Generalized Incremental Orthant Search: Towards Efficient Steady-State Evolutionary Multiobjective Algorithms

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

ABSTRACT

Some of the modern evolutionary multiobjective algorithms have a high computational complexity of the internal data processing. To further complicate this problem, researchers often wish to alter some of these procedures, and to do it with little effort.

The problem is even more pronounced for steady-state algorithms, which update the internal information as each single individual is computed. In this paper we explore the applicability of the principles behind the existing framework, called generalized offline orthant search, to the typical problems arising in steady-state evolutionary multiobjective algorithms.

We show that the variety of possible problem formulations is higher than in the offline setting. In particular, we state a problem which cannot be solved in an incremental manner faster than from scratch. We present an efficient algorithm for one of the simplest possible settings, incremental dominance counting, and formulate the set of requirements that enable efficient solution of similar problems. We also present an algorithm to evaluate fitness within the IBEA algorithm and show when it is efficient in practice.

CCS CONCEPTS

• **Applied computing** → **Multi-criterion optimization and decision-making**; • **Theory of computation** → **Sorting and searching**.

KEYWORDS

Pareto dominance, orthant search, steady-state algorithms, IBEA.

ACM Reference Format:
Maxim Buzdalov. 2019. Generalized Incremental Orthant Search: Towards Efficient Steady-State Evolutionary Multiobjective Algorithms. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326880

1 INTRODUCTION

Software systems for evolutionary computation shall be both easy to use and performant. It is not obvious which of these two points is more important. A software system with a steep learning curve repels new users that wish to try it for their purposes. A slow software system pushes away some of the existing users, which appear to be unsatisfied with the speed. Such users have already invested their time into getting acquainted with the system and into tuning their workflows to work with it, so the amount of frustration, as well as the associated losses, can also be quite high.

The nature of metaheuristics implies that the end users often wish to play with the algorithms, which involves not just tuning their parameters but also altering and replacing and their components, often at the fundamental level. This necessitates open software architectures and requires the components to be easily understandable and manageable. As a result, performance and convenience are typically in conflict.

Some of the parts are less likely to be intercepted by users, as they feel them as monolithic building blocks, which enables their optimization. For instance, the hypervolume indicator [43] is mostly used as is after simple data preparation, such as coordinate normalization or selection of the reference point [2, 20]. This faciliates development of efficient algorithms [21, 24, 33] to this #P-complete problem [5]. Similarly, the end users are unlikely to alter eigendecomposition in the core of the CMA-ES algorithms [18, 19].

Other computationally-demanding subroutines of evolutionary algorithms are altered much more often. For instance, the Pareto-dominance relation, which is used in a large number of evolutionary multiobjective algorithms, is sometimes replaced by epsilon-dominance [25], and the procedure of reference vector creation, which is common to decomposition-based many-objective algorithms [40], is altered in NSGA-III [12] in order to adapt the boundaries in every objective based on the current population state.

To enable efficient implementation of such ever-changing procedures, designing generic algorithmic frameworks is one of the solutions. Already in 2003 a divide-and-conquer scheme is proposed [22] which encapsulated non-dominated sorting [12, 13], archiving of non-dominated solutions [1] and dominance counting [42], all with the complexity of $O(N(\log N)^{K-1})$ or smaller for N points and dimension K. A more recent paper [8] introduced the formalism called *generalized offline orthant search* to cover these problems, as well as computation of the ε-indicator [44], initial fitness assignment for the IBEA algorithm [41] and the variants of the R2 indicator [6, 34], that enabled a single well-optimized implementation, running in $O(N(\log N)^{K-1})$, to serve all the purposes.

The contribution of this paper amounts to investigation of generalized orthant search in the settings common to steady-state evolutionary multiobjective algorithms. We introduce *generalized incremental orthant search*, which appears to be an even richer problem, and study several applications which appear to be the easiest, the hardest and the medium-complexity cases.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326880

2 PRELIMINARIES

In this paper, we will mostly investigate algorithmic problems arising when developing and implementing evolutionary multiobjective algorithms. Furthermore, we are going to operate exclusively with the objective vectors representing fitness values of individuals, so we abstract away from the typical issues related to evolutionary computation, such as solution representations, variation operators and genotype-to-phenotype mapping.

2.1 Basic Definitions and Notation

We consider solving multiobjective problems with the number of objectives $K \geq 2$, also denoted as the *dimension*. Without loss of generality, we assume that the fitness vectors, which we will often refer to as *points*, come from the space $\mathbb{R}^K$, and we will refer to the i-th component of a point, $1 \leq i \leq K$, as the *i-th objective* or the *i-th coordinate*. We assume minimization problems, that is, the smaller the objective, the better.

A point p is said to *strictly dominate* another point q in the Pareto sense, denoted as $p \prec q$, if the following conditions are satisfied:

$$p \prec q \leftrightarrow \begin{cases} \forall i, 1 \leq i \leq K & p_i \leq q_i, \\ \exists j, 1 \leq j \leq K & p_j < q_j. \end{cases}$$

If we remove the second condition, we will get the definition of *weak dominance*, denoted as $p \preceq q$. We will also need the *partially strict* dominance parameterized by the set S of *strict coordinates*:

$$p \prec_S q \leftrightarrow \begin{cases} \forall i, (1 \leq i \leq K) \wedge (i \notin S) & p_i \leq q_i, \\ \forall i, (1 \leq i \leq K) \wedge (i \in S) & p_i < q_i, \\ \exists j, 1 \leq j \leq K & p_j < q_i. \end{cases}$$

2.2 Several Important Algorithms

In this subsection we recall some algorithms that either appear as parts of evolutionary multiobjective algorithms, or are themselves evolutionary multiobjective algorithms, which we will use further.

Dominance counting. The algorithm SPEA2 [42] features a two-phase procedure called *dominance counting*. In the first phase, every non-dominated point is assigned a value called the *dominance strength*, which is equal to the number of other points it dominates. The less points it dominates, the better it is thought to be, as in this case it represents a less-populated region. In the second phase, the dominance strength of every dominated point p is evaluated by summing up the dominance strengths of all non-dominated points that dominate p. Although the reference implementation of SPEA2 uses $\Theta(N^2K)$ algorithms to complete both of the phases, it is shown in [22] that each of these phases can be completed in $O(N(\log N)^{K-1})$, which is also possible to achieve as in [8].

