-3}$	$1.13 \cdot 10^{-1}$	$2.86 \cdot 10^{-3}$	**$4.88 \cdot 10^{-2}$**	**$3.08 \cdot 10^{-3}$**	$5.21 \cdot 10^{-2}$	$5.21 \cdot 10^{-3}$	$1.69 \cdot 10^{0}$	$2.83 \cdot 10^{-2}$
cmp	**$1.02 \cdot 10^{6}$**	**$2.27 \cdot 10^{4}$**	$3.73 \cdot 10^{6}$	$5.80 \cdot 10^{4}$	$1.07 \cdot 10^{7}$	$7.87 \cdot 10^{3}$	**$2.21 \cdot 10^{6}$**	**$2.02 \cdot 10^{5}$**	$4.17 \cdot 10^{6}$	$2.42 \cdot 10^{5}$	$2.56 \cdot 10^{8}$	$2.19 \cdot 10^{5}$
DTLZ2												
time	**$5.41 \cdot 10^{-2}$**	**$4.08 \cdot 10^{-3}$**	$9.51 \cdot 10^{-2}$	$4.11 \cdot 10^{-3}$	$9.71 \cdot 10^{-2}$	$3.78 \cdot 10^{-3}$	**$6.75 \cdot 10^{-2}$**	**$3.92 \cdot 10^{-3}$**	$9.35 \cdot 10^{-2}$	$5.37 \cdot 10^{-3}$	$1.53 \cdot 10^{0}$	$2.15 \cdot 10^{-2}$
cmp	**$1.09 \cdot 10^{6}$**	**$2.81 \cdot 10^{4}$**	$6.24 \cdot 10^{6}$	$2.29 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.24 \cdot 10^{3}$	**$4.19 \cdot 10^{6}$**	**$5.00 \cdot 10^{4}$**	$7.91 \cdot 10^{6}$	$1.77 \cdot 10^{5}$	$2.58 \cdot 10^{8}$	$4.97 \cdot 10^{4}$
DTLZ3												
time	**$4.30 \cdot 10^{-2}$**	**$1.83 \cdot 10^{-3}$**	$5.37 \cdot 10^{-2}$	$4.27 \cdot 10^{-3}$	$1.13 \cdot 10^{-1}$	$9.16 \cdot 10^{-4}$	**$4.18 \cdot 10^{-2}$**	**$2.07 \cdot 10^{-3}$**	$4.78 \cdot 10^{-2}$	$4.52 \cdot 10^{-3}$	$1.94 \cdot 10^{0}$	$5.69 \cdot 10^{-2}$
cmp	**$1.08 \cdot 10^{6}$**	**$3.78 \cdot 10^{4}$**	$3.81 \cdot 10^{6}$	$1.11 \cdot 10^{5}$	$1.06 \cdot 10^{7}$	$8.25 \cdot 10^{3}$	**$1.31 \cdot 10^{6}$**	**$1.06 \cdot 10^{5}$**	$3.19 \cdot 10^{6}$	$1.02 \cdot 10^{5}$	$2.55 \cdot 10^{8}$	$1.07 \cdot 10^{5}$
DTLZ4												
time	**$5.60 \cdot 10^{-2}$**	**$2.81 \cdot 10^{-3}$**	$9.47 \cdot 10^{-2}$	$3.34 \cdot 10^{-3}$	$1.02 \cdot 10^{-1}$	$9.92 \cdot 10^{-3}$	**$6.87 \cdot 10^{-2}$**	**$6.93 \cdot 10^{-3}$**	$9.30 \cdot 10^{-2}$	$4.99 \cdot 10^{-3}$	$1.58 \cdot 10^{0}$	$5.08 \cdot 10^{-2}$
cmp	**$1.08 \cdot 10^{6}$**	**$3.32 \cdot 10^{4}$**	$6.21 \cdot 10^{6}$	$3.57 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$4.89 \cdot 10^{3}$	**$3.95 \cdot 10^{6}$**	**$8.01 \cdot 10^{4}$**	$7.67 \cdot 10^{6}$	$2.31 \cdot 10^{5}$	$2.58 \cdot 10^{8}$	$8.87 \cdot 10^{4}$
DTLZ5												
time	**$5.36 \cdot 10^{-2}$**	**$2.49 \cdot 10^{-4}$**	$9.30 \cdot 10^{-2}$	$1.24 \cdot 10^{-3}$	$9.66 \cdot 10^{-2}$	$3.03 \cdot 10^{-3}$	**$6.76 \cdot 10^{-2}$**	**$4.36 \cdot 10^{-3}$**	$9.40 \cdot 10^{-2}$	$4.35 \cdot 10^{-3}$	$1.54 \cdot 10^{0}$	$2.02 \cdot 10^{-2}$
cmp	**$1.09 \cdot 10^{6}$**	**$3.29 \cdot 10^{4}$**	$6.24 \cdot 10^{6}$	$2.29 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.24 \cdot 10^{3}$	**$4.19 \cdot 10^{6}$**	**$5.26 \cdot 10^{4}$**	$7.91 \cdot 10^{6}$	$1.77 \cdot 10^{5}$	$2.58 \cdot 10^{8}$	$4.97 \cdot 10^{4}$
DTLZ6												
time	**$4.96 \cdot 10^{-2}$**	**$6.55 \cdot 10^{-3}$**	$7.09 \cdot 10^{-2}$	$1.94 \cdot 10^{-3}$	$1.03 \cdot 10^{-1}$	$1.79 \cdot 10^{-3}$	**$5.88 \cdot 10^{-2}$**	**$2.52 \cdot 10^{-3}$**	$7.18 \cdot 10^{-2}$	$5.43 \cdot 10^{-3}$	$1.70 \cdot 10^{0}$	$1.96 \cdot 10^{-2}$
