Incremental Non-Dominated Sorting with $O(N)$ Insertion for the Two-Dimensional Case

Ilya Yakupov, Maxim Buzdalov
ITMO University
49 Kronverkskiy av.
Saint-Petersburg, Russia, 197101
Email: iyakupov93@gmail.com, mbuzdalov@gmail.com

***Abstract*—We propose a new algorithm for incremental non-dominated sorting of two-dimensional points. The data structure which stores non-dominating layers is based on a tree of Cartesian trees. If there are N points in M layers, the running time for of an insertion is $O(M(1 + \log(N/M)) + \log M \log(N/\log M))$, which is $O(N)$ in the worst case.**

This algorithm can be a basic building block for efficient implementations of steady-state multiobjective algorithms such as NSGA-II.

I. 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 [1], PESA [2], PESA-II [3], SPEA2 [4], PAES [5], PDE [6], 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 [7], the algorithm for determining the non-dominated solutions is proposed with the 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 $O(N)$.

978-1-4799-7492-4/15/$31.00

Jensen [8] 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 [9]. The corrected (or, as in [9], "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 [10] 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 become asynchronous. Particularly, a steady-state version of the NSGA-II algorithm was developed [11] which showed good convergence rate and high quality of Pareto front approximation. However, the running time of this variation 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: $O(KN^3)$ when the fast non-dominated sorting [1] is used, or $O(N^2 \log^{K-1} N)$ when the sorting from [10] 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 presents our first results for incremental non-dominated sorting. The presented algorithm is developed for two-dimensional case ($K = 2$) and has $O(N)$ worst-case complexity for single insertion. More precisely, if there are M layers, the worst-case complexity for single insertion is $O(M(1 + \log(N/M)) + \log M \log(N/\log M))$, which is smaller than $O(N)$ if M is small.

II. Used Data Structures

To implement our 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);
- 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 [13] 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.

III. Algorithm Description

In this section, we describe the proposed algorithm and the data structure which supports it. The data structure design is described in Section III-A. The procedure of solution lookup (finding which layer a solution belongs to) is described in Section III-B. The procedure of solution insertion is described in Section III-C. The worst solution deletion is described in Section III-D.

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.

A. 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. The pseudocode for the resulting "tree of trees" data structure is given in Fig. 1, and the data structure itself is presented graphically in Fig. 2.

Note that the tree of layers can be an ordinary balanced tree, while every tree of layer elements 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 tree of layers. Additionally, to move between adjacent layers in $O(1)$, nodes of the tree of layers should be augmented with pointers to the next-in-order node (which can be done without affecting $O(\log N)$ performance of basic operations).

```
1:  structure SOLUTION
2:      – a solution to the optimization problem
3:      X – the first objective
4:      Y – the second objective
5:  end structure
6:  structure LLTNODE
7:      – a node of a low-level tree
8:      L : LLTNODE – the left child
9:      R : LLTNODE – the right child
10:     V : SOLUTION – the node key
11: end structure
12: structure HLTNODE
13:     – a node of a high-level tree
14:     L : HLTNODE – the left child
15:     R : HLTNODE – the right child
16:     N : HLTNODE – the next-in-order node
17:     V : LLTNODE – the node key
18:     S : INTEGER – the subtree size
19: end structure
```

Fig. 1. A pseudocode for the data structure

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.

B. 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.

To prove the latter fact, consider two cases. If u is not found, then for any solution t from T it holds that $t_X > s_X$, so t does not dominate s. If u is found, all solutions from T which are not equal to u can be divided into two groups: $V = \{v \,|\, v_X < u_X\}$ and $W = \{w \,|\, w_X > u_X\}$. For every solution v in V it also holds that $v_Y > u_Y$. If u does not dominate s, then $u_Y > s_Y$, because $u_X \le s_X$. This means that for every solution v from V it holds that $v_Y > u_Y > s_Y$, so no solution from V can dominate s. At the same time, for every solution w from W it holds that $w_X > s_X$ by construction (u_X is maximum possible such that $u_X \le s_X$), so no solution from W can dominate s as well.

