Reducing the Run-time Complexity of Multi-Objective EAs: The NSGA-II and other algorithms

Mikkel T. Jensen

Mikkel T. Jensen is a research assistant professor in the EVALife group at the Department of Computer Science, University of Aarhus, Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark. E-mail: mjensen@daimi.au.dk

Abstract

The last decade has seen a surge of research activity on multi-objective optimization using evolutionary computation, and a number of well performing algorithms have been published. The majority of the algorithms use fitness assignment based on Pareto-domination: Non-dominated sorting, dominance counting or identification of the non-dominated solutions. The success of these algorithms indicates that this type of fitness is suitable for multi-objective problems, but so far the use of Pareto-based fitness has lead to program run-times in $O(GMN^2)$, where G is the number of generations, M is the number of objectives and N is the population size. The N^2 factor should be reduced if possible, since it leads to long processing times for large population sizes.

This paper presents a new and efficient algorithm for non-dominated sorting, which can speed up the processing time of some multi-objective EAs substantially. The new algorithm is incorporated into the NSGA-II, and reduces the overall run-time complexity of this algorithm to $O(GN \log^{M-1} N)$, much faster than the $O(GMN^2)$ complexity published by Deb et al. Experiments demonstrate that the improved version of the algorithm is indeed much faster than the previous one.

The paper also points out that multi-objective EAs using fitness based on dominance counting and identification of non-dominated solutions can be improved significantly in terms of running time by using efficient algorithms known from computer science instead of inefficient $O(MN^2)$ algorithms. Furthermore, the archives currently used in multi-objective EAs (usually based on lists) are slow and can be improved by using orthogonal range-searching data-structures. Finally, the paper discusses the use of niching in multi-objective EAs, and points out that the use of efficient algorithms for nearest neighbor identification can be used to speed up multi-objective EAs using certain types of niching.

The directions given in this paper can be used to improve the run-time complexity of the NSGA-II, DMOEA, SPEA2, PDE, PAES, PESA, and a multi-objective VLSI-GA.

I. Introduction

Over the last ten years, there has been an increasing interest in applying evolutionary algorithms to multi-objective optimization problems. This research is highly relevant for real world applications, since real world optimization problems often involve several conflicting objectives for which a trade-off must be found. The presence of multiple conflicting objectives in an optimization problem means that no single solution is globally optimal, unless priorities can be assigned to the objectives. It is usually difficult or even impossible to assign priorities a priori, and this makes an algorithm returning a set of promising solutions preferable to an algorithm returning only one solution based on some weighting of the objectives. For this reason, most contemporary multi-objective evolutionary algo-

rithms (MOEAs) are designed to return a set of promising solutions, from which a solution can be picked by a human expert.

Most MOEAs use *Pareto domination* to guide their search. A solution s_1 is said to dominate another solution s_2, if s_1 is no worse than s_2 in all objectives, and better than s_2 in at least one objective. A solution is said to be *non-dominated* if it is not dominated by any other solution. Ideally, a MOEA returns the *Pareto optimal* set, the solutions not dominated by any other solution in the search-space. If the Pareto optimal set is infinite or very large, the algorithm returns a set of non-dominated solutions covering the Pareto set as well as possible. Examples of this work include Corne et al.'s PESA [8], Knowles and Corne's PAES [16], Zitzler and Thiele's SPEA [27] and Deb et al.'s NSGA-II [10].

Even though contemporary MOEAs work with several objectives simultaneously, they still transform all of the objectives into one fitness measure. This is necessary, since ultimately what makes an EA work is the selection of highly fit individuals over less fit individuals. This transformation is usually made in an explicit way, e.g., by assigning each solution a measure of its non-dominatedness (e.g. NSGA-II [10], SPEA [27]), but it can also be done implicitly, e.g., in the form of Pareto domination tournaments (NPGA [14]).

The transformation of multiple objectives into a single fitness measure is usually a costly matter in terms of processing time. Most Pareto-based fitness assignment schemes require that each solution is compared to a large number of other solutions. Many of the MOEAs published in recent years have running times bounded only by $O(GMN^2)$ [10], [27], [26], [1], [19], where G is the number of generations, M is the number of objectives and N is the population size. In contrast, single-objective evolutionary algorithms usually have processing times bounded by $O(GN)$. The N^2 factor in multi-objective processing means that for large population sizes the processing time becomes prohibitively large. Since in some situations it is desirable to use large population sizes (e.g., when the number of conflicting objectives is large [9, section 8.8.2]), it is important to reduce this processing time.

The high computational demand of many published MOEAs is partly due to the fact that multi-objective fitness assignment is a harder computational problem than single-objective fitness assignment. Another explanation lies in the fact that often MOEA research has

disregarded run-time complexity. The focus has been on coming close to the true Pareto-optimal set and on achieving a good spread of solutions.

In this paper, a different approach is taken. The focus will be on improving the running time of a number of well-known MOEAs. Many of the MOEAs published can be improved in terms of running time using standard computer science algorithms and data-structures.

The main result of this paper is the development of an efficient algorithm for non-dominated sorting. Since non-dominated sorting is used for fitness assignment in NSGA-II [10] and a few other MOEAs, the algorithm can be used to improve running times of these algorithms. The new algorithm runs in time $O(N \log^{M-1} N)$, which is much faster than the previously used $O(MN^2)$ algorithm. This drops the run-time complexity of the NSGA-II algorithm from $O(GMN^2)$ to $O(GN \log^{M-1} N)$. Experiments with implementations of the old and the new algorithm confirm that for moderate and large population sizes the savings in processing time can be very large. Additionally, directions for reducing the running times of PESA [8], PDE [1], SPEA2 [26], PAES [16], DMOEA [19], and the algorithm of [24] will be given.

The outline of the paper is as follows. The next section gives an overview of MOEA research from a processing time perspective. Fitness assignment, archiving and niching methods are discussed, and directions are given for how to improve the run-time complexity of a number of well-known algorithms. Section III discusses how to improve the run-time performance of the NSGA-II. It has a brief description of the NSGA-II, followed by the introduction of an efficient algorithm for non-dominated sorting. Experiments demonstrate that the proposed algorithm can be much faster than the previous algorithm. Section IV concludes the paper.

II. A brief overview of MOEAs and their running-times

This section will give a brief survey on current MOEA research. The emphasis will be on the run-time complexity of the algorithms, and on improving the run-time complexities whenever this is possible. Since most MOEA research has not been focused on this issue so far, many of the papers cited below do not explicitly deal with the run-time complexity of the algorithms they present[1]. These papers give no directions for how to make an

[1] noteworthy exceptions are [16], [10], [9], [22]

efficient implementation of the algorithm they describe, and they give no references to this effect either. In these cases I take the liberty of assuming that the authors were considering an algorithm not employing sophisticated algorithms and data-structures from computer science. In any case, a person without solid knowledge of computer science implementing the algorithm will end up with an inefficient $O(GMN^2)$ algorithm without sophisticated algorithms and data-structures. The ability of the algorithms to come close to the true Pareto-front and achieve diverse solutions will not be treated in this text, for more comprehensive surveys of MOEA research see [9], [7].

As presented by their original authors, most MOEAs have running times bounded only by $O(GMN^2)$ or $O(GMNA)$. This is the case for the NSGA-II [10], SPEA [27], SPEA2 [26], DMOEA [19], PESA [8], PAES [16], M-PAES [17], PDE [1] and the maximin MOEA of [2]. In all of these algorithms fitness assignment or archive maintenance takes time proportional to N^2 or NA. The author is not aware of a single publication describing in detail a MOEA running faster than $O(GMN^2)$ or $O(GMNA)$.

Common to many MOEAs published these days are three features: *i*) Fitness assignment based on Pareto-domination, *ii*) elitism and *iii*) niching. We will deal with these features in turn in the next subsections. Each subsection will describe the methods used in current MOEAs and propose algorithms doing exactly the same but in a more time-efficient manner.

Most MOEAs employ mechanisms for rewarding or punishing individuals for their "non-dominatedness" and for being located in a sparsely or densely populated part of the search-space (niching). These two mechanisms govern selection and replacement in the algorithms, and strictly speaking they are both part of the fitness assignment of individuals. However, in the following subsections we will take the liberty of referring to the measure of "non-dominatedness" as *fitness*, while dealing separately with niching.

