Towards Large-Scale Multiobjective Optimisation with a Hybrid Algorithm for Non-dominated Sorting

Margarita Markina and Maxim Buzdalov(✉)

ITMO University, 49 Kronverkskiy prosp., Saint-Petersburg 197101, Russia
margaritam2706@gmail.com, mbuzdalov@gmail.com

Abstract. We present an algorithm for non-dominated sorting that is suitable for large-scale multiobjective optimisation. This algorithm is a hybrid of two previously known algorithms: the divide-and-conquer algorithm initially proposed by Jensen, and the non-dominated tree algorithm proposed by Gustavsson and Syberfeldt.

While possessing the good worst-case asymptotic behaviour of the divide-and-conquer algorithm, the proposed algorithm is also very efficient in practice. In our experimental study it is shown to outperform both of its parents on the majority of problem instances, both sampled uniformly from a hypercube and having a single front, with as large as 10^6 points and up to 15 objectives.

Keywords: Multiobjective optimisation · Non-dominated sorting
Large-scale optimisation

1 Introduction

Many real-world optimisation problems are inherently multiobjective, that is, they require maximizing or minimizing not a single objective, but several ones, which often conflict with each other. For this reason, there are typically many optimal solutions which are incomparable and trade one objective for another. Even in the conditions that only one of these solutions must be chosen, this choice is often advised to be done lately, as the acquired knowledge of the problem can influence the preferences of the decision maker [1].

According to the tutorial [1], most general-purpose evolutionary multiobjective algorithms that do not try to incorporate the prior knowledge or user preferences belong to three categories: Pareto-based [5–7,26], indicator-based [25], and decomposition-based [23] algorithms.

In turn, Pareto-based algorithms can be classified by how they rank or select solutions. Some of them maintain an archive of non-dominated solutions [3,5,13], others perform non-dominated sorting [6,7], use domination count [9] or domination strength [26] to assign fitness values. In this paper, we consider non-dominated sorting, as many popular algorithms rely on this procedure [6,7].

A. Auger et al. (Eds.): PPSN 2018, LNCS 11101, pp. 347–358, 2018.
https://doi.org/10.1007/978-3-319-99253-2_28

1.1 Non-dominated Sorting: Definition and Algorithms

From now on we assume, without loss of generality, that we need to minimize all objectives. We also explicitly state that in this paper we consider only the objective space and ignore the existence of decision variables and the questions of genotype-to-phenotype mapping. Throughout the paper, we denote as M the number of objectives.

To define non-dominated sorting, we first need to introduce the Pareto dominance relation. A point p is said to *dominate* a point q, denoted as $p \prec q$, if for every objective index i, $1 \le i \le M$, it holds that $p_i \le q_i$, and there exists an index j such that $p_j < q_j$.

Non-dominated sorting assigns *ranks* to solutions from the solution set P in the following way: every solution from P that is not dominated by any other solution from P gets rank 0, and every solution which is dominated by at least one solution of rank i gets rank $i+1$. A set of all points having the same rank is often called a *front*, a *level* or a *layer*. In the work where this procedure was originally proposed [20], it was performed in $O(N^3M)$, where N is the population size. This was later improved to be $O(N^2M)$ in a subsequent work that introduced the famous NSGA-II algorithm [7].

In NSGA-II, non-dominated sorting determines the computational complexity of a single iteration, as all other parts of an iteration scale better as N grows. This poses a problem either when fitness evaluation and variation operators are cheap, or when the population size N is large. As a result, there is quite a number of works dedicated to reduction of either theoretical complexity or practical running times of non-dominated sorting. Due to space limitations, we cannot consider each work in detail, nor can we cite all of them, so we just briefly describe the two prevailing directions.

The first direction aims at developing algorithms that work efficiently on inputs common to evolutionary multiobjective optimisation, but their worst-case time is still $\Omega(N^2M)$. A remarkable number of papers belongs to this direction [8,11,16,18,22,24], where most of the algorithms have $\Theta(N^2M)$ worst-case complexity, while Deductive Sort [16] can be forced to run in $\Theta(N^3M)$ time. Among these, the best performing algorithms to date are Best Order Sort [18] and the ENS-NDT algorithm [11].