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.

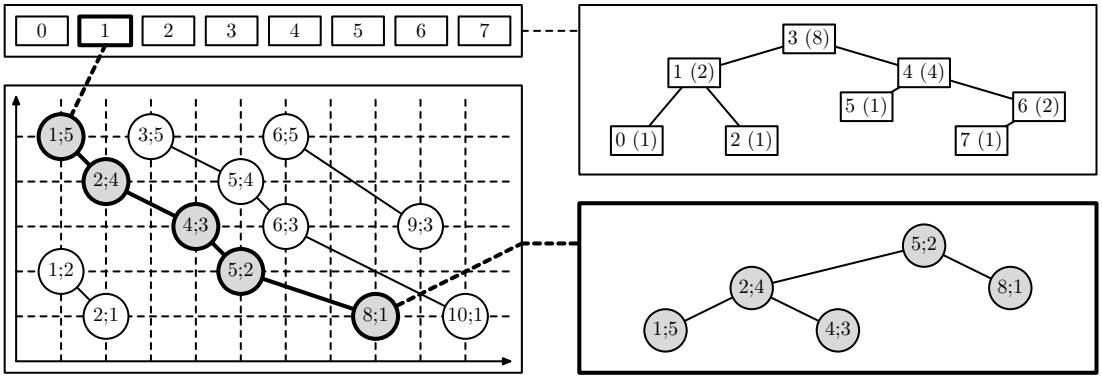

Fig. 2. Data structure for the algorithm – the "tree of trees". Nodes of the "high-level" tree correspond to the layers. Each layer is, in turn, represented by a "low-level" tree, where nodes are sorted by the first objective. In each node of the "high-level" tree, the subtree size is stored as well (the numbers in parentheses shown near to the layer number). Note that layer numbers are not stored in nodes explicitly, they are just shown for convenience.

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 $L_1 + \ldots + L_k \le N$. 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.

C. 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 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 "lookup" operation.

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 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 validity of such splitting will be proven below in Lemma 1.
- 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.
- 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.

To prove correctness of this algorithm, we prove the following lemma first.

```
1: function LowLevelDominates(T, s)
2:     – returns whether any solution from T dominates s
3:     T : LLTNode – the root node of the low-level tree
4:     s : Solution – the solution to test for domination
5:     B ← NULL – the best node so far
6:     while T ≠ NULL do
7:         if T.V.X ≤ s.X then
8:             B ← T
9:             T ← T.R
10:        else
11:            T ← 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 : HLTNode – the root node of the high-level tree
23:    s : Solution – the solution to find a layer for
24:    I ← 0 – the number of dominating layers so far
25:    B ← NULL – the best node so far
26:    while H ≠ NULL do
27:        if LowLevelDominates(H.V, s) then
28:            I ← I + H.S
29:            H ← H.R
30:            if H ≠ NULL then
31:                I ← I − H.S
32:            end if
33:        else
34:            B ← H
35:            H ← H.L
36:        end if
37:    end while
38:    return (B, I)
39: end function
```

Fig. 3. A pseudocode for determining the smallest layer which doesn't dominate the given solution

Lemma 1. *Consider a two-dimensional space of solutions. Let there be two sets of solutions, A and B, such that no two solutions from A dominate each other, no two solutions from B dominate each other and every solution from B is dominated by at least one solution from A.*

Let a subset $A' \subseteq A$ be defined as $\{a : a \in A, a.X \geq X_A, a.Y \geq Y_A\}$ for some X_A and Y_A. Let a subset $B' \subseteq B$ be defined as $\{b : b \in B, \exists a \in A' : a \text{ dominates } b\}$. Then, there exist some X_B and Y_B such that $B' = \{b : b \in B, b.X \geq X_B, b.Y \geq Y_B\}$.