A. Fitness assignment

Many early MOEAs used non-Pareto based fitness such as weighted sums or considering the objectives in turns. A number of problems have been identified in these approaches (such as only finding one solution per run, being unable to identify non-convex parts of the trade-off surface), and they are rarely used in contemporary algorithms. Now

most MOEAs use fitness based on Pareto-domination: domination counts, non-dominated sorting or identification of the non-dominated solutions. Domination counts, identification of the non-dominated set and non-dominated sorting can all be solved using strait-forward $O(MN^2)$ algorithms, and these have been widely used. However, divide-and-conquer algorithms for identifying the non-dominated front running in time $O(N \log^{M-2} N)$ [18], [4] and dominance counting in time $O(N \log^{M-1} N)$ [4] are known. Section III-B of this paper presents an $O(N \log^{M-1} N)$ algorithm for non-dominated sorting. To the best of my knowledge, these algorithms are not used in any MOEAs today, although Deb [9] is aware of the efficient algorithm for identifying non-dominated solutions.

Examples of algorithms using inefficient algorithms for fitness assignment are the SPEA [27] and SPEA2 [26] algorithms, in which each solution is assigned a "strength" reflecting its degree of non-dominatedness. The strength is based on dominance counts: a solution not dominated by any other solution is assigned a low strength (i.e., a high fitness) if it dominates few other solutions. This is believed to assign higher fitness to solutions located in sparsely populated parts of the search-space. A dominated solution is assigned a strength based on the accumulated strength of the solutions that dominate it. The papers on SPEA [27] and SPEA2 [26] give no directions on which algorithms to use to calculate dominance counts and strengths, but the SPEA implementation made available by Zitzler uses $O(MN^2)$ algorithms. However, the dominance counting algorithm of [4] can be transformed into calculating dominance counts and solution strengths in parallel, and this will reduce the running time of fitness assignment in these algorithms to $O(N \log^{M-1} N)$.

The PDE algorithm [1] need to identify the non-dominated set for fitness assignment. A naive implementation of this uses time $O(MN^2)$, while the efficient algorithm of Kung et al. [18] will only require $O(N \log^{M-2} N)$.

The NSGA-II, DMOEA [19], and the algorithm for VLSI design [24] all use non-dominated sorting for fitness assignment. Using the fast non-dominated sorting algorithm developed in this paper, the running times will drop from $O(MN^2)$ to $O(N \log^{M-1} N)$.

The run-time complexities discussed in this subsection are summed up in table I.

TABLE I

RUN-TIME COMPLEXITIES FOR EVALUATING THE FITNESS IN ONE GENERATION USING NAIVE AND MORE EFFICIENT ALGORITHMS.

Algorithm	Naive fitness	Improved fitness	Remark
NSGA-II [10]	$O(MN^2)$	$O(N\log^{M-1}N)$	non-dominated sorting
DMOEA [19]	$O(MN^2)$	$O(N\log^{M-1}N)$	non-dominated sorting
VLSI-GA [24]	$O(MN^2)$	$O(N\log^{M-1}N)$	non-dominated sorting
SPEA [27]	$O(MN^2)$	$O(N\log^{M-1}N)$	dominance counting
SPEA2 [26]	$O(MN^2)$	$O(N\log^{M-1}N)$	dominance counting
PDE [1]	$O(MN^2)$	$O(N\log^{M-2}N)$	identification of non-dom solutions

B. Archive maintenance / elitism

It has been established that preserving the best individuals is important in multi-objective optimization [25]. This is usually accomplished either by having a huge elite within the population (e.g. NSGA-II, DMOEA), or by having an explicit archive separate from the population in which the elite is stored (e.g., SPEA [27], PAES [16]). When an external archive is used, a data-structure is needed to hold it.

Most papers do not explicitly state how to implement the archive, but give the impression that it should be done by having a list of solutions and comparing a new candidate solution to all of them[2]. This requires time $O(MA)$ to test a candidate solution, where A is the size of the archive, while insertions and deletions can be done in constant time.

The archive can be maintained much more efficiently using a data-structure for dynamic orthogonal range searching, since checking for non-dominance is equivalent to an orthogonal range query. An orthogonal range query is a query in a data-structure of points in a space $\Re^d$ asking if any points exist in a rectangular region $\{\mathbf{x} \in \Re^d | x_0^{min} \leq x_0 \leq x_0^{max}, \ldots, x_d^{min} \leq x_d \leq x_d^{max}\}$. Whether the solution s is dominated by any point in the archive can be checked using the objectives as dimensions in the data-structure, and by asking if any archive solutions are in the region $\{\mathbf{x} \in \Re^d | -\infty < x_0 \leq x_0(s), \ldots, -\infty < x_d < x_d(s)\}$[3]. The data-structure needs to support insertions and deletions, since the

[2]For PAES, Knowles and Corne [16] explicitly state that they use linear search in the archive.

[3]We are assuming that all objectives are minimized. The query will only check whether $\mathbf{x}^{new}$ is covered by any of the points in the data-structure. True domination can be tested by checking if all of the points covering $\mathbf{x}^{new}$

TABLE II

NAIVE AND IMPROVED RUN-TIME COMPLEXITIES PER GENERATION FOR ARCHIVE MAINTENANCE.

Algorithm	Naive archive	Improved archive	Remark
SPEA [27]	$O(MNA)$	$O(N \log^{M-1} A \log\log A)$	Identification of non-dom solutions also viable. Time: $O((N+A)\log^{M-2}(N+A))$.
SPEA2 [26]	$O(MNA)$	$O(N \log^{M-1} A \log\log A)$	
PESA [8]	$O(MNA)$	$O(N \log^{M-1} A \log\log A)$	
PAES [16]	$O(MNA)$	$O(N \log^{M-1} A \log\log A)$	

archive is updated from time to time. Dynamic orthogonal range searching data-structures satisfy all of these requirements, and using a dynamic range tree [20] the archive can do a query/update operation in time $O(\log^{M-1} A \log\log A)$, much faster than the $O(MA)$ complexity of the linear list. Another possibility is to use a randomized segment tree [23, section 8.1], giving update and query times in $O(\log^M A)$. This suggestion is similar to the suggestion of Mostaghim et al. [22], who proposed the use of quad-trees to maintain the archive. Mostaghim et al. did not consider the run-time complexity of their approach, but for large population sizes they achieve substantial speedups in their experiments with the SPEA algorithm.

The algorithms PAES [16], SPEA [27], SPEA2 [26], and PESA [8] all use archives, and can all be improved by using a more efficient data-structure. Some of the algorithms update their archives in large batches. This is the case for SPEA, SPEA2 and PESA, in which the program alternates between filling up the population and inserting new non-dominated individuals into the archive. For these algorithms another efficient way of implementing the archive would be by identifying the entire non-dominated set from scratch every time the population and the archive are merged. This will be effective as long as the archive is not much larger than the population.

The run-time complexities discussed in this subsection are summed up in table II.

C. Niching

In order to reach a nice spread of solutions, most MOEAs employ some kind of niching. Usually phenotypical niching is used, meaning that the spread is supposed to be in the

are identical to $\mathbf{x}^{new}$. Using an extra search tree with counts of phenotypes, this additional check can be done in time $O(\log A)$.

objective functions values and not necessarily on the genotypical level. In most cases, niching is used as a secondary measure of fitness: If individual s_1 is more non-dominated than s_2, s_1 is preferred regardless of niching, while if s_1 and s_2 have the same degree of non-dominatedness, the one residing in the most sparsely populated part of the search-space is preferred. In archive-based algorithms such as PESA [8], PAES [16] and SPEA [27], new non-dominated solutions are inserted into the archive until a maximum size is reached. After this, whenever a new non-dominated individual is found, it is inserted in the archive and a solution perceived to be in the most crowded region of the archive is removed.

In many algorithms, the niching mechanisms used are just as time-consuming as fitness assignment, but harder to improve in terms of time-complexity. Many algorithms use niching mechanisms that require the calculation of the nearest neighbor (or k nearest neighbors) for all solutions in the archive or population, leading to an $O(MN^2)$ processing time for a naive implementation. Examples of such algorithms are SPEA [27], SPEA2 [26], PAES [16], and PDE [1]. In the general case, it is difficult to improve this, but for the case of two objectives, efficient algorithms are known from computational geometry. According to Mulmuley [23, section 7.7], ray-shooting methods can be used to answer nearest neighbor queries. Using this approach, a data-structure for two-dimensional nearest neighbor computation can be built in time $O(N^{1+\epsilon})$, where ϵ is an arbitrarily small constant. Using the data-structure, a k'th nearest neighbor query can be answered in expected time $O(k \log N)$.