cmp	**$1.16 \cdot 10^{6}$**	**$2.38 \cdot 10^{4}$**	$5.17 \cdot 10^{6}$	$4.81 \cdot 10^{4}$	$1.07 \cdot 10^{7}$	$4.27 \cdot 10^{3}$	**$2.90 \cdot 10^{6}$**	**$2.13 \cdot 10^{5}$**	$5.95 \cdot 10^{6}$	$2.46 \cdot 10^{5}$	$2.57 \cdot 10^{8}$	$2.13 \cdot 10^{5}$
DTLZ7												
time	**$5.64 \cdot 10^{-2}$**	**$4.39 \cdot 10^{-3}$**	$9.43 \cdot 10^{-2}$	$3.82 \cdot 10^{-3}$	$1.02 \cdot 10^{-1}$	$1.80 \cdot 10^{-3}$	**$7.06 \cdot 10^{-2}$**	**$2.16 \cdot 10^{-3}$**	$9.07 \cdot 10^{-2}$	$2.93 \cdot 10^{-3}$	$1.59 \cdot 10^{0}$	$1.65 \cdot 10^{-2}$
cmp	**$1.13 \cdot 10^{6}$**	**$2.07 \cdot 10^{4}$**	$6.12 \cdot 10^{6}$	$2.86 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.85 \cdot 10^{3}$	**$3.55 \cdot 10^{6}$**	**$6.15 \cdot 10^{4}$**	$7.21 \cdot 10^{6}$	$1.32 \cdot 10^{5}$	$2.57 \cdot 10^{8}$	$6.50 \cdot 10^{4}$
WFG1												
time	**$4.90 \cdot 10^{-2}$**	**$1.48 \cdot 10^{-3}$**	$7.91 \cdot 10^{-2}$	$1.87 \cdot 10^{-3}$	$1.00 \cdot 10^{-1}$	$1.12 \cdot 10^{-3}$	**$6.11 \cdot 10^{-2}$**	**$4.02 \cdot 10^{-3}$**	$8.15 \cdot 10^{-2}$	$1.15 \cdot 10^{-2}$	$1.57 \cdot 10^{0}$	$2.80 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$1.99 \cdot 10^{4}$**	$5.48 \cdot 10^{6}$	$1.62 \cdot 10^{5}$	$1.07 \cdot 10^{7}$	$1.14 \cdot 10^{4}$	**$2.94 \cdot 10^{6}$**	**$4.30 \cdot 10^{5}$**	$6.27 \cdot 10^{6}$	$4.71 \cdot 10^{5}$	$2.57 \cdot 10^{8}$	$5.18 \cdot 10^{5}$
WFG2												
time	**$5.94 \cdot 10^{-2}$**	**$2.67 \cdot 10^{-3}$**	$1.04 \cdot 10^{-1}$	$5.11 \cdot 10^{-3}$	$1.02 \cdot 10^{-1}$	$1.87 \cdot 10^{-3}$	**$9.73 \cdot 10^{-2}$**	**$4.72 \cdot 10^{-3}$**	$1.23 \cdot 10^{-1}$	$8.75 \cdot 10^{-3}$	$1.54 \cdot 10^{0}$	$1.98 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$3.65 \cdot 10^{4}$**	$6.97 \cdot 10^{6}$	$6.68 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$2.14 \cdot 10^{3}$	**$7.05 \cdot 10^{6}$**	**$1.25 \cdot 10^{5}$**	$1.13 \cdot 10^{7}$	$1.64 \cdot 10^{5}$	$2.61 \cdot 10^{8}$	$8.74 \cdot 10^{4}$
WFG3												
time	**$5.63 \cdot 10^{-2}$**	**$3.37 \cdot 10^{-3}$**	$9.40 \cdot 10^{-2}$	$5.75 \cdot 10^{-4}$	$9.83 \cdot 10^{-2}$	$2.29 \cdot 10^{-3}$	**$7.59 \cdot 10^{-2}$**	**$5.90 \cdot 10^{-3}$**	$9.56 \cdot 10^{-2}$	$6.60 \cdot 10^{-3}$	$1.53 \cdot 10^{0}$	$1.72 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$2.69 \cdot 10^{4}$**	$6.16 \cdot 10^{6}$	$2.88 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.42 \cdot 10^{3}$	**$4.98 \cdot 10^{6}$**	**$7.89 \cdot 10^{4}$**	$8.69 \cdot 10^{6}$	$1.94 \cdot 10^{5}$	$2.59 \cdot 10^{8}$	$7.67 \cdot 10^{4}$
WFG4												
time	**$5.34 \cdot 10^{-2}$**	**$1.46 \cdot 10^{-3}$**	$9.22 \cdot 10^{-2}$	$2.35 \cdot 10^{-3}$	$9.90 \cdot 10^{-2}$	$1.19 \cdot 10^{-3}$	**$7.51 \cdot 10^{-2}$**	**$4.20 \cdot 10^{-3}$**	$9.78 \cdot 10^{-2}$	$3.68 \cdot 10^{-3}$	$1.53 \cdot 10^{0}$	$2.04 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$2.11 \cdot 10^{4}$**	$6.07 \cdot 10^{6}$	$5.90 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$2.56 \cdot 10^{3}$	**$4.76 \cdot 10^{6}$**	**$1.92 \cdot 10^{5}$**	$8.43 \cdot 10^{6}$	$2.49 \cdot 10^{5}$	$2.59 \cdot 10^{8}$	$1.85 \cdot 10^{5}$