Non-dominated sorting. This procedure was proposed in the multiobjective algorithms NSGA [35] and NSGA-II [13] as one of the ranking procedures. It assigns *ranks* to points using the following logic: let P_i be the part of the population at the i-th iteration, $i \geq 0$ (initially P_0 is the entire population), then the point set R_i (called the i-th *level* or the i-th *layer*) is defined as the non-dominated subset of P_i, e.g.: $R_i = \{p \mid p \in P_i \wedge \nexists q \in P_i : q \prec p\}$, and $P_{i+1} = P_i \setminus R_i$ is the remainder of the population which is processed on the next iteration. The process continues until P_i is empty for some i.

In NSGA [35], this was implemented straightforwardly in time $O(N^3K)$, where N is the number of points, and NSGA-II [13] contained an improved algorithm, called *fast non-dominated sorting*, that required $\Theta(N^2K)$ time and memory. Even with this reduced complexity, it was still an asymptotic bottleneck in NSGA-II. For this reason, as well as because of the huge popularity of NSGA-II and the appearance of more algorithms that used non-dominated sorting [11, 12, 42], a large number of more efficient algorithms appeared. Some of them did not attempt to improve the worst-case performance, but instead concentrated on improving the performance "on average" regarding typical scenarios from evolutionary computation; the best algorithms of this sort are arguably ENS-NDT [17] and Best Order Sort [31, 32]. Others aimed at improving the worst-case performance, which started in [22] with the time complexity of $O(N(\log N)^{K-1})$ and subsequently refined in a number of works, featuring e.g. the algorithm requiring $O(N(\log N)^{K-2} \log\log N)$ time and a word RAM computation model [9]. These works are based on the divide-and-conquer scheme dating back to [23].

Note that in the computational geometry community this problem is known as *finding the layers of maxima* and has been extensively investigated for $K = 3$, resulting in efficient algorithms with complexity as small as $O(N(\log\log N)^2)$ [7, 29]; however, the attempts to generalize these ideas to higher dimensions did not succeed so far.

Incremental non-dominated sorting. A steady-state version of NSGA-II recently received considerable attention and found out to deliver noticeable improvements both on benchmark functions [28] and on certain real-world problems [10] in terms of both fitness function evaluations and the resulting diversity. One of the key differences between it and the classical NSGA-II is that one individual is synthesized and evaluated on each iteration, then it is added back to the population. Unfortunately, a straightforward implementation requires re-running non-dominated sorting, already a bottleneck, on each individual insertion, which influences the running time negatively even with quite expensive fitness functions.

A number of algorithmic approaches have been developed to replace the conventional offline non-dominated sorting with a data structure to maintain incremental changes, e.g. insertion of a newly computed individual and removal of the worst individual. The approach called *efficient non-dominated level update* [26, 27] modifies fast non-dominated sorting for this purpose, which does not improve the worst-case insertion time but significantly improves performance on the real data. Another approach [38] is particularly efficient in the two-dimensional case (the worst-case insertion time is $O(N)$, which drops to $O(\log N)$ in typical evolutionary computation scenarios), but has also been generalized to arbitrary dimensions [39] with the insertion complexity of $O(N(\log N)^{K-2})$.

Note that even with these improvements the complexity of inserting N points into the data structure for incremental non-dominated sorting is much higher than doing it once with an offline sorting. It is not known as of now whether it is a fundamental property, or just a lack of attention from algorithmists.

ε-indicator. In [44] it was proposed to compare the performance of evolutionary multobjective algorithms by means of *binary indicators*. An *indicator* is a function that takes one or more point

sets and returns a single number. The well-known hypervolume indicator [43] is an example of an unary indicator. One of the binary indicators proposed in [44] is the *additive ε-indicator* which takes two point sets, the moving set M and the fixed set F, and computes the following value:

$$\varepsilon(M,F) = \max_{m\in M}\min_{f\in F}\max_{i=1}^{K}(f_i - m_i),$$

which, informally speaking, tells that $\varepsilon(M,F)$ is the minimum value that shall be subtracted from each point from M such that every point from F gets dominated by at least one point from M.

It may seem unlikely to enhance the performance of three tight loops that evaluate the ε-indicator in $\Theta(N^2K)$, however, an algorithm was proposed in [37] to compute it in time $O(KN(\log N)^{K-2})$ using K runs of a divide-and-conquer algorithm similar to the one used for non-dominated sorting, which appeared to be efficient in practice for small K. This work served as a basis for developing the concept of generalized offline orthant search [8].

Indicator-Based Evolutionary Algorithm. In [41] it was proposed to use binary indicators not only for assessment of performance of evolutionary multiobjective algorithms, but for the optimization itself. The indicator-based evolutionary algorithm, or IBEA, can use any binary indicator I which is *dominance-preserving*, that is, if $a \prec b$, then $I(\{a\},\{b\}) < I(\{b\},\{a\})$, and for any other point c it holds that $I(\{c\},\{a\}) \geq I(\{c\},\{b\})$. Note that the ε-indicator, as well as the binary version of the hypervolume also proposed in [44], satisfy these conditions. With the use of such an indicator I, IBEA assigns its own fitness values to an arbitrary set of points P in the following way:

$$F(p) = -\sum_{q\in P\setminus\{p\}} e^{-I(\{q\},\{p\})/\kappa}.$$

IBEA maintains a population of individuals P of size N. On every iteration it synthesizes N more individuals using variation operators, adds to this population, and then removes the worst individuals (the ones with the smallest fitness) one by one until the population size gets back to N. Once an individual is removed, fitness is recomputed for the remaining individuals by removing the corresponding addends, amounting in $\Theta(NK)$ operations per individual, assuming the ε-indicator is used.

