Evolutionary Computation Just Accepted MS.
doi:10.1162/EVCO_a_00204

A New Algorithm Using the Non-dominated Tree to improve Non-dominated Sorting

Patrik Gustavsson patrik.gustavsson@his.se
School of Engineering, University of Skövde, Skövde, 54134, Sweden

Anna Syberfeldt anna.syberfeldt@his.se
School of Engineering, University of Skövde, Skövde, 54134, Sweden

Abstract

Non-dominated sorting is a technique often used in evolutionary algorithms to determine the quality of solutions in a population. The most common algorithm is the Fast Non-dominated Sort (FNS). This algorithm, however, has the drawback that its performance deteriorates when the population size grows. The same drawback applies also to other non-dominating sorting algorithms such as the Efficient Non-dominated Sort with Binary Strategy (ENS-BS). An algorithm suggested to overcome this drawback is the Divide-and-Conquer Non-dominated Sort (DCNS) which works well on a limited number of objectives but deteriorates when the number of objectives grows. This paper presents a new, more efficient, algorithm called the Efficient Non-dominated Sort with Non-Dominated Tree (ENS-NDT). ENS-NDT is an extension of the ENS-BS algorithm and uses a novel Non-Dominated Tree (NDTree) to speed up the non-dominated sorting. ENS-NDT is able to handle large population sizes and a large number of objectives more efficiently than existing algorithms for non-dominated sorting. In the paper, it is shown that with ENS-NDT the runtime of multi-objective optimization algorithms such as the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) can be substantially reduced.

Keywords

Evolutionary computation, k-d tree, multi-objective evolutionary algorithms, non-dominated sorting, Pareto optimality, run-time complexity.

1 Introduction

Almost all real-world problems involve the simultaneous optimization of multiple objectives, and it is rare for only a single objective to be considered (Zitzler, 1999), (Deb, 2001) and (Mehnen et al., 2004). A multi-objective problem comprises multiple objective functions to be optimized. The difficulty with this type of problem is that there is usually no single optimal solution with respect to all objectives, as improving the performance of one objective means worsening the performance of another, if the objectives are conflicting (Srinivas and Deb, 1994).

To handle multiple optimization objectives, it is common to use a so-called Pareto approach. The Pareto approach to multi-objective optimization provides the user with information about the trade-offs among various objectives (Deb, 2001) and (Murata and Ishibuchi, 1995). Instead of a single optimum, there is a set of optimal trade-offs between conflicting objectives called Pareto-optimal or non-dominated solutions. Figure 1 illustrates the Pareto concept for a minimization problem with two objectives f_1 and f_2. In this example, solutions A–D are non-dominated, i.e., Pareto optimal, because for each of these solutions no other solution exists that is superior in one objective without being worse in another. Solution E is dominated by B and C (but not by A or D, because E is better than these two in f_1 and f_2, respectively). Different Pareto ranks can also be identified among solutions. Rank 1 includes the Pareto-optimal solutions in the complete population, and rank 2 the Pareto-optimal solutions identified when temporarily discarding all solutions of rank 1, and so on. A set of solutions belonging to one Pareto rank is called a Pareto front.

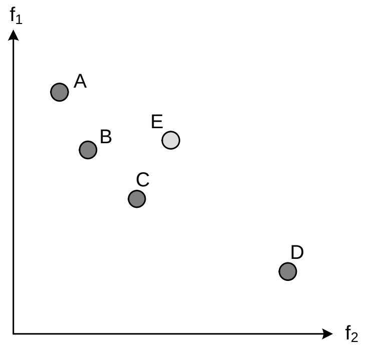

Figure 1: Illustration of dominance.

Evolutionary algorithms are very well suited for handling multi-objective problems (Deb, 2001) and (Coello Coello, 1999). Because evolutionary algorithms maintain a population of solutions, it is possible to find multiple Pareto-optimal solutions in a single optimization run. Many multi-objective evolutionary algorithms use Pareto ranking to assign some kind of quality measure. If all solutions belong to the same Pareto front then all solution will be similarly ranked. This means that different solutions will offer no advantages over one another and that the search will stagnate. Deb (2001) demonstrated that, by increasing the size of a random population then fewer solutions are likely to belong to the first front. Deb (2001) further demonstrated that if the percentage of solutions in the first front is kept constant while increasing the number of objectives, the population size will need to increase exponentially. If 30% of the solutions are to end up in the first front, this will require a population size of 100 on four objectives, of nearly 800 for six objectives, and of over 6000 for eight objectives.

Although multi-objective evolutionary algorithms using the Pareto concept have been very successful in recent years, the computationally expensive non-dominated sorting procedures needed to derive the Pareto ranks of the solutions are a problem. The ordinary approach for this sorting algorithm has a time complexity of $O(MN^2)$, where M is the number of objectives and N is the number of solutions in the population, causing long processing times for larger populations. However, a divide-and-conquer approach exists that has a time complexity of $O(N \log^{M-1} N)$, but that has longer processing times when increasing the number of objectives. This paper presents a new non-dominated sorting algorithm having considerably faster processing time for both larger population sizes and more objectives.

The rest of the paper is structured as follows. Section 2 reviews related work, existing algorithms for non-dominated sorting, and research in the non-dominated sorting area. Section 3 describes how the Efficient Non-dominated Sort with Non-Dominated

Tree (ENS-NDT) algorithm works, why it works, and details of its implementation. Section 4 describes the computational complexity of the ENS-NDT algorithm. Section 5 describes several experiments in order to evaluate the ENS-NDT algorithm, doing so in several steps including comparison with other state-of-the-art non-dominated sorting algorithms and comparisons with mathematical equations. Section 6 concludes the paper and discusses future work.

2 Related Work

Deb et al. (2002a) presented a Fast Non-dominated Sorting (FNS) algorithm that has a worst, average and best case time complexity of $O(MN^2)$ where M is the number of objectives and N the population size. For each solution, the FNS algorithm stores a domination count variable and a set of dominated solutions. In the first step of the FNS algorithm, all solutions are compared with one another, which requires N^2 comparisons. For each comparison, either the domination count variable or the set of dominated solutions are updated. In the second step all solutions are added one-by-one to the front set. All solutions with a domination count of zero are added to the first front. Each of these solutions have a set of dominated solutions, these dominated solutions have their domination count decreased. In the next iteration new solutions have a domination count of zero, all these solutions are added to the second front, and the domination count of the dominated solutions are updated. This procedure is repeated until all solutions are added to the front set. All solutions need to be compared with one another, each time up to M objectives are compared generating a time complexity of $O(MN^2)$.

Jensen (2003) presented a Two-objective Non-dominated Sort (TNS) that runs with a time complexity of $O(N \log N)$, illustrated in Figure 2. The algorithm uses the strategy of presorting all solutions so that the latter solutions do not dominate the former. This is achieved by presorting the solutions in ascending order; both objectives are assumed to be minimized. Each comparison in the presort starts by comparing the first objective between two solutions, and if the first objective of the two solutions have identical values then the second objective is compared between the two solutions. After the presorting then the first solution can be directly added to the first front, since the latter solutions cannot dominate the former. Then the remaining solutions in the population are added to the front set one-by-one. A binary search is used to find the first front which does not contain any solutions that dominates the current solution, and then the current solution is added to that front. If no such front exists, then the current solution will be added to a new front which is appended to the end of the front set. Since all solutions are presorted, this means that all solutions in a front are in ascending order for the first objective and descending order for the second, as illustrated in Figure 2, (since all solutions in one front are non-dominated). The domination comparison can therefore be determined by only comparing the current solution with the last solution in a front. This is possible because the first objective of the current solution is always larger than the previous solutions and the last solution in a front has the lowest value of the second objective in that front. The TNS algorithm is the optimal sorting procedure when only two objectives exist (Jensen, 2003); however, this algorithm does not work for three or more objectives.

Zhang et al. (2015) proposed a method called the Efficient Non-dominated Sort (ENS), which is similar to TNS but can handle more than two objectives. The ENS algorithm uses the same strategy as the TNS algorithm by presorting the population, ensuring that the latter solutions cannot dominate the former. All the solutions are

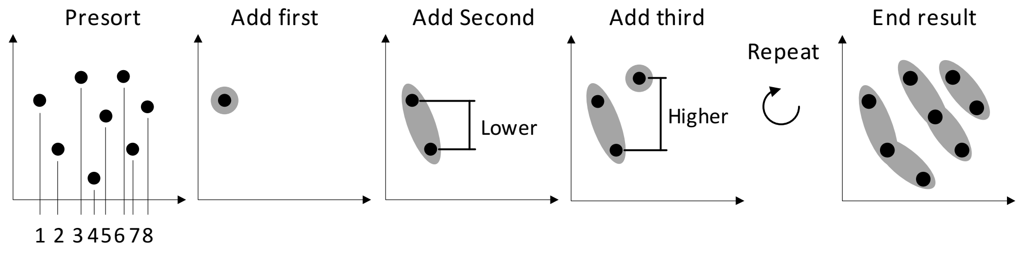

Figure 2: Overview of the TNS procedure.

then added one by one to the first front that does not dominate the solution. Zhang et al. (2015) proposed two strategies to find the front index, the Sequential Strategy (ENS-SS) and the Binary Strategy (ENS-BS). The Binary Strategy has the advantage when there are many fronts, because the search performance is $O(log|\mathcal{F}|)$, $|\mathcal{F}|$ being the number of fronts, while the Sequential Strategy has a worse performance of $O(|\mathcal{F}|)$. However, the Sequential Strategy can perform better than the Binary Strategy when increasing the number of objectives, as demonstrated by Zhang et al. (2015). There are two main differences between TNS and ENS. First, the presort in ENS considers all objectives (TNS considers only two objectives). Second, the front domination check needs to check all solutions in the front, instead of just the last solution as with the TNS algorithm. The ENS-BS algorithm works well when the solutions are distributed across many fronts but has the same drawback as FNS, with a worst time complexity of $O(MN^2)$, when all solutions belong to the same front.

An extension to the ENS algorithms has recently been created called T-ENS as mentioned in Zhang et al. (2016). This algorithm first sorts the population to ensure that the latter solutions cannot dominate the former solutions based on the first objective. Then the algorithm finds all solutions that belong to the first front, then the second front, until all fronts have been created. To quicker determine if a solution belongs to one front or not, each front is instead represented with a tree. This tree keeps track of the non-domination relationships between solutions and with this tree the number of comparisons can be considerably reduced. The T-ENS algorithm is more efficient than the ENS algorithms when most solutions are non-dominated with each other. However, the main drawback is that T-ENS cannot sort solutions sharing identical values for any of the objectives.