The second direction tries to reduce not only the running times, but also the computational complexity. Jensen [12] was the first to adapt the earlier result of Kung et at. [14], who solved the problem of finding non-dominated solutions in $O(N(\log N)^{\max(1,M-2)})$, to non-dominated sorting. This algorithm has the worst-case complexity of $O(N(\log N)^{M-1})$. However, this algorithm could not handle coinciding objective values, which was later corrected in subsequent works [2,10].

We shall also note that in a different community, where this problem is called *layers of maxima*, an algorithm for $M = 3$ was found [17], whose complexity is $O(N(\log\log N)^2)$ with the use of randomized data structures, or $O(N(\log\log N)^3)$ for deterministic ones. Whether this algorithm is useful in practice is still an open question.

Finally, we should mention our own recent work [15], where we tried to unify the benefits of the two directions above under the cover of a single algorithm. Our current paper builds on some of the insights of that paper and pushes these ideas towards a new level of quality.

1.2 Our Motivation and Contribution

Apart from a purely fundamental desire to develop efficient algorithms for hard problems, our research is motivated by a very important practical problem: the *multiobjective in-core fuel management optimisation* problem, instances of which needs to be solved during the functioning of a nuclear reactor. This problem is a hard combinatorial optimisation problem, the solutions of which need to optimise a number of contradicting objectives, such as the power received from the reactor, the amount of neutrons flying out from the reactor and more.

In a multiobjective setting, this problem attracted significant attention in the recent years. Several approaches used in practice use algorithms that employ non-dominated sorting. The reader is directed to the dissertation of Evert Schlünz for further reading [19]. An application of simulated annealing to this problem recommended numbers of samples up to 10^5 already in 1995 [21], which become population sizes in multiobjective settings and can nowadays rise up to 10^6.

In this paper, we consider hybridising the divide-and-conquer approach, initially proposed by Jensen [12] and subsequently refined by Fortin et al. [10] and Buzdalov and Shalyto [2], and the recently proposed ENS-NDT approach by Gustavsson and Syberfeldt [11]. The latter algorithm is used to solve subproblems, which are created by the divide-and-conquer algorithm and have small enough sizes. This particular scheme resembles the production-ready implementations of the mergesort algorithm, which delegate small sub-arrays to the insertion sort.

In the case of non-dominated sorting, however, the subproblems are not completely equivalent to the initial non-dominated sorting problem. The straightforward adaptation of the ENS-NDT algorithm to solving these subproblems has rendered invalid a number of its invariants, which appear to be necessary for fast operation of the algorithm. This forced us to develop a slightly different version of ENS-NDT, which also appeared to be interesting on its own: in particular, it appeared to be more efficient than the original version for smaller values of M.

Our experiments show that our hybrid algorithm tends to outperform both its origins, namely, the ENS-NDT algorithm (including its variation developed by us) and the divide-and-conquer algorithm, especially for large problem sizes ($N > 10^5$). This claim is supported by experimental results on two types of data (the "uniform hypercube", also known as the "cloud dataset", and the "uniform hyperplane" that consists of a single front) with M up to 15 and N up to 10^6.

The rest of the paper is structured as follows. Section 2 describes the necessary details of the divide-and-conquer algorithm, as well as of ENS-NDT. Section 3 presents the modified version of the ENS-NDT algorithm, that is used in the hybrid, as well as the hybrid itself. Experiments are presented and discussed in Sect. 4. Finally, Sect. 5 concludes.

2 Preliminaries: The Algorithms to Hybridise

In this section, we describe the two algorithms, that we are going to use, in more detail. We start with the divide-and-conquer approach by Jensen [12], however, we use the version taken from [2] which is provably correct on every input unlike the algorithm from [12] and unlike the algorithm from [10] has a provably fast asymptotic behaviour. The second algorithm will be the non-dominated tree approach from [11], which is also known as ENS-NDT.