Proof: Let $a_{\min}$ be a solution from A' with the minimum X possible, and let $a_{\max}$ be a solution from A' with the minimum Y possible. Let $X'_A = a_{\min}.X$ and $Y'_A = a_{\max}.Y$. As no two solutions from A' dominate each other, the value $X_M = a_{\max}.X$ is the maximum possible X for solutions from A', and the value $Y_M = a_{\min}.Y$ is the maximum possible Y for solutions from A'.

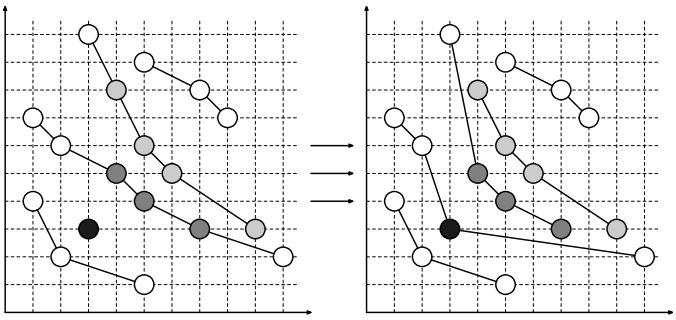

Fig. 4. An example of insertion process. Solutions which don't change their layer nodes (not numbers!) during the insertion process are white. A solution which is being inserted is black. Two clusters of solutions which together change their layer node are dark-gray and light-gray, correspondingly.

Obviously, for every solution $b \in B'$ it holds that $b.X \geq X'_A$ and $b.Y \geq Y'_A$, because if either of these conditions is violated, b can not be dominated by any element from A'. So, $B' \subseteq \{b : b \in B, b.X \geq X'_A, b.Y \geq Y'_A\}$.

We need to prove that every solution $b \in B$ for which $b.X \geq X'_A$ and $b.Y \geq Y'_A$ belongs to B' as well. This will be done by contradiction. Assume that there exists some $b \in B$ such that $b.X \geq X'_A$ and $b.Y \geq Y'_A$, but it does not belong to B'. By definition, there exists a solution $a \in A$ such that a dominates b. By definition of B', $a \notin A'$, so either $a.X < X'_A$ or $a.Y < Y'_A$. Consider the cases separately:

- If $a.X < X'_A$, then $a.Y > Y_M$, otherwise a dominates $a_{\min}$. As a consequence, $b.Y > Y_M$ as well. However, $b.X \geq X'_A$, so the solution $a_{\min} \in A'$ actually dominates b. Contradiction.
- If $a.Y < Y'_A$, then $a.X > X_M$, otherwise a dominates $a_{\max}$. As a consequence, $b.X > X_M$ as well. However, $b.Y \geq Y'_A$, so the solution $a_{\max} \in A'$ actually dominates b. Contradiction.

Each case terminates with a contradiction, so the assumption about existence of b is wrong, which proves the lemma. ■

In other words, we just proved that, given two successive layers, a contiguous fragment of the first layer always dominates a contiguous fragment of the second layer. This explains why splitting the current layer in three parts always yields the middle set which is completely dominated by the solutions which are to be pushed into this layer.

Theorem 1 (Correctness of an iteration)**.** *At the beginning of an iteration t ($t \geq 1$), denote by L_i^t the layer with index i, by S^t an index of the layer to which the solutions are pushed, by C^t the set of solutions pushed to $L_{s^t}^t$. Let M be the number of layers at the beginning of the first iteration.*

If initially the layers form a correct set of non-dominating layers (i.e. no solution from L_1^1 is dominated by any other solutions, for every $k \geq 2$ and for every $a \in L_k^1$ there exists