Jensen (2003) proposed a Divide-and-Conquer Non-dominated Sort (DCNS) algorithm. The algorithm assigns each solution in the population with a front index, and then the front index is updated based on solutions it has been compared with so far. By comparing the solutions in a divide-and-conquer approach, this front number increases as a solution is dominated. Finally, when the algorithm has done all necessary comparisons, the solutions are added to the front set based on their front indices. The time complexity of the DCNS algorithm is $O(N \log^{M-1} N)$ in the worst, average and best cases. This algorithm suffers from the same problem as T-ENS, that it cannot sort solutions sharing identical values for any of the objectives. Compared with the FNS (Deb et al., 2002a), the DCNS algorithm performs much better with larger population sizes. However, when using, for example, a population size of 100 and eight objectives, then

DCNS takes 60% more processing time than FNS, as demonstrated by Jensen (2003). If there are eight objectives and the population size is increased to 1000, then DCNS is twice as fast as FNS. Fortin et al. (2013) generalized the DCNS algorithm so that it can also handle solutions sharing identical values. This algorithm will henceforth be referred to as Fortin's extension of the DCNS algorithm (DCNS-F). DCNS-F has the same average and best time complexities as DCNS, of $O(N \log^{M-1} N)$, but the worst time complexity is $O(MN^2)$. Buzdalov and Shalyto (2014) continued the work and created an algorithm that has a time complexity of $O(N \log^{M-1} N)$ in the best, average and worst cases.

Fang et al. (2008) proposed a method for faster non-dominated sorting of a population by adopting a special tree. This tree keeps the domination relationships between solutions and is used to reduce the number of redundant domination checks. By using this tree, it is possible to eliminate domination checks already conducted in earlier generations. Also, if solution a dominates solution b, then solution a does not need to be checked against solutions dominated by solution b. This works well for a population with several fronts, though this algorithm suffers from the same drawback as FNS when the population has only one front with a time complexity of $O(MN^2)$.

Drozdík et al. (2015) proposed another approach to reduce the time complexity of evolutionary algorithms, an approach that uses Pareto ranking. This diverges from the standard approach by trying to eliminate the need for non-dominated sorting. At the start of the evolutionary algorithm, the non-dominated solutions are added to a special list and to a so-called *k*-d tree. During the generational loop, one solution at a time is checked against this list, which contains only the non-dominated solutions. If the new solution is non-dominated, it is added to the list and to the *k*-d tree; otherwise, it is discarded. All the solutions in the list that are dominated by the new non-dominated solution are also discarded from this list and *k*-d tree. The domination check process uses the special list to check whether a solution is non-dominated, though the check would not be efficient if there were not a good reference solution from which the checking starts. This reference solution is found by using the nearest neighbor search in the *k*-d tree. For each loop in which there are more than N non-dominated solutions, there is no need for non-dominated sorting. In those cases in which there are fewer than N non-dominated solutions, the ordinary non-dominated sorting method is used.

The aim of a multi-objective optimization algorithm is to find increasingly better solutions that eventually converge towards a Pareto-optimal front (Deb, 2001) and (Coello Coello, 1999). When using the ENS algorithms in such optimization, the time complexity approaches $O(MN^2)$ in the late stages, i.e. when the algorithm produces only populations with few fronts. This means that the ENS algorithms are not efficient when used with larger population sizes; for example, for a population size of 1000, the worst case entails one million comparisons. The DCNS algorithm and its extensions by Fortin et al. (2013), and Buzdalov and Shalyto (2014) are more efficient when increasing the population size. The main drawback of those algorithms are their time complexity of $O(N \log^{M-1} N)$ when increasing the number of objectives M, because the power of $M - 1$ grows quickly.

3 The Efficient Non-dominated Sort with Non-Dominated Tree

A new algorithm was developed in the present study called the Efficient Non-dominated Sort with Non-Dominated Tree (ENS-NDT). This algorithm extends the ENS-BS algorithm and increases the efficiency when sorting a population that contains few fronts. The ENS-NDT algorithm adopts a novel tree that was developed as part of

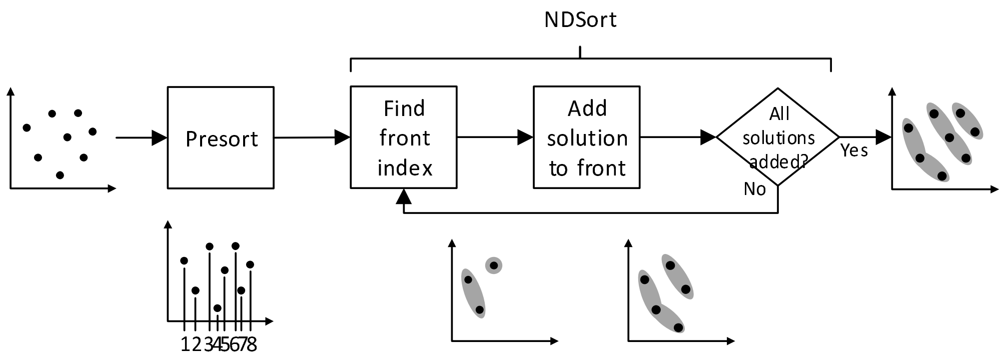

Figure 3: Overview of the ENS procedure.

the study, called the Non-Dominated Tree (NDTree), NDTree quickly searches through a front to confirm whether or not a solution is dominated and is designed to overcome the problem of ENS-BS with long processing times on few fronts. The NDTree is a variant of a bucket *k*-dimensional tree (*k*-d tree) and the motivation for using a *k*-d tree is its quick multidimensional search performance that involves, for example, *range* and *nearest neighbor* searches (Bentley, 1975).

In the following subsections the ENS-NDT algorithm is further explained. The basis of ENS-NDT, namely the ENS-BS algorithm, is first described in more detail in subsection 3.1, followed by a description of the novel NDTree in subsection 3.2. The ENS-NDT algorithm is then described in subsection 3.3, and last a number of examples and illustrations of how the ENS-NDT algorithm works is presented in subsection 3.4.

3.1 Description of the Efficient Non-dominated Sort with Binary Strategy

The ENS-BS algorithm developed by Zhang et al. (2015) follows the procedure illustrated in Figure 3. In this example there are two minimization objectives. The algorithm starts by presorting the population in an ascending order based on a lexicographic comparison. The lexicographic comparison starts by comparing the first objective value of two solutions, if the values are identical then the comparison continues to the next objective value, and iterates through all objective values if necessary. The ascending sort based on lexicographic comparison ensures that the latter solutions have at least one objective value that is worse than the former (unless all objective values are identical). The domination criterion requires all objective values to be equally good or better, and at least one objective value to be strictly better. Therefore, the presort ensures that the latter solutions in the sorted population cannot dominate the former solutions.

After the presort, then the main loop starts which henceforth is referred to the ndsort. The ndsort iterates through all solutions in the sorted population and adds one solution at a time to the front set. The ndsort is split up in two main steps, as shown in Figure 3. First the front index is found for the solution in the current iteration, then the solution is added to that front and when all solutions have been added, the front set is returned. The front index is identified as the index of the first front which does not contain any solutions that dominates the current solution. For each front, the domination check compares the new solution with the solutions in the front, until one solution is found to dominate the new solution. If the front contains a solution that dominates the new solution, then the new solution is dominated by that front. However, if no solution in the front is found to dominate the new solution, then the new solution is regarded as not dominated by the front. In the case of non-domination, the new solution needs to be compared with all the solutions in the front. When the ndsort has added all solutions in the population to the front set, then the ENS-BS algorithm returns the front set.

It can be commented that Zhang et al. (2015) originally proposed two strategies to find the front index, one being the sequential strategy ENS-SS which uses a sequential search, and the other being the binary strategy ENS-BS which uses a binary search. While ENS-BS has a better time complexity than ENS-SS, ENS-SS still has advantages when optimizing a larger number of objectives. However, ENS-BS was selected in this study as the basis for the ENS-NDT algorithm because it has better overall time-complexity.

Regarding the time complexity, the ENS algorithms approach the worst case complexity of $O(MN^2)$ when used in traditional optimizations, because the populations in the later stages of an optimization tend to have few fronts. The ENS-NDT algorithm with the NDTree has been developed to solve this problem.

3.2 Description of the Non-Dominated Tree

The NDTree is a variant of a bucket *k*-d tree illustrated in Figure 4. A tree is a data structure that stores values in nodes. The tree starts with a root node, which can have node children. These node children can also be connected to other nodes, as long as circles in the tree are not created. A bucket *k*-d tree is a special type of binary tree (a tree where each node can have at most two node children) that stores points of *k* dimensions. The points are stored within nodes of the tree and each node is associated with one of the *k* dimensions. Each node has a maximum size called the bucket size. When the bucket overflows, i.e. the node contains more points than the bucket size, the points are split into two new nodes (a left and a right node) based on the node's associated dimension. First, the median is calculated based on the values in the current dimension of all the node's points. Then the points lower in value than the median are put in the left node and the points higher in value are put in the right node. The points equal in value to the median are put in either the left or right node depending on the settings of the bucket *k*-d tree. The new nodes are then associated with another dimension separately from the parent node (Bentley, 1975), usually the next dimension (i.e. dimension of the parent node + 1, or the first dimension if the parent node is associated with the last dimension). In the left illustration of Figure 4 the associated dimension is shown next to the nodes. Node A is associated with X, node B and C are associated with Y, and node D and E are associated with X again.

The NDTree uses a prebalanced split set, where all possible median values are calculated before adding points to the tree. This differs from a normal bucket *k*-d tree which calculates the split when a node overflows. It is possible to prebalance the splits because the entire set of solutions is known beforehand when used in non-dominated sorting. The advantage of using a prebalanced split set is that it keeps the tree balanced and thereby greatly improves the search performance. The prebalanced split set can be structured as the left illustration of Figure 4 but without the buckets, which makes it easy for the NDTree to keep track of its splits.

