Computational Cost Reduction of Non-dominated Sorting Using the M-front

Martin Drozdík, Youhei Akimoto, Hernán Aguirre, and Kiyoshi Tanaka *Member, IEEE,*

***Abstract*—Many multi-objective evolutionary algorithms rely on the *non-dominated sorting* procedure to determine the relative quality of individuals with respect to the population. In this paper we propose a new method to decrease the cost of this procedure. Our approach is to determine the non-dominated individuals at the start of the evolutionary algorithm run and to update this knowledge as the population changes. In order to do this efficiently we propose a special data structure called the *M-front*, to hold the non-dominated part of the population. The M-front uses the geometric and algebraic properties of the Pareto dominance relation to convert *orthogonal range queries* into *interval queries* using a mechanism based on the nearest neighbor search. These interval queries are answered using dynamically sorted linked lists. Experimental results show that our method can perform significantly faster than the state of the art Jensen-Fortin's algorithm, especially in many-objective scenarios. A significant advantage of our approach is that if we change a single individual in the population we still know which individuals are dominated and which are not.**

***Index Terms*—Computational cost reduction, Data structures, Evolutionary computation, K-d tree, Many-objective optimization, Multi-objective optimization, Nearest neighbor searches, Non-dominated sorting, Pareto optimization.**

I. Introduction

An INDIVIDUAL solution of a multi-objective optimization problem is said to *Pareto dominate* a different solution if it is *better* with respect to at least one objective while not being worse with respect to *any* objective. Non-dominated sorting is the process of dividing the population of a multi-objective evolutionary algorithm (MOEA) into *fronts* with respect to *Pareto dominance*. Belonging to a specific front gives us a measure of relative quality of an individual with respect to the rest of the population. Many MOEAs such as NSGA-II [2], GDE3 [3] or DEMO [4] require that non-dominated sorting is performed at each generation to determine the individuals that *survive into the next generation*.

This procedure often becomes time-consuming compared to the rest of the algorithm. This is especially the case with large populations and/or high number of objectives. One can parallelize the objective function evaluations for the population into available processors, however the sorting has to be performed in serial and costs more computational time as the number of objectives grows due to the nature of Pareto dominance. As an example we profiled a run of a MOEA algorithm called GDE3 [3] implemented on a single processor on a WFG9 [5] problem with 4 objectives and an initial population size of 1000 individuals for 500 generations. We found out that approximately 82% of computer time and 79% of computer instructions[1] were used on non-dominated sorting. In computationally expensive problems this ratio will change. Nonetheless, it is desirable to reduce the cost of non-dominated sorting, especially for large populations.

The first to recognize and address this problem were Deb et al. in [2]. Their method called the *fast non-dominated sorting* compares each individual with each other and caches the result of these comparisons in order to avoid comparing the same two individuals twice.

For further improvement, several researchers have proposed different approaches, which are mainly categorized into the following three.

Divide and conquer: These methods are based on an article by Kung et al. [6]. They divide the problem with respect to both *dimension* (number of objectives) and *population size*. The first attempt by Jensen [7] in 2003 achieved significant speedup and a reduction in computational complexity, but it failed to deal with the case where two individuals have a same value for a certain objective. Treating this case turned out to be more difficult than it seemed. Luckily, this problem was recently solved by Fortin et al. [8] while preserving both speed and computational complexity. These algorithms achieve astonishing improvement in both theory (computational complexity) as well as in practice (computational wall clock time). Unfortunately the performance declines to the level of fast non-dominated sorting for a high number of objectives. Moreover these algorithms are rather complicated and *static* in the sense that when one individual is changed there is no easy way to determine the non-dominated fronts of the modified population other than to run the algorithm again.

Reducing the number of dominance comparisons: These methods try to *infer* domination relationships using the transitivity property of Pareto dominance. Notable recent algorithms are the *climbing sort* and the *deductive sort* by McClymont

Martin Drozdík is with the Interdisciplinary Graduate School of Science and Technology, Shinshu University. All other authors are with the Faculty of Engineering, Shinshu University, 4-17-1 Wakasato, Nagano 380-8553, Japan (e-mail: martin@iplab.shinshu-u.ac.jp).

The work of Martin Drozdík has been supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

This article is an expansion and revision of our earlier work [1].

Manuscript created on April 9th, 2014, revised August 12th 2014, accepted October 2nd 2014.

[1]We used a computer profiler Callgrind, which runs the program on a virtual machine and counts the instructions executed.

and Keedwell [9]. These algorithms achieve very significant speedup, but they are specifically designed for populations where the domination relationship between individuals is *relatively common*. Unfortunately this assumption does not hold for problems with a large number of objectives. The fewer domination relationships there are, the fewer such relationships can be inferred and the performance suffers. Even so, these methods constitute a significant innovation since they are *dynamic* in the sense that when one individual changes, the information about the non-dominated fronts can be efficiently updated.

Archiving the non-dominated individuals: The problem these methods try to solve is different from the original non-dominated sorting. Schütze [10] calls this problem the *dynamic non-dominance problem*. Instead of starting from scratch and computing the non-dominated fronts for a certain population, these methods concentrate on *keeping* and *updating* a single non-dominated front. These methods are of course *dynamic* as in the previous paragraph. Notable research has been done by Fieldsend et al. [11] and by Schütze [10]. Both studies propose original data structures to hold and maintain the set of non-dominated individuals. Although the speedup achieved by these methods over the brute force method is significant, it is not competitive with the divide and conquer methods.

In this paper we propose a new method to reduce the cost of non-dominated sorting. This method is closely linked with the dynamic non-dominance problem.

The main idea is to compute which individuals are non-dominated *at the beginning* of the MOEA and then *update* this knowledge *each time an individual changes*. This way the non-dominated individuals are known at *all times*. Thus we do not need to call non-dominated sorting to compute the first front. In the case there is more than one front to compute then we apply non-dominated sorting just on the subset of dominated individuals. When the number of objectives grows, there are fewer and fewer dominated individuals which reduces the need to call non-dominated sorting. Therefore our method thrives on problems in which the number of non-dominated solutions is large.

We keep track of the non-dominated individuals by storing them in a special data structure which we call an *archive*. We update this archive whenever an individual in the population changes. The computational cost of updating the archive is critical, therefore a major part of our work is dedicated to an efficient implementation of a fast archive which we call an M-front.

The M-front keeps track of all the non-dominated individuals in the population. This means that when a new individual is generated, the M-front needs to determine *if* this individual is dominated by any individual from the M-front and if it is not, to determine *which* individuals are dominated by the new individual. The M-front uses the geometric and algebraic properties of the Pareto dominance relation to reduce the number of individuals which need to be compared. It converts the *orthogonal range* queries related to Pareto dominance to *interval* queries. In order to answer these interval queries *efficiently*, the M-front keeps all its individuals *sorted* in linked lists. There is one linked list for each objective and this list keeps all the individuals sorted with respect to that objective. In order to convert an orthogonal range query into interval queries, an *auxiliary individual* needs to be chosen from the M-front. The role of this individual is just to perform the conversion. We found out that the *closer* this auxiliary individual is to the new individual, the *smaller* are the resulting interval queries and the *faster* is the computation. In order to find an auxiliary individual which is *as close as possible* to the new individual, the M-front keeps an internal K-d tree data structure to perform approximate *nearest neighbor* search. The M-front can be also used as a stand-alone archive for algorithms which use *unbounded archives* such as [12] or [11].

Experiments confirm that our method can outperform the state of the art Jensen-Fortin's divide and conquer algorithm up to certain population sizes. The performance of our method scales well, especially for a large number of objectives. Since our approach is dynamic in nature, the non-dominated individuals are known at *all times* which is a significant advantage over the state of the art method.

There are algorithms which use more precise methods, such as the *hypervolume* [13], to estimate the quality of individuals in the population. These algorithms can yield better solutions than algorithms which use less precise mechanisms [14]. However, the main problem with such algorithms is that the hypervolume is extremely costly to compute for big populations and large numbers of objectives [15]. In addition, even if the hypervolume could be computed fast, there would still be the need to determine the non-dominated individuals because the hypervolume is computed from them.