1: **function** SPLITX(T, s)
2: – splits a tree T into two trees L, R
3: – such that for all $l \in L$ holds $l.X \leq s.X$
4: – and for all $r \in R$ holds $r.X > s.X$
5: T : LLTNODE
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 $l.Y > s.Y$
11: – and for all $r \in R$ holds $r.Y \leq s.Y$
12: T : LLTNODE
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 : LLTNODE
19: R : LLTNODE
20: **end function**
21: **function** INSERT(H, s)
22: – inserts a solution s into a high-level tree H
23: H : HLTNODE
24: s : SOLUTION
25: $C \leftarrow$ NEW LLTNODE(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 HLTNODE(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 HLTNODE(C) after last node of H
46: **end function**

Fig. 5. A pseudocode for insertion of a solution into a high-level tree

some $b \in L^1_{k-1}$ which dominates a, but no two solutions from L^1_k dominate each other), and S^1 is chosen by the SMALLESTNONDOMINATINGLAYER *function, the following statements are true:*

1) *The number of layers at the beginning of an iteration t is exactly M.*
2) *The layers $L^t_1 \ldots L^t_M$ form a correct set of non-dominating layers.*
3) *Every solution from C^t is dominated by at least one solution from $L^t_{S^t-1}$, if $S^t > 1$.*
4) *If $t > 1$, then $S^t = S^{t-1} + 1$ and there exist X^t and Y^t such that $C^t = \{c : c \in L^{t-1}_{S^t-1}, c.X \geq X^t, c.Y \geq Y^t\}$.*
5) *If $t > 1$, then for any $1 \leq i \leq M$, $i \neq S^{t-1}$, $L^{t-1}_i = L^t_i$.*

Proof: The first statement is easy to prove, as every level addition is immediately followed by termination of the insertion algorithm. The fifth statement holds because the only layer changed at an iteration t has the number of S^t.

The other statements are proved by induction. The induction base is $t = 1$, where:

- the second and the third statements are true by definition.
- as $t = 1$, the fourth statement is not checked.

Let's prove correctness of these statements for $t+1$ if they hold for t. The next iteration will be performed if the layer $L^t_{S^t}$ is split by the minimum X and Y from C^t such that the middle set T_M is not empty and at least one of the sets T_L and T_R is not empty as well.

As $C^{t+1} = T_M$, by Lemma 1 the values of X^{t+1} and Y^{t+1} exist. The condition $S^{t+1} = S^t + 1$ is fulfilled by line 43 on Fig. 5, so the fourth statement for $t+1$ is true.

As $C^{t+1} = T_M$, for every solution $a \in C^{t+1}$ there exists a solution $b \in C^t$ such that b dominates a. However, $C^t \subset L^{t+1}_{S^t} = L^{t+1}_{S^{t+1}-1}$, so the third statement is true as well.

Finally, the second statement has to be proven. Due to the fifth statement, we need to prove the following statements only:

- In $L^{t+1}_{S^t}$ no two solutions dominate each other. As $L^{t+1}_{S^t} = T_L \cup C^t \cup T_R$ and $T_L \subset L^t_{S^t}$, $T_R \subset L^t_{S^t}$, we need to show that:
  - For every $a \in C^t$ and $b \in T_L$, a and b don't dominate each other. If $t = 1$, this holds by definition of S^t. Otherwise, $C^t \subset L^{t-1}_{S^t-1}$ and $T_R \subset L^{t-1}_{S^t}$ by induction assumption, so b cannot dominate a. However, $b.X < a.X$ by construction, so a cannot dominate b.
  - For every $a \in C^t$ and $b \in T_R$, a and b don't dominate each other. This proof is symmetrical to the previous one.
- For every $a \in L^{t+1}_{S^t}$ there exists $b \in L^{t+1}_{S^t-1}$ such that b dominates a. This is true because $L^{t+1}_{S^t} = T_L \cup C^t \cup T_R$, $T_L \subset L^t_{S^t}$, $T_R \subset L^t_{S^t}$, both $L^t_{S^t}$ and C^t are dominated by $L^t_{S^t-1}$ and $L^{t+1}_{S^t-1} = L^t_{S^t-1}$.
- For every $a \in L^{t+1}_{S^t+1}$ there exists $b \in L^{t+1}_{S^t}$ such that b dominates a. As $L^{t+1}_{S^t+1} = L^t_{S^t+1}$, for every a there exists $b' \in L^t_{S^t}$ such that b' dominates a. As $L^{t+1}_{S^t} = T_L \cup C^t \cup T_R$, $T_L \subset L^t_{S^t}$, $T_R \subset L^t_{S^t}$, the statement is true if $b' \in T_L \cup T_R$. The only alternative is $b' \in T_M$. However, for every $t \in T_M$ there is $t' \in C^t$ such that t' dominates t, so t' dominates a as well.