The PDE [1] algorithm uses a nearest neighbor approach to do niching. Non-dominated solutions are stored in an archive, and when the archive reaches a maximal size archive solutions are selected based on the distance to their two closest neighbors. Using a naive approach, this takes time $O(MN^2)$, while for two objectives it can be done in time $O(N^{1+\epsilon} \log N)$ using the ray-shooting data-structure.

The SPEA2 [26] algorithm uses a related approach: When the archive becomes too large, the individuals with the lowest nearest neighbor distances are removed. In case of a tie, the individual with the closest second-nearest neighbor is removed, etc. Using a naive implementation this gives an expected time of $O(MN^2)$. Considering a two-

objective problem, using the ray shooting approach of [23] gives an expected processing time of $O(N^{1+\epsilon} \log N)$, assuming that ties can be broken within a fixed number of nearest-neighbor queries. However, it is possible to construct degenerate inputs that will result in a $O(N^2 \log N)$ processing time for this approach. For niching within the population, SPEA2 uses an approach based on a k-th nearest neighbor-rule, where $k = \sqrt{N}$. The algorithm uses tournament selection, and of two solutions with the same strength (non-dominatedness), the solution with the most distant $\sqrt{N}$'th nearest neighbor is preferred. In the two-dimensional case, the k'th nearest neighbor can be found in time $\sqrt{N} \log N$ assuming that the ray-shooting data-structure has been constructed first. This is much faster than using the strait-forward approach requiring $O(N)$ time, but since in practice only few of these calculations may be needed, it is unclear whether the more efficient approach is really worthwhile.

The SPEA [27] algorithm uses a clustering approach in which the distance between all individuals in the archive is calculated, using time $O(MN^2)$. After the calculation of distances, each individual is considered a cluster, and the clusters are merged one by one based on their distances (the two clusters closest to each other are merged) until the number of clusters is equal to the maximal archive size. The distance between two clusters containing several solutions is defined to be the average distance between all solution pairs, one solution from each cluster. A representative solution from each cluster is kept in the archive, while the rest is removed. It is difficult to improve the run-time complexity of this approach, even for two objectives. The difficulty stems from the use of average distances between clusters of multiple solutions. A distance measure of this kind cannot be efficiently handled by the ray shooting data-structure, and the processing time remains $O(MN^2)$, even for $M = 2$.

The NSGA-II algorithm uses a much faster niching scheme than the algorithms discussed above. After non-dominated sorting, each individual is assigned a *crowding-distance*, and individuals with high crowding-distances are assumed to reside in sparsely populated areas and preferred over individuals with low crowding-distances. The crowding distances are assigned by sorting the population on each objective in turn, and setting the crowding distance of an individual to the sum of distances along the objective axis to the closest

individual. Individuals with extreme positions are assigned infinite crowding-distances. The crowding-distances can be calculated with an $O(MN \log N)$ algorithm.

The above algorithms have all used distance measures of various types to achieve diverse solutions. Another popular approach is dividing the objective space into cells using a hyper-grid, and simply counting the number of solutions in each cell. Individuals located in cells with no or few other individuals are assumed to be in sparsely populated areas and preferred over individuals sharing their cell with many other individuals. In terms of processing time, this idea can be handled more efficiently than most distance-based niching methods, for instance the grid-based method used in DMOEA [19] can run in time $O(MN)$ per generation.

The PESA [8] and PAES [16] algorithms use a closely related approach, but contrary to DMOEA, they sometimes need to locate the most crowded cell in the hyper-grid. However, if the hyper-grid is implemented as a quad-tree in which each tree is annotated with the highest grid-count found below it, locating the most crowded cell can be done in time $O(M)$, and as for DMOEA the total cost of maintaining the hyper-grid can be linear, $O(M(N + A))$ per generation. PAES sometimes recalculates the hyper-grid used. This happens when the extremal objective values of the archive changes. The $O(M(N + A))$ processing time assumes that this happens infrequently, otherwise the processing time will be larger.

Another approach for niching is used in MOGA [13], where phenotypical sharing [3, section 6.1] is used; individuals closer to each other than a certain distance σ_{share} get their fitness reduced. In the strait-forward implementation this will run in time $O(MN^2)$ per generation. The procedure can be optimized by distributing the solutions in a hypergrid similar to the one used by PAES [16], PESA [8] and DMOEA [19]. When calculating the sharing function of and individual, only individuals in neighbor cells need to be considered. If the sharing is successful and only few individuals occupy each cell, the processing time drops to $O(MN)$, but in the worst case the running time will still be $O(MN^2)$ per generation.

Table III summarizes the algorithms and niching methods discussed in this section. The run-time complexity varies widely for the niching methods, ranging from expensive schemes

TABLE III

NAIVE AND IMPROVED RUN-TIME COMPLEXITIES FOR NICHING.

Algorithm	Naive niching	Improved niching	Remark
NSGA-II	$O(MN \log N)$	-	
DMOEA [19]	$O(MN)$	-	
SPEA [27]	$O(MN^2)$	-	
SPEA2 [26] ($M = 2$)	$O(N^2)$	$O(N^{\frac{3}{2}} \log N)$	Assuming ties broken in few steps.
SPEA2 [26] ($M \geq 3$)	$O(MN^2)$	-	
PDE [1] ($M = 2$)	$O(N^2)$	$O(N^{1+\epsilon} \log N)$	
PDE [1] ($M \geq 3$)	$O(MN^2)$	-	
MOGA [13]	$O(MN^2)$	$O(MN)$	Assuming efficient geometric hashing.
PAES [16]	$O(M(N + A))$	-	Assuming infrequent recalculation.
PESA [8]	$O(M(N + A))$	-	

proportional to N^2 to much cheaper methods with linear running times. Currently, there is little experimental evidence to support the use of the expensive $O(N^2)$ schemes over the cheaper ones [10], [27].

D. Total run-time complexities

Table IV sums up the run-time complexities of the algorithms treated in this section. In the complexities given, we are assuming M to be fixed. The table reveals that the directions given in the last three subsections can be used to improve the overall run-time complexities of the NSGA-II, DMOEA, MOGA, PESA, PAES and the VLSI-GA. Additionally, the SPEA2 and PDE algorithms can be improved for two objectives.

III. IMPROVING THE RUN-TIME COMPLEXITY OF THE NSGA-II

The rest of the paper is devoted to improving the run-time complexity of the NSGA-II. The complexity of the algorithm as described in [10] is $O(GMN^2)$, but this section will demonstrate how to reduce this to $O(GN \log^{M-1} N)$. This is achieved by a new and faster algorithm for non-dominated sorting. The next subsection introduces the NSGA-II. Section III-B describes the improved non-dominated sorting algorithm, while the experiments in section III-C demonstrate that the proposed algorithm can be much faster than the previous one.

TABLE IV

NAIVE AND IMPROVED TOTAL RUN-TIME COMPLEXITIES FOR THE ALGORITHMS..

Algorithm	Naive run-time	Improved run-time	Improvement
NSGA-II	$O(GMN^2)$	$O(GN\log^{M-1}N)$	fitness
DMOEA [19]	$O(GMN^2)$	$O(GN\log^{M-1}N)$	fitness
SPEA [27]	$O(GM(N+A)^2)$	-	fitness+archive
SPEA2 [26] ($M=2$)	$O(G(N+A)^2)$	$O((N+A)^{\frac{3}{2}}\log(N+A))$	fitness+archive+niching $\star$
SPEA2 [26] ($M\geq 3$)	$O(GM(N+A)^2)$	-	fitness+archive
PDE [1] ($M=2$)	$O(MN^2)$	$O(GN^{1+\epsilon}\log N)$	fitness+niching
PDE [1] ($M\geq 3$)	$O(MN^2)$	-	fitness
MOGA [13]	$O(MN^2)$	-	niching
PAES [16]	$O(GMNA)$	$O(GN\log^{M-1}A\log\log A)$	archive †
PESA [8]	$O(GMNA)$	$O(GN\log^{M-1}A\log\log A)$	archive
VLSI-GA [24]	$O(GMN^2)$	$O(GN\log^{M-1}N)$	fitness

$\star$: Assuming ties broken in few steps in niching. †: Assuming infrequent recalculation of hypergrid.