This paper is a significant revision and extension of our previous work [1], in which we were restricted to *differential evolution* [16] algorithms. In this work we generalize our method to any multi-objective evolutionary algorithm (MOEA) which uses non-dominated sorting. In addition, as we mentioned before, now our approach can be applied to handle archives of MOEAs. We have also improved the implementation of our method. In our original work we use *skip lists* to keep the individuals sorted by each objective. Now we use simple *linked lists* and a hash-table to retrieve the positions of individuals in the linked lists. This results in faster insertions and removals, smaller memory usage and simpler implementation. The K-d tree is a data structure which gets unbalanced after many insertions and deletions. In our previous work we mitigated this problem by rebuilding the K-d tree from scratch after a fixed number of insertions and deletions. Now we propose a new, more frugal mechanism which detects if the K-d tree is unbalanced. We also now include a comparison with the state of the art Jensen-Fortin's algorithm on both practical and conceptual level. Lastly, we added a theoretical section which derives the average case expected computational complexity on a random model.

The organization of this paper is as follows: first in Section II we introduce a general framework of our approach. This framework introduces the concept of an archive and explains its usage in detail, while the concrete implementation of one such archive is described in Section III. The computational complexity of the proposed approach is theoretically explored in Section IV. In Section V we provide a conceptual comparison with Jensen-Fortin's algorithm while the experimental comparison is shown in Section VI. The experimental section also contains a comparison with fast non-dominated sorting.

II. Proposed method

A. Notation

First, let us establish some basic notation that we shall use in the rest of this article.

We have a *minimization* problem F consisting of M objective functions:

$$F = (f_1, f_2, \ldots, f_M).$$

Each function has n variables:

$$f_i : D \subseteq \mathbb{R}^n \mapsto \mathbb{R} \quad \text{for } i = 1, \ldots, M.$$

The problem may contain arbitrary constraints. We say that F maps the *decision* space D to the *objective* space $\mathbb{R}^M$. We call the members of the *decision* space *decision vectors* and the members of objective space *objective vectors*.

We define an *individual* to be a pair (id, X) where $id \in \mathbb{N}$ is a *unique identifier* and $X \in \mathbb{R}^n$ is a *decision vector*. This way we can distinguish between several individuals with the same value in the *decision* space. [2]

To avoid confusion when using subscripts and to simplify notation we use the following convention: when we have an individual $a = (id_a, X_a)$ then instead of writing $f_i(X_a)$ to denote the value of i-th objective for individual a, we simply write $f_i(a)$. Similarly, instead of $F(X_a)$ we simply write $F(a)$ to denote the objective vector of a.

We say that an individual a *dominates* an individual b if

$$f_i(a) < f_i(b) \text{ for } \textit{some } i \in \{1, \ldots, M\}$$

and

$$f_i(a) \leq f_i(b) \text{ for } \textit{all } i \in \{1, \ldots, M\}.$$

We call this relation *Pareto dominance* and denote it as:

$$a \prec b : \quad a \text{ dominates } b.$$

If for two individuals a, b neither $a \prec b$ nor $b \prec a$, we call a and b *mutually non-dominated.*

B. Applicability

Our method requires some modifications to the usual computational flow of the MOEA. To illustrate this, we restrict ourselves to an evolutionary algorithm following the scheme in Fig. 1. We demonstrate our method on this generic version of a MOEA. Popular algorithms such as NSGA-II [2] or GDE3 [3] mentioned earlier both follow this scheme.

The algorithm has a generating phase on lines 3 to 7 and a survival selection phase on lines 8 to 10. In order for an algorithm to benefit from our method we need to modify *both phases*. We get an *equivalent algorithm* described in Fig. 2.

The individual-by-individual steady-state generation and insertion of individuals may be confusing, but indeed also algorithms which generate their individuals in large chunks, such as the NSGA-II, can be rearranged to conform to Fig. 1. The mechanism which generates a new individual on line 4 is completely arbitrary. Therefore for NSGA-II the step on line 4 can be just *taking* the new individual from the offspring population. NSGA-II does not remove individuals in the generating phase, but some algorithms such as GDE3 do. Therefore we allow for such removals on line 6.

Note that in the generating phase we can remove only individuals which are dominated. This is a limitation of our approach. Later we shall explain why we need this limitation.

```
Algorithm: MOEA that can use our method
   Input: Initial population size $N$
   Output: Approximation of the Pareto front by a
           population $P$
1  initialize population $P = \{a_1, ..., a_N\}$
2  while arbitrary stopping condition not satisfied do
   Evolutionary loop
3      while arbitrary stopping condition not satisfied do
       Generating phase
4          generate new individual $a'$
5          insert $a'$ into $P$
6          remove arbitrary dominated $a \in P$ if needed
7      end
8      non-dominated sorting
9      remove worst non-dominated fronts
10     trim $P$ back to size $N$ using a secondary criterion
11 end
12 report $P$
```

Fig. 1. Generic MOEA which can benefit from our method.

C. Overnondomination

It is a well known fact that there is a tendency for a large proportion of the population to become *non-dominated* when the number of objectives increases [17]. This may also happen for 2 and 3 objective problems, especially in the later stage of the search, depending on the problem [18]. We call this phenomenon *overnondomination.*

Overnondomination is usually a bad thing. It causes some MOEAs to stagnate and even recede from the Pareto front. As we mentioned in the introduction it also causes problems for novel non-dominated sorting methods that try to infer domination relationships, such as the deductive sort.

In this work on the other hand, we use overnondomination *to our advantage*. We use it to bridge the dynamic non-dominance problem to the non-dominated sorting problem.

D. Using an archive to avoid non-dominated sorting

In the following we use the term *archive* to mean a data structure which holds a set of *mutually non-dominated individuals*. Later we provide an exact implementation of an archive which we call the *M-front*, but all results in this section are applicable to any archive.

[2] In our implementation the id is simply a C++ pointer to the vector X in memory. In order for the id to remain valid, the individuals do not change their addresses in the memory once they are created.

The algorithm in Fig. 1 uses non-dominated sorting to determine which individuals get discarded after the end of the generating phase. What we need to realize is that the algorithm *does not need to know all the fronts*. It only needs enough fronts so that they contain *at least N individuals*.

If the population suffers from overnondomination such as in the right side of Fig. 3, there is a good chance that the algorithm only needs to know the *first non-dominated front* i.e. the non-dominated part of the population.

Our method is to keep track of the *non-dominated* part of the population at *all times*. We do this by keeping it in *an archive A*. We update the archive with each single change to the population so that it contains *only non-dominated* individuals. We can see this in lines 5 to 10 of Fig. 2.

If we know into which front an individual a belongs, we say that a has a *determined front*. All the individuals in the archive have *determined fronts*, since they belong to the *first* front.

We can now see the explanation of our limitation of removing only dominated individuals in the generation phase of algorithm in Fig. 1. If we would remove a non-dominated individual a from the archive A, there is a possibility that some individual $a' \in P \setminus A$ (in P but not in A) which was dominated by a might become *no longer dominated* and we would need to insert it into A to keep the integrity of the archive. Then we would need a mechanism to detect such individuals a' efficiently. In this work however, we limit ourselves to the case where non-dominated individuals cannot be removed from the population if there are some dominated individuals. If $P \setminus A$ is empty, then we are free to remove even *non-dominated* individuals without breaking the integrity of the archive.

When it comes to discarding the worst non-dominated fronts at the end of the evolutionary loop of Fig. 2, we have a good chance that we *do not need to perform any non-dominated sorting at all* (line 11). We just discard all the individuals which are not in the archive.

If the archive contains fewer than N individuals (line 14), we need to determine additional fronts. We do this using the simple method described in Fig. 4. We initialize an empty set S to hold candidates for non-dominated individuals (line 1). For each individual a_i in the population of size k we determine whether the individual is dominated by an individual from S (lines 3 to 9). If it is not dominated (line 10) we add this individual to S and check which individuals in S it *dominates*. These individuals are then removed from S (line 14). We run this procedure repeatedly until enough individuals have their front determined.

Even if the archive does not contain more individuals than we need for the next generation (at least N) our task is greatly reduced since we need to compute the additional non-dominated fronts from a smaller set $P \setminus A$.