This case analysis finishes proving this theorem. ■

Theorem 2 (Correctness of the algorithm)**.** *If before running the algorithm the layers formed a correct set of non-dominating layers, then when the algorithm finishes:*

1) *the layers will form a correct set of non-dominating layers;*
2) *every solution which was in the data structure before running the algorithm will remain in the data structure;*
3) *the inserted solution will be in the data structure.*

Proof: At every exit point of the algorithm, there are no solutions which should be pushed to any layers, and no solution is ever removed by the insertion algorithm, so the second statement is true. As initially the inserted solution is in the set of solutions which should be pushed, it will be in the data structure when the algorithm terminates, so the third statement is true.

To prove the first statement, consider three exit points of the algorithm:

- The algorithm exits on Line 34 of Fig. 5. At this point, there are no solutions which should be pushed to any layers. By Theorem 1, the layers will form a correct set of non-dominating layers.
- The algorithm exits on Line 40 of Fig. 5. Here, the layer formed by C dominates the entire layer formed by $T_M = G.V$, which, in turn, dominates the entire subsequent layer (by the second statement of Theorem 1). So the new layer formed by T_M can be inserted after the layer formed by C without violation of the first statement of this theorem.
- The algorithm exits on Line 46 of Fig. 5. Before insertion of the new layer, all solutions from C don't dominate each other, as C either consists of a single solution or is a fragment of a layer. Additionally, every solution from C is dominated by the last layer (third statement of Theorem 1). So if this new layer is inserted, the layers will form a correct set of non-dominating layers.

All the cases are proven, so the theorem is proven as well. ■

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 $P \le M$ 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:

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

D. Deletion of the Worst Solution

In most multiobjective algorithms, deletion of an arbitrary solution is not needed and hence is not necessary to support. The only solutions which are deleted are the "worst" solutions, which are stored at the last layer and can be deleted without rebuilding the whole data structure. The running time of the algorithm for deletion of the worst solution (an arbitrary one from the last layer) is $O(\log N + \log M)$.

IV. EXPERIMENTS

To perform comparison of the proposed algorithm with the existing methods, we generated a number of benchmark problems. Each benchmark problem is a list of two-dimensional integer points (solutions to a hypothetical optimization problem) which needs to be added to non-dominating layers one by one in the specified order.

We used the following problem generators, where N is the problem size:

- "square": generates N random points from an $N \times N$ square;
- "parallel": generates N random points, $N/2$ of which lie on a line $y = N - x$, while the remaining points lie on a line $y = N - x + 1$;
- "diag1": generates a sequence of N points (x, x), starting from the biggest x;
- "diag2": generates a sequence of N points $(x, x+5)$ and $(x+5, x)$ one after another, starting from the biggest x;
- "parper": generates a "parallel-perpendicular" test which consists of $N/6$ points on a line $y = x+5$, $N/6$ points on a line $y = x-5$, $N/3$ points on a line $y = N/3-x-4$, and $N/3$ points on a line $y = N/3-x-6$, laid out as shown on Fig. 6.

We evaluate the following algorithms:

- the fast non-dominated sorting from [1], which is run once on all points;
- the ENLU approach from [12], which is used to add one point at a time;
- the proposed method, which is also used to add one point at a time.