The NDTree uses the same logic as a bucket *k*-d tree when inserting new solutions. Each node has a given bucket size; when the node overflows, the solutions are split into

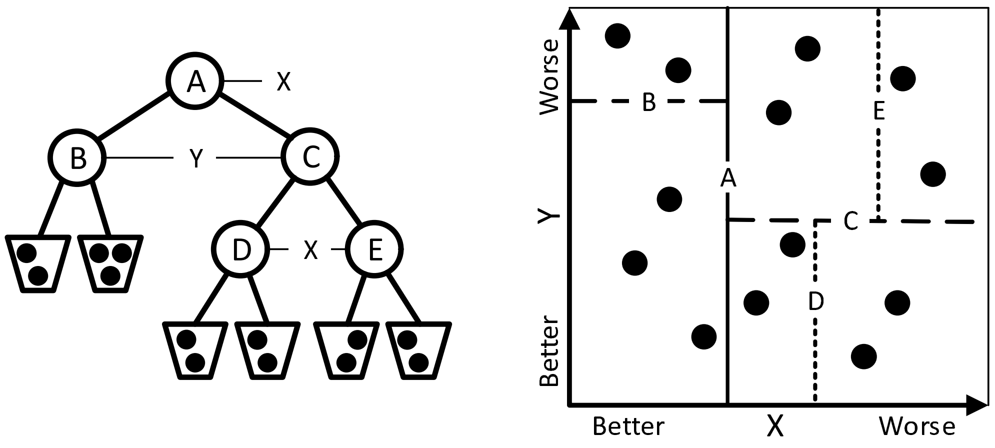

Figure 4: Illustration of the internal structure of the NDTree which follows the structure of the bucket k-d tree. The tree structure is shown to the left and the representation in Euclidian space is shown to the right.

two new nodes. Unlike a bucket k-d tree, the solutions are split based on the node's prebalanced split. The solutions are put into one of the two nodes based on whether or not the solution dominates the split. The dominance check considers the prebalanced split value and the node's associated objective. The solutions that dominate the split are put in the node in the better direction, henceforth called BetterNode. All the other solutions, which do not dominate the split, are put in the node in the worse direction, henceforth called WorseNode. In Figure 4 the bucket size is set to three, when a fourth solution arrives then the node is split in two transferring the solutions that dominates the split to the BetterNode and the solutions that does not dominate the split to the WorseNode. Solutions might have similar values in several objectives and to ensure that the insertion does not create too many level of nodes the NDTree has a maximum depth. This means that a node on the maximum depth cannot be split and therefore does not overflow even if it gets larger than the bucket size.

All the solutions added to the NDTree need to be non-dominated, fulfilling the constraint $T_i \not\prec T_j$ where $1 \leq i, j \leq |T|$, i.e. none of the solutions in the tree can dominate any other in the same tree. A domination check is therefore necessary before adding a solution to the NDTree.

3.3 Description of the Efficient Non-dominated Sort with Non-Dominated Tree

The overall flow of the ENS-NDT procedure is shown in Figure 5. The main difference between ENS-NDT in Figure 5, and ENS in Figure 3, is the use of NDTrees to speed up the domination checks. The NDTrees require prebalanced splits which are created in the "Create splits" step. The presort works the same way but uses a reverse lexicographic comparison, i.e. starts by comparing with the last objective value and if the values are equal then continue to the previous objective value, and so on. Then the main loop starts in the "NDSort" phase, where in each iteration, first the front index is found and then the solution is added to that front. To find the front index, a binary

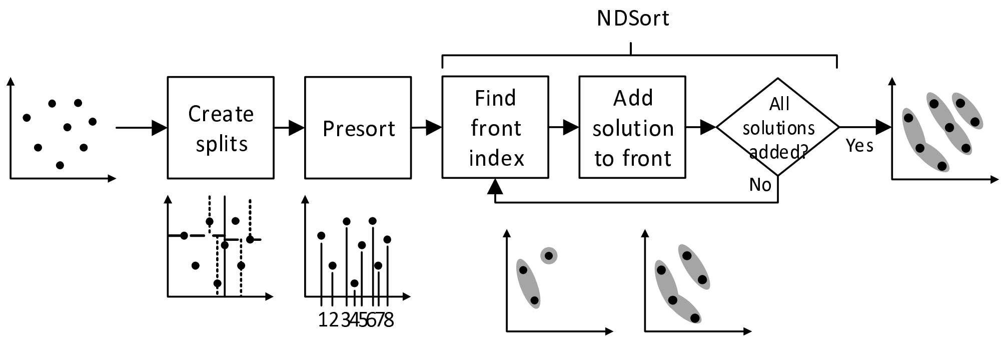

Figure 5: Overview of the ENS-NDT procedure which is similar to the ENS procedure but uses the NDTree front set to speed up the domination checks.

search is used which is almost the same as in the ENS-BS algorithm except that ENS-NDT finds the front index by comparing each new solution with the NDTree front set, instead of the regular front set.

The entire procedure of the new non-dominated sorting algorithm is described with pseudo codes in Figures 6-9. The main procedure of the ENS-NDT algorithm, shown in Figure 6, uses three arguments: the population P, the number of objectives M, and the bucket size of the NDTrees B. The algorithm is similar to the ENS-BS algorithm but uses the NDTree for each front domination check. The prebalanced split set for the NDTrees are calculated using the CreateSplits procedure and stored in variable S on line 2 in Figure 6. The splits are calculated using the population P, the bucket size B and the number of objectives $M-1$ ignoring the Mth objective. The Mth objective is not necessary in the NDTrees because the population P is later presorted, ensuring that $P_j^M \not\prec P_i^M$ when $j > i$, i.e. the latter solutions cannot dominate the former ones based on the Mth objective. The procedure then presorts the population P in the reverse lexicographic objective dominance order on line 3. This means that the last objective M is compared between two solutions, and that the solution with the dominant objective value is placed first. However, if the objective values are equal, then the next objective $M-1$ is compared. This continues until the procedure reaches the first objective. The presort ensures that $P_i \not\prec P_j$ when $j > i$, i.e. the latter solutions cannot dominate the former ones. The reverse lexicographic objective dominance order is selected for the presort for two reasons. First, the CreateSplits procedure can easily ignore the last objective. Second, all future domination checks go in the direction objective $1 \to M$. This reduces the number of comparisons needed in each domination check, because the comparison of the Mth objective can be ignored. This is possible because the Mth objective value of the latter solutions is larger than the Mth objective value of the former.

In the next step, the front set $\mathcal{F}$ and the NDTree set $\mathcal{T}$ are created on lines 4–6 in Figure 6. The front set $\mathcal{F}$ is the sorted set of Pareto fronts resulting from the non-dominated sort. The NDTree set $\mathcal{T}$ is a representation of the front set but with NDTrees to speed up the domination checks. The NDTree set $\mathcal{T}$ and the front set $\mathcal{F}$ have almost identical distributions of solutions during the main loop but with different storage structures. The only thing that differs between $\mathcal{F}$ and $\mathcal{T}$ is that only unique solutions with regards

```
 1: procedure ENS-NDT($P, M, B$)
 2:     $S \leftarrow$ CreateSplits($P, M-1, B$)
 3:     $P \leftarrow$ Sort($P, a^M \prec b^M, ..., a^1 \prec b^1$)
 4:     $\mathcal{F} \leftarrow \{\{P_1\}\}$
 5:     $\mathcal{T} \leftarrow \{$new NDTree($S, B$)$\}$
 6:     InsertIntoNDTree($\mathcal{T}_1, P_1$)
 7:     $j \leftarrow 1$
 8:     for $i = 2, ..., |P|$ do
 9:         if $P_{i-1} \neq P_i$ then
10:             $j \leftarrow$ FrontIndexBinarySearch($\mathcal{T}, P_i$)
11:             if $j > |\mathcal{T}|$ then
12:                 $\mathcal{F}_j \leftarrow \emptyset$
13:                 $\mathcal{T}_j \leftarrow$ new NDTree($S, B$)
14:             end if
15:             InsertIntoNDTree($\mathcal{T}_j, P_i$)
16:         end if
17:         $\mathcal{F}_j \leftarrow \mathcal{F}_j \cup \{P_i\}$
18:     end for
19:     return $\mathcal{F}$
20: end procedure
```

Figure 6: The ENS-NDT procedure which is similar to the ENS-BS procedure but uses NDTrees for non-dominated checks.

to all objective values are stored in $\mathcal{T}$. Solution P_1 is directly added to both the front set $\mathcal{F}$ and the NDTree set $\mathcal{T}$ on lines 4–6 in the first front. Solution P_1 always belongs to the first front because none of solutions $P_2, \ldots, P_N$ can dominate P_1 due to the presort. The NDTree is created with the prebalanced split set S and the bucket size B.

Variable j is declared on line 7 in Figure 6 and is used for the front index. It is declared before the main loop to keep track of the front index of the previous solution. The main loop starts on line 8 where the remaining solutions are added to the correct front. First the current solution P_i is compared with the previous solution $P_(i-1)$ on line 9 to determine whether they have identical values. If they have identical objective values then solution P_i is added to the same front as the previous solution on line 17. If they do not have identical objective values then solution P_i is checked against the NDTree set $\mathcal{T}$ on line 10 using the FrontIndexBinarySearch procedure. This procedure finds the lowest value of j for which criterion $\mathcal{T}_j \nprec P_i$ is true, i.e. the first front which does not contain any solution that dominate solution P_i. When front index j has been determined, solution P_i is added to $\mathcal{F}$ and $\mathcal{T}$ on lines 15 and 17, respectively. However, solution P_i is only added to the NDTree set $\mathcal{T}$ when solution $P_(i-1) \neq P_i$ in order to store only unique solutions. If front index j is larger than the front set size then a new front and a new NDTree is added to $\mathcal{F}$ and $\mathcal{T}$ on lines 12 and 13, respectively. The NDTree is created with the prebalanced split set S and the bucket size B.

The CreateSplits procedure is further described in Figure 7 which shows an example of how the prebalanced split set can be calculated. This procedure uses the same strategy as building a balanced *k*-d tree, but instead of building the *k*-d tree only the split values are stored. A general approach to building a balanced *k*-d tree is through the divide-and-conquer approach as explained by Blum et al. (1973). In the example in Figure 7 the prebalanced split set has a tree structure similar to the NDTree, which

```
1:  procedure CreateSplits($P, M, B, d \leftarrow 0$)
2:      $o \leftarrow 1 + (d \mod M)$
3:      $P \leftarrow$ Sort($P, a^o \prec b^o$)
4:      $m \leftarrow P_{1+\lfloor |P|/2 \rfloor}$
5:      $S \leftarrow$ new NDSplit($o, m$)
6:      if $|P| > B$ then
7:          $Better \leftarrow \{P_i, i < 1 + \lfloor |P|/2 \rfloor\}$
8:          $Worse \leftarrow \{P_i, i \geq 1 + \lfloor |P|/2 \rfloor\}$
9:          $S.BetterSplit \leftarrow$ CreateSplits($Better, M, B, d + 1$)
10:         $S.WorseSplit \leftarrow$ CreateSplits($Worse, M, B, d + 1$)
11:     end if
12:     return $S$
13: end procedure
```

```
NDSplit structure($S$){
    Objective
    Median
    BetterSplit
    WorseSplit
}
```

Figure 7: An example of the CreateSplits procedure that calculates the prebalanced split set for the NDTree.

makes it easier to associate the split with the correct NDTree node. The procedure takes four arguments: the population P, the number of objectives M, the bucket size B, and the depth d. When calling the procedure for the first time, the depth variable is set to the default value of zero. First, the objective o is selected on line 2 in Figure 7 based on the depth d of the procedure. Then the population P is sorted on line 3 based on the oth objective. The oth objective value is selected from the middle solution $P_{1+\lfloor |P|/2 \rfloor}$ and is stored as the split value m on line 4. The split is created with the objective index o and the median m on line 5, to keep track of both the split value and the objective with which the node will be associated. If the number of solutions in the population is larger than the bucket size on line 6, then the population is split into two equal sized populations, $Better$ and $Worse$, on lines 7 and 8, respectively. After that, CreateSplits is called two more times on both the $Better$ and the $Worse$ populations with increasing depth. The resulting splits are stored in the sub-splits $S.BetterSplit$ and $S.WorseSplit$ on lines 9 and 10, respectively. This recursively creates all the splits until the population size is less than or equal to the bucket size. Finally, the prebalanced split set is returned on line 12.