The computational complexity of our approach depends on how fast we can perform insertions and removals from the archive. In the following section we shall provide a fast archive whose usage can result in average case complexity of $O(M^2 N^{2-\frac{1}{M-1}})$ which we prove in Section IV.

```
Algorithm: Generic MOEA using our method
  Input: Initial population size $N$
  Output: Approximation of the Pareto front by a
          population $P$
1  initialize population $P = \{a_1, ..., a_N\}$
2  determine non-dominated individuals in $P$
3  construct archive $A$ from the non-dominated individuals
4  while arbitrary stopping criterion not satisfied do
   Evolutionary loop
5      while arbitrary stopping criterion not satisfied do
       Generating phase
6          generate new individual $a'$
7          insert $a'$ into $A$ and $P$
8          remove dominated individuals from $A$
9          remove arbitrary $a \in P \setminus A$ if needed
10     end
11     if $A$ contains more than $N$ individuals then
12         remove all individuals not in $A$ from $P$
13         trim $A$ and $P$ to size $N$ using a secondary
           criterion
14     else
15         while # individuals with determined front $< N$ do
16             determine next non-dominated front
17         end
18         remove individuals with undetermined fronts
19         trim $P$ back to size $N$ using a secondary criterion
20     end
21 end
22 report $P$
```

Fig. 2. Generic MOEA using our method.

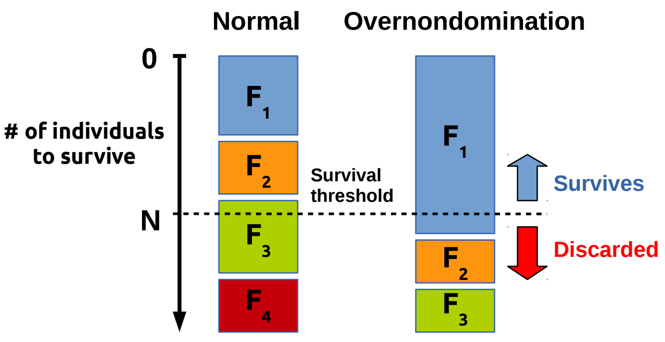


Fig. 3. Survival selection based on non-dominated sorting.

III. Implementation of the Archive

A. Characterization

In the previous section we argued that the problem of reducing the cost of non-dominated sorting can be approached if we had a sufficiently fast data-structure that manages a mutually non-dominated part of the population. Here we describe such a data structure, which we call an M-front. In the following sections, we gradually build up the main ideas behind the M-front.

Algorithm: Determine non-dominated individuals

```
Input: Population $P = \{a_1, \ldots, a_k\}$
Output: $S$ - set of non-dominated individuals $S \subseteq P$
$S \leftarrow \varnothing$
for $i := 1$ to : $k$ do
    bool ai_non_dominated ← true
    forall the $a \in S$ do
        if $a \prec a_i$ then
            ai_non_dominated ← false
            break
        end
    end
    if ai_non_dominated then
        insert $a_i$ into $S$
        forall the $a \in S$ do
            if $a_i \prec a$ then
                remove $a$ from $S$
            end
        end
    end
end
report $S$ as the non-dominated front
```

Fig. 4. Determining a single non-dominated front

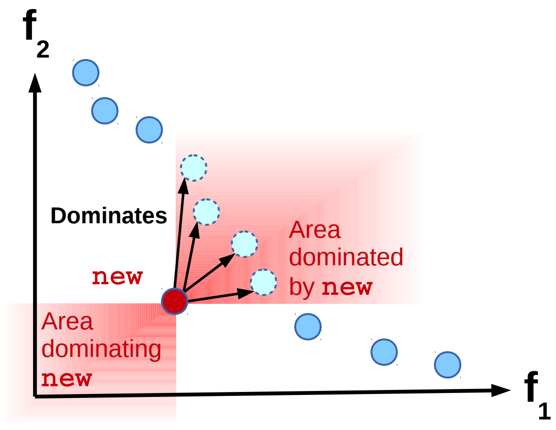


Fig. 5. Insertion into a 2-dimensional archive.

The M-front data structure is a container which holds a *set* (in the mathematical sense) of *individuals*. It has the important invariance property that *all individuals it contains are mutually non-dominated*. This means that if we insert a new individual into the data structure, we need to determine and remove all individuals which have *become dominated*. Next we shall explain how to do this efficiently.

B. Geometric motivation

We illustrate our ideas on a two-dimensional (2 objectives) example. The circles in Fig. 5 represent individuals in an archive of mutually non-dominated individuals. We insert an individual `new` into the archive A. In order to preserve the invariant of A we need to find out:

1) if `new` *dominates* any individuals in A
2) if `new` *is dominated* by any individuals in A

Note that a positive answer to one of these questions implies a negative answer to the other, but there is a case when both answers are negative.

In the geometrical sense, we need to find out which individuals are in the **areas** *dominated by* or *dominating* `new`. These areas are rectangles aligned with the axes. The task of finding all the vectors which lie in such a rectangle is called an *orthogonal range query*. This is a well researched subject in the area of computational geometry and many clever techniques were developed to perform this task efficiently. Some are described in [19].

However the specific nature of our problem allows us to use a different approach. As we shall see in the following section, we can *transform* the *orthogonal range query* into an *interval query*, which is simpler. This is thanks to the specific shape of the orthogonal queries which come from the domination computation.

C. Transformation of orthogonal queries

First we select an *arbitrary* individual in the archive. We call this individual the **reference individual** and mark it with the symbol `ref`. Next we compare the `ref` and `new` for dominance. There are three possible outcomes:

1) `ref` dominates `new`
2) `new` dominates `ref`
3) `ref` and `new` are mutually non-dominated

The first case is simple. We simply abort inserting `new`.

Let us look at the second case which is illustrated in Fig. 6. We already know that `new` belongs into the archive since it cannot be dominated by any other individual. We still need to determine *all* the individuals which are dominated by `new`.

We see that the area *dominated by* `ref` *does not* contain any individuals from the archive, since this would violate the invariance. Therefore we can *subtract* this area from the area dominated by `new` and we get two *intervals* which contain all the individuals which are dominated by `new`.

If an individual a is dominated by `new` then:

$$\begin{aligned} & f_1(a) \in [f_1(\texttt{new}); f_1(\texttt{ref})] \\ \text{or} \quad & f_2(a) \in [f_2(\texttt{new}); f_2(\texttt{ref})] \end{aligned}$$

The 2-dimensional case where `new` dominates `ref` is misleadingly simple. We see that the individuals which lie in at least one of the intervals are *exactly* the ones that are dominated by the `new`. In more than 2 dimensions the individuals which lie in at least one of the intervals form a *superset* of the individuals dominated by `new` and therefore we need to compare all of them for dominance.

Let us move to the last case. The `ref` and `new` are mutually non-dominated. Here we need to find out both *if* `new` itself is dominated and if it is not, *which* individuals `new` dominates. We do this by constructing the areas which are dominating and dominated by `new`, choosing a `ref`, constructing these

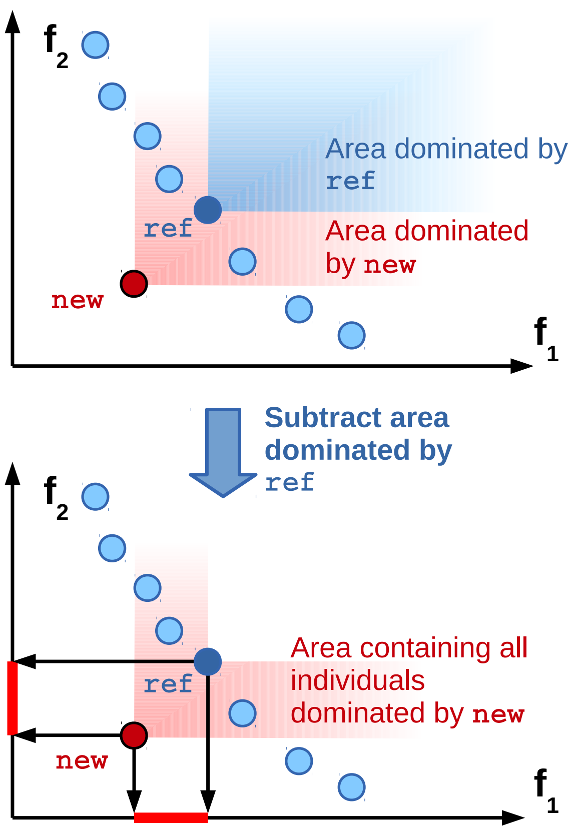

f_2
f_1
Area dominated by ref
Area dominated by new
ref
new
Subtract area dominated by ref
Area containing all individuals dominated by new

Fig. 6. Transformation of an orthogonal query into interval queries (case where `new` dominates `ref`).