Two measures are used: the total wall-clock running time and the total number of comparisons made. Problem sizes are taken from the set $\{250, 500, 1000, 2000, 4000\}$. For each problem size, each problem generator and each algorithm, 100 runs are performed and the measures are averaged.

For the "square" test, the results are presented on Fig. 7 for comparisons and on Fig. 12 for wall-clock time. We

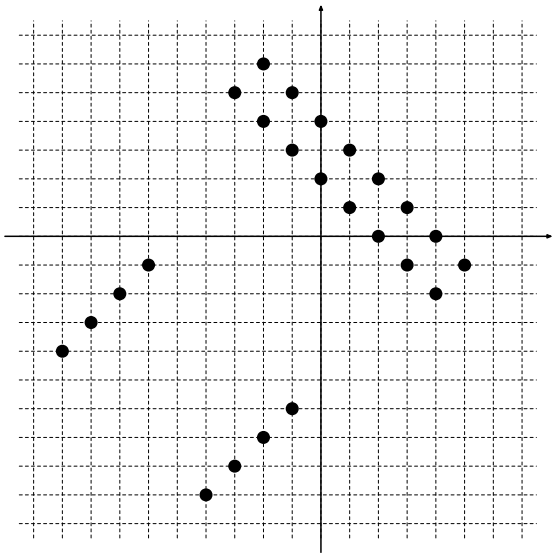

Fig. 6. An example of the "parper" test for $N = 24$

can see that the ENLU approach outperforms the plain fast non-dominated sorting, and the proposed method, in turn, outperforms ENLU.

For the "parallel" test, the results are presented on Fig. 8 for comparisons and on Fig. 13 for wall-clock time. This is an example when the proposed method is a clear winner both in absolute and in asymptotical sense. Indeed, the small constant number of layers renders the insertion time to be $O(\log N)$ which is virtually unreachable for ENLU when the size of the first layer is $O(N)$.

For the "diag1" test, the results are presented on Fig. 9 for comparisons and on Fig. 14 for wall-clock time. This is the "best-case" test for ENLU: every insertion is processed in $O(1)$ time. The proposed method performs in $O(\log N)$. However, the wall-clock times of these two methods are almost identical. This probably can be explained by some auxiliary operations whose complexity overtakes the complexity of comparisons.

For the "diag2" test, the results are presented on Fig. 10 for comparisons and on Fig. 15 for wall-clock time. This test seems to be the worst-case for the proposed method, where it demonstrates $O(N)$ insertion times. Even in these conditions, it makes fewer comparisons than its competitors and it is on par with the fast non-dominated sorting by wall-clock time.

Finally, for the "parper" test, the results are presented on Fig. 11 for comparisons and on Fig. 16 for wall-clock time. This test was constructed specially to challenge the ENLU approach. Here it demonstrates $O(N^2)$ insertion complexity, and its overall performance is $O(N^3)$. In fact, for $N = 4000$ the number of comparisons exceeded $3 \cdot 10^9$. For the proposed method, this test is not difficult.

V. Conclusion

A new algorithm for incremental non-dominated sorting is proposed, which has a worst case insertion complexity of $O(M(1 + \log(N/M)) + \log M \log(N/\log M))$, where N is the number of solutions and M is the number of layers. In the worst possible case, this evaluates to $O(N)$. Experiments show that the proposed algorithm is efficient not only from theoretical point of view, but in practice too.

A side effect of this research is that a test is generated for the competing ENLU approach [12] where it demonstrates an $O(N^2)$ insertion complexity for $O(N)$ insertions.

As a future work, we consider relaxing the two-dimensional condition and evaluating more information, such as crowding distance, which will allow to construct efficien steady-state versions of multiobjective evolutionary algorithms.

The code which can be used to reproduce the experiments is published at GitHub[1].

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

References

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

[2] 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*, ser. Lecture Notes in Computer Science. Springer, 2000, no. 1917, pp. 839–848.

[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*. Morgan Kaufmann Publishers, 2001, pp. 283–290.