The described sorting approach is used to create the prebalanced split set because it is easy to implement and efficient. It should, however, be noted that other methods can be used as well. For example, Brown (2015) proposes a method that presorts the population M times, once for each objective. For each level in the recursion, the solutions are split into two but shifted to maintain the internal order. A divide-and-conquer approach can also be used in which each recursion finds the median (Bentley, 1975) via the selection procedure explained by Blum et al. (1973). The population is then split into two based on the median value.

The binary search procedure FrontIndexBinarySearch is shown in Figure 8. This procedure searches through the existing NDTree front set $\mathcal{T}$, to determine the front to which solution s belongs, i.e. the first front that does not dominate the solution. The procedure works with two indices, the lower index i and the upper index j on lines 2 and 3, respectively, in Figure 8. The upper index j is set to $|\mathcal{T}| + 1$ because the solution can be dominated by all the fronts, which indicates that a new front needs to be created. The procedure continues until $i = j$; then both indices point to the front index to which the solution belongs. On line 5, the index k is calculated based on the center

```
1:  procedure FrontIndexBinarySearch($\mathcal{T}, s$)
2:      $i \leftarrow 1$
3:      $j \leftarrow |\mathcal{T}| + 1$
4:      while $i \neq j$ do
5:          $k \leftarrow \lfloor i + (j - i)/2 \rfloor$
6:          if FrontDominates($\mathcal{T}_k, s$) then
7:              $i \leftarrow k + 1$
8:          else
9:              $j \leftarrow k$
10:         end if
11:     end while
12:     return $i$
13: end procedure
```

Figure 8: Procedure to find the index of the first NDTree (representing a front) which does not contain any solution that dominates solution s.

index between the two indices i and j. Then on line 6 the procedure checks whether the NDTree $\mathcal{T}_k$ dominates the solution s using the FrontDominates procedure. If the solution s is dominated then the lower index i is set to $k + 1$ on line 7. It is set to $+1$ because, in this case, it is dominated, so it cannot belong to the front index i. If it is not dominated then the upper index j is set to k on line 9. In this case, the solution is not dominated, which means that the front index k is a possible candidate. Finally, on line 12 the procedure returns the front index i. It is worth to notice that since the procedure is based on a binary search algorithm it searches through at most $\log |\mathcal{T}|$ fronts, where $|\mathcal{T}|$ is the number of fronts, and is therefore efficient.

The FrontDominates procedure is shown in Figure 9 and takes two arguments as input: the NDTree T, and the solution s. It calls the NodeDominates procedure on line 2 to check whether or not the root node of T dominates solution s. The NodeDominates procedure in Figure 9 takes two arguments: node N and solution s. If the node is a branch on line 5, i.e. containing node children, then the objective and median from the node's associated split is stored in o on line 6 and m on line 7. Next, three conditions are checked in the IF-statement on line 8; if any condition is false, the procedure skips directly to line 11 without evaluating the remaining conditions in the IF-statement. The first condition of the IF-statement checks whether the node N has a WorseNode. The second condition checks whether the node's split is not dominated by the solution s, by checking if the objective value o of solution s does not dominates the median value m. The third condition checks whether the WorseNode dominates the solution s by recursively calling the NodeDominates procedure. If all conditions are true, then the procedure returns "true" on line 9. Otherwise, the procedure continues to line 11 and checks whether BetterNode exists; if so, it checks whether BetterNode dominates solution s. If solution s is dominated, then the procedure returns "true"; otherwise, it returns "false". On line 5, the procedure checks if node N is a branch, however, if the node is a leaf, i.e. a bucket containing the solutions, then the procedure skips directly to line 13 calling the LeafDominates procedure. The LeafDominates procedure is a straightforward procedure that takes two arguments, the leaf node N, and the solution s. This procedure checks solution s against the entire content of leaf node N on lines 15–20. If the solution s is dominated on line 17, then the procedure return "true" directly on line 18 without having to check the rest of the solutions in the node.

```
1:  procedure FrontDominates($T, s$)
2:      return NodeDominates($T.RootNode, s$)
3:  end procedure

4:  procedure NodeDominates($N, s$)
5:      if $N$ is branch then
6:          $o \leftarrow N.Split.Objective$
7:          $m \leftarrow N.Split.Median$
8:          if $N.WorseNode \neq \emptyset$ and $s^o \nprec m$ and
                NodeDominates($N.WorseNode, s$) then
9:              return true
10:         end if
11:         return $N.BetterNode \neq \emptyset$ and
                NodeDominates($N.BetterNode, s$)
12:     end if
13:     return LeafDominates($N, s$)
14: end procedure

15: procedure LeafDominates($N, s$)
16:     for $i = 1, ..., |N|$ do
17:         if $N_i \prec s$ then
18:             return true
19:         end if
20:     end for
21:     return false
22: end procedure
```

$T.RootNode$ top Node of NDTree T

Node structure(N){
  Split
  BetterNode
  WorseNode
}

NDSplit structure(S){
  Objective
  Median
  BetterSplit
  WorseSplit
}

s^o is the oth objective value in s
N_i is the ith solution in Node N

Figure 9: The special procedure to determine whether a solution s is dominated by the NDTree T, which represents a front.

If it is not dominated, then the procedure returns “false” on line 21.

This search procedure exploits the fact that if solution s dominates the split, then the solutions in the WorseNode can never dominate solution s. This constraint is true because domination is fulfilled only when all objectives are equally good or better and at least one objective is strictly better.

3.4 Examples of the ENS-NDT algorithm

Figure 10 illustrates how the ENS-NDT algorithm sorts a population step by step. This example is simplified with only two minimization objectives, and the bucket size is set to one. The example is meant to illustrate the basic functionality and therefore includes both objectives in the NDTree, but in the real algorithm the last objective would be excluded in the NDTree.

In the algorithm, all split values first need to be created, as shown in the “Create splits” image of Figure 10. Then the population is presorted to ensure that the latter solutions cannot dominate the former solutions. When the presort is done the first solution can be added to the first front directly, this is possible because the latter solutions cannot dominate the former. After these first steps the main loop, called ndsort, starts. In this loop, all remaining solutions are added into the front set, as shown from image “NDSort start” to image “NDSort end” in Figure 10. Each iteration of the ndsort includes two steps; the first step finds the first front that does not contain any solution

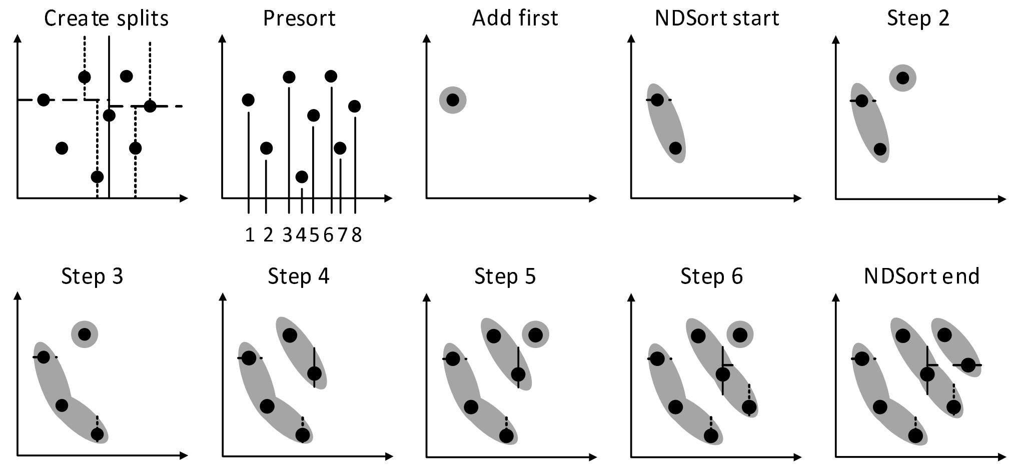

Figure 10: Illustration of the process of ENS-NDT with bucket size one. The example has two minimization objectives, each solution is represented as a point, and each front is represented with a grey area containing points. The lines in each front represents the splits, from the create splits procedure, that are used in that front.

that dominates the new solution, and the second step adds the new solution to that front (both the NDTree front and the regular front). As each solution is added to a front, the NDTree in that front will split the solutions when a bucket overflows. In Figure 10 this is shown by the split line in the grey area, where the grey areas represents the fronts. Each front will not automatically have the same splits activated, because the splits are determined by the added solutions. This also means that if several fronts exists the NDTree in each front will be unbalanced, as seen in the image "NDSort end" in Figure 10 where the second front only have three splits active out of seven.

Two examples are presented in Figure 11 to demonstrate the domination check procedure. In these examples, the solutions have three minimization objectives: f1, f2 and f3. All the solutions in the population are non-dominated and the same population is used in both examples. In the examples, only f1 and f2 are used in the NDTrees; the last objective, f3, can be ignored because the presort has already sorted the population based on objective f3. The prebalanced split values are illustrated using lines: vertical lines represents split values in objective f1, while horizontal lines represents split values in objective f2. The solid line represents the first depth, the dashed lines the second depth and the dotted lines the third depth. Furthermore each line has a letter assigned to it, from A to G, which is used later on when further explaining the domination check procedure. The arrows indicate the direction of the procedure, while the number indicates the step in the process. Only minimization objectives are used and therefore right and upwards are the "worse" directions, while left and downwards are the "better" directions. The points represent solutions used in the tree; filled points represent solutions that were used in the domination check and the crossed point represents the new solution. Points with an arrow represent solutions which have the same objective value as the split line and the arrow indicates which bucket the solution belongs to.