A. A brief description of the NSGA-II

Pseudo-code for the NSGA-II is given in Fig. 1. The algorithm is based on the idea of transforming the M objectives to a single fitness measure by the creation of a number of fronts, sorted according to non-domination. During fitness assignment, the first front $\mathcal{F}_1$ is created as the set of solutions not dominated by any solutions in the population. These solutions are given the highest fitness and are temporarily removed from the population. After this, a second non-dominated front $\mathcal{F}_2$ consisting of the solutions that are now non-dominated is built, assigned the second-highest fitness etc. This is repeated until all of the solutions have been assigned a fitness. After each front has been created, its members are assigned *crowding distances* (normalized distance to closest neighbors in the front in objective space) later to be used for niching.

Selection is performed in tournaments of size two: The solution with the lowest front number wins. If the solutions come from the same front, the solution with the highest crowding distance wins, since a high distance to the closest neighbors indicates that the solution is located at a sparsely populated part of the front. Reproduction occurs in generations. In each generation N new individuals are generated, where N is the population

```
NSGA-II() {
   generate P_0 at random.
   set P_0 = (F_1, F_2, ...) =non-dominated-sort(P_0)
   for all F_i ∈ P_0
      crowding-distance-assignment(F_i)
   set t = 0
   while(not done) {
      generate child population Q_t from P_t
      set R_t = P_t ∪ Q_t
      set F = (F_1, F_2, ...) =non-dominated-sort(R_t)
      set P_{t+1} = ∅
      set i = 1
      while |P_{t+1}| + |F_i| < N {
         crowding-distance-assignment(F_i)
         set P_{t+1} = P_{t+1} ∪ F_i
         set i = i + 1
      }
      sort F_i on crowding distances
      set P_{t+1} = P_{t+1} ∪ F_i[1 : (N − |P_{t+1}|)]
      set t = t + 1
   }
   return F_1
}
```

Fig. 1. *The NSGA-II algorithm.*

size. Of the $2N$ individuals, the N best individuals are kept for the next generation. In this way a huge elite can be kept from generation to generation.

The processing time used by the `non-dominated-sort` procedure as suggested in [10] is $O(MN^2)$, since the procedure involves comparing the objective values of every solution to the objective values of all other solutions in the population. The time used by the `crowding-distance-assignment` procedure is $O(MN \log N)$, since the procedure involves sorting the elements in fronts $\mathcal{F}_i$ one time for every objective, and since the front size is only limited by N. The time used by the rest of the steps in the algorithm can be expected to be in $O(N)$, and the overall running time of the algorithm is dominated by `non-dominated-sort`, and runs in time $O(GMN^2)$, where G is the number of generations.

B. A faster way to do non-dominated sorting

From the arguments above we realize that if a faster way to do non-dominated sorting can be found, the run-time complexity of the NSGA-II can be reduced. The following subsections will present a sweep line algorithm for the two-objective case, and a divide-

```
non-dom-sort-on-two-objectives() {
   sort the solutions to a sequence s_1, s_2 ... s_N satisfying
     i < j ⇒ (x_1(s_i) < x_1(s_j) ∨ (x_1(s_i) = x_1(s_j) ∧ x_2(s_i) ≤ x_2(s_j))
   set F_1 = {s_1}
   set A = 1
   for(i = 2; i ≤ N; i = i + 1) {
   /* Invariant: F_1 ... F_A hold a non-dominated sorting of the solutions s_1 ... s_{i-1} */
   /* The solutions in F_1 ... F_A are not dominated by any of the solutions s_i ... s_N. */
     if(s_i ⊀ F_A) {
       find lowest b such that s_i ⊀ F_b
       set F_b = F_b ∪ {s_i}
     } else {
       set A = A + 1
       set F_A = {s_i}
     }
   }
   return F_1, ... , F_A
}
```

Fig. 2. *Non-dominated sorting algorithm for two objectives.*

and-conquer algorithm for three or more objectives.

B.1 Non-dominated sorting on two objectives

Kung et al. [18] present a simple algorithm for locating the non-dominated set of a set of points in $\Re^2$. This algorithm runs in time $O(N \log N)$. The algorithm presented in [18] only extracts the first non-dominated front $\mathcal{F}_1$, but it is not difficult to modify it to extract all of the fronts $\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \dots)$.

In the following the notation $s_i \prec s_j$ is used to indicate that s_i is dominated by s_j, and $s_i \not\prec s_j$ to indicate that s_i is not dominated by s_j. The notation $s \prec F$ is used to indicate that the solution s is dominated by some solution in the set F. We are assuming that all objectives are to be minimized. The algorithm for non-dominated sorting in two objectives is given in Fig. 2. The idea in the algorithm is to construct all of the fronts simultaneously by sweeping the solutions one by one in a way that guarantees that if solution s_j is swept after solution s_i, we know that s_j cannot dominate s_i. This is achieved by presorting all of the solutions on x_1 (and x_2 if two solutions share identical x_1 values), so that $s_i \not\prec s_j$ will always hold if $i < j$.

After sorting the solutions, they are considered one by one. The solution s_1 is guaranteed to be in the first front (since it cannot be dominated by any of the solutions $s_2 \dots s_N$), so

we set $\mathcal{F}_1 = \{s_1\}$. The variable A holds the current number of fronts, and is initialized to 1. In the main loop of the algorithm, we get the next solution s_i, and check if it is dominated by any solution in the worst front seen so far, $\mathcal{F}_A$. If this is not the case, s_i belongs to one of the fronts $\mathcal{F}_1 \ldots \mathcal{F}_A$, so we locate the proper front $\mathcal{F}_b$ by binary search in $\mathcal{F}_1 \ldots \mathcal{F}_A$ and insert s_i. If s_i was dominated by $\mathcal{F}_A$, a new front containing s_i is created.

The fronts $\mathcal{F}_{(j)}$ can be implemented as lists. If we implement the $\mathcal{F}_b = \mathcal{F}_b \cup \{s_i\}$ operation by simply appending s_i to the list, we know that each list $\mathcal{F}_{(j)}$ will hold solutions in increasing order of x_1. Since the solutions in $\mathcal{F}_{(j)}$ do not dominate each other, they must also hold solutions in decreasing order of x_2. As the solutions have been presorted according to x_1, we know that the solution s_i can never have a smaller x_1 value than any of the solutions in $\mathcal{F}_1 \ldots \mathcal{F}_A$. This means that in order to check the condition $s_i \prec \mathcal{F}_j$, we simply need to compare $x_2(s_i)$ to the x_2 value of the last element of $\mathcal{F}_{(j)}$. A run of the two-objective algorithm is illustrated in Fig. 3.

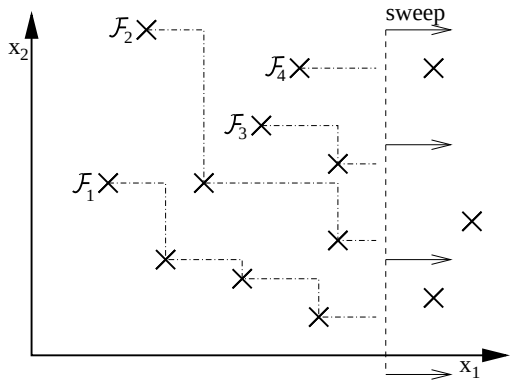

Fig. 3. Illustration of the two-objective algorithm.

In order to do the binary search for $\mathcal{F}_b$, we need to do at most $O(\log N)$ checks of the $s_i \prec \mathcal{F}_j$ condition. Since this check and all other operations on $\mathcal{F}_j$ can be done in constant time, the total running time of the algorithm becomes $O(N \log N)$, which is clearly optimal since ordinary sorting is a special case of two-objective non-dominated sorting.

B.2 A divide-and-conquer algorithm for $M \geq 3$

A divide-and-conquer algorithm for identifying the first non-dominated front of a M-objective problem is presented by Kung et al. in [18]. A less complex presentation of

the same algorithm is given in [4]. This algorithm can be used to identify all of the non-dominated fronts by repeatedly finding and removing the solutions that are currently non-dominated. In the best case, in which all of the solutions are non-dominated, this leads to a running time of $O(N \log^{M-2} N)$. However, the worst case performance of such an approach could be $O(N^2 \log^{M-2} N)$, since if there is only one solution in each non-dominated front, the $O(N \log^{M-2} N)$ algorithm will have to be applied N times. Thus, a more clever approach is needed.