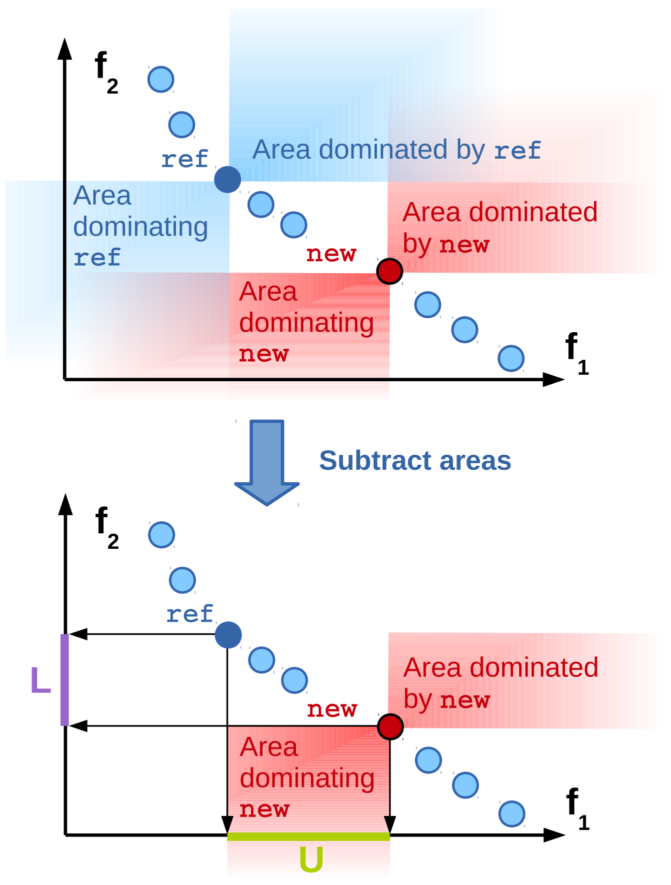


Fig. 7. Transformation of an orthogonal query into interval queries (case where `new` and `ref` are mutually non-dominated).

areas for `ref` and subtracting them from the areas for `new`. This process is illustrated in Fig. 7.

Instead of searching the orthogonal areas which are left by this subtraction, we search the intervals marked by letters U and L in Fig. 7. More formally:

If an individual a dominates `new` then:

$$f_1(a) \in [f_1(\texttt{ref}); f_1(\texttt{new})] =: U$$

and if it is dominated by `new` then:

$$f_2(a) \in [f_2(\texttt{new}); f_2(\texttt{ref})] =: L.$$

We can see that there are individuals which belong to these intervals, but neither dominate nor are dominated by `new`. Therefore we need to compare all of them for dominance with `new`. In the future we hope to find a way how to avoid having to do this. Next, we formalize our findings in a more rigorous manner.

D. Reference sets

In the previous section we described a method to transform the areas that need to be searched when we insert a new individual into the archive. Here we formalize our ideas. First we define the *area* that needs to be searched.

Definition 1 (Reference areas and reference sets)**.** *Let $A = \{a_1, ..., a_N\}$ be a set of mutually non-dominated individuals. Let `new` be an individual which does not belong to A and let `ref` be an arbitrary individual from A.*

The upper reference **area** *for individual `new` induced by individual `ref` is the set $RA_U(\texttt{new}, \texttt{ref}) \subseteq \mathbb{R}^M$ given by:*

$$RA_U(\texttt{new}, \texttt{ref}) := \bigcup_{\substack{i \text{ such that} \\ f_i(\texttt{ref}) < f_i(\texttt{new})}} \{Y \in \mathbb{R}^M \mid y_i \in [f_i(\texttt{ref}); f_i(\texttt{new})]\}. \tag{1}$$

We call the set of all individuals in A whose objective vector lies in $RA_U(\texttt{new}, \texttt{ref})$ the upper reference **set** *of individual `new` induced by individual `ref`. We shall denote it by:*

$$RS_U(\texttt{new}, \texttt{ref}) := \{a \in A \mid F(a) \in RA_U(\texttt{new}, \texttt{ref})\}. \tag{2}$$

Conversely, the lower reference **area** *for individual `new` induced by individual `ref` is the set $RA_L(\texttt{new}, \texttt{ref}) \subseteq \mathbb{R}^M$ given by:*

$$RA_L(\texttt{new}, \texttt{ref}) := \bigcup_{\substack{i \text{ such that} \\ f_i(\texttt{new}) < f_i(\texttt{ref})}} \{Y \in \mathbb{R}^M \mid y_i \in [f_i(\texttt{new}); f_i(\texttt{ref})]\}. \tag{3}$$

Analogously we have the lower reference **set***:*

$$RS_L(\texttt{new}, \texttt{ref}) := \{a \in A \mid F(a) \in RA_L(\texttt{new}, \texttt{ref})\}. \tag{4}$$

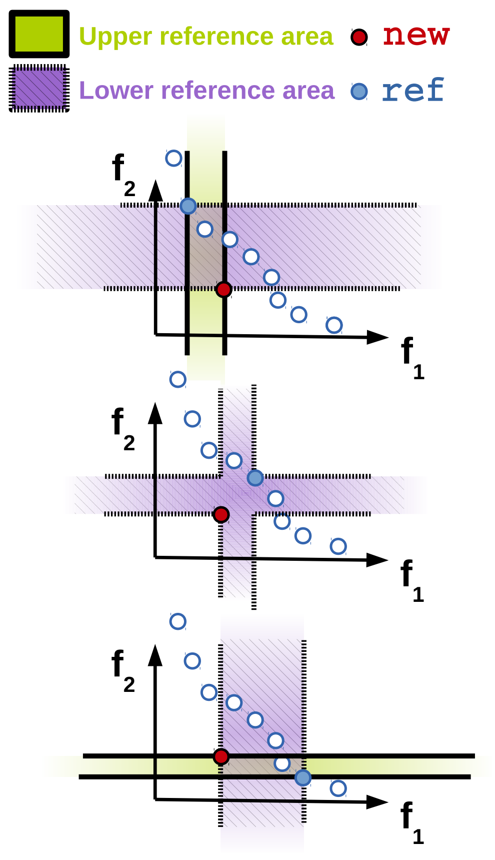


Fig. 8. Reference areas induced by different choices of **reference individual**.

This definition is slightly more strict than in our previous work [1]. We can see an illustration of both upper and lower reference areas for several different choices of reference individual in Fig. 8.

The following theorem says that when we are inserting a new individual into the archive, we need to compare it only to individuals from the upper and lower reference set.

Theorem 1 (Properties of reference sets). *Let* $A = \{a_1, ..., a_N\}$ *be a set of mutually non-dominated individuals. Let* $\mathtt{new}$ *be an individual which does not belong to* A *and let* $\mathtt{ref}$ *be an arbitrary individual from* A.

Then:

1) *if* $\mathtt{new}$ *dominates some* $a \in A$ *then*

$$a \in RS_L(\mathtt{new}, \mathtt{ref}),$$

2) *if some* $a \in A$ *dominates* $\mathtt{new}$ *then*

$$a \in RS_U(\mathtt{new}, \mathtt{ref}).$$

We shall prove only the first statement. The proof of the second statement is analogical.

Proof: We assume that $\mathtt{new}$ dominates some a. The situation is illustrated in Fig. 9.