While the initial fitness assignment, which can be run after all children are evaluated, was shown in [8] to be possible in time $O(KN(\log N)^{K-2})$, applying the same approach during the removal phase seems to be impossible. In Section 4.3 of this paper we propose an algorithm that improves the running time of this operation, but, unfortunately, only by a constant factor.

2.3 Several Important Data Structures

This subsection explains the basics of two data structures, which we will subsequently need in the proposed algorithms.

k-d tree. This is a data structure which efficiently partitions the k-dimensional space using hyperplanes formed by fixing a particular coordinate to a certain value [3]. It accelerates various queries, most often from the practical point of view, by ignoring the unrelated sections of the space whenever possible. Its name is a set expression, which appeared as a short form of a term *k-dimensional tree*.

The *k-d* tree consists of leaves and internal nodes. The leaves contain k-dimensional points (typically at most a constant number of them), often augmented with problem-dependent information. The internal nodes contain the coordinate index i, the value of the coordinate v, and the pointers to the subtrees which contain only the points with the i-th coordinate respectively smaller and greater than v. One can also store some subtree-related information (such as e.g. the sum of values associated with all the points in the subtree) in the internal nodes, which improves both theoretical and practical performance in certain applications.

Adding a new point generally requires the time proportional to the height of the tree. If a leaf contains too many points after insertion of the new point, it needs to be split. The shape of the tree depends on the strategy which chooses the coordinate index and the value at split events, but randomized choices generally work well, and the height of the tree is roughly logarithmic to the number of contained points. There is no particular time bounds for an arbitrary query, but many of them are still logarithmic.

In the context of evolutionary computation, *k-d* trees are used in the ENS-NDT algorithm for non-dominated sorting [17], as well as in several algorithms for maintaining the non-dominated set of points [14, 16]. The space complexity of a *k-d* tree is $O(N)$.

K-dimensional range query tree. This is a close relative of the multi-dimensional Fenwick tree [15] that supports dynamic insertion and deletion of points. A conventional Fenwick tree stores N numbers at indices $1, 2, \ldots, N$, which it can update in $O(\log N)$ time, and it also can retrieve the sum on indices $[1..i]$ also in $O(\log N)$ time. It can be generalized to an arbitrary dimension K, such that the structure stores K-dimensional points as well as the associated values, with update and query times of $O((\log N)^{K-1})$. while the memory complexity remains $O(N)$.

To support dynamic insertion and deletion of points, which would correspond to indices of the Fenwick tree, a randomized splitting scheme similar to the one used in *k-d* trees is typically used, with the exception that the data structure consists of multiple levels of trees, where each level corresponds to a dedicated coordinate, and every node of the level i also contains a tree of the level $i-1$ that contains all the points from the subtree. This makes it possible to insert and remove points in time $O((\log N)^{K-1})$, as well as execute orthant sum queries in the same time, but the space requirements increase and reach $O(N(\log N)^{K-1})$. Additional information stored in the nodes also enables modifications that, as effectively observed, simultaneously add some value to all the points within a certain product of ranges also in time $O((\log N)^{K-1})$. This family of data structures is known as *range query trees* [4].

2.4 Generalized Offline Orthant Search

In [8] a generalization of a number of existing algorithms, which were based on the same divide-and-conquer scheme as in [22], was proposed to be able to embrace them all with a single piece of code. *Generalized offline orthant search* was defined as follows:

- A collection of points P of size N from the K-dimensional space is given. We distinguish different points with coinciding coordinates, so we assign to them indices i, $1 \leq i \leq N$.
- Each point can independently be a *data point* and a *query point*. A set D contains the indices of data points, and a set Q

contains the indices of query points. Note that $D \cap Q$ need not be empty, and in some applications $D = Q = [1..N]$.
- The partially strict dominance relation $\prec_S$, defined by a set $S \subseteq [1..K]$ as in Section 2.1, is used.
- A commutative monoid $\mathcal{K}$, having the neutral element $\square$ and the aggregation operation $\oplus : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$, is given, which is the domain of the values associated with the data points, and also of the query results associated with the query points.
- A mapping $V : D \to \mathcal{K}$, that defines the values associated with the data points, is initially defined, and it can be subsequently updated while solving the problem.
- The problem is to compute, for each query point with index q, an aggregation of values associated with all data points which dominate $P[q]$ according to the $\prec_S$ relation:

$$A(q) = \bigoplus_{\substack{d \in D \\ P[d] \prec_S P[q]}} D(d).$$

- Once $A(q)$ is found for a query point q, it is possible to update, in an arbitrary way, the mapping V for a constant number of data points d, such that $Q[q] \prec_S d$. This is captured by defining the *query-to-data* operation $Q2D_q : (D \to \mathcal{K}) \times \mathcal{K} \to (D \to \mathcal{K})$, associated with a query point q, which takes the old mapping V and the query result $A(q)$ and produces the new mapping V by updating a few entities. In most applications, $Q2D_q$ is the identity function on the second argument (i.e. it does nothing). In some applications, it is associated only with those query points, which are also data points, and updates only their own data values.
- The order of computing the queries for the above reason, and the problem definition requires that an answer to a query may be computed only once all necessary data points are completely evaluated and will never change. This means that any two queries $q_1 \prec_S q_2$ must be answered in the order of dominance. As the $\prec_S$ relation is acyclic, there exists an order of query answering which satisfies this requirement.

This complicated definition differs from the conventional problem of answering orthant minimum queries in the following points:

- It enables on-the-fly updates of the data points in order to capture non-dominated sorting and the similar problems.
- It uses a configurable dominance relation, which is heavily used, for instance, in the reduction of IBEA's fitness assignment to ensure correctness for all inputs.

A divide-and-conquer algorithm with $O(N(\log N)^{K-1})$ time and $O(NK)$ memory was proposed in [8] along with an implementation in Java which enables defining new configurations of generalized offline orthant search by just extending a few abstract methods of a configuration class, enabling a cheap reduction of many problems to be solved by a well-optimized algorithmic core. This is similar to reduction of complicated problems to solvers, such as SAT solvers [36, Chapter 2], common to other computer science fields.