The algorithm presented here is related to Kung et al.'s algorithm, but finds all of the fronts in one run. The algorithm is shown in Fig. 4. The algorithm as presented in the figure assumes that no solutions share identical values for any coordinates, but removing this assumption is not difficult. The basic idea is to assign to each solution $s \in S$ an additional value $f[s]$. This value holds the front-number of s for the solutions s has been compared to so far. Thus, the main procedure `non-dominated-sort` sets $f[s] = 1$ for all $s \in S$ (since in the beginning no solutions have been compared to any other solutions, so all solutions are assumed to belong to the first front), and calls `ND-helper_A`. When `ND-helper_A` returns, the solutions have been sorted, and the front numbers are in f. The fronts $\mathcal{F}_1, \mathcal{F}_2, \ldots$ are made based on f and returned.

The recursive procedures `ND-helper_A` and `ND-helper_B` are where the real sorting takes place. `ND-helper_A` takes as arguments a set of solutions S and a number of objectives M, and creates a non-dominated sorting of S based on the first M objectives $x_1 \ldots x_M$. The front numbers of the solutions are stored in the $f[s]$ data structure. If the number of solutions is two (line marked $\lceil\heartsuit\rceil$) the two solutions are compared to each other. If s_1 is found to dominate s_2, then $f[s_2]$ is set to $\max(f[s_2], f[s_1] + 1)$. Conversely, if s_2 is found to dominate s_1, then $f[s_1]$ is set to $\max(f[s_1], f[s_2] + 1)$.

If the number of solutions is higher than two, `ND-helper_A` splits the problem into smaller subproblems and solves them recursively. This is done by splitting S into two equally sized sets L and H around the median of the M objective, x_M^{split}. The `(`L`,`H`)= split(`S`,`x_M^{split}`,`M`)` call of the `split` routine puts all elements of S with an x_M value lower or equal to x_M^{split} into L and all elements with a higher x_M value into H. No element of H can ever dominate an element in L (since $s_1 \in L \wedge s_2 \in H \Rightarrow x_M(s_1) < x_M(s_2)$), and

```
non-dominated-sort(S,M) {
   foreach (s ∈ S) set f[s] = 1
   ND-helper_A(S,M)
   set F_1 = ∅, F_2 = ∅, ...
   foreach (s ∈ S) set F_{f[s]} = F_{f[s]} ∪ {s}
   return(F_1, F_2, ...)
}

/* This procedure creates a non-dominated sorting of S on the first M
   objectives. The frontnumbers in f[] are taken as a basis for the sorting */
ND-helper_A(S,M) {
   if (|S| = 2) <<sorting s_1 and s_2:update f[s_1] and f[s_2] >> ♡
   else if(|S| > 2) {
     set x_M^split =median(x_M(s_1)...x_M(s_|N|))
     set (L,H)= split(S,x_M^split,M)
     ND-helper_A(L,M)
     ND-helper_B(L,H,M − 1)
     ND-helper_A(H,M)
    }
 }

/* This procedure assigns front numbers to the solutions in
   H according to the solutions in L. The solutions in L are
   assumed to have the correct front numbers in f[] already. */
ND-helper_B(L,H,M) {
   if (|L| = 1)
     <<go through H, comparing to l_1:update f[h] for h ∈ H >> ♣
   else if (|H| = 1)
     <<go through L, comparing to h_1:update f[h_1] >> ♣
   else if (M = 2) <<do 2D sorting of H according to L:
                        update f[h] for h ∈ H >> ♠
   else {
     if (max(x_M(l_1)...x_M(l_|L|)) ≤ min(x_M(h_1)...x_M(h_|H|))
       ND-helper_B(L,H,M − 1)
     else if (min(x_M(l_1)...x_M(l_|L|)) ≤ max(x_M(h_1)...x_M(h_|H|))) {
       if (|L| > |H|) set x_M^split=median(x_M(l_1)...x_M(l_|L|))
       else set x_M^split=median(x_M(h_1)...x_M(h_|H|))
       set (L_1,L_2)= split(L,x_M^split,M)
       set (H_1,H_2)= split(H,x_M^split,M)
       ND-helper_B(L_1,H_1,M)
       ND-helper_B(L_1,H_2,M − 1)
       ND-helper_B(L_2,H_2,M)
      }
    }
 }
```

Fig. 4. *Recursive non-dominated sorting algorithm for $M \geq 3$.*

therefore the elements in L can be sorted according to non-domination without considering H. This is done in the recursive call `ND-helper_A(`L`,`M`)`. A solution in H may or may not be dominated by solutions in L, so in order to get the proper front numbers of H, L needs to be considered. Because of the construction of L and H, a solution $s_1 \in H$ is dominated by a solution $s_2 \in L$ iff s_2 dominates s_1 in the first $M-1$ objectives. The call to `ND-helper_B(`L`,`H`,`$M-1$`)` compares the solutions in H to the solutions in L for the remaining $M-1$ objectives, and if a solution $s_1 \in H$ is found to be dominated by $s_2 \in L$, then its front-number $f[s_1]$ is set to $\max(f[s_1], f[s_2]+1)$. After the call to `ND-helper_B`, the front numbers assigned to the solutions in H would be correct if they were only to be assigned front numbers according to the solutions in L. In order to complete the non-dominated sorting, the solutions in H need to be compared to each other. The sorting is finished by recursively calling `ND-helper_A(`H`,`M`)`, creating a non-dominated sorting of the solutions in H while respecting the front-numbers assigned earlier in the algorithm.

The procedure `ND-helper_B` works in a way similar to `ND-helper_A`, but instead of sorting a set by comparing it to itself, it takes two sets L and H as arguments, and sorts H by comparing the solutions in H to the solutions in L. The solutions in H are not compared to each other, since this is supposed to happen after `ND-helper_B` terminates. The solutions in L are not compared to each other, since they already have the right front numbers. If there is only one solution in L or one solution in H (lines marked ⌜♣⌝), all of the solutions are compared to the single solution one by one using the first M objectives. If $s_2 \in H$ is dominated by $s_1 \in L$, it is assigned the front-number $\max(f[s_2], f[s_1]+1)$.

If the problem has two objectives (line marked ⌜♠⌝), a sweep-line procedure akin to the algorithm of Fig. 2 is used to assign front-numbers to H according to dominance by solutions in L. This is done by building fronts ("stairs") from the solutions in L while assigning new front-numbers to solutions in H; if $s_1 \in L$ dominates $s_2 \in H$ then the front number of s_2 is set to $\max(f[s_2], f[s_1]+1)$. Since the stairs must respect the front-numbers previously assigned to the solutions in L, "holes" will sometimes appear in the front-numbers represented by the stairs (e.g., front-numbers 1, 2 and 4 may be present in the stairs, while 3 is missing). Additionally, in some cases a stair may be discontinued, if a stair representing a higher front number reaches a lower x_2-value. A run of the algorithm is

illustrated on Fig. 5. The solutions from L have been annotated with their front-numbers, while the solutions from H have been annotated with their front-numbers before and after the sweep. Note that the stair corresponding to $\mathcal{F}_2$ is discontinued when the x_2-value of $\mathcal{F}_4$ drops below the x_2-value of $\mathcal{F}_2$. The $\mathcal{F}_2$-stair later reappears because a solution with an even lower x_2 value belonging to $\mathcal{F}_2$ is found.

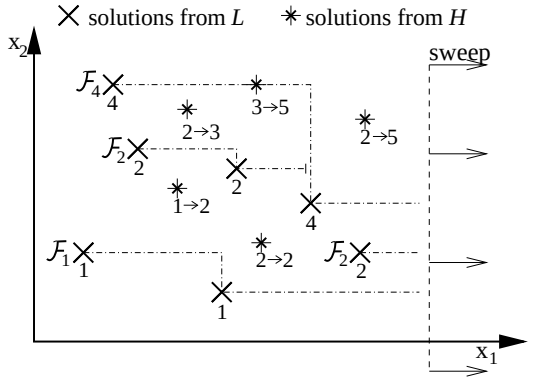

Fig. 5. Illustration of the two-objective algorithm used in `ND-helper_B`.