First we establish that $F(\mathtt{new}) \neq F(\mathtt{ref})$. If $F(\mathtt{new}) = F(\mathtt{ref})$ were true, this would imply that $\mathtt{ref}$ dominates a, which is in conflict with the assumption that all individuals in A are mutually non-dominated.

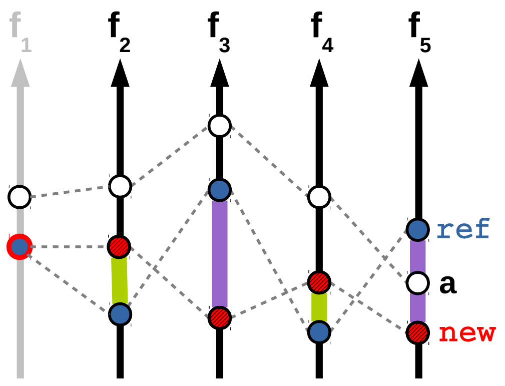


Fig. 9. Illustration of the proof of Theorem 1. $f_1, \ldots, f_5$ are the axes of individual objective functions.

Now let us handle the trivial case where $F(a) = F(\mathtt{ref})$. Since $\mathtt{new}$ dominates a, there must exist an objective f_i such that $f_i(\mathtt{new}) < f_i(a) = f_i(\mathtt{ref})$. Therefore the union on the right side of (3) is not empty and contains $F(a)$.

Now we establish that there exists an objective f_i such that:

$$f_i(a) < f_i(\mathtt{ref}) \tag{5}$$

If the opposite were true, then either $F(a) = F(\mathtt{ref})$ would hold, or $\mathtt{ref}$ would dominate a. We already handled the first case and the second is in conflict with the assumption of the theorem.

Since $\mathtt{new}$ dominates a, we have:

$$f_i(\mathtt{new}) \leq f_i(a). \tag{6}$$

By combining (5) and (6) we get:

$$f_i(\mathtt{new}) < f_i(\mathtt{ref}) \text{ and } f_i(a) \in [f_i(\mathtt{new}); f_i(\mathtt{ref})],$$

which means that a belongs to the lower reference set. ■

E. M-list

1) Data structure: Now we describe how to construct the reference sets for a given pair $(\mathtt{new}, \mathtt{ref})$ *efficiently*.

To construct a reference set, we need to find all the individuals whose objective values lie in certain intervals. If we want to determine all the items whose attribute lies in a certain interval, it is helpful to have that data *sorted by that attribute*. We need to search according to all objectives, therefore we keep the population *sorted by each objective*. We keep a *sorted doubly linked list* for each objective. An illustration of these lists is shown in Fig. 10. The upper and lower reference sets can be constructed simply by iterating between the positions of $\mathtt{ref}$ and $\mathtt{new}$.

However, the lists do not support random access. There is also no mechanism for fast insertion in logarithmic time such as with RB trees or skip-lists. This is again a slight modification to our previous work [1] where we used skip-lists. In order to remove or insert an individual into these lists while maintaining the ordering we use an alternative mechanism. We maintain a

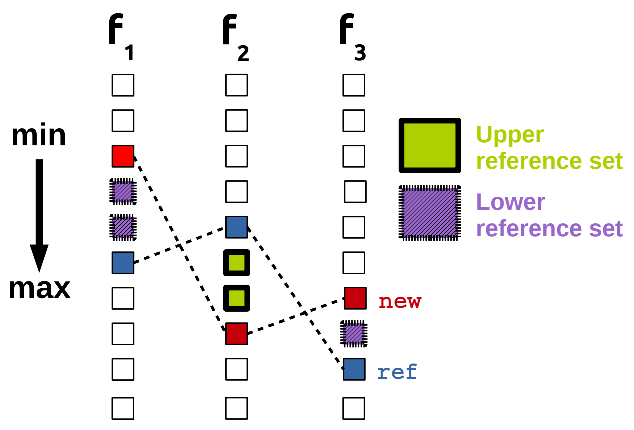


Fig. 10. Constructing reference sets using linked lists.

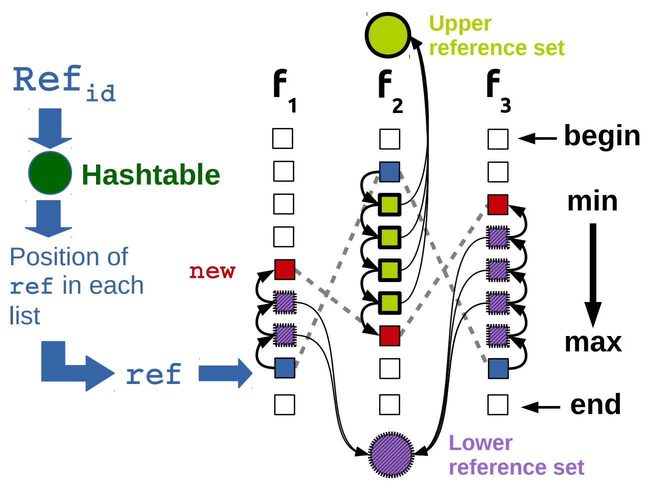


Fig. 11. Insertion into the M-list.

hash-table that maps the id's of the individuals in the archive to an *object which holds the positions of that individual in each list.* [3] We call the resulting data structure consisting of M linked lists and a hash-table an **M-list**.

2) Insertions and removals: The M-list supports these two fundamental operations:

- `remove(a)` removes an arbitrary individual
- `insert(new, ref)` inserts a new individual using a reference individual

Removal of an arbitrary individual is simple. We just:

1) Retrieve the positions of a from the hash-table
2) Remove the entry for a from the hash-table
3) Remove the entry for a from each list

In the average case retrieval and removal from the hash-table is an $O(1)$ operation. Removal from a linked list is also an $O(1)$ operation. Since we have M lists, removal is a $O(M)$ operation.

On the other hand, the `insert` procedure is significantly more complex. This procedure is illustrated in Fig. 11 and described in Fig. 12. The key idea is the combination of the creation of reference sets with finding the correct position for the `new` in each list.

In the description of the algorithm we use the programming concept of an *iterator*. An iterator marks the *position* of an element in a data structure. If `it` is an iterator, then by `*it` we denote the *item* at that position. In our example iterator `it` marks the position of a certain individual in a linked list and `*it` is the individual itself. We establish the convention that the linked lists in the M-list are sorted in an *ascending* order.

Now we explain the steps of Fig. 12. When we insert a new individual into the M-list, we first check if it is dominated by the reference individual (line 2). If this is the case, we know immediately that the new individual cannot be inserted into the archive and abort the insertion. If `new` dominates `ref` (line 7) we know that `new` is not dominated by any individual in the M-list. We store this information for future optimization (line 8).

If `new` is not dominated by `ref`, we proceed with constructing the reference sets. We initialize the upper and lower reference sets, $RS_U(\texttt{new}, \texttt{ref})$ and $RS_L(\texttt{new}, \texttt{ref})$, as empty. Then for each list (line 13) we must determine the correct position for `new`. We initialize an iterator to the position of `ref` in the current list. If the value of the current objective f_i is the same for `new` and `ref`, we know that objective is not relevant for creation of the reference sets. We also know that the position of `ref` is also the correct position of `new`. In this case we just insert `new` to a neighboring position to `ref` and move to the next objective (line 16).

If $f_i(\texttt{new}) < f_i(\texttt{ref})$, we know that f_i is relevant for the creation of the *lower* reference set. There may be more than one individual a with $f(a) = f_i(\texttt{ref})$ in the list, so we increment[4] the iterator to the last position where this holds since we want to capture *all* individuals from the interval $[f_i(\texttt{new}); f_i(\texttt{ref})]$. We then start to decrement the iterator, which moves it toward the place where $f_i(\texttt{new})$ belongs in the list. Simultaneously we are creating the proper reference set (line 20). After each decrement, we check if $f_i(\star\texttt{it}) \geq f_i(\texttt{new})$, which is equivalent to the *lower* reference set requirement: $f_i(\star\texttt{it}) \in [f_i(\texttt{new}); f_i(\texttt{ref})]$. If this condition holds, we insert the individual at position `it` into the lower reference set. Once the condition fails, we know that we have found the right place to insert `new`. We insert `new` and move to the next list (line 23).