3 GENERALIZED INCREMENTAL ORTHANT SEARCH: THE PROBLEM DEFINITION

In this section we define the problem of *generalized incremental orthant search* closely following the previous section.

The following components of the definition are similar to the ones that constitute generalized offline orthant search:

- The domain for values $\mathcal{K}$, which is a commutative monoid with the aggregation operation $\oplus : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ and the neutral element $\square$. This time, we may also profit from a *negation operation* $\overline{\cdot} : \mathcal{K} \to \mathcal{K}$.
- The partial dominance relation $\prec_S$.
- The collections of data points D and query points Q. For a dynamic data structure it is not easy to maintain contiguous indices for points that can be added or removed. However, we still need to distinguish points that are coordinatewise equal but appeared by different queries.
- The data values associated with the data points $V : D \to \mathcal{K}$, as well as the query answers maintained for every query $A : Q \to \mathcal{K}$.
- The query-to-data operation $Q2D$ that updates a constant number of data points once a query is evaluated: $Q2D_q : (D \to \mathcal{K}) \times \mathcal{K} \to (D \to \mathcal{K})$.

However, it is not enough to stop with this set of components. In the offline search, answers to the queries, as well as the updated values associated with the data points, can be directly used with the $O(1)$ lookup complexity once the algorithm terminates, so we do not have to account for operations that consume them. The situation is different in the incremental case. For instance, incremental non-dominated sorting is used within the steady-state NSGA-II in conjunction with crowding distance, and finding the individual with the smallest crowding distance in the last level may now be of the same complexity order as insertion of one element, so special tricks are needed to maintain a data structure that can efficiently locate the worst individual [30].

Another example is the IBEA's fitness assignment procedure. As we will see later, we could achieve the $O(K(\log N)^{K-2})$ time for insertion/removal of a point, as well as for reading the answer to a query. However, after modifications we need to identify the point which has the smallest sum of the answers of associated queries, which would require scanning all query points and result in the increase of the overall complexity, so this approach is not useful for the intended application.

For this reason, we need to introduce an additional component, the *query selector*, which can be asked to find a query point that satisfies some requirement. This component can as well be associated with a *system* of generalized incremental orthant search instances instead of each single instance. For this reason, we do not specify its function type, but merely signify its existence.

The following operations need to be supported by a data structure implementing generalized incremental orthant search:

- adding a data point;
- removing a data point;
- adding a query point;
- removing a query point;
- retrieving the answer for a query point;
- retrieving the value associated with a data point;
- asking the query selector.

The first four operations are modification operations. The data structure needs to ensure that once an operation is finished, the values associated with each data point, as well as the answers to

all queries, are exactly the same as if generalized *offline* orthant search was called on the same inputs. That is, the state of the data structure shall not depend on the order of modification operations, with the sole exception of resolving ties that may appear in the query selector.

In the presence of the non-trivial query-to-data operation, it is not easy to maintain this property. This is why incremental non-dominated sorting, which is an instance of the generalized incremental orthant search with the following parameters:

- $\mathcal{K} = \mathbb{N}$, $\square = 0$, $\oplus = \max$, $S = [1..K]$,
- all points are both query points and data points,
- the initial value of a data point is $\square$,
- $Q2D_p$ performs $V(p) \leftarrow A(p) + 1$,
- no query selector is used,

is noticeably harder than the conventional offline non-dominated sorting, as some of the updates tend to propagate for long distances through the data structure.

On the other hand, we can get rather good bounds for the settings with a trivial query-to-data operation, no query selectors and the presence of the negation operator using range query trees.

Theorem 3.1. *Generalized incremental orthant search is possible to implement with $O((\log N)^{K-1})$ time required to perform all modification operations, as well as for retrieving the answer for a query point, within $O(N(\log N)^{K-1})$ memory.*

Proof. We use two K-dimensional range query trees with values from $\mathcal{K}$ and the aggregation operation $\oplus$. The first one stores the data points and is used to produce answers to the queries, in $O((\log N)^{K-1})$ each, every time a new query arrives.

The second one stores the changes in the answers to the queries resulting from adding and removing the data points. It stores negated query points along with their corresponding answers (initialized by the answer at the time of the query addition), and also features additional values in the internal nodes that need to be added to the entire subtree of that node. Addition (or removal) of the data point results in adding (or subtracting) its value from the orthant range originating at the negated data point, which takes $O((\log N)^{K-1})$ time. When an answer to a query is requested, the value associated with that query is summed up with all the additional values in the internal nodes on the way to the root. When a new query is inserted, the additional values in the internal nodes on the way to the location of that query from the root are pushed down to the children of these nodes, which retains the correctness of the data structure and allows accommodation of the new query. □

While using range query trees seems promising, their benefits quickly disappear once advanced features, such as non-trivial query-to-data operations or query selector, need to be implemented. This is why we limit ourselves by implementations based on *k-d* trees in the experimental sections of this paper.

The following sections investigate the easiest case of generalized incremental orthant search with the setting from Theorem 3.1 (Section 4.1, experimental), the hardest case where nothing can be done more efficiently than from scratch (Section 4.2, theoretical), and the important case of IBEA's fitness assignment with a constant-factor improvement compared to the naive implementation (Section 4.3, experimental).

4 APPLICATION EXAMPLES

4.1 An Easy Example: Dominance Counting

For an easy example, we considered the simplified version of the dominance counting setup. In this example, we will have two separate sets of points: one for data points that would represent the non-dominated points and would carry some value uniformly randomly sampled from $[0; 1]$, and one for query points that would represent the dominated points and would request the sum of the data points that dominate them.

We consider dimensions $K \in \{2, 3, 4, 5, 7, 10\}$ and the characteristic problem sizes $N = \lfloor 10^{n/2} \rfloor$ for $n \in [2..7]$, so that the maximum N is 3162. We generate both the query points and the data points using three types of generators:

- "cube" (each coordinate is sampled uniformly and independently from $[0; 1]$);
- "plane" (coordinates from $[2..K]$ are sampled as above, the first coordinate is equal to one minus the sum of all other coordinates; all such points reside on a hyperplane and thus are non-dominated);
- "line" (a single value for all coordinates of a point is sampled uniformly from $[0; 1]$).