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

[5] J. D. Knowles and D. W. Corne, "Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy," *Evolutionary Computation*, vol. 8, no. 2, pp. 149–172, 2000.

[6] 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*. IEEE Press, 2001, pp. 971–978.

[7] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *Journal of ACM*, vol. 22, no. 4, pp. 469–476, 1975.

[8] M. T. Jensen, "Reducing the Run-time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms," *Transactions on Evolutionary Computation*, vol. 7, no. 5, pp. 503–515, 2003.

[9] 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*, ser. GECCO '13. ACM, 2013, pp. 615–622.

[10] 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*, ser. Lecture Notes in Computer Science, 2014, no. 8672, pp. 528–537.

[11] 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*, ser. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2009, no. 193, pp. 435–456.

[12] K. Li, K. Deb, Q. Zhang, and S. Kwong, "Efficient non-domination level update approach for steady-state evolutionary multiobjective optimization," Tech. Rep., 2014. [Online]. Available: www.egr.msu.edu/~kdeb/papers/c2014014.pdf

[13] J. Vuillemin, "A unifying look at data structures," *Communications of ACM*, vol. 23, no. 4, pp. 229–239, 1980.

[14] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[1] https://github.com/mbuzdalov/papers/tree/master/2015-cec-nds

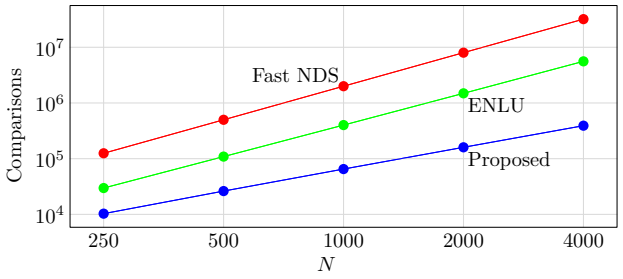

Fig. 7. Number of comparisons for "square"

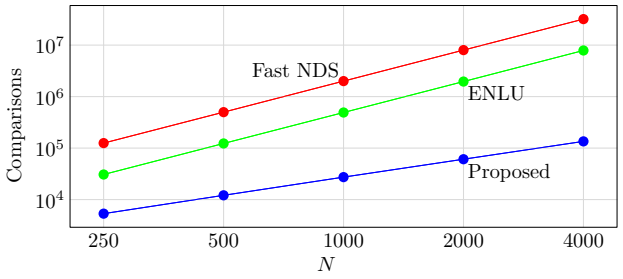

Fig. 8. Number of comparisons for "parallel"

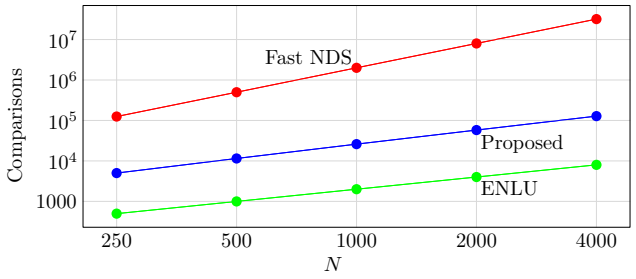

Fig. 9. Number of comparisons for "diag1"

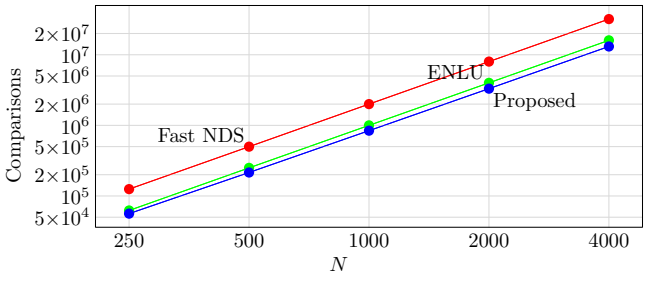

Fig. 10. Number of comparisons for "diag2"