If $f_i(\texttt{new}) > f_i(\texttt{ref})$ the situation is symmetrical. We perform the exact opposite of all operations from the previous case while constructing the *upper* reference set.

Once the insertion of `new` into the lists and the construction of the reference sets is complete we create a position object holding the position of `new` in each list. We insert this object into the hash-table (line 33) so that we can retrieve `new` from the lists in constant time.

[3] We implemented this in C++ using a `std::unordered_map` mapping `Individual*` pointers to an object that held M iterators of type `std::list::iterator`, one for each list.

[4] In Fig. 11 incrementing may be seen as a *downward* movement by one box.

```
Algorithm: insert(ref, new)
  Input: list_1, ..., list_M, hash-table H, ref, new
  Output: R - set of removed individuals from the M-list
1  $R \leftarrow \varnothing$
2  if ref dominates new then
3      insert new into $R$
4      return $R$
5  end
6  bool new_non_dominated $\leftarrow$ false
7  if new dominates ref then
8      new_non_dominated $\leftarrow$ true
9  end
10 $RS_U(\texttt{new}, \texttt{ref}) \leftarrow \varnothing$
11 $RS_L(\texttt{new}, \texttt{ref}) \leftarrow \varnothing$
12 retrieve the position object of ref from $H$
13 for $i := 1$ to : $M$ do
14     initialize iterator it to the position of ref in list$_i$
15     if $f_i(new) = f_i(ref)$ then
16         insert new before or after it
17     if $f_i(new) < f_i(ref)$ then
18         increment it to the last position where
           $f_i(\star\texttt{it}) = f_i(\texttt{ref})$
19         while $f_i(\star it) \geq f_i(new)$ do
20             insert $\star$it into $RS_L(\texttt{new}, \texttt{ref})$
21             decrement it
22         end
23         insert new right after it
24     if $f_i(new) > f_i(ref)$ then
25         decrement it to the last position where
           $f_i(\star\texttt{it}) = f_i(\texttt{ref})$
26         while $f_i(\star it) \leq f_i(new)$ do
27             insert $\star$it into $RS_U(\texttt{new}, \texttt{ref})$
28             increment it
29         end
30         insert new right before it
31     end
32 end
33 insert position object of new into $H$
34 forall the $a \in RS_L(new, ref)$ do
35     if new dominates $a$ then
36         remove $a$ from the archive
37         insert $a$ into $R$
38         new_non_dominated $\leftarrow$ true
39     end
40 end
41 if new_non_dominated then
42     return $R$
43 end
44 forall the $a \in RS_U(new, ref)$ do
45     if $a$ dominates new then
46         remove new from the archive
47         insert new into $R$
48         return $R$
49     end
50 end
51 return $R$
```

Fig. 12. Insertion into the M-list.

Then we check if new dominates some individuals in the M-list by comparing new to each individual in the *lower* reference set. If we find such an individual, we remove it from the M-list immediately (removal from M-list is fast as mentioned above) and set the new_non_dominated flag. If there is an individual that is dominated by new, that means that new is not dominated by any individual in the entire M-list and we can skip the following step.

Last of all we check if new is dominated by some individual in the archive. We do this by comparing new to each individual in the *upper* reference set.

3) Important programming details: We conclude this subsection with an important tip to implement an efficient M-list.

In the previous section we did not explain how to implement the data structures symbolizing the reference sets. At first sight this does not look like a significant problem. The most obvious solution is to use a container that represents a *mathematical set*. For example the std::set or std::unordered_set data structure from the C++ standard library. In general, a container that is implemented either as a sorted list or as a hash-table. This would assure that

- We can perform insertions quickly
- We know if the item being inserted already is in the set

Using a computer program profiler[5] we found out that the operations involving insertions and traversal of reference sets are critical to the performance of the entire algorithm.

Armed with this knowledge we tried to implement the reference sets using several alternatives. We tried to use programming classes which model the mathematical concept of a set using either a *hash-table* or a *red-black* tree internally.[6] We also tried our own implementation of hash-table with size growing according to powers of two and linear probing collision resolution.

Lastly we tried to implement the set as a simple array based stack, while relaxing the unique entry property of a mathematical set. If we insert a particular individual more than once, it will be in the set more than once. This may happen if an individual is between the new and ref in more than one list in the M-list.

Surprisingly, the stack implementation outperformed all other implementations in almost all instances. The additional cost of having to compare some individuals for dominance more than once was outweighed by the speed of the stack data structure. Consequently, we perform all our experiments using the stack data structure.

F. K-d trees

1) Nearest neighbor problem: In this section we turn ourselves to the question of selecting a *reference individual*. We understand intuitively that if the reference individual is *close* to the new individual, the reference set induced will be *small* and we will not have much work comparing the new

[5]Callgrind.

[6]We tried the std::set and std::unordered_set containers from the gcc 4.8.1 implementation of the C++ Standard Library. The std::set is implemented as a red-black tree and std::unordered_set is implemented as a hash-table with a prime size and separate chaining collision resolution.

individual to all individuals in it. In our previous work [1] we provide experimental justification for this. Therefore we try to choose the reference individual *as close as possible* to the newly inserted individual.

Formally, we define this problem of finding a close reference individual as follows. Given an M-list ML, a metric d and an individual being inserted $\texttt{new} \notin \mathrm{ML}$ we hope to find $\texttt{ref}$ satisfying:

$$\texttt{ref} \in \mathrm{ML} \text{ such that } \forall a \in \mathrm{ML} : d(\texttt{ref}, \texttt{new}) \leq d(a, \texttt{new})$$

For example, if the metric is the L_1-distance in the objective space, i.e., $d(x, y) := \sum_{i=1}^{M} |f_i(x) - f_i(y)|$, this measures the sum of widths of the reference areas in each objective, as illustrated in Fig. 6. Then, a reference point close to $\texttt{new}$ w.r.t. this metric tends to provide a small reference set. This task is a well studied problem with name *nearest neighbor search*.

Importantly, our objective here is not to find out the nearest individual in a strict manner, but a relatively close reference point. Especially, we do not want to spend much more time to find out the exact solution than to perform the operations described in the previous section. Therefore, an approximation approach to the nearest neighbor search is a good candidate for our purpose.

We employ the K-d tree approach [19] to perform an approximate nearest neighbor search. K-d tree is a hierarchical data structure which supports fast *nearest neighbor* and *approximate nearest neighbor* queries. The K-d tree structure has been reported to have a good performance when the *dimension of the problem is small* [20]. Dimensions until about 8 are considered very small in the nearest neighbor community, whereas the MOEA community considers problems with more than 4 objectives as many-objective. Therefore, we maintain a K-d tree *along* the M-list.

We will not describe the general mechanism of the K-d tree since there are many publications that provide an excellent description (see e.g. [21]). We describe only the details of the particular K-d tree implementation that we tested ourselves and to which our experimental results apply.

2) Implementation details: With each insertion to the M-list we need to perform an insertion into the K-d tree and the same goes for removals. Because we need to use the K-d tree in such a dynamic manner, we use a slightly modified version. We keep the data *only in the leaves* of the tree. This results in somewhat simpler removals and insertions into the tree.

The approximate nearest neighbor procedure proceeds exactly as the standard exact nearest neighbor procedure, but allows for only 4 evaluations of metric d. Once these 4 evaluations have been spent, the procedure returns the closest individual found so far. We chose 4 evaluations ad hoc, since it seemed to perform well in many problem instances.

3) Re-balancing the K-d tree: The major problem with using K-d trees is that it is primarily a *static* data structure. That is, it is not well suited for the dynamic character of the multi-objective optimization. When many insertions and deletions are performed, the tree tends to become *unbalanced*. To our best knowledge there is no efficient method to detect the fact that the tree is unbalanced and to perform the re-balancing. Therefore we developed a heuristic to determine if the tree is out of balance and if it is, we simply destroy it and construct it again.