We assume that we perform non-dominated sorting on a set of points P from the M-dimensional objective space. Since non-dominated sorting is based entirely on the Pareto dominance relation, we can safely assume that this objective space is $\mathbb{R}^M$, as otherwise we can sort all points in every objective and transform objectives into integers while preserving Pareto dominance.

2.1 The Divide-and-Conquer Algorithm

The divide-and-conquer algorithm is based on the following observation. Assume we took some value q of the j-th objective and we split the set of points P into two sets, the set $P_L = \{p \in P \mid p_j \leq q\}$ and $P_R = \{p \in P \mid p_j > q\}$. Then no point from P_R can dominate any point from P_L, because every point from P_L is less than any point from P_R in the j-th objective. So we can find the ranks for points P_L on their own, then perform the necessary comparisons between points from P_L and from P_R, always having points from P_L on the left side of the dominance relation to be checked, and, finally, refine the ranks for points from P_R by comparing them one to another.

The operations on P_L and P_R alone can be implemented in mostly the same way (again choosing an objective, splitting into halves and performing the same actions on the halves), thus allowing a recursive implementation. The operation on two arguments, P_L and P_R, is different, but it can also benefit from divide-and-conquer: if we split both sets of points, using the same value q of the same objective, into sets L_L, L_R, R_L and R_R, we can use the same procedure on pairs L_L and R_L, L_L and R_R, L_R and R_R, but we can avoid calling it on L_R and R_L.

For performance reasons, the value q is always chosen to be a median of the set of j-th objectives, and all sets are split into three parts (less than q, equal to q and greater than q). What is more, the objective j is always chosen to be the maximum objective in which the comparison still makes sense: in HelperA all points have the same value for every objective greater than j, and in HelperB every $l \in L$ dominates every $r \in R$ in all objectives greater than j.

To complete the algorithm, one needs to provide recursion terminators. There are two types of them: the first ones trigger when one of the sets becomes too small, the second ones are called when only two meaningful objectives remain. The former case is solved straightforwardly. For the latter case, a special sweep-line algorithm is used, which is described in detail in [12].

The outline of the algorithm is given in Algorithm 1. The runtime of the sweep line subroutines is known to be $O(n \log n)$ where n is the number of points supplied. Using this fact, and by noticing that $\max(|P_L|, |P_R|) \leq 1/2 \cdot |P|$

Algorithm 1. The outline of the divide-and-conquer algorithm

```
function DivideConquerSorting(P, M)
    HelperA(P, M)
end function
function HelperA(P, m)
    if |P| ≤ 1 then
        return
    else if |P| = 2 then
        Compare points in first m objectives
    else if m = 2 then
        Run the sweep line subroutine
    else
        q ← Median({p_m | p ∈ P})
        ⟨P_L, P_M, P_R⟩ ← Split(P, m, q)
        HelperA(P_L, m)
        HelperB(P_L, P_M, m − 1)
        HelperA(P_M, m − 1)
        HelperB(P_L ∪ P_M, P_R, m − 1)
        HelperA(P_R, m)
    end if
end function
function HelperB(L, R, m)
    if |L| ≤ 1 or |R| ≤ 1 then
        Compare all pairs of points in first m objectives
    else if m = 2 then
        Run the sweep line subroutine
    else
        q ← Median({p_m | p ∈ L ∪ R})
        ⟨L_L, L_M, L_R⟩ ← Split(L, m, q)
        ⟨R_L, R_M, R_R⟩ ← Split(R, m, q)
        HelperB(L_L, R_L, m)
        HelperB(L_R, R_R, m)
        HelperB(L_L ∪ L_M, R_M ∪ R_R, m − 1)
    end if
end function
```

and $\max(|L_L| + |R_L|, |L_R| + |R_R|) \leq 1/2 \cdot (|L| + |R|)$, one can use the Master theorem for solving recurrence relations [4] and prove the $O(|P| \cdot (\log |P|)^{M-1})$ worst-case running time bound.