The two examples in Figure 11 are described step-by-step below (note that the

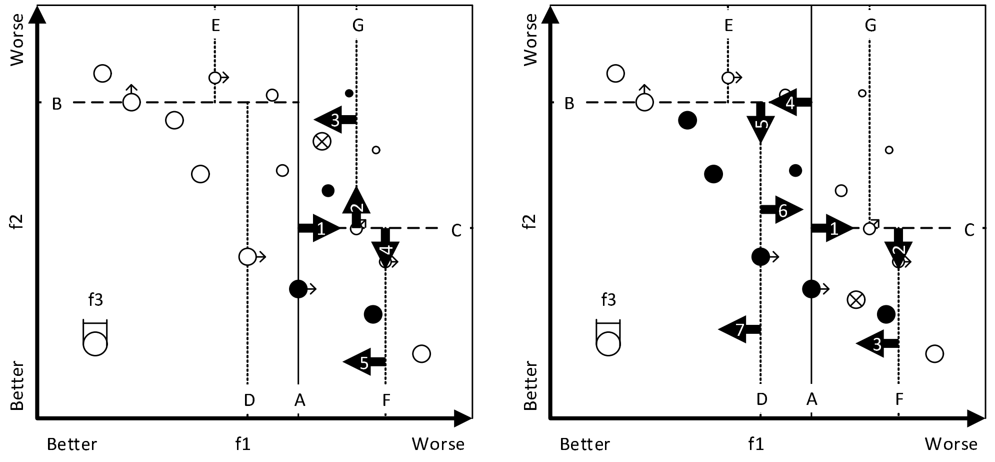

Figure 11: Two examples of the domination check procedure with an NDTree. To the left a dominated solution and to the right a non-dominated solution that is tested against the same NDTree. There are three objectives and the bucket size is set to two.

number preceding each step correlates to the numbers in the arrows in Figure 11). The notation $s^{f1} \nprec A$ means that the value in objective f1 of solution s does not dominate split A, while $s^{f1} \prec A$ means that it does. The left image in Figure 11 demonstrates the domination check procedure when a dominated solution s is compared with the NDTree T:

1. $s^{f1} \nprec A \rightarrow$ "worse" direction
2. $s^{f2} \nprec C \rightarrow$ "worse" direction
3. $s^{f1} \prec G \rightarrow$ "better" direction
   The leaf node does not contain any solution that dominates solution s
4. Go to last "worse" turn at split C $\rightarrow$ "better" direction
5. $s^{f1} \prec F \rightarrow$ "better" direction
   The leaf node contain a solution that dominates solution s, return true

The right image in Figure 11 demonstrates the domination check procedure when a non-dominated solution s is compared with the NDTree T:

1. $s^{f1} \nprec A \rightarrow$ "worse" direction
2. $s^{f2} \prec C \rightarrow$ "better" direction
3. $s^{f1} \prec F \rightarrow$ "better" direction
   The leaf node does not contain any solution that dominates solution s
4. Go to last "worse" turn at split A $\rightarrow$ "better" direction
5. $s^{f2} \prec B \rightarrow$ "better" direction
6. $s^{f1} \nprec D \rightarrow$ "worse" direction
   The leaf node does not contain any solution that dominates solution s
7. Go to last "worse" turn at split D $\rightarrow$ "better" direction
   The leaf node does not contain any solution that dominates solution s
   No more "worse" turn, i.e. solution s is non-dominated, return false

The advantage of this procedure is that it is not necessary to compare solution s with the entire front's solutions due to the NDTree. These examples show that less than half of the solutions are compared (it should, however, be noted that the number of comparisons varies when the population size is increased).

4 Complexity analysis

This section analyzes the complexity of the ENS-NDT algorithm, starting with an in-depth analysis of the time complexity followed by an analysis of the space complexity at the end of the section.

The time complexity of the ENS-NDT algorithm $T(N, M)$ differs between its three main steps (presort, prebalance splits, and ndsort) and is therefore analyzed separately for each of these steps. The factors to be considered in the analysis are the population size N and the number of objectives M.

$$T(N, M) = T_{presort}(N, M) + T_{splits}(N, M) + T_{ndsort}(N, M)$$

In the presort step $T_{presort}(N, M)$, the worst, average, and best case time complexities are $O(N \log N)$ if using for example the MergeSort or the HeapSort algorithm (Weiss, 2006). Each comparison, however, compares all M objectives in reverse lexicographic objective dominance order. In the worst case, when all objective values are the same between all solutions, the total time complexity of the presort is $O(MN \log N)$.

$$T_{presort}(N, M) = O(MN \log N)$$

When considering the time complexity in the second step, the prebalanced splits $T_{splits}(N, M)$, the variable M does not need to be considered as it does not affect the time complexity (the execution time to prebalance the splits is not affected by the number of objectives). To calculate the prebalanced split set, a recursive sorting algorithm is used to find all the median values. First, the population of size N is sorted based on one objective, then all solutions are split into two new populations of size $N/2$ and the split function is called for those two populations. When the population size N is less than or equal to the bucket size, then the recursion stops. The time complexity of this kind of recursive sorting procedure is $O(N \log^2 N)$, if a sort method that can guarantee $O(N \log N)$ is used (Brown, 2015).

$$T_{splits}(N, M) = O(N \log^2 N)$$

This split function approach has a time complexity of $O(N \log^2 N)$ but can be changed to $O(MN \log N)$ if using the algorithm developed by Brown (2015). The time complexity can even be reduced to $O(N \log N)$, if the median of each recursion step is found within a time of $O(N)$ (Bentley, 1975). This is possible using selection algorithms that use linear search time (Blum et al., 1973); however, these algorithms are quite complex, as mentioned by Brown (2015).

The third step is the ndsort loop $T_{ndsort}(N, M)$, in which all solutions are added to the front set. This step comprises two parts. First, the front index needs to be determined using the binary search procedure, which for each comparison checks whether an NDTree dominates the solution. Second, the solution is added to the Front set and inserted to the NDTree set.

$$T_{ndsort}(N, M) = T_{search}(N, M) * T_{check}(N, M) + T_{add}(N, M) + T_{insert}(N, M)$$

Adding all solutions to the Front set $T_{add}(N, M)$, has a time complexity of $O(N)$ because each addition is constant. However, the NDTree has a tree structure that, if perfectly balanced, has a worst case insertion time complexity of $O \log N$ (Bentley, 1975). When accumulated for N solutions, the NDTree insertion $T_{insert}(N, M)$, will result in a total time complexity of $O(N \log N)$.

$$T_{add}(N, M) = O(N)$$
$$T_{insert}(N, M) = O(N \log N)$$

The time complexity of the ndsort loop $T_{ndsort}(N, M)$, is dependent on the number of fronts. Two extreme cases are therefore analyzed, one in which all solutions belong to different fronts and one when all solutions belong to the same front. When all solutions belong to different fronts, then the time complexity of the ndsort loop is the same as the best time complexity of the ENS-BS algorithm $T_{best}(N, M)$, namely $O(MN \log N)$. The split function has a time complexity of $O(N \log^2 N)$, which means that the best case time complexity of the ENS-NDT algorithm $T_{best}(N, M)$, is $O(N \log^2 N)$ if $\log N > M$; otherwise, it is $O(MN \log N)$.

$$T_{ndsort}(N, M) = O(MN \log N)$$
$$T_{best}(N, M) = O(MN \log N) + O(N \log^2 N) + O(MN \log N)$$
$$T_{best}(N, M) = \begin{cases} O(MN \log N), & M > \log N \\ O(N \log^2 N), & otherwise \end{cases}$$

When all solutions belong to one front, then the binary search for the front will be constant but the NDTree domination check will instead run on N solutions. The worst case scenario is when the NDTree is forced to have only one bucket, e.g. when $M \geq \log N + 2$) and all objective values are the same except the last two which fulfill the non-dominated criteria. Each check will go through the depth of the tree of $\log N$ and then compare with all solutions N. Each comparison requires checking all M objectives since only the last two objectives fulfill the non-dominated criteria. The check has in this case a time complexity of $O(MN^2)$, which will be the time complexity of the ndsort $T_{ndsort}(N, M)$. Since ndsort in this case sets the upper bound of the whole algorithm, the worst case time complexity of the ENS-NDT algorithm $T_{worst}(N, M)$, is $O(MN^2)$.

$$T_{ndsort}(N, M) = O(MN^2)$$
$$T_{worst}(N, M) = O(MN \log N) + O(N \log^2 N) + O(MN^2)$$
$$T_{worst}(N, M) = O(MN^2)$$

Calculating the average case time complexity of the NDTree domination check is non-trivial and is therefore estimated based on experiments. These experiments are presented in section 5. The average time complexity is the largest of the calculated best case time complexity $T_{best}(N, M)$, and $T_{search}(N, M) * T_{check}(N, M)$.

$$T_{average}(N, M) = O(MN \log N) + O(N \log^2 N) + T_{search}(N, M) * T_{check}(N, M)$$
$$T_{average}(N, M) = \max \left(\begin{Bmatrix} O(MN \log N) \\ O(N \log^2 N) \\ T_{search}(N, M) * T_{check}(N, M) \end{Bmatrix} \right)$$

Table 1: Summary of the space and time complexities for the algorithms ENS-NDT, DCNS-F, ENS-BS and FNS.

Algorithm	Space complexity	Time complexity
ENS-NDT	Worst: $O(N \log N)$	Worst: $O(MN^2)$
	Average: $O(N)$	Average: $\max\left(\left\{\begin{array}{c} O(MN\log N) \\ O(N\log^2 N) \\ T_{search}(N,M) * T_{check}(N,M) \end{array}\right\}\right)$
	Best: $O(\log N)$	Best: $T(N,M) = \begin{cases} O(MN\log N), & M > \log N \\ O(N\log^2 N), & otherwise \end{cases}$
DCNS-F	$O(MN)$	Worst: $O(MN^2)$
		Average: $O(N\log^{M-1} N)$
		Worst: $O(N\log N)$
ENS-BS	$O(1)$	Worst: $O(MN^2)$
ENS-BS		*Average: $O(MN^2)$
		Best ENS-BS: $O(MN\log N)$
		Best ENS-SS: $O(MN\sqrt{N})$
FNS	$O(N^2)$	Worst: $O(MN^2)$

* When used in evolutionary optimization, in these cases the population eventually converges toward one Pareto optimal front.

Besides the two analyzed cases there are additional cases that are of interest to consider. One such case is when the solutions are grouped into different fronts then all the NDTrees will be unbalanced, degrading the performance of each NDTree check. However, due to the decreased size of each NDTree, there are fewer solutions to check, so the overall performance is assumed to remain balanced. The time complexity of the search procedure and the NDTree domination check, when solutions are grouped into different fronts is estimated using experiments in section 5.

The space complexity of the algorithm is dependent solely on the extra memory usage for the NDTree set. This is because the three stages only work to a maximum depth of $O(\log N)$, where N is the population size, while the extra memory usage has a space complexity of at least $O(N)$ when only unique solutions exist. If there are $B + 1$ solutions, where B is the bucket size, in each front and the splits are configured so that each NDTree has its solutions in the same bucket, i.e. only one leaf node, then each NDTree has a depth of $\log N - \log B$. The size of the NDTree set is in this case $N(\log N - \log B)/(B+1)$, and because B is constant the space complexity is $O(N \log N)$. If all solutions are identical then the NDTree set would contain only one solution, which means that the extra memory usage would be $O(\log N)$ for the sorting algorithms. So the worst case space complexity is $O(N \log N)$, the average case is $O(N)$, and the best case is $O(\log N)$.