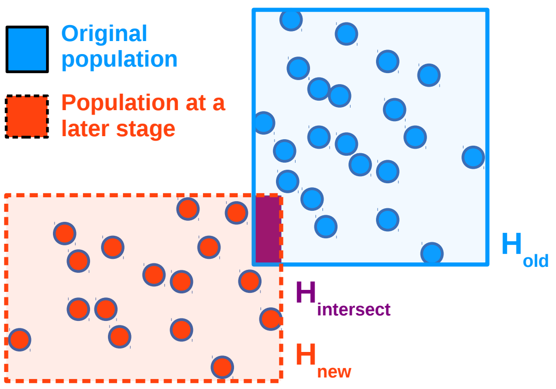


Fig. 13. Computing α in (7).

Our mechanism to detect the loss of balance of a K-d tree is as follows. At the beginning of the algorithm, we compute the bounding hyper-box of the population in the M-list (illustrated in Fig. 13) and denote it by H_{old}. During optimization, we periodically compute the bounding hyper-box H_{new} of the population which is currently in the M-list. Then we compute the ratio of the volume of the *intersection* of these boxes to their *union*:

$$\alpha := \frac{\mathrm{vol}(H_{\text{new}} \cap H_{\text{old}})}{\mathrm{vol}(H_{\text{new}} \cup H_{\text{old}})} . \tag{7}$$

A value of α from (7) near 1 means that the boxes are almost identical, while a value close to 0 means that they are quite different. This can happen either by the box H_{new} becoming too small, too big, or moving away from H_{old}. If the value drops bellow a predefined level (we have chosen 0.5 ad hoc), we rebuild the tree from scratch and replace H_{old} by H_{new}.

A nice feature of the α indicator in (7) is that $\alpha \in [0; 1]$ and that it is invariant to the scaling of the axes.

G. M-front

There is another reason why we chose the K-d tree in particular. As [19, p.101] suggests, the construction of the K-d tree can get expensive because of the need to compute the medians at each node to split the data uniformly. On the same page, the author suggests pre-sorting the data with respect to each dimension, in order to avoid the costly computation of the medians. The M-list is a data structure where the data *already is sorted* with respect to each dimension, which reduces the computational cost even further. To express this affinity of the M-list and K-d tree, we call the combined structure the M-front.

An insertion into the M-front is described in Fig. 14. When we insert $\texttt{new}$ into the M-front, a reference individual is chosen as the *approximate nearest neighbor* to $\texttt{new}$ (line 1). $\texttt{new}$ is next inserted into both parts of the M-front, i.e. the M-list and the K-d tree. The individuals that are removed from M-list

Algorithm: Insertion into the M-front

```
Input: M-front internals: K-d tree $K$, M-list ML, new
Output: $R$ - set of removed individuals from the M-front
1 ref ← retrieve approximate nearest neighbor to new
  using $K$
2 insert new into $K$
3 R ← insert(ref, new) into ML
4 forall the $a \in R$ do
5     remove $a$ from $K$
6 end
7 return $R$
```

Fig. 14. Insertion into the M-front.

(they may contain also new) must be also removed from the K-d tree (line 5).

We can also remove arbitrary individuals from the M-front by removing them from the M-list and from the K-d tree. This is particularly useful when we want to prune the set of non-dominated individuals. Here one can take advantage of the M-front internals. For example, one method to prune non-dominated individuals is the *partitioned quasi-random selection (PQRS)* [11]. The computational cost of this procedure is decreased if the population is sorted with respect to each objective. Hence the M-list can be used to decrease the cost of PQRS. Similarly, one can take advantage of the K-d tree, which is a data structure suitable for efficient nearest neighbor computation, to reduce the computational cost of pruning procedures which perform these computations, such as *the M nearest neighbors pruning* [22].

Source code for all mentioned data structures can be found in [23].

IV. Computational complexity

In this section we theoretically explore the computational complexity, i.e. the number of required operations, especially floating point number comparisons, in our algorithm. Since the core of our method is the insertion into the M-front, we investigate the computational complexity of one such insertion.

A. Best and worst case complexity of our method

The M-front is composed of two data structures, namely the M-list and the K-d tree. Each insertion starts with the retrieval of the reference individual from the K-d tree. If the K-d tree containing N items is balanced, then approximate nearest neighbor queries can be performed in $O(\ln(N))$ time. Therefore if we restrict our usage of the K-d tree to computing just the approximate nearest neighbors, the cost of retrieving the reference individual is $O(\ln(N))$. After the reference individual is retrieved, it is compared to the newly inserted individual for dominance. If the reference individual dominates the newly inserted individual, the insertion is aborted, leaving the computational cost at $O(\ln(N))$. If the reference individual does not dominate the new individual, its position object is retrieved from the hash-table in the M-list. This operation is $O(1)$ in the *average* case and $O(N)$ in the *worst* case.

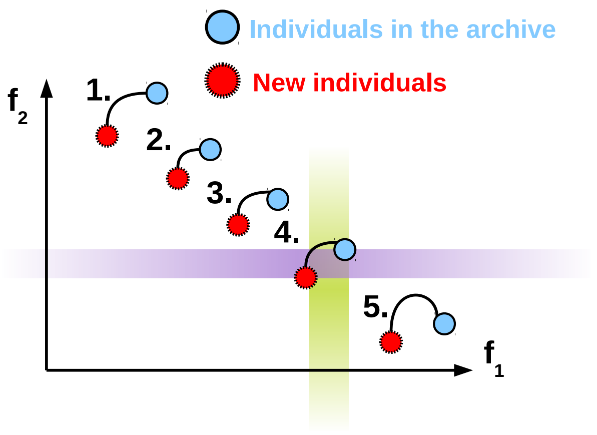


Fig. 15. Sequence of insertions with empty reference sets.

Afterwards, the reference sets are constructed while inserting the new individual into the linked lists of the M-list. There is a non negligible chance that the upper and lower reference sets are *empty* or contain only the reference individual itself. In this case there are no more operations needed. This is especially likely if there is a reference individual which is particularly closer to the new individual than the other individuals in the M-front. We can see an illustration of a sequence of 5 best case insertions in Fig. 15. Each time the new individual is paired with its reference individual in the archive. All insertions yield empty upper and lower reference sets. Intuitively we see that the probability of such a sequence is not infinitesimally small, but it may actually happen. Especially if the underlying MOEA is evolving the population by perturbing one individual at a time. The individual being perturbed may serve as a reference individual, saving thus the cost of the K-d tree.

In the worst case, the reference sets contain all the individuals in the archive. In this case there are N domination comparisons needed, i.e. $O(MN)$ floating point number comparisons, and the complexity is the same as the brute force comparison.

An insertion could cause the K-d tree to be considered unbalanced. In that case it needs to be rebuilt. The cost of rebuilding a K-d tree is $O(MN\ln(N))$ [19]. However if the tree is checked for imbalance periodically each T insertions and if $T \geq N$ then there is at most 1 *rebuilding* of the tree for N insertions. Then the average cost for one insertion is $O(M\ln(N))$ which does not change the worst case complexity.

B. Average case complexity of our method

We model the *average case* complexity of our method using the concept of *random variables* from probability theory. Similarly as when establishing the best case complexity, we estimate the computational complexity of inserting a single individual into an M-front archive. We model the population using *random vectors*. Therefore the complexity that we estimate is itself a random variable. We estimate its *expected value* and *asymptotic properties*.

We try to use familiar naming conventions but from now on, we deal with *random* individuals.

Definition 2 (Random individual)**.** *A* random individual a *is the ordered pair* (id_a, Y_a) *where:*

1) $id_a \in \mathbb{N}$ *is the identifier,*
2) $Y_a = (y_{a,1}, \dots, y_{a,M}) \in \mathbb{R}^M$ *is a random* vector.

The vector Y_a is supposed to model the *objective vector*. In this section we are not interested in the decision vectors at all. We are just trying to model a snapshot of the population in a MOEA.

The most important property of an M-front is that all the individuals within are *mutually non-dominated*. We model the population of the M-front using the following definition:

Definition 3 (Random front)**.** *Let* $\mathrm{RF} = \{a_1, \dots, a_n\}$ *be a set of random individuals. If the probability that there are individuals* $a_i, a_j \in \mathrm{RF}$ *such that* a_i *dominates* a_j *is zero, then we call* RF *a* random front*.*

We shall estimate the computational complexity of inserting an individual into an M-front whose population is a certain specific type of a *random front*.