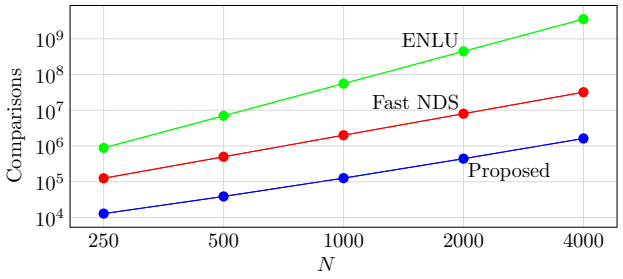

Fig. 11. Number of comparisons for "parper"

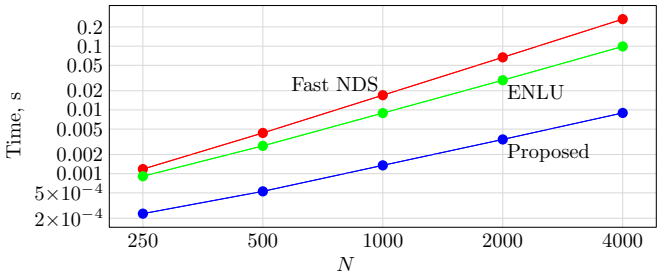

Fig. 12. Wall-clock time for "square"

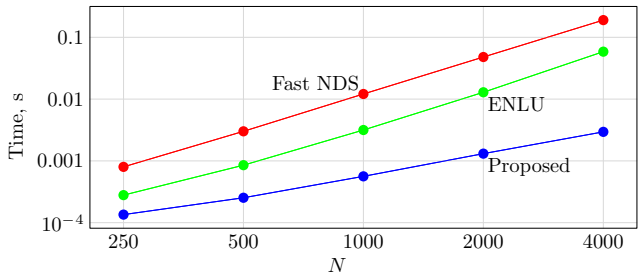

Fig. 13. Wall-clock time for "parallel"

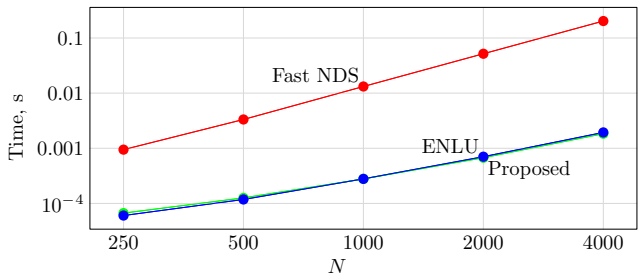

Fig. 14. Wall-clock time for "diag1"

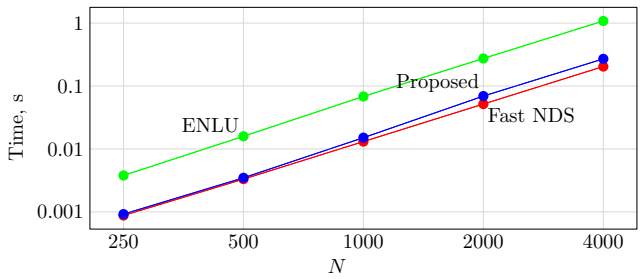

Fig. 15. Wall-clock time for "diag2"

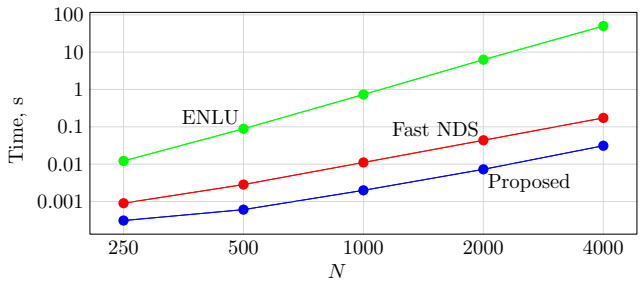

Fig. 16. Wall-clock time for "parper"