If there are several solutions in both sets and the number of objectives is higher than 2, recursive calls may be needed. If $\max(x_M(l_1) \ldots x_M(l_{|L|})) \leq \min(x_M(h_1) \ldots x_M(h_{|H|}))$, objective M can be effectively ignored since all of the solutions in L have lower x_M objectives than the solutions in H. We simply call `ND-helper_B(`L`,`H`,`$M-1$`)` recursively.

If $\min(x_M(l_1) \ldots x_M(l_{|L|})) \leq \max(x_M(h_1) \ldots x_M(h_{|H|}))$ does not hold we do not need to do anything; the solutions in H all have lower x_M values than the solutions in L; they cannot be dominated by solutions in H. If the x_M values of L and H form overlapping intervals, the sets are split into sets L_1, L_2, H_1 and H_2 around the median of the x_M-coordinate. Recursive calls of `ND-helper_B` are made to assign front-numbers to H_1 according to L_1, H_2 according to L_1 (only $M-1$ objectives need to be considered since $x_M(l) < x_M(h)$ for $l \in L_1, h \in H_2$) and H_2 according to L_2. Note that we do not need to assign front-numbers to H_1 according to L_2, since by construction the solutions in H_1 cannot be dominated by solutions in L_2.

The correctness of the algorithm follows from the fact that whenever a solution s_a is assigned a front-number by comparing it to another solution s_b, the solution s_b has already been assigned the correct front-number. The recursive calls of the algorithms

make sure that all solutions are assigned front-numbers according to all other solutions. Fig. 6 illustrates the algorithm for a three-objective problem.

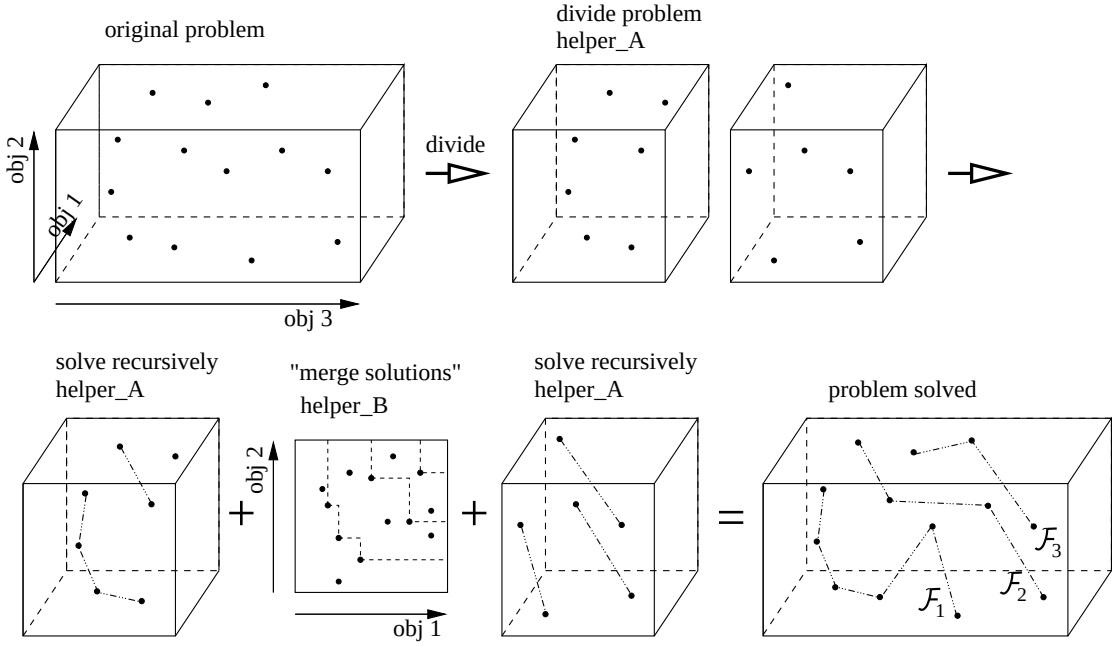

Fig. 6. Illustration of the multi-objective sorting algorithm on a three-objective problem. Solutions belonging to the same front have been connected by a dashed line.

As presented in Fig. 4, the algorithm assumes that no solutions share identical objective values. Removing this assumption is strait-forward; all we need to do is check if all x_M values of a call to the `ND-helper` procedures are identical, in which case the objective M can effectively be ignored in the present call. This means that a sweep-line algorithm has to be added to the `ND-helper_A` procedure for the $M = 2$ case. Besides, a number of special cases have to be handled, e.g. in the `split` procedure. The analysis and time complexity derived below can be shown to hold also for this case.

We now investigate the time-complexity of the algorithm. When using the big-oh notation we will assume M to be fixed. We start by considering the procedure `ND-helper_B`. If we denote by L and H the sizes of inputs, the recursion equations for the running time

become

$$\begin{aligned} T_B(L,H,2) &\in O((L+H)\log(L+H)) \\ T_B(1,H,M) &\in O((L+H)\,M) \\ T_B(L,1,M) &\in O((L+H)\,M) \\ T_B(L,H,M) &= T_B(L_1,H_1,M) + T_B(L_2,H_2,M) + T_B(L_1,H_2,M-1) + T_{\texttt{split}}(L,H), \end{aligned}$$

where $L_1 + H_1 \leq \frac{3}{4}(L+H)$, $L_2 + H_2 < \frac{3}{4}(L+H)$, $L_1 + L_2 = L$ and $H_1 + H_2 = H$. $T_{\texttt{split}}(L,H)$ is the time needed to find the median and call `split` on L and H. Using the algorithm of Blum et al. [6] this can be achieved in time $O(L+H)$. Recurrence relations similar to the one above were investigated by Monier in [21]. According to his results, T_B is bounded by $O(N \log^{M-1} N)$, where $N = L + H$.

For `ND-helper_A`, the recursion relation is

$$\begin{aligned} T_A(2,M) &\in O(M) \\ T_A(N,M) &= 2\,T_A(\tfrac{1}{2}N,M) + T_B(N,M-1) + T_{\texttt{split}}(N) \end{aligned}$$

The total running time becomes $T_A(N,M) \in O(N \log^{M-1} N)$. Note that the time complexity of this algorithm is higher than the complexity of the algorithm for identifying the first non-dominated front presented in [18], [4] by a factor of $\log N$. The reason for this is the existence of an $O(N \log N)$ algorithm for identifying the maxima of a three objective problem. We have not been able to find a $O(N \log N)$ algorithm for non-dominated sorting when $M = 3$, but if such an algorithm can be found, it can be used as a base case in `ND-helper_A` and `ND-helper_B`, lowering the processing times by a factor of $\log N$ for $M \geq 3$.

When incorporating the fast non-dominated sorting algorithm into the NSGA-II, the processing time of the entire algorithm becomes

$$T_{\text{NSGA-II}}(N,M) \in O(G(MN\log N + N\log^{M-1} N)) = O(G\,N\log^{M-1} N).$$

The $MN\log N$ term comes from the `crowding-distance-assignment` taking place in the main loop.

Considering the storage complexity of the new sorting algorithm, recursion equations related to the ones above can be stated. Solving these leads to a storage complexity

of $O(MN)$ for the entire sorting algorithm. This is a significant improvement over the $O(MN + N^2)$ storage required by the procedure suggested in [10].

Kung et al. [18] present a lower bound on the problem of identifying the non-dominated set. According to them, the processing time is bounded from below by $O(N \log N)$, and it is trivial to see that this bound must also hold for non-dominated sorting. The lower bound indicates that for the case $M = 2$ the algorithm presented in this paper is optimal, while for $M \geq 3$ this may not be the case.

C. Experiments

In order to test the improvement of running time when using the new non-dominated sorting algorithm, the NSGA-II was implemented in two versions, one using the non-dominated sorting algorithm proposed in [10], and one using the faster algorithm of this paper. Both algorithms were implemented in `C++`, and experiments were carried out on a 450MHz pentium III computer running Linux.