Table 1 summarizes the time and space complexities for ENS-NDT, DCNS-F, ENS-BS, ENS-SS and FNS. Zhang et al. (2015) have not defined an average case time complexity for the ENS-BS and the ENS-SS algorithms. However, these algorithms are normally used in evolutionary optimization and since evolutionary optimization eventually produces populations with few fronts, ENS-SS and ENS-BS converge towards the worst case time complexity which is $O(MN^2)$. Since this is the normal usage the average time complexity of ENS-SS and ENS-BS is $O(MN^2)$.

5 Evaluation

To verify how the new algorithm performs in comparison with the existing algorithms, the FNS (Deb et al., 2002a), ENS-BS and ENS-SS (Zhang et al., 2015), and DCNS-F (Fortin et al., 2013) algorithms were implemented. All the algorithms and experiments were implemented using the C# programming language and the experiments were executed on an Intel Core i7-4790 3.6GHz processor. Three experiments were conducted to evaluate the new algorithm. For the first two experiments the runtime was measured for all five algorithms and compared to evaluate the performance difference between them. For the last experiment the numbers of comparisons were counted to estimate the average time complexity of $T_{search}(N, M) * T_{check}(N, M)$, which was difficult to mathematically prove as stated in section 4.

The bucket size of the NDTrees in the ENS-NDT algorithm was set to two in all experiments. This bucket size was selected because the performance difference between bucket size one and two is noticeable (approximately 25% more processing time using bucket size one), but setting the bucket size to more than two offers no substantial improvement.

The evaluation consists of three parts. Subsection 5.1 describes the first part, which evaluates how the different algorithms perform with regards to different Pareto-front setups (randomly generated population e.g. several fronts and population with only one front). Subsection 5.2 describes the second part, which evaluates how the different sorting algorithms perform in a multi-objective evolutionary algorithm. Subsection 5.3 describes the third part, which evaluates the average time complexity of the ENS-NDT algorithm, because it could not be determined in the complexity analysis.

5.1 Experiment with different Pareto front setups

To evaluate the algorithm, it was tested with two different population types representing the population at the start and at the end of a typical optimization procedure. The start type was a completely randomly generated population in which the solutions end up in several different fronts. The end type was a population of solutions that all belong to the same front but are randomly generated. The solutions were generated with (1) for the start population and (2) for the end population, where s is the solution, i the objective index, and M the number of objectives. By testing the algorithms on these population types, the behavior of the algorithms could be analyzed.

$$s^i = Rand(0,1), \quad i = 1, ..., M \tag{1}$$

$$s^i = \begin{cases} Rand(0,1), & i < M \\ 1 - \sum_{j=1}^{M-1} \dfrac{s^j}{M-1}, & i = M \end{cases} \tag{2}$$

All experiments were repeated 20 times for each population size and the average processing time in CPU milliseconds was calculated. The population size starts from 100 and doubles until it reaches 3200. All experiments were conducted using three to eight objectives and the results with three and eight objectives are shown in Figure 12, Table 2 and Table 3. For the sake of visibility only the best performing strategy for the ENS algorithm is shown in the log-log plots. To make the tables easier to read only population sizes 200, 800 and 3200 are shown.

The results indicate that the ENS-NDT algorithm outperforms both the FNS and DCNS-F algorithms for all population types, population sizes, and numbers of objectives. When increasing the number of objectives, the performance differences between

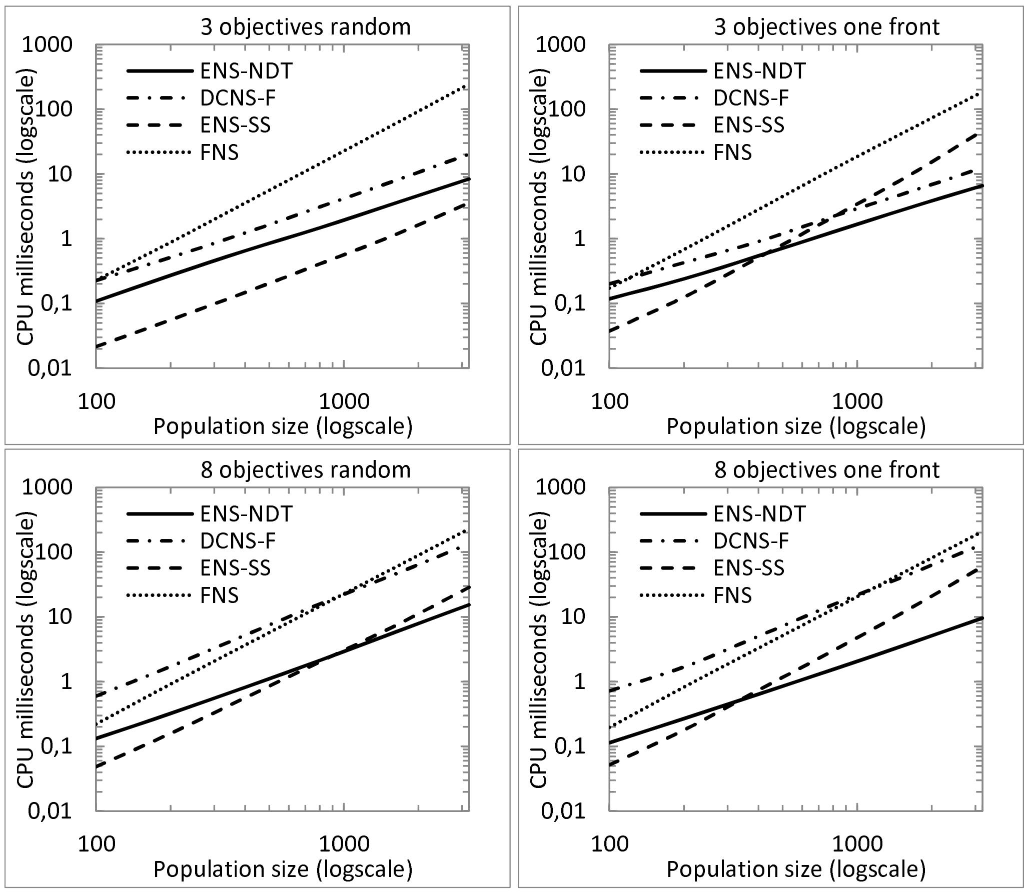

Figure 12: Processing times of the four algorithms for populations with randomly generated solutions to the left and randomly generated populations where all solutions belong to one front to the right for three and eight objectives.

DCNS-F and ENS-NDT expand in favor of ENS-NDT. For three objectives, DCNS-F is approximately two times slower than ENS-NDT for all population types and population sizes. For eight objectives and populations with one front, DCNS-F is approximately six times slower than ENS-NDT with a population size of 200 but almost 14 times slower with a population size of 3200 when the population has only one front.

The results indicate that the ENS algorithms are better than the ENS-NDT algorithm for smaller population sizes. ENS-SS outperforms ENS-NDT up to a population size of at least 3200, when the population is completely random with three objectives. With a population size of 200, ENS-NDT is approximately 4.5 times slower and with a population size of 3200 it is approximately 2.3 times slower. If the number of objectives is increased to eight then ENS-NDT is almost 2.5 times slower than ENS-SS with a population size of 200. If the population size is increased to 3200, then ENS-SS is approximately 1.8 times slower than ENS-NDT. If all the solutions belong to the same front, then the relationship between the ENS algorithms and the ENS-NDT algorithm

Evolutionary Computation Just Accepted MS.
doi:10.1162/EVCO_a_00204

Table 2: Means and Standard Deviations of Runtimes in milliseconds of all algorithms on three objectives.

Algorithm	Random population type			One front population type		
	200	800	3200	200	800	3200
ENS-NDT	0.27 (0.05)	1.47 (0.12)	8.33 (0.79)	0.24 (0.03)	1.27 (0.21)	6.57 (0.69)
DCNS-F	0.52 (0.10)	3.10 (0.22)	20.19 (0.90)	0.43 (0.20)	2.22 (0.25)	12.48 (0.86)
ENS-BS	0.07 (0.01)	0.56 (0.03)	5.74 (0.26)	0.15 (0.002)	2.57 (0.07)	53.15 (0.45)
ENS-SS	0.06 (0.01)	0.41 (0.02)	3.57 (0.13)	0.12 (0.002)	2.16 (0.07)	46.19 (0.43)
FNS	0.88 (0.04)	14.40 (0.47)	245.10 (7.51)	0.68 (0.03)	11.90 (0.44)	185.07 (1.76)

Table 3: Means and Standard Deviations of Runtimes in milliseconds of all algorithms on eight objectives.

Algorithm	Random population type			One front population type		
	200	800	3200	200	800	3200
ENS-NDT	0.32 (0.04)	2.12 (0.07)	15.31 (0.93)	0.27 (0.02)	1.54 (0.15)	9.56 (0.47)
DCNS-F	1.71 (0.10)	15.86 (0.84)	134.67 (1.85)	1.68 (0.10)	15.41 (1.04)	131.39 (1.79)
ENS-BS	0.20 (0.01)	2.79 (0.10)	45.39 (1.57)	0.20 (0.02)	3.47 (0.07)	67.22 (0.94)
ENS-SS	0.16 (0.01)	2.02 (0.11)	28.47 (0.92)	0.18 (0.01)	3.01 (0.08)	59.74 (0.66)
FNS	0.92 (0.02)	14.63 (0.17)	230.02 (1.62)	0.82 (0.02)	13.12 (0.23)	208.05 (1.00)

is similar, even if the number of objectives is increased. At a population size of 200, ENS-NDT is 1.5-2 times slower than ENS-BS; the threshold at which ENS-NDT performs better is at a population size of approximately 300–500, while at a population size of 3200, the ENS algorithms are at least six times slower.

These results clearly indicate that to obtain the best performance from the non-dominated sort, only the ENS-SS and ENS-NDT algorithms should be used. The threshold for when to use one or the other is dependent on the number of objectives, how many fronts exist, and the population size. For three objectives it is difficult to determine the population size threshold because it differs too much between the population types. However, for eight objectives, the population size threshold lies between 300 and 800.

The ENS-NDT algorithm is slower than the ENS algorithms for smaller population sizes because of the overhead processing. ENS-NDT has some processes that ENS lacks, such as calculating the prebalanced split set and the insertion of solutions into the NDTree. If these processes have processing times longer than the time saved by the improved front domination, then the ENS-NDT algorithm will be slower.

5.2 Experiment with a multi-objective evolutionary algorithm

The algorithms must be evaluated using a multi-objective optimization algorithm applying the Pareto concept, because this is the primary use of non-dominated sorting. Several such algorithms have been suggested, for example, Multi-Objective Genetic Algorithms (MOGA) (Murata and Ishibuchi, 1995), the Niched-Pareto Genetic Algorithm (NPGA) (Horn et al., 1994), the Pareto-Archived Evolution Strategy (PAES) (Knowles and Corne, 1999) and the Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002a). In this study, NSGA-II was implemented because it is a commonly used state-of-the-art algorithm in multi-objective evolutionary optimization and has been

Table 4: Settings for the NSGA-II algorithm.