Note that even the HelperA function solves a more general problem than non-dominated sorting: this function must cope with the existing lower bounds on ranks, arising from comparisons of points from the set P with points outside this set. From this point of view, HelperB can be seen as the function that upgrades ranks of points from the set R by comparing them with points from the set L, whose ranks are known and will not subsequently change. It is possible to switch to other algorithms instead of HelperA and HelperB, for instance on smaller sizes to improve performance, if they produce the expected result.

2.2 The ENS-NDT Algorithm

This algorithm belongs to another family of algorithms for non-dominated sorting, termed Efficient Non-dominated Sorting, or ENS [24]. The main idea is to first sort all points lexicographically (by comparing the first objectives, move on to the second objectives if the first are equal, and continuing this way). A point cannot dominate any other point which comes before in the lexicographical order. The algorithm then traverses the points in the sorted order, while maintaining some data structure that makes comparisons with the previous points faster. For each point, first a rank query is performed against the data structure, then the point with the determined rank is added to that data structure.

Two algorithms from this family, ENS-SS and ENS-BS [24], maintain a list of already ranked points for each rank value, and for each such list the dominance check is performed, starting with the most recently added point. They are different in that ENS-SS performs the sequential search for a rank, starting with the first one, and ENS-BS performs binary search for a rank.

The ENS-NDT algorithm proposed by Gustavsson and Syberfeldt [11], instead of a list, uses a *k-d tree* (this name comes from a "k-dimensional tree") to store points of each rank. To do this efficiently, the objective space is partitioned in advance: first all points are split by the M-th objective into two approximately equal parts (using the median similarly to the divide-and-conquer algorithm), then every such part is further partitioned into halves using the $(M-1)$-th objective and so on. After the second objective, the M-th objective comes again, as splitting in the first objective never makes sense. Every tree that stores the points will subsequently use this space partitioning scheme.

Ranking a newly inserted point is performed by running binary search for the rank, and for each rank a query to the k-d tree is made. The tree is traversed from the root towards the leaves. When the branching node is visited, its child corresponding to smaller objective values is always visited, while its other child is visited only if the splitting value stored in the node is not greater than the corresponding objective of the query point. Dominance comparisons in leaves are made straightforwardly, and if one of them succeeds, the procedure terminates.

The possibility of skipping entire subtrees determines the impressive performance of this algorithm. In particular, for many distributions of input points one can show a constant upper bound α on the probability of entering a node child corresponding to a higher objective value. This immediately gives the upper bound of $O(M \cdot N^{\log_2(1+\alpha)})$ per one query and $O(M \cdot N^{1+\log_2(1+\alpha)})$ for the entire run, which is strictly faster than $\Theta(N^2M)$ when $\alpha < 1$.

It is, however, possible to observe the $\Theta(N^2M)$ running time of this algorithm on an input described by three numbers N, M and k, where N is the number of points, M is the number of objectives and $1 \le k \le M$ is the index of the "special" objective. The point $P^{(i)}$, $1 \le i \le N$, of this input will have the objective value $P^{(i)}_j = i$ for all $j \ne k$ and $P^{(i)}_k = N - i$. The choice of k that degrades the performance most prominently depends on the implementation, but $k = 1$ or $k = M$ may be good choices. With this input, there will always be one front, and each ranking query will visit almost the entire tree.

3 The Proposed Algorithms

In this section, we first explain the problems which arise when adapting ENS-NDT, and many more algorithms, to serve as the replacements for HelperA and HelperB of the divide-and-conquer algorithm. Then we introduce ENS-NDT-ONE, a modification of ENS-NDT that uses only one k-d tree instance and is capable of working as HelperA and HelperB. Finally, we describe the hybrid algorithm.

3.1 Loss of Monotonicity in HelperB

At the first glance, it should be trivial to adapt ENS-NDT, as well as other algorithms from the ENS family, to serve as HelperB. Given the point sets L and R, one has to traverse their union in the lexicographical order. When a point from L is encountered, it is added to the data structure with its already known rank. When a point from R is encountered, one needs to query the data structure for the rank of this point, but one must not add this point to the data structure. This way, all necessary comparison between the points from L and from R will be performed.