We compare two algorithms: the naive one which compares the added/removed point for dominance with all points from the opposite class, and the one which uses two *k-d* trees as follows:

- the first *k-d* tree stores the data points and produces answers to the queries when they arrive;
- the second *k-d* tree stores the (negated) query points, and on each added/removed data point the algorithm runs an orthant, made from a negated data point, on the second tree, and updates all the query points which were found to be dominated by the negated data point.

As in both algorithms the time to get the answer to a query is $O(1)$ and the data values do not change, we only measure the time needed to perform modification queries. The protocol for measuring running times is as follows:

- The total number of queries for each test instance is $4N$.
- The current number of data points $N_d \le N$, as well as of query points $N_q \le N$, is tracked.
- The modifications are generated with the following randomized procedure:
  - With probability $\frac{1}{2}$, the data point set is modified. With probability $(1 - 2^{N_d - N})/(1 - 2^N)$, a new data point is randomly generated and added; otherwise, a randomly chosen existing data point is deleted.
  - With probability $\frac{1}{2}$, the query point set is modified. With probability $(1 - 2^{N_q - N})/(1 - 2^N)$, a new query point is randomly generated and added; otherwise, a randomly chosen existing query point is deleted.
- Three independently generated datasets are used for each configuration.
- The algorithms are implemented in Scala. The running time is measured by the Java Microbenchmark Harness tool with five forks of the Java Virtual Machine.

The experimental results can be seen in Fig. 1–3. One can see that, apart from the smallest problem sizes, the implementation based on the *k-d* tree outperforms the naive implementation, and, based on the observed difference of the slopes, the speed-up is asymptotic for the "cube" and the "plane" datasets, while in the "line" dataset the performance is of the same order, which is expected as the *k-d* tree has to traverse each dominating point.

Again matching the expectations, the speed-up on the "cube" dataset increases with the dimension as fewer points dominate each other for larger K (the speed-up at $N = 3162$ is 2.8 for $K = 2$ and 5.9 for $K = 10$). On the "plane" dataset the speed-ups are better and slightly decrease as K grows (28.8 for $K = 2$, 13.6 for $K = 10$). The speed-ups on the "line" dataset are less than 2x for $K = 2$ and converge to 1x when K grows.

4.2 A Hard Artificial Example: Not Easier than Computing from Scratch

In this section, we will exemplify a situation when, even without the query selection mechanism, generalized incremental orthant search will have to spend $O(N(\log N)^{K-1})$ time per *each* update out of a sequence of $\Theta(N)$ updates.

For this we will develop a special monoid $\mathcal{K}$. An element of $\mathcal{K}$ will be an integer consisting of $2m$ bits, where $m > 2\log_2 N$. The lower m bits represent the *value part* of the element, while the upper m bits represent the *salt part*. Beforehands, 2^m integers consisting of 2^m bits are generated using a cryptographically strong keystream. The $\oplus$ operation will behave as follows: it chooses as s the *largest* salt part of the arguments, then it applies the bitwise exclusive OR operation between the s-th crypto integer and the value part for each of the arguments, then it multiplies them together and takes the result modulo 2^m to form the value part of the result. The salt part of the result is chosen to be s. The query-to-data operation copies the salt part of the answer to the query to the value of the coinciding data point.

After that, to demonstrate the declared properties, the first N points (each is both a query point and a data point) are added arbitrarily, ensuring that the salt part is equal to zero. The following N points are inserted in such a way that the i-th of these points ($i \geq 1$) has the salt part equal to i, and the point itself dominates all other points in the data structure.

The described procedure ensures that, on each insertion of a point from the second pool, all data values are simultaneously replaced with new pseudo-random values. The algorithm has to either recompute every query from scratch, or to maintain the possible update scenarios for $\Omega(N^2)$ different salts. An arbitrary combination of these strategies ensures the running time of at least the time needed to recompute the queries from scratch, that is, $O(N(\log N)^{K-1})$.

4.3 An Attempt to Accelerate IBEA's Fitness Assignment

The reduction of the IBEA's fitness assignment to the generalized incremental orthant search follows the principle detailed in [8], which we give only shortly for the space reasons. We maintain K instances of the orthant search problem, one for each coordinate, such that in the i-th of these instances the points that dominate the point p will determine the ε-indicator by the difference of their i-th coordinates. In fact, the i-th problem is an orthant search problem defined over *projected* points $(x_1 - x_i, x_2 - x_i, \ldots, x_K - x_i)$, where the identity-zero $x_i - x_i$ coordinate is removed.

To account for the points where the differences between the coordinates may coincide, we use the partially strict dominance with the set $S = [i; K]$ in the i-th coordinate, so that each pair of points is found to be dominated in a single instance only.

The experimental setup is mostly similar to the one in Section 4.1, except that there was only one type of the event, namely, generation of a random point according to the chosen generator. Once the number of points stored in the data structure was equal to $2N$, the worst points were removed from the data structure, one by one, until the size gets back to N, closely mimicking the IBEA's logic.

As finding the worst point is performed by iterating over the answer values, the lower bound on the entire runtime is $\Omega(N^2)$ and the upper bound is $O(N^2K)$, thus we may hope only for a constant-factor improvement. The results of measurements are presented in Fig. 4–6. One can see that for $K \leq 4$ there is indeed an improvement by a constant factor of roughly 1.8, starting from some problem size. The greater K is, the worse is the performance of orthant search. The only exception is the "line" dataset, where the orthant-based implementation is always slightly faster, which can be explained by the fact that in this case all points in each of the K orthant search instances simply coincide. One of the reasons for the speed-up may be that the proposed scheme performs $\Theta(K)$ exponentiations for each individual, while the original scheme performs $\Theta(N)$ of them.