Settings	Value
Generations	300
Crossover algorithm	Simulated Binary Crossover (SBX)
Crossover probability	0.95
SBX distribution index	2
Mutation algorithm	Polynomial mutation (PM)
Mutation probability	0.05
PM distribution index	5

proven to successfully solve complex real-world problems in various domains.

Unlike an ordinary evolutionary algorithm, NSGA-II selects solutions for the next generation of the population based on Pareto ranks (Deb et al., 2002a). More specifically, the solutions for the next generation are selected from set R, which is the union of the parent and offspring populations (both of size N). Non-dominated sorting is applied to R, and the next generation of the population is formed by selecting solutions from one of the Pareto fronts at a time. The selection starts with solutions in the best Pareto front, then continues with solutions in the second best front, and so on, until N solutions have been selected. If there are more solutions in the last front than there are remaining to be selected, the crowding distance is calculated to determine which solutions should be chosen; all of the remaining solutions are discarded.

To evaluate the non-dominated sort using the NSGA-II algorithm, benchmark problems are necessary. The DTLZ problems (Deb et al., 2002b) are commonly used as benchmark problems when evaluating multi-objective optimization. These problems have known Pareto-optimal fronts, which are useful when determining the quality of the optimization algorithms. The DTLZ problems are scalable and can handle multiple objectives, which is necessary in this case. For these experiments, DTLZ1 and DTLZ2 were selected. DTLZ1 has a linear Pareto-optimal front in hyper-space while DTLZ2 has a spherical Pareto-optimal front in hyper-space.

To optimize DTLZ1 and DTLZ2 benchmark problems the NSGA-II algorithm had the settings shown in Table 4. These settings were experimentally selected by testing different settings on a population size of 100 with three objectives and then visually observing the outcome. In this case, settings were selected based on whether NSGA-II could produce a population with as optimal solutions as possible. This also means that the produced population will have as few fronts as possible, or ultimately one Pareto-optimal front. To put pressure on the different non-dominated sorting algorithms, the population should go from having many different fronts to only a few fronts. When increasing the population size, the number of fronts initially increases, but because there are more diverse solutions, NSGA-II will find a near Pareto-optimal front more quickly. This means that the non-dominated sorting algorithms will be tested with more Pareto-front setups when increasing the population size.

The DTLZ problems have two settings: number of inputs, and number of outputs. When running the experiments the number of outputs is already known. However, the number of inputs needs to be determined. In the DTLZ problems the number of inputs n is determined by $n = M+k-1$, where M is the number of outputs and k the constant, which varies between DTLZ problems. As suggested by Deb et al. (2002b), the input constant was set to $k = 5$ for the DTLZ1 problem and $k = 10$ for the DTLZ2 problem.

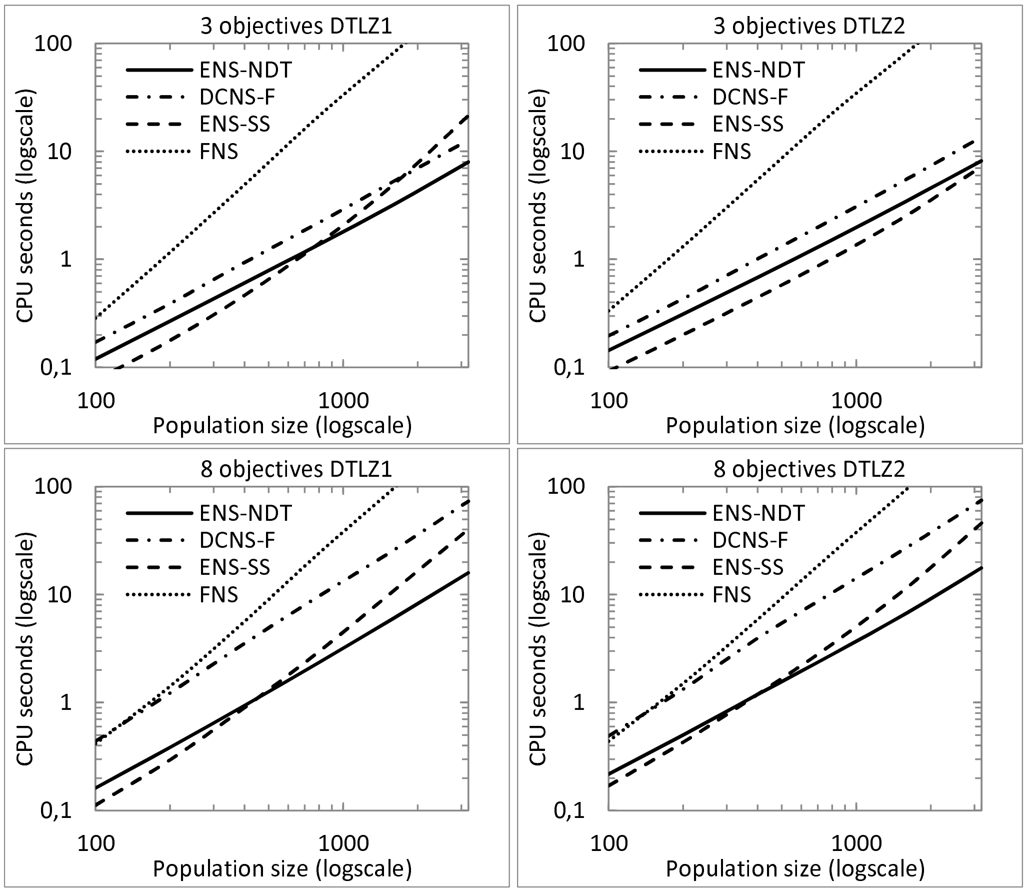

Figure 13: Processing times of the NSGA-II algorithm for the DTLZ1 problem to the left and the DTLZ2 problem to the right using the four algorithms for three and eight objectives.

All experiments were repeated 20 times for each population size and the average processing time in CPU seconds was calculated. The population size starts from 100 and doubles until it reaches 3200. The population size in these experiments refers to the population size for the NSGA-II algorithm. NSGA-II performs all non-dominated sorts on two combined populations, which means that the non-dominated sorts are performed using a doubled population size. All experiments have been conducted using three to eight objectives and the results with three and eight objectives are shown in Figure 13, Table 5 and Table 6. For the sake of visibility only the best performing strategy for the ENS algorithm is shown in the log-log plots. To make the tables easier to read only population sizes 200, 800 and 3200 are shown.

The results indicate a behavior similar to that described in subsection 5.1. However, the performance differences are smaller in these experiments, which can be explained by other time-consuming operations of the NSGA-II algorithm. It can be seen that changing only the non-dominated sort can substantially change the runtime per-

Table 5: Means and Standard Deviations of Runtimes in milliseconds of all algorithms used in NSGA-II, on three objectives.

Algorithm	DTLZ1			DTLZ2		
	200	800	3200	200	800	3200
ENS-NDT	268 (2)	1376 (20)	7964 (202)	312 (5)	1521 (13)	8165 (213)
DCNS-F	390 (6)	2198 (30)	12711 (68)	433 (6)	2342 (55)	13594 (227)
ENS-BS	206 (5)	1915 (97)	29681 (1136)	229 (8)	1376 (57)	12595 (579)
ENS-SS	178 (5)	1408 (79)	21381 (1061)	201 (4)	1026 (36)	7242 (262)
FNS	1163 (53)	21357 (633)	303446 (4456)	1320 (97)	22552 (219)	309149 (3168)

Table 6: Means and Standard Deviations of Runtimes in milliseconds of all algorithms used in NSGA-II, on eight objectives.

Algorithm	DTLZ1			DTLZ2		
	200	800	3200	200	800	3200
ENS-NDT	385 (15)	2359 (85)	15895 (355)	503 (11)	2792 (43)	17630 (205)
DCNS-F	1228 (54)	9723 (286)	73067 (1923)	1328 (58)	10399 (191)	74465 (768)
ENS-BS	334 (30)	3757 (401)	55661 (5489)	471 (20)	4348 (169)	61755 (1184)
ENS-SS	294 (30)	2977 (359)	40258 (4427)	430 (20)	3477 (150)	45720 (1013)
FNS	1420 (23)	24352 (408)	387095 (9550)	1508 (17)	24216 (207)	383062 (2689)

formance of the NSGA-II algorithm. The results indicate that the ENS-NDT algorithm outperforms both FNS and DCNS-F for all population types, population sizes, and numbers of objectives. For three objectives, NSGA-II using DCNS-F takes 30–70% more processing time than when using ENS-NDT. When the number of objectives is increased to eight, then NSGA-II using DCNS-F is two to five times slower than when using ENS-NDT.

The ENS-SS sort is the fastest when using a population size below 300 for the NSGA-II algorithm. When using only three objectives, NSGA-II with ENS-SS outperforms all other algorithms up to a population size of 800 for the DTLZ1 problem and up to a population size of at least 3200 for the DTLZ2 problem. With a population size of 200, NSGA-II requires 10–50% more processing time with ENS-NDT than with ENS-SS; however, with a population size of 3200, NSGA-II is up to 2.6 times slower with ENS-BS than with ENS-NDT.

These results clearly indicate that the ENS-NDT and ENS-SS algorithms are the only algorithms necessary to obtain the best performance, as was the case in subsection 5.1. The population size threshold for when to use one algorithm or the other is not clear for three objectives but is well defined for eight objectives. For three objectives, the population size threshold differs between the problems and lies from 800 and somewhere above 3200; however, for eight objectives, the threshold lies at a population size in between 400 and 500. With an increasing number of objectives, the ENS-NDT algorithm becomes better than the ENS-SS and ENS-BS algorithms. The reason is, with more objectives the number of fronts decreases, and the efficiency of the ENS algorithms deteriorates with fewer fronts.

Table 7: Numbers of Comparisons for ENS-NDT.

Population size	3 objectives		8 objectives	
	One front	Random	One front	Random
100	1186 (72)	2245 (104)	2393 (157)	5293 (665)
400	6620 (186)	14531 (493)	16104 (640)	42938 (2601)
1600	35037 (859)	84822 (2228)	112241 (2613)	343214 (13247)
6400	174876 (1726)	465723 (8329)	783960 (8528)	2537209 (107908)
25600	850135 (3055)	2442700 (24241)	5366916 (61227)	16919556 (546709)
102400	4028589 (13606)	12416252 (84873)	35145911 (211264)	108253751 (4204601)

5.3 Experiment to estimate the average time complexity of ENS-NDT

As the average case time complexity is difficult to calculate mathematically it is instead estimated using experiments. The parts that could not be calculated were the time complexity of the search and domination check procedures $T_{search}(N, M) * T_{check}(N, M)$. To estimate the time complexity for these parts the numbers of comparisons are measured in the algorithm, the measurement include both the comparisons with the splits, and the comparisons for each objective value. The reason comparisons are measured instead of runtime, is to remove the impact of the create splits, presort and insertion procedures. All experiments were repeated 20 times with population sizes starting with 100 and doubling up to 102,400. The population types used in this experiment are the same as those defined in subsection 5.1 being one random population and one population where all solution belong to the same front. The algorithm does not need to be tested against populations where all solution belongs to different fronts, because that time complexity can be mathematically proven to be $O(N \log^2 N)$. The large population size of 102,400 were used to get a more precise estimation of the time complexity. The experiments were conducted using three to eight objectives and the results with three and eight objectives are shown in Figure 14 and Table 7. Figure 14 shows the results in a log-log plot comparing the measured numbers of comparisons with fitted mathematical equations. Table 7 shows the mean and standard deviation of numbers of comparisons, rounded to the closest integer. To make the table easier to read, every other population size are shown starting with population size 100.