Definition 4 (Uniform front)**.** *Let* $Pf : \mathbb{R}^{M-1} \to \mathbb{R}$ *be a function that is* strictly decreasing *with respect to each variable.*

Let $RF_{Pf} = \{a_1, \dots, a_n\}$ *be a set of random individuals whose objective vectors are independent identically distributed (i.i.d.) random vectors with the following distribution: Each vector's first* $M - 1$ *components are i.i.d. uniform random variables on* $[0; 1]$*. The last component is the value of function* Pf *of the first* $M - 1$ *components*

$$Y = (y_1, \dots, y_{M-1}, Pf(y_1, \dots, y_{M-1})) \ .$$

We call RF_{Pf} *a* uniform front with shape Pf*.*

The mutual non-domination of individuals is guaranteed thanks to the decreasing nature of the shape function Pf, as is formally described in the following theorem.

Theorem 2 (Correctness of uniform front)**.** *A uniform front is indeed a* random front *in the sense of Definition 3.*

The proof can be found in the Appendix A.

The uniform random front models the individuals in an M-front. Let us now investigate the computational cost of an insertion into such an M-front. As we explained earlier, this cost is proportional to the number of individuals in the reference sets. Therefore we estimate the *expected cardinality of the reference sets.* In order to keep things as simple as possible, we shall assume that the inserted individual is in the *center* of the uniform random front. That is:

$$Y_{\mathtt{new}} = (0.5, \dots, 0.5, Pf(0.5, \dots, 0.5)) \ .$$

Furthermore we shall add some assumptions on the shape function of the front Pf.

Theorem 3 (Expected cardinality of the reference sets)**.** *Let* $Pf : \mathbb{R}^{M-1} \to \mathbb{R}$ *be a*

- *Lipschitz function with respect to the maximum metric with constant L, i.e.* $\forall X, Y \in \mathbb{R}^{M-1}; |Pf(X) - Pf(Y)| \le L \cdot \max_{i \in [\![1; M-1]\!]} |x_i - y_i|$*, and*
- *there exists an* $S \in \mathbb{R}$ *such that the probability density function* f_{Pf} *of the random variable* $Pf(y_1, \dots, y_{M-1})$ *where* $y_i \sim U[0; 1]$ *are i.i.d., is bounded by* S*.*

Let $d_{M-1} : \mathbb{R}^M \times \mathbb{R}^M \mapsto [0; \infty)$ *be the maximum pseudo-metric defined by:*

$$d_{M-1}(X, Y) := \max_{i \in [\![1; M-1]\!]} |x_i - y_i| \ .$$

Let RF_{Pf} *be a* uniform random front with shape Pf *containing* N *individuals,* `new` *be a newly inserted individual with objective vector*

$$Y_{\mathtt{new}} = (0.5, \dots, 0.5, Pf(0.5, \dots, 0.5))$$

and `ref` *be a reference individual chosen as the closest individual to* `new` *with respect to* d_{M-1}*.*

Let the number of individuals which fall either into the upper or lower reference sets denoted by $C_{M,N}$*. Then* $E[C_{M,N}]$ *exists for all* $M,\ N \in \mathbb{N}$*,* $M > 1$*, and*

$$E[C_{M,N}] \in O(MN^{1-\frac{1}{M-1}}) \tag{8}$$

for a fixed M*.*

The proof can be found in Appendix B.

The requirements imposed on Pf may seem complicated and heavy handed but we do it for the sake of simplicity of the proof. One example of such a function is an arbitrary linear function with all coefficients negative.

Since each dominance comparison has the computational cost of $O(M)$, we see that the *expected computational cost of inserting into an M-front* is

$$O(M^2 N^{1-\frac{1}{M-1}}) \tag{9}$$

floating point comparisons.

C. Summary

We have estimated the computational cost of insertions into the M-front. The most costly operation is comparing against individuals in the reference sets. Therefore we were concerned with the *cardinality of the reference sets*.

If we look at the algorithm in Fig. 2 and assume overnondomination, there are approximately the same number of individuals in the M-front as there are individuals in the initial population. Therefore the N in both contexts is roughly the same. In order to be able to compare the computational complexity to an algorithm that performs non-dominated sorting, we shall multiply the costs of inserting one individual (9) by N, since in the course of one generation of algorithm in Fig. 2 there are N insertions. We summarize the computational complexities in Table I. As we see in Table I, our approach scales better with respect to M than Jensen-Fortin's algorithm in the average case. On the other hand Jensen-Fortin's algorithm scales better in terms of N for a fixed M.

V. Comparison with Jensen-Fortin's method

Jensen-Fortin's algorithm [7] [8] is one of the fastest non-dominated sorting algorithms so far. Our algorithm is different on many levels other than speed. In this section we shall go into depth on all the details in which the two algorithms differ.

TABLE I
COMPUTATIONAL COMPLEXITIES WHERE
N IS THE POPULATION SIZE AND M IS THE DIMENSION.

	Jensen-Fortin	M-front
Best case	$O(MN)$	$O(MN)$ or $O(MN\ln(N))$ using the K-d tree
Average case	$O(N\ln^{M-1}(N))$	$O(M^2N^{2-\frac{1}{M-1}})$
Worst case	$O(MN^2)$	$O(MN^2)$

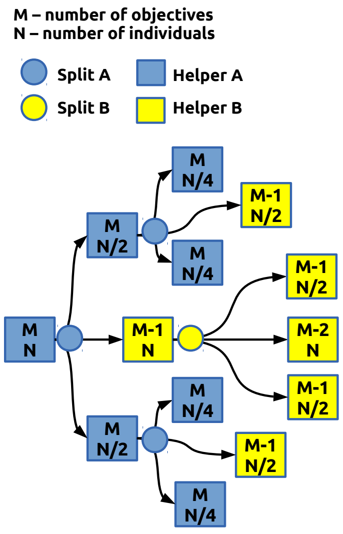


Fig. 16. Jensen-Fortin's algorithm.

A. Description of Jensen-Fortin's algorithm

We mentioned Jensen-Fortin's algorithm in the introduction. Here we shall describe it in a little more detail. Jensen-Fortin's algorithm is based on Kung's algorithm [6]. This algorithm computes the set of non-dominated individuals in a population. Instead of repeatedly using Kung's algorithm to compute non-dominated fronts one by one, Jensen chose a cleaner approach which constructs all non-dominated fronts in one run.

Unfortunately the algorithm did not work well in the general case where more than one individual has the same value for some objective. This was recently fixed by Fortin et al. From now on, we shall work only with the Fortin's version of the algorithm. We call this algorithm the Jensen-Fortin's algorithm. The algorithm uses a divide-and-conquer strategy. This is illustrated in Fig. 16.

In the notation of Fortin et al., the algorithm has two main procedures, `Helper_A` and `Helper_B`, and two splitting procedures, `Split_A` and `Split_B`. After some preprocessing, the algorithm calls the `Helper_A` procedure which does essentially all the work. This procedure splits the problem into problems of smaller size and merges the results using the `Helper_B` procedure which is itself a recursive divide-and-conquer algorithm. `Helper_B` splits the problem again using the `Split_B` procedure. The problem is further divided until either the dimension M or the problem size N gets reduced to 2, which are handled as final cases.

B. Conceptual comparison

The main difference between the two methods is that Jensen's method is a *procedure* while our method is essentially a *data structure*. Our data structure keeps track of the non-dominated individuals at *all times* and this knowledge is updated with each change in the population. This is not possible with Jensen's algorithm. Once a single individual changes the entire computation needs to be executed again.

The advantage of Jensen's method is that it computes *all* the non-dominated fronts, while our method computes only those fronts that *are needed by the trimming procedure*.

C. Computational speed

In the section on experimental results we show that Jensen's algorithm performs well on *large populations*, while our algorithm works well with *a large number of objectives.*

Another main difference is that Jensen's algorithm performs the entire computation at once, while our method allows the cost to be distributed along the entire run of the algorithm. As soon as an individual's objective value is evaluated, we can insert it into the M-front.

D. Flexibility

The Jensen's algorithm is fairly difficult to modify. It took 10 years