5 CONCLUSION

We proposed generalized incremental orthant search, a formalism which may be a basis of algorithmic frameworks intented to be efficient "solvers" for various components of evolutionary multiobjective algorithms. Unfortunately, unlike generalized offline orthant search, the variety of possible problem configurations is much higher in our case, and it seems necessary to maintain at least several codebases for different subsets of the generic problem.

In particular, we have investigated the easiest problem class, similar to dynamic dominance counting, and obtained both good theoretic complexity bounds and noticeable asymptotic speed-ups in practice, reaching up to 28x on the investigated problem sizes. The medium-complexity instances may enjoy only a very small asymptotic speed-up compared to the naive approach, such as incremental non-dominated sorting, or just a constant speed-up, like in our experimental study on the IBEA's fitness assignment procedure. We also exemplified the hardest setup, where a nearly arbitrary consecutive number of operations would require at least the same time as needed to re-evaluate everything from scratch.

The source code of the experiments is available on GitHub[1]. The particular version used to produce the presented results is accessible under the `v2019.04` release tag[2].

ACKNOWLEDGMENTS

This research was supported by the Russian Scientific Foundation, agreement No. 17-71-20178.

[1] https://github.com/mbuzdalov/incremental-orthants
[2] https://github.com/mbuzdalov/incremental-orthants/tree/v2019.04

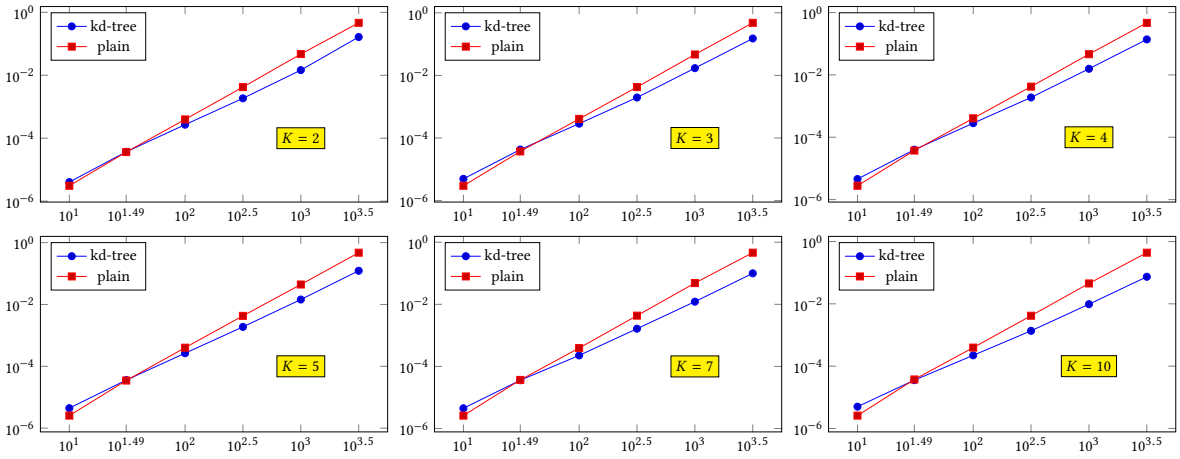

Figure 1: Running times for simplified dominance counting, in seconds, the "cube" dataset generator

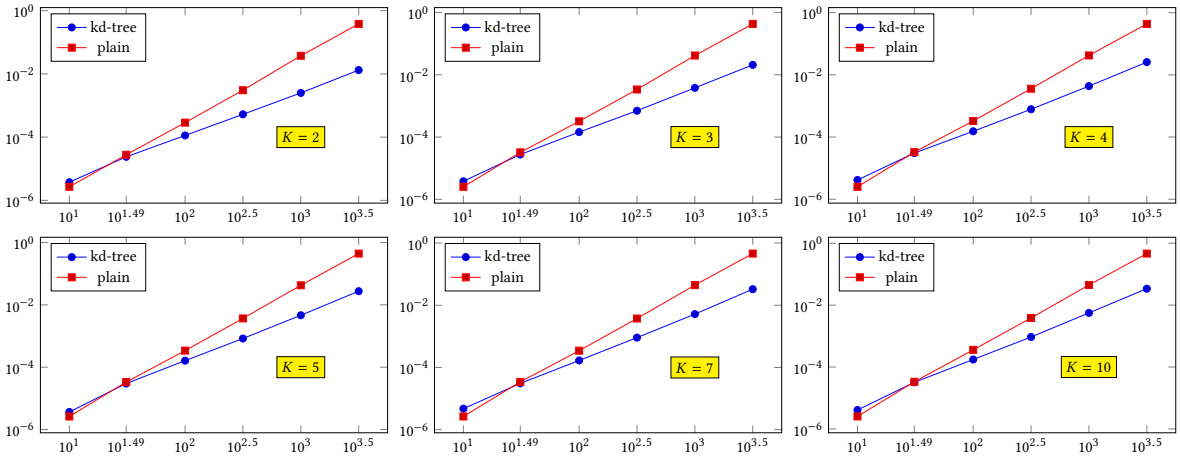

Figure 2: Running times for simplified dominance counting, in seconds, the "plane" dataset generator