To find the average time complexity of $T_{search}(N, M) * T_{check}(N, M)$ in the ENS-NDT algorithm, different mathematical equations are fitted to the measurements from the experiment. The time complexity can then be visually estimated by confirming whether or not the growth of the equation, when increasing N, is the same as the growth of the measured number of comparisons. The equations are shown in the legends of each plot in Figure 14. For three objectives the equations $N \log N$ and $N \log^2 N$ was used, and for eight objectives the equations $N \log^3 N$ and N^x was used, where x is the calculated exponent to best fit the measurements. For the random population type x was calculated to approximately 1.43 and for the population type with one front x was calculated to approximately 1.38. To fit the curve of each mathematical equation with the measured values (numbers of comparisons), a constant is calculated for each mathematical equation. This constant, when multiplied with the mathematical equations, will make all curves start at the same point as the measured values at population size 100.

The results indicate that when the number of objectives are three, the estimated average case time complexity of $T_{search}(N, M) * T_{check}(N, M)$ is $O(N \log^2 N)$. However,

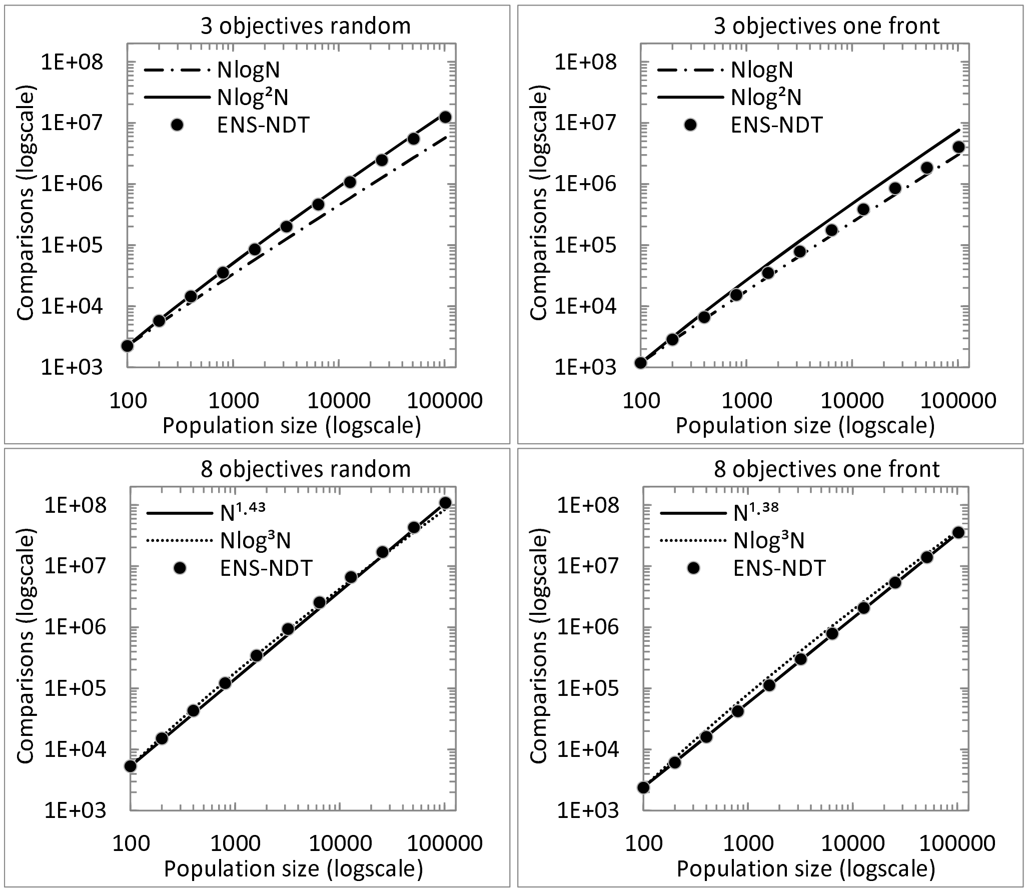

Figure 14: Numbers of comparisons of the ENS-NDT algorithm compared with curves from different mathematical equations curves, for three and eight objectives, for populations with randomly generated solutions to the left and for randomly generated populations where all solutions belong to one front to the right.

when the number of objectives increases to eight, the average time complexity changes and fits better in a power equation, where the random population type shows an estimated time complexity of $O(N^{1.43})$ and the one front population shows an estimated time complexity of $O(N^{1.38})$. The average time complexity shows a small difference for different population types, where random populations generates a slightly higher estimated time complexity. The reason for this increase is most likely because of the NDTrees, which cannot be perfectly balanced when several fronts exist, making the domination checks less efficient.

The average time complexity of the ENS-NDT algorithm is $O(N \log^2 N)$ when there are three objectives because the precalculated splits procedure sets the upper bound to $O(N \log^2 N)$. However, when increasing the number of objectives to eight, the average time complexity is estimated to $O(N^{1.43})$, because this time complexity grows more quickly than $O(N \log^2 N)$.

6 Conclusions

This paper proposes an extension of the Efficient Non-dominated Sort (ENS) algorithm with Binary Strategy (ENS-BS), called the Efficient Non-Dominated Sort using Non-Dominated Tree (ENS-NDT). This algorithm uses the same base procedure as ENS-BS but adopts a novel tree, developed as part of this study, called the Non-Dominated Tree (NDTree) for quicker non-domination checks. Experiments have demonstrated that this algorithm is efficient when dealing with large population sizes, even with a large number of objectives. The algorithm outperforms the Divide-and-Conquer Non-dominated Sort (DCNS) algorithms in all cases, and with larger population sizes and number of objectives it outperforms the ENS algorithms with both the binary strategy (ENS-BS) and the sequential strategy (ENS-SS).

The best performance of non-dominated sorting can be achieved by using the new ENS-NDT algorithm developed in this study and the ENS algorithms. For smaller population sizes, ENS-SS is preferable, but when increasing the population size then ENS-NDT performs the best. ENS-NDT is an extension of the ENS-BS algorithm and the following approach is suggested for implementing the non-dominated sort: the first step is to implement the ENS-SS algorithm; when the population size needs to be increased, the algorithm can then be extended to ENS-NDT by implementing the binary strategy and the NDTree.

Experiments have been conducted solely on populations in which the solutions have unique objective values. In these cases, the ENS-NDT algorithm has an average time complexity of $O(N \log^2 N)$ for three objectives and an estimated time complexity of $O(N^{1.43})$ for eight objectives. However, when the solutions share identical objective values, the time complexity worsens, possibly reaching $O(MN^2)$ in the worst case. The main problem lies in creating the prebalanced split set for the NDTrees. Configuring the prebalanced split set in such a way that the efficiency of each front domination check is less affected, might improve the algorithm.

This work focused on improving the non-dominated sort procedure and the suggested algorithm was therefore compared with other non-dominated sorting algorithms. However, Drozdík et al. (2015) proposed an alternative approach that reduces the need for non-dominated sorting by keeping track of all non-dominated solutions by dynamically updating a *k*-d tree. When there are enough solutions in the first front, there is no need for non-dominated sorting. In the future, it would be interesting to try and combine Drozdík et al. (2015)'s approach with the ENS-NDT algorithm as both use a *k*-d tree. If the *k*-d tree could be dynamically updated while being integrated into the ENS-NDT algorithm, the non-dominated sorting could be much quicker. This would mean that at the beginning of an evolutionary algorithm when non-dominated sorting is necessary, the ENS-NDT algorithm would be used, and then when the population has enough solutions in the first front, there would be no need for non-dominated sorting.

References

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461.

Brown, R. A. (2015). Building a balanced k-d tree in $o(kn \log n)$ time. *Journal of Computer Graphics Techniques (JCGT)*, 4(1):50–68.

Buzdalov, M. and Shalyto, A. (2014). *A Provably Asymptotically Fast Version of the Generalized Jensen Algorithm for Non-dominated Sorting*, pages 528–537. Springer International Publishing, Cham.

Coello Coello, C. A. (1999). A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1(3):269–308.

Deb, K. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002a). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2002b). Scalable multi-objective optimization test problems. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 1, pages 825–830.

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

Fang, H., Wang, Q., Tu, Y. C., and Horstemeyer, M. F. (2008). An efficient non-dominated sorting method for evolutionary algorithms. *Evolutionary Computation*, 16(3):355–384.

Fortin, F.-A., Grenier, S., and Parizeau, M. (2013). Generalizing the improved run-time complexity algorithm for non-dominated sorting. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 615–622, New York, NY, USA. ACM.

Horn, J., Nafpliotis, N., and Goldberg, D. E. (1994). A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 82–87 vol.1.

Jensen, M. T. (2003). Reducing the run-time complexity of multiobjective eas: The nsga-ii and other algorithms. *IEEE Transactions on Evolutionary Computation*, 7(5):503–515.

Knowles, J. and Corne, D. (1999). The pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimisation. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 1, pages 1–105 Vol. 1.

Mehnen, J., Michelitsch, T., Bartz-beielstein, T., and Henkenjohann, N. (2004). Systematic analyses of multi-objective evolutionary algorithms applied to real-world problems using statistical design of experiments. In *Proceedings Fourth International Seminar Intelligent Computation in Manufacturing Engineering (CIRP ICME'04*, pages 171–178.

Murata, T. and Ishibuchi, H. (1995). Moga: multi-objective genetic algorithms. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 1, page 289.

Srinivas, N. and Deb, K. (1994). Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248.

Weiss, M. A. (2006). *Data structures and algorithm analysis in C++*. Pearson Addison-Wesley, Boston, 3rd edition.

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

Zhang, X., Tian, Y., Cheng, R., and Jin, Y. (2016). A decision variable clustering-based evolutionary algorithm for large-scale many-objective optimization. *IEEE Transactions on Evolutionary Computation*, PP(99):1–1.

Zitzler, E. (1999). Evolutionary algorithms for multiobjective optimization: Methods and applications.