The algorithms were tested on two problems:

• The DTLZ1 benchmark from [11] was used, since it is a standard multi-objective benchmark and since it is scalable (any number of objectives can be used). Besides, phenotype construction and genetic operators take very little time in this problem, meaning that a substantial speedup should be possible by faster non-dominated sorting. The problem is defined as follows:

$$
\begin{aligned}
\text{Minimize} \quad f_1(\mathbf{x}) &= \tfrac{1}{2} x_1 x_2 \dots x_{M-1}(1 + g(\mathbf{x}_M)), \\
f_2(\mathbf{x}) &= \tfrac{1}{2} x_1 x_2 \dots (1 - x_{M-1})(1 + g(\mathbf{x}_M)), \\
&\vdots \\
f_{M-1}(\mathbf{x}) &= \tfrac{1}{2} x_1 (1 - x_2)(1 + g(\mathbf{x}_M)), \\
\text{where} \quad 0 \leq x_i &\leq 1, \quad \text{for } i = 1, 2, \dots, n \\
\text{and} \quad g(\mathbf{x}_M) &= 100 \left[|\mathbf{x}_M| + \sum_{x_i \in \mathbf{x}_M} (x_i - \tfrac{1}{2})^2 - \cos(20\pi(x_i - \tfrac{1}{2})) \right].
\end{aligned}
$$

The number of decision variables is $|\mathbf{x}| = M + k - 1$, where k is a problem parameter. The notation $\mathbf{x}_M$ is used to denote the last k elements of $\mathbf{x}$, i.e., $\mathbf{x}_M = (x_M, x_{M+1}, \dots, x_{M+k-1})$.

The Pareto-optimal solution corresponds to $\mathbf{x}_M = \mathbf{0}$ with objective function values on the hyperplane $\sum_{m=1}^{M} f_m = 0.5$. As suggested in [11], k was set to 5 in the experiments.

• In order to test the algorithm on a problem for which phenotype construction and genetic operators take significant amounts of processing time, a job shop scheduling problem (JSSP) was transformed into a multi-objective problem. The job shop problem is a combinatorial optimization problem in which n jobs and m machines are given. Each job consists of a sequence of operations, and each operation is to be processed at a specific machine for a specific processing time. Solving the problem implies deciding when to process each of the operations while respecting a number of constraints: *i*) the processing sequences of the jobs must be respected, *ii*) each machine can process only one operation at a time, *iii*) each job can have only one operation processed at a time, *iv*) there can be no preemption. The objectives used in the experiments were the end-of-processing times for the last operation of each job. A comprehensive description of the JSSP is beyond the scope of this paper, the interested reader is referred to [5]. The problem instance `ft20` [12] with 20 jobs and 5 machines was used in the experiments.

C.1 The DTLZ1 problem

The algorithms used a binary representation scheme and except for the number of fitness evaluations and population size, the operators and parameter settings from [10] were used. The problem was used with $M = 2, 3, 5$ and 8 objectives, and 13 population sizes in the range 100 ... 2000. For each combination of M and population size, 10 runs were made with the NSGA-II program using the ordinary and the improved non-dominated sorting algorithms, and the average processing time in CPU-seconds was calculated.

The results for two and eight objectives are graphically displayed on Fig. 7. There are three graphs in both diagrams: Two graphs showing the processing time for the improved and the ordinary algorithms, and one showing the processing time spent on other things than non-dominated sorting in both algorithms. The plots have been drawn in logarithmic scales and show the average processing time as a function of population size. Plots equivalent of the plots in Fig. 7 were made for $M = 3$ and $M = 5$. The plot for $M = 3$ was similar to the plot for $M = 2$, but for the improved algorithm the processing times were a little higher. For $M = 5$ the plot was similar to the plot for $M = 8$, but the

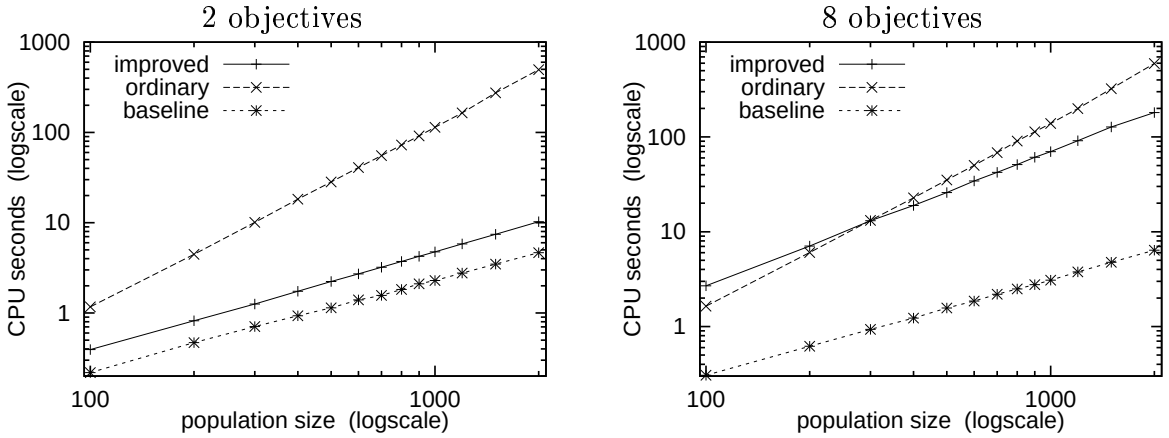

Fig. 7. Processing times of the improved and ordinary algorithms on the DTLZ1 problem for 2 and 8 objectives. The graph marked 'baseline' shows the time not spent on non-dominated sorting in both algorithms.

processing times of the improved algorithm were somewhat lower.

Judging from the plots, the relative performance of the two algorithms is highly dependent on the number of objectives. For two objectives, the improved algorithm is much faster than the ordinary algorithm even for a small population size (for $N = 100$, the processing times are 0.4 and 1.2 seconds respectively). As the population size increases, the difference between the two algorithms becomes larger and larger; for $N = 2000$, the improved algorithm is 48 times faster than the ordinary algorithm.

For eight objectives the improved algorithm is slower than the ordinary algorithm for population sizes smaller than $N = 300$. For a population size of $N = 100$, the improved algorithm spends roughly 60% more processing time than the ordinary algorithm. For large population sizes, the improved algorithm is much faster than the ordinary algorithm; for a population size of $N = 1000$ the improved algorithm is twice as fast as the ordinary algorithm, and for larger population sizes the difference is even bigger.

In practical applications the inferior performance of the improved algorithm for many objectives and small population sizes is not important, since for many objectives a large population size is needed. In order to have sufficient selection pressure in the algorithm, a certain proportion of the population needs be dominated. If the entire population (or close to it) is non-dominated the algorithm has no way to distinguish good solutions from bad ones, and the search will stagnate. Let us follow Deb [9, section 8.8.2], and assume that

at most 30% of the population should be non-dominated. Assuming randomly distributed objectives, for a six-objective problem, we need a population size of almost 800. For a seven objective problem, the population size should be in the 1500-2000 range [9, figure 276].

The plots of Fig. 7 all seem linear, and this is also the case for $M = 3$ and $M = 5$. Since the plots are log-scale, this indicates that the processing times follow a $T = \beta N^{\alpha}$ relation. The α parameters were calculated using linear regression, and this revealed that for the ordinary algorithm α was very close to 2 in all of the experiments. This confirms the $O(N^2)$ processing time of this algorithm. For the improved algorithm, α ranged between 1.1 ($M = 2$) and 1.4 ($M = 8$), confirming that the asymptotic run-time of this algorithm is indeed better than for the ordinary algorithm.

The speedup of the fast algorithm over the slow algorithm has been plotted in Fig. 8. The speedup is defined as the processing time of the ordinary algorithm divided by the processing time of the improved algorithm. As expected the largest speedup is achieved for $M = 2$, while the speedup becomes smaller as M increases, but for large population sizes the speedup is substantial, regardless of the number of objectives.

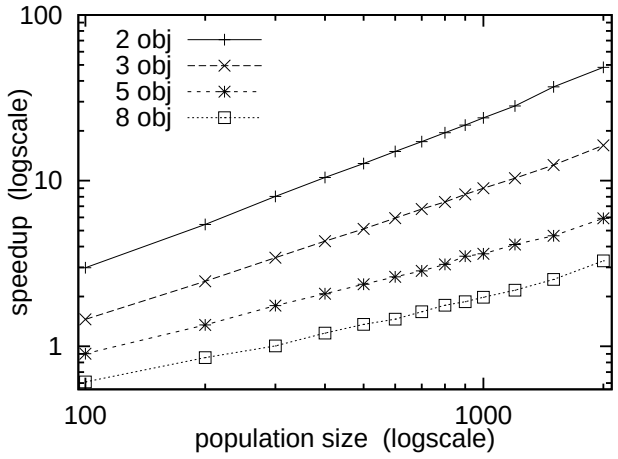

Fig. 8. The speedup achieved by the improved algorithm as a function of population size for 2,3,5 and 8 objectives for the DTLZ1 problem.