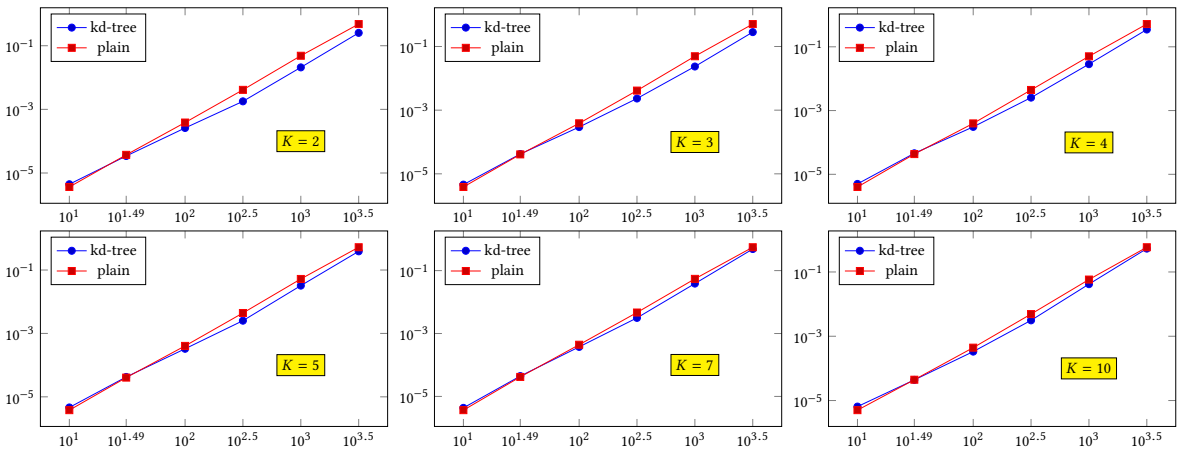

Figure 3: Running times for simplified dominance counting, in seconds, the "line" dataset generator

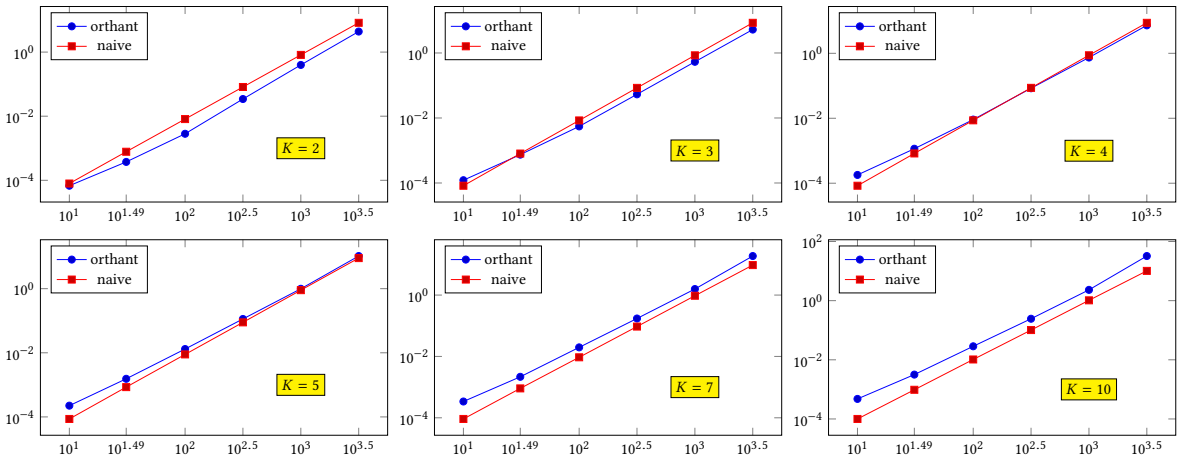

Figure 4: Running times for IBEA's fitness assignment, in seconds, the "cube" dataset generator

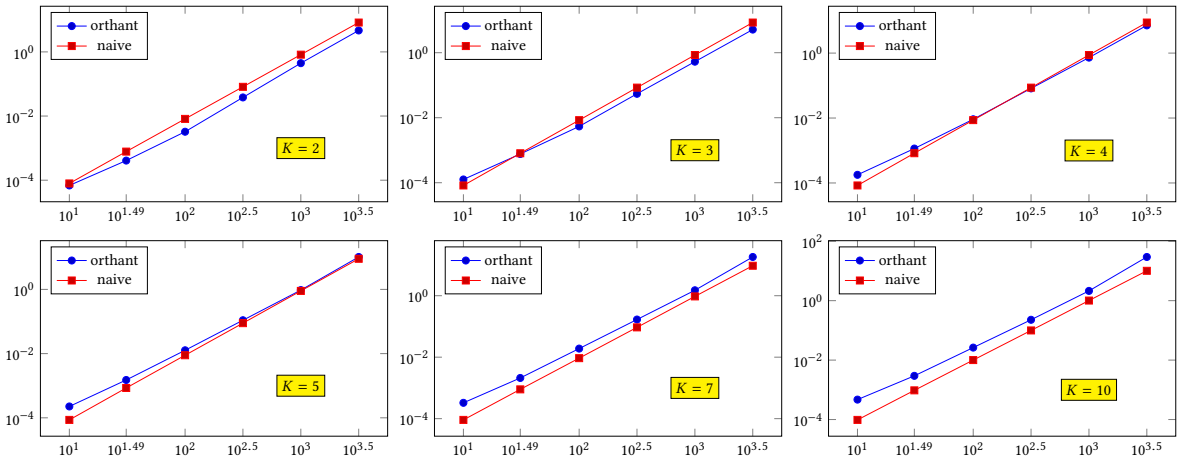

Figure 5: Running times for IBEA's fitness assignment, in seconds, the "plane" dataset generator

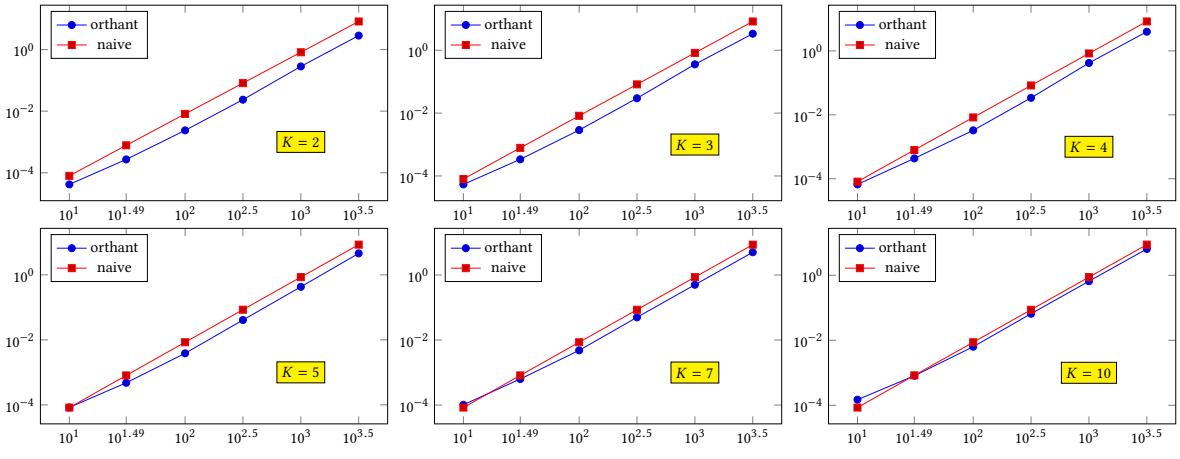

Figure 6: Running times for IBEA's fitness assignment, in seconds, the "line" dataset generator