The problem with this approach is that, in order to work correctly, the implementations of the algorithms shall stop relying on certain invariants that improve performance, and, as a result, the performance can significantly degrade. In the case of ENS-NDT and ENS-BS, this important invariant is *monotonicity*, which enables binary search. The invariant can be formulated as follows: *if the front* $k+1$ *dominates the point, then the front* k *also dominates it.* We now show that this invariant can be violated inside HelperB.

Consider points $p_0 = (1, 3, 9, 1)$, $p_1 = (1, 5, 5, 3)$, $p_2 = (1, 6, 2, 4)$, $p_3 = (1, 6, 7, 4)$, $p_4 = (1, 6, 7, 7)$, $p_5 = (1, 9, 1, 5)$, $p_6 = (2, 1, 6, 7)$, $p_7 = (2, 6, 5, 6)$, $p_8 = (4, 8, 2, 7)$, $p_9 = (5, 3, 3, 8)$. The first call to HelperA splits them into $P_L = \{p_0, p_1, p_2, p_3\}$, $P_M = \{p_5\}$, $P_R = \{p_4, p_6, p_7, p_8, p_9\}$. By the time HelperB$(P_L \cup P_M, P_R, 3)$ is called, p_0 will have rank 0 and p_3 will have rank 1. This call will partition these sets around the median of the third objective, which is 5, such that $L_R = \{p_0, p_3\}$ and $R_R = \{p_4, p_6\}$. Once HelperB$(L_R, R_R, 3)$ is called, the point p_4 will be found to be dominated by p_3 of rank 1, but the front corresponding to rank 0 will consist only of point p_0, which does not dominate p_4. This means that there is no monotonicity anymore, and binary search for the rank is no longer valid.

The same problem makes it impossible for ENS-SS to test ranks in the increasing order. The original ENS-SS stops once a front is found that does *not* dominate the point. We now know that inside HelperB this can result in a preliminary termination. The valid strategy in these conditions is to test ranks in the *decreasing* order, and to stop once the front is found that *does* dominate the point. Again, this reduces the performance of the ENS-SS algorithm, as, unlike the original version, most fronts are now traversed to their very end.

We overcome this problem by adapting ENS-NDT in such a way that it does not have to rely on monotonicity of fronts, while retaining a decent performance.

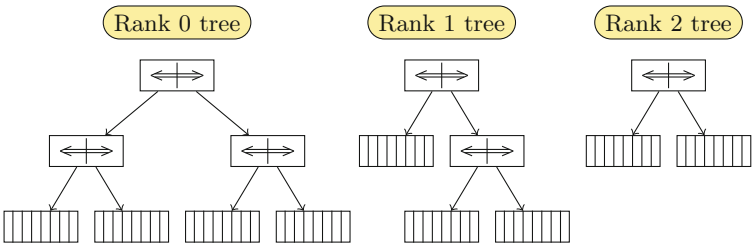

Fig. 1. The way ENS-NDT uses its k-d trees. Each tree is associated with a rank value, and stores only points with that rank.

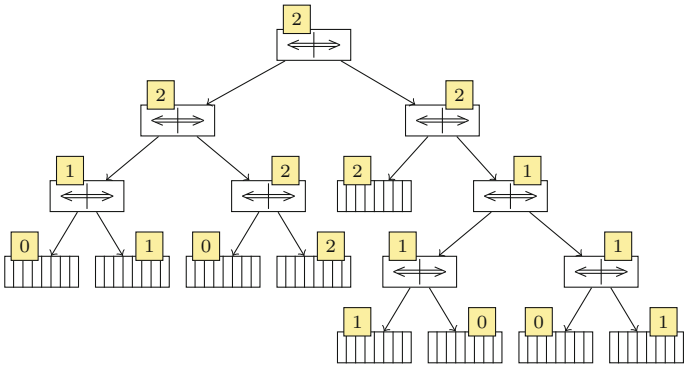

Fig. 2. The way ENS-NDT-ONE uses its only k-d tree. All points reside in the same tree, and each node additionally stores the maximum rank of a point in its subtree.