WFG5												
time	**$5.62 \cdot 10^{-2}$**	**$4.48 \cdot 10^{-3}$**	$1.03 \cdot 10^{-1}$	$2.08 \cdot 10^{-3}$	$9.73 \cdot 10^{-2}$	$5.45 \cdot 10^{-3}$	**$9.01 \cdot 10^{-2}$**	**$3.21 \cdot 10^{-3}$**	$1.20 \cdot 10^{-1}$	$3.73 \cdot 10^{-3}$	$1.54 \cdot 10^{0}$	$1.60 \cdot 10^{-2}$
cmp	**$1.08 \cdot 10^{6}$**	**$4.42 \cdot 10^{4}$**	$7.10 \cdot 10^{6}$	$1.80 \cdot 10^{5}$	$1.08 \cdot 10^{7}$	$7.08 \cdot 10^{3}$	**$6.71 \cdot 10^{6}$**	**$4.86 \cdot 10^{5}$**	$1.11 \cdot 10^{7}$	$7.35 \cdot 10^{5}$	$2.61 \cdot 10^{8}$	$4.73 \cdot 10^{5}$
WFG6												
time	**$5.46 \cdot 10^{-2}$**	**$3.50 \cdot 10^{-3}$**	$9.19 \cdot 10^{-2}$	$6.80 \cdot 10^{-3}$	$1.01 \cdot 10^{-1}$	$3.19 \cdot 10^{-3}$	**$6.92 \cdot 10^{-2}$**	**$2.74 \cdot 10^{-3}$**	$9.29 \cdot 10^{-2}$	$7.66 \cdot 10^{-3}$	$1.53 \cdot 10^{0}$	$1.88 \cdot 10^{-2}$
cmp	**$1.07 \cdot 10^{6}$**	**$1.97 \cdot 10^{4}$**	$5.96 \cdot 10^{6}$	$7.38 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$4.13 \cdot 10^{3}$	**$4.48 \cdot 10^{6}$**	**$3.28 \cdot 10^{5}$**	$8.10 \cdot 10^{6}$	$4.59 \cdot 10^{5}$	$2.58 \cdot 10^{8}$	$3.29 \cdot 10^{5}$
WFG7												
time	**$5.58 \cdot 10^{-2}$**	**$1.64 \cdot 10^{-3}$**	$9.80 \cdot 10^{-2}$	$2.45 \cdot 10^{-3}$	$9.96 \cdot 10^{-2}$	$4.25 \cdot 10^{-3}$	**$7.81 \cdot 10^{-2}$**	**$1.67 \cdot 10^{-3}$**	$1.02 \cdot 10^{-1}$	$2.84 \cdot 10^{-3}$	$1.52 \cdot 10^{0}$	$1.63 \cdot 10^{-2}$
cmp	**$1.06 \cdot 10^{6}$**	**$2.98 \cdot 10^{4}$**	$6.38 \cdot 10^{6}$	$2.43 \cdot 10^{4}$	$1.08 \cdot 10^{7}$	$1.11 \cdot 10^{3}$	**$5.53 \cdot 10^{6}$**	**$7.61 \cdot 10^{4}$**	$9.47 \cdot 10^{6}$	$1.35 \cdot 10^{5}$	$2.59 \cdot 10^{8}$	$7.71 \cdot 10^{4}$
WFG8												
time	**$5.50 \cdot 10^{-2}$**	**$4.76 \cdot 10^{-3}$**	$8.04 \cdot 10^{-2}$	$1.46 \cdot 10^{-3}$	$1.08 \cdot 10^{-1}$	$1.40 \cdot 10^{-3}$	**$4.79 \cdot 10^{-2}$**	**$3.04 \cdot 10^{-3}$**	$6.28 \cdot 10^{-2}$	$6.48 \cdot 10^{-3}$	$1.52 \cdot 10^{0}$	$2.64 \cdot 10^{-2}$
cmp	**$1.05 \cdot 10^{6}$**	**$4.14 \cdot 10^{4}$**	$4.59 \cdot 10^{6}$	$1.25 \cdot 10^{5}$	$1.07 \cdot 10^{7}$	$8.61 \cdot 10^{3}$	**$1.66 \cdot 10^{6}$**	**$1.32 \cdot 10^{5}$**	$4.12 \cdot 10^{6}$	$1.61 \cdot 10^{5}$	$2.56 \cdot 10^{8}$	$1.23 \cdot 10^{5}$
WFG9												
time	**$5.67 \cdot 10^{-2}$**	**$2.99 \cdot 10^{-3}$**	$9.68 \cdot 10^{-2}$	$3.83 \cdot 10^{-3}$	$1.01 \cdot 10^{-1}$	$5.63 \cdot 10^{-3}$	**$7.46 \cdot 10^{-2}$**	**$3.45 \cdot 10^{-3}$**	$1.04 \cdot 10^{-1}$	$3.93 \cdot 10^{-3}$	$1.53 \cdot 10^{0}$	$1.56 \cdot 10^{-2}$
cmp	**$1.06 \cdot 10^{6}$**	**$2.69 \cdot 10^{4}$**	$6.23 \cdot 10^{6}$	$1.05 \cdot 10^{5}$	$1.08 \cdot 10^{7}$	$4.66 \cdot 10^{3}$	**$4.91 \cdot 10^{6}$**	**$4.58 \cdot 10^{5}$**	$8.56 \cdot 10^{6}$	$6.21 \cdot 10^{5}$	$2.59 \cdot 10^{8}$	$4.31 \cdot 10^{5}$