REFERENCES

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

[2] Anne Auger, Johannes Bader, Dimo Brockhoff, and Eckart Zitzler. 2012. Hypervolume-based multiobjective optimization: Theoretical foundations and practical implications. *Theoretical Computer Science* 425 (2012), 75–103.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Jon Louis Bentley. 1980. Multidimensional Divide-and-conquer. *Communications of ACM* 23, 4 (1980), 214–229.

[5] Karl Bringmann and Tobias Friedrich. 2009. Approximating the Least Hypervolume Contributor: NP-Hard in General, But Fast in Practice. In *Proceedings of the 5th International Conference on Evolutionary Multi-Criterion Optimization*. 6–20.

[6] Dimo Brockhoff, Tobias Wagner, and Heike Trautmann. 2012. On the properties of the R2 indicator. In *Proceedings of Genetic and Evolutionary Computation Conference*. 465–472.

[7] Adam L. Buchsbaum and Michael T. Goodrich. 2004. Three-Dimensional Layers of Maxima. *Algorithmica* 39 (2004), 275–286.

[8] Maxim Buzdalov. 2018. Generalized offline orthant search: One code for many problems in multiobjective optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*. 593–600.

[9] Maxim Buzdalov. 2019. Make Evolutionary Multiobjective Algorithms Scale Better with Advanced Data Structures: Van Emde Boas Tree for Non-Dominated Sorting. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*. Number 11411 in Lecture Notes in Computer Science. 66–77.

[10] Alexander W. Churchill, Phil Husbands, and Andrew Philippides. 2013. Tool Sequence Optimization using Synchronous and Asynchronous Parallel Multi-Objective Evolutionary Algorithms with Heterogeneous Evaluations. In *Proceedings of IEEE Congress on Evolutionary Computation*. 2924–2931.

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

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

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

[14] M. Drozdik, Y. Akimoto, H. Aguirre, and K. Tanaka. 2015. Computational cost reduction of nondominated sorting using the M-front. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 659–678.

[15] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24, 3 (1994), 327–336.

[16] Tobias Glasmachers. 2017. A Fast Incremental BSP Tree Archive for Non-dominated Points. In *Proceedings of Evolutionary Multiobjective Optimization*. Number 10173 in Lecture Notes in Computer Science. 252–266.

[17] Patrik Gustavsson and Anna Syberfeldt. 2018. A New Algorithm Using the Non-dominated Tree to Improve Non-dominated Sorting. *Evolutionary Computation* 26, 1 (2018), 89–116.

[18] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. 2003. Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES). *Evolutionary Computation* 11, 1 (2003), 1–18.

[19] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9 (2001), 159–195.

[20] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. 2018. How to Specify a Reference Point in Hypervolume Calculation for Fair Performance Comparison. *Evolutionary Computation* 26, 3 (2018), 411–440.

[21] Andrzej Jaszkiewicz. 2018. Improved quick hypervolume algorithm. *Computers & Operations Research* 90 (2018), 72–83.

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

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

[24] Renaud Lacour, Kathrin Klamroth, and Carlos M. Fonseca. 2017. A Box Decomposition Algorithm to Compute the Hypervolume Indicator. *Computers & Operations Research* 79 (2017), 347–360.

[25] Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. 2002. Combining Convergence and Diversity in Evolutionary Multi-Objective Optimization. *Evolutionary Computation* 10, 3 (2002), 263–282.

[26] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Qiang Zhang. 2017. Efficient Non-domination Level Update Method for Steady-State Evolutionary Multiobjective Optimization. *IEEE Transactions on Cybernetics* 47, 9 (2017), 2838–2849.

[27] Sumit Mishra, Samrat Mondal, and Sriparna Saha. 2017. Improved solution to the non-domination level update problem. *Applied Soft Computing* 60 (2017), 336–362.

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

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

[30] Niyaz Nigmatullin, Maxim Buzdalov, and Andrey Stankevich. 2016. Efficient Removal of Points with Smallest Crowding Distance in Two-dimensional Incremental Non-dominated Sorting. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*. 1121–1128.

[31] Proteek Chandan Roy, Kalyanmoy Deb, and Md. Monirul Islam. 2019. An Efficient Nondominated Sorting Algorithm for Large Number of Fronts. *IEEE Transactions on Cybernetics* 49, 3 (2019), 859–869.

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

[33] Luis M. S. Russo and Alexandre P. Francisco. 2014. Quick Hypervolume. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014).

[34] Ke Shang, Hisao Ishibuchi, Min-Ling Zhang, and Yiping Liu. 2018. A new R2 indicator for better hypervolume approximation. In *Proceedings of Genetic and Evolutionary Computation Conference*. 745–752.

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

[36] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter (Eds.). 2007. *Handbook of Knowledge Representation*. Vol. 1. Elsevier Science.

[37] Andrey Vasin and Maxim Buzdalov. 2016. A Faster Algorithm for the Binary Epsilon Indicator Based on Orthant Minimum Search. In *Proceedings of Genetic and Evolutionary Computation Conference*. 613–620.

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

[39] Ilya Yakupov and Maxim Buzdalov. 2017. Improved Incremental Non-dominated Sorting for Steady-State Evolutionary Multiobjective Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*. 649–656.

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

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

[42] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Proceedings of the EUROGEN'2001 Conference*. 95–100.

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

[44] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. 2003. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation* 7, 2 (2003), 117–132.