3.2 The ENS-NDT-ONE Algorithm

We propose a new algorithm for non-dominated sorting, termed ENS-NDT-ONE, that is based on ENS-NDT, however, unlike its ancestor, it does not maintain separate trees for storing points of different ranks. Instead, all points now reside in a single k-d tree.

One of the performance advantages of ENS-NDT is that, while completing the rank query for a point p, once a point in a tree is found to dominate p, it is possible to quit that tree immediately, since no more points from that tree can influence the rank of p. This is not so in ENS-NDT-ONE, as there can be points with the same or a greater rank compared to the updated rank of p.

To compensate for this performance loss, we propose to store in each tree node the value of the maximum rank among all points in the subtree rooted at this node. With this information in hand, we can now refrain from visiting the node (and all nodes in its subtree) if the maximum rank is less than the current rank of the point p being queried. The maximum ranks of nodes are also straightforwardly updated on insertion of a point. See Figs. 1 and 2 for comparison of the principles beneath ENS-NDT and ENS-NDT-ONE.

The worst-case running time of this algorithm is $\Theta(N^2M)$, which is demonstrated by the same construction as we used for ENS-NDT. However, for many cases the running time is much smaller. For instance, if the points to be sorted are sampled uniformly from a hypercube $[0;1]^M$, then we can see that $\Theta(N)$ points will have a probability of at most $1/2$ to enter both children of every particular branching node of the tree. This immediately gives the $O(MN^{1+\log_2(3/2)}) \approx O(MN^{1.585})$ runtime bound. Similar results can be shown for other distributions, and the bounds can be further reduced by considering the distribution of ranks.

3.3 The Hybrid Algorithm

We can now formulate the hybrid algorithm. We take the divide-and-conquer algorithm as a basis, however, before we enter the main parts of HelperA or HelperB, we check whether the subproblem is *small enough*. If it is, we use the ENS-NDT-ONE algorithm to solve this subproblem. Since ENS-NDT-ONE is immune to the features of these subproblems, such as the loss of monotonicity, the resulting algorithm will always produce correct results.

More formally, we define, for every number of objectives, a threshold which signifies that every subproblem with this number of objectives and the size below the threshold should be delegated to ENS-NDT-ONE. For HelperA, the size of the problem is the size of the set P, while for HelperB this is the sum of sizes of the sets L and R.

We shall note that, since we define thresholds to be constants, the asymptotic estimation of the running time of this algorithm is still $O(N(\log N)^{M-1})$. However, we note that more careful choices for thresholds, that possibly depend on the number of objectives or on other properties of the subproblems, may possibly result in smaller runtime bounds. Due to the complexity of this issue, including strong dependency on inputs, we leave this for possible future work.

4 Experiments and Discussion

All mentioned algorithms were implemented in Java within the same algorithmic framework, which enabled sharing large code amounts between the algorithms. These implementations are available on GitHub[1] along with performance plots.

All the algorithms, except for the original divide-and-conquer algorithm, feature parameters that influence their performance. In particular, the ENS-NDT algorithm and its derivatives have the *split threshold* parameter which regulates the maximum possible size of the terminal node. In [11] this parameter was fixed to the value of 2, however, our preliminary experiments found that the value of 8 brings generally better performance. This difference can be attributed to the differences in implementations. We used the split threshold of 8 for ENS-NDT, ENS-NDT-ONE, as well as in the ENS-NDT-ONE part of the hybrid algorithm.

[1] https://github.com/mbuzdalov/non-dominated-sorting/releases/tag/v0.1.

Table 1. Average running times of the algorithms in seconds. The smallest running time, for each category, is marked grey. All standard deviations are less than 2%.