4. CONCLUSION

We proposed a new approach to implementation of steady-state multiobjective evolutionary algorithms for two dimensions. This approach is based on a data structure which is able to perform fast incremental non-dominated sorting and track other valuable data such as crowding distance. Experiments with the NSGA-II algorithm showed that the new approach offers running times similar to or better than those of generational versions while retaining the quality of steady-state versions.

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

5. REFERENCES

[1] H. A. Abbass, R. Sarker, and C. Newton. PDE: A Pareto Frontier Differential Evolution Approach for Multiobjective Optimization Problems. In *Proceedings of the Congress on Evolutionary Computation*, pages 971–978. IEEE Press, 2001.

[2] M. Buzdalov and A. Shalyto. A provably asymptotically fast version of the generalized Jensen algorithm for non-dominated sorting. In *International Conference on Parallel Problem Solving from Nature*, number 8672 in Lecture Notes in Computer Science, pages 528–537. 2014.

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

[4] D. W. Corne, J. D. Knowles, and M. J. Oates. The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization. In *Parallel Problem Solving from Nature Parallel Problem Solving from Nature VI*, number 1917 in Lecture Notes in Computer Science, pages 839–848. Springer, 2000.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *Transactions on Evolutionary Computation*, 6:182–197, 2000.

[6] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. *Scalable Test Problems for Evolutionary Multiobjective Optimization*, pages 105–145. Springer, 2005.

[7] F.-A. Fortin, S. Grenier, and M. Parizeau. Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting. In *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO '13, pages 615–622. ACM, 2013.

[8] 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.

[9] 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.

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

[11] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4):469–476, 1975.

[12] K. Li, K. Deb, Q. Zhang, and S. Kwong. Efficient non-domination level update approach for steady-state evolutionary multiobjective optimization. Technical report, 2014.

[13] 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.

[14] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of ACM*, 32(3):652–686, 1985.

[15] J. Vuillemin. A unifying look at data structures. *Communications of ACM*, 23(4):229–239, 1980.

[16] 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*, 2015 (to appear).

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

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

[19] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the Strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.