C.2 The job shop problem

The representation, decoding, genetic operators and parameters were identical to the algorithm used in [15]. Since these details are not important for what follows, the are left out for brevity. Phenotype construction and genetic operators are much slower for this problem than they were for the DTLZ1 problem. For the DTLZ1 problem, constructing a new genotype and phenotype took in the range $2 * 10^{-5}$ to $5 * 10^{-5}$ CPU seconds. For the JSSP, it took almost ten times longer.

The processing time of the ordinary and the improved algorithm have been plotted in Fig. 9. The figure indicates that if non-dominated sorting is not the bottleneck in the algorithm, only little can be gained by using the improved algorithm. For small population sizes, genetic operators and phenotype construction use almost all the processing time spent by the algorithms (as indicated by the graph marked 'baseline' on Fig. 9), and the speedup offered by the improved algorithm is negligible. For $N = 100$, both programs spend only 15% of their processing time doing non-dominated sorting. However, for larger population sizes the non-dominated sorting begins taking substantial processing time, and for a population size of 2000 a speedup of 4.3 is reached.

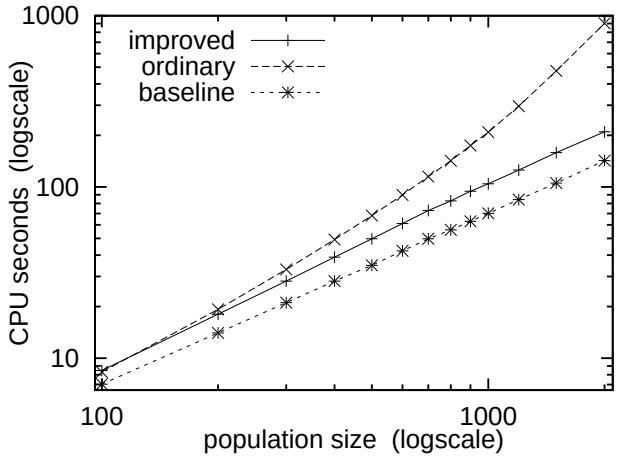

Fig. 9. The processing times of the ordinary and improved algorithm on the job shop problem. The graph marked 'baseline' shows the time not spent on non-dominated sorting in both algorithms.

C.3 Program availability

The `C++` source of the program used in the experiments is available for download at `http://www.daimi.au.dk/~mjensen/`.

IV. Conclusion

This paper has treated the issue of run-time complexity in MOEAs. Most MOEAs published so far have running times bounded only by $O(GMN^2)$ or $O(GMNA)$, where G is the number of generations, M is the number of objectives, N is the population size and A is the size of the archive. In some applications using an algorithm with a large population size is desirable, but the N^2 and NA terms in the algorithm processing times can lead to very high computational demands for large population sizes.

We have developed a new algorithm for non-dominated sorting, which is used for fitness assignment in a number of MOEAs including the NSGA-II. This algorithm runs in time $O(N \log^{M-1} N)$, and improves the run-time complexity of the NSGA-II from $O(GMN^2)$ to $O(GN \log^{M-1} N)$. Experiments with the NSGA-II using the previous and the new non-dominated sorting algorithms demonstrated that for problems in which phenotype construction and genetic operators are fast, the speedup achieved by the new algorithm can be very substantial, especially for large population sizes and few objectives. For problems where phenotype construction and genetic operators take more processing time, the new algorithm can still save substantial amounts of processing time, provided that the population is large. The new implementation of the NSGA-II is available for download.

A number of algorithms and data-structures known from computer science can be used to decrease the computational demand of a number of MOEAs. For MOEAs using identification of non-dominated solutions for fitness assignment, the running time can be improved to $O(GN \log^{M-2} N)$, while for MOEAs using dominance counting the improved running time becomes $O(GN \log^{M-1} N)$.

Archive-based MOEAs, can be improved by implementing the archive using a data-structure for dynamic orthogonal range-searching. If a dynamic range tree is used, this can improve the running time of certain MOEAs from $O(GMNA)$ to $O(GN \log^{M-1} A \log \log A)$.

Some MOEAs use niching mechanisms with running times proportional to N^2. These

mechanisms are difficult to improve, but for mechanisms relying on nearest neighbor calculations the case of two objectives can be improved to $O(N^{\frac{3}{2}} \log N)$, and in certain cases hypergrids can be used to speed up the calculation of sharing-functions.

Acknowledgement

I wish to thank the anonymous reviewers for helpful suggestions and my coworkers at EVALife for help in revising the paper. I also thank Gerth Stølting Brodal from BRICS for a helpful discussion.

This work was partly supported by the Danish Technical Research Council, grant no. 26-02-0140.

References

[1] H. A. Abbass, R. Sarker, and C. Newton. PDE: A Pareto-frontier Differential Evolution Approach for Multi-objective Optimization Problems. In *Proceedings of CEC 2001*, volume 2, pages 971–976, 2001.

[2] R. Balling and S. Wilson. The Maximin Fitness Function for Multi-objective Evolutionary Computation: Application to City Planning. In L. Spector et al., editors, *Proceedings of GECCO 2001*, pages 1079–1084. Morgan Kaufmann, 2001.

[3] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation.* IOP Publishing and Oxford University Press, 1997.

[4] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[5] J. Błazewicz, K. H. Ecker, G. Schmidt, and J. Węglarz. *Scheduling in Computer and Manufacturing Systems.* Springer, 1994.

[6] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bound for selection. *Journal of Comput. Syst. Sci.*, 7:448–461, 1973.

[7] C. A. C. Coello. A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. *Knowledge and Information Systems*, 1(3):269–308, 1999.

[8] D. W. Corne, J. D. Knowles, and M. J. Oates. The Pareto Envelope-Based Selection Algorithm for Multiobjective Optimization. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 839–848. Springer, 2000.

[9] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley & sons, 2001.

[10] K. Deb, A. Pratab, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[11] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. In *Proceedings of the 2002 IEEE Congress on Evolutionary Computation (CEC 2002)*, 2002.

[12] H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, 1963.

[13] C. M. Fonseca and P. J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discus-

sion and Generalization. In S. Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, 1993.

[14] J. Horn, N. Nafpliotis, and D. E. Goldberg. A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In *In Proceedings of the First IEEE Conference on Evolutionary Computation*, volume 1, pages 82–87, 1994.

[15] M. T. Jensen. Guiding Single-Objective Optimization using Multi-Objective Methods. In *Proceedings of EvoCOP 2003*, 2003.

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

[17] Joshua Knowles and David Corne. M-PAES: A Memetic Algorithm for Multiobjective Optimization. In *2000 Congress on Evolutionary Computation*, volume 1, pages 325–332, 2000.

[18] H. T. Kung, R. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the Association for Computing Machinery*, 22(4):469–476, 1975.

[19] H. Lu and G. G. Yen. Dynamic Population size in Multiobjective Evolutionary Algorithms. In *Proceedings of CEC 2002*, volume 2, pages 1648–1653, 2002.

[20] K. Mehlhorn and S. Näher. Dynamic Fractional Cascading. *Algorithmica*, 5:215–241, 1990.

[21] L. Monier. Combinatorial Solutions of Multidimensional Divide-and-Conquer Recurrences. *Journal of Algorithms*, 1:60–74, 1980.

[22] S. Mostaghim, J. Teich, and A. Tyagi. Comparison of Data Structures for Storing Pareto-sets in MOEAs. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 1, pages 843–848. IEEE, 2002.

[23] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, 1994.

[24] R. Thomson and T. Arslan. An Evolutionary Algorithm for the Multi-objective Optimisation of VLSI Primitive Operator Filters. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 1, pages 37–42. IEEE, 2002.

[25] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, 2000.

[26] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization. In K. C. Giannakoglou et al., editors, *EUROGEN 2001 - Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, pages 95–100, 2001.

[27] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.