N	M	Divide&Conquer		ENS-NDT		ENS-NDT-ONE		Hybrid	
		hypercube	hyperplane	hypercube	hyperplane	hypercube	hyperplane	hypercube	hyperplane
$5 \cdot 10^5$	3	1.52	0.85	1.95	0.73	1.66	0.76	1.17	0.67
10^6	3	2.82	1.60	5.25	1.61	4.25	1.65	2.63	1.50
$5 \cdot 10^5$	5	22.7	16.6	8.31	2.01	6.25	2.22	6.43	4.68
10^6	5	45.2	33.0	26.3	5.22	18.2	5.82	17.2	12.8
$5 \cdot 10^5$	7	89.6	55.1	17.1	6.96	15.5	6.78	9.29	7.02
10^6	7	191.5	120.2	55.4	19.4	46.1	18.9	26.8	20.1
$5 \cdot 10^5$	10	197.7	99.9	27.6	15.9	36.7	17.7	14.5	11.5
10^6	10	478.8	228.6	84.8	48.1	104.8	55.0	41.0	33.0
$5 \cdot 10^5$	15	190.0	116.1	40.8	23.0	62.1	25.9	22.6	15.7
10^6	15	587.9	337.5	135.4	76.3	206.8	85.4	64.5	46.0

The hybrid algorithm also depends on the switch-to-tree threshold values. Based on our preliminary investigations, we chose this threshold for three objectives to be 100, and for more than three objectives to be 20 000.

We have investigated the performance of all these algorithms, including the ENS-NDT-ONE alone, on several artificial inputs. We used two types of data. The first one is the "uniform hypercube", which is also known as the "cloud dataset" in the literature, where points are sampled uniformly at random from the $[0; 1]^M$ hypercube. The second one is the "uniform hyperplane", where points are sampled uniformly at random from the piece of a hyperplane, such that all coordinates are non-negative and sum up to 1. The following values of N were tested: $\{1, 2, 5\} \times \{10, 10^2, 10^3, 10^4, 10^5\}$ and 10^6. We considered M to be from the set $\{3, 5, 7, 10, 15\}$, which covers the most widely used range.

For every input configuration, 10 instances were created with different but fixed random seeds. We measured the total times on all these instances and divided them by 10 to achieve an approximation of the average time. The time measurements were done using the Java Microbenchmark Harness suite with one warmup iteration of at least 6 seconds, which was enough for the entire bytecode to be translated to the native code, and one measurement iteration of at least one second. For each pair of algorithm and input, five measurement runs were conducted. A high-performance server with AMD Opteron™ 6380 processors and 512 GB of RAM was used, and the code was run with the OpenJDK virtual machine 1.8.0_141.

The already mentioned GitHub release features the plots of the running times, which could not fit in this paper due to space restrictions. In Table 1, we show only the average results for two largest N, $5 \cdot 10^5$ and 10^6. One can see that the hybrid algorithm wins in all cases except for $M = 5$ and the hyperplane instance of $M = 7$. One more insight is that ENS-NDT-ONE runs faster than ENS-NDT on hypercube instances with $M \leq 7$, which means that the maximum subtree rank heuristic is indeed efficient. The implementation constant of ENS-NDT-ONE seems to be slightly larger, however.

5 Conclusion

We proposed a highly efficient algorithm for non-dominated sorting based on hybridisation of two previously known algorithms, the divide-and-conquer algorithm by Jensen and the non-dominated tree (ENS-NDT) by Gustavsson and Syberfeldt. It typically outperforms both of its parents on large population sizes, except for certain ranges of population sizes in several dimensions. Our modification of ENS-NDT is also of interest, as it can outperform the original ENS-NDT.

We are probably the first to report results on 10^6 solutions. Some industrial applications of evolutionary multiobjective optimisation already require population sizes that are this large. As divide-and-conquer algorithms often offer parallelisation benefits, and our algorithm is not an exception, we hope to get further speed-ups by adapting our algorithm to multicore computers.

The optimal choice of thresholds to decide when to switch to ENS-NDT is an open and difficult question. We expect that adaptation of thresholds while running the algorithm can overcome this issue.

Acknowledgment. We would like to acknowledge the support of this research by the Russian Scientific Foundation, agreement No. 17-71-20178.

References

1. Brockhoff, D., Wagner, T.: GECCO 2016 tutorial on evolutionary multiobjective optimization. In: Proceedings of Genetic and Evolutionary Computation Conference Companion, pp. 201–227 (2016)
2. Buzdalov, M., Shalyto, A.: A provably asymptotically fast version of the generalized jensen algorithm for non-dominated sorting. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 528–537. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10762-2_52
3. Coello Coello, C.A., Toscano Pulido, G.: A micro-genetic algorithm for multiobjective optimization. In: Zitzler, E., Thiele, L., Deb, K., Coello Coello, C.A., Corne, D. (eds.) EMO 2001. LNCS, vol. 1993, pp. 126–140. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44719-9_9
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
5. Corne, D.W., Jerram, N.R., Knowles, J.D., Oates, M.J.: PESA-II: Region-based selection in evolutionary multiobjective optimization. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 283–290. Morgan Kaufmann Publishers (2001)
6. Deb, K., Jain, H.: An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. IEEE Trans. Evol. Comput. **18**(4), 577–601 (2013)
7. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
8. Fang, H., Wang, Q., Tu, Y.C., Horstemeyer, M.F.: An efficient non-dominated sorting method for evolutionary algorithms. Evol. Comput. **16**(3), 355–384 (2008)

9. Fonseca, C.M., Fleming, P.J.: Nonlinear system identification with multiobjective genetic algorithm. In: Proceedings of the World Congress of the International Federation of Automatic Control, pp. 187–192 (1996)
10. Fortin, F.A., Grenier, S., Parizeau, M.: Generalizing the improved run-time complexity algorithm for non-dominated sorting. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 615–622. ACM (2013)
11. Gustavsson, P., Syberfeldt, A.: A new algorithm using the non-dominated tree to improve non-dominated sorting. Evol. Comput. **26**(1), 89–116 (2018)
12. Jensen, M.T.: Reducing the run-time complexity of multiobjective EAs: the NSGA-II and other algorithms. IEEE Trans. Evol. Comput. **7**(5), 503–515 (2003)
13. Knowles, J.D., Corne, D.W.: Approximating the nondominated front using the pareto archived evolution strategy. Evol. Comput. **8**(2), 149–172 (2000)
14. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. J. ACM **22**(4), 469–476 (1975)
15. Markina, M., Buzdalov, M.: Hybridizing non-dominated sorting algorithms: divide-and-conquer meets best order sort. In: Proceedings of Genetic and Evolutionary Computation Conference Companion, pp. 153–154 (2017)
16. McClymont, K., Keedwell, E.: Deductive sort and climbing sort: new methods for non-dominated sorting. Evol. Comput. **20**(1), 1–26 (2012)
17. Nekrich, Y.: A fast algorithm for three-dimensional layers of maxima problem. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 607–618. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22300-6_51
18. Roy, P.C., Islam, M.M., Deb, K.: Best Order Sort: a new algorithm to non-dominated sorting for evolutionary multi-objective optimization. In: Proceedings of Genetic and Evolutionary Computation Conference Companion, pp. 1113–1120 (2016)
19. Schlünz, E.B.: Multiobjective in-core fuel management optimisation for nuclear research reactors. Ph.D. thesis, Stellenbosch University, December 2016
20. Srinivas, N., Deb, K.: Multiobjective optimization using nondominated sorting in genetic algorithms. Evol. Comput. **2**(3), 221–248 (1994)
21. Stevens, J., Smith, K., Rempe, K., Downar, T.: Optimization of pressurized water reactor shuffling by simulated annealing with heuristics. Nucl. Sci. Eng. **121**(1), 67–88 (1995)
22. Wang, H., Yao, X.: Corner sort for pareto-based many-objective optimization. IEEE Trans. Cybern. **44**(1), 92–102 (2014)
23. Zhang, Q., Li, H.: MOEA/D: a multiobjective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. **11**(6), 712–731 (2007)
24. Zhang, X., Tian, Y., Cheng, R., Jin, Y.: An efficient approach to nondominated sorting for evolutionary multiobjective optimization. IEEE Trans. Evol. Comput. **19**(2), 201–213 (2015)
25. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 832–842. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30217-9_84
26. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: improving the strength pareto evolutionary algorithm for multiobjective optimization. In: Proceedings of the EUROGEN 2001 Conference, pp. 95–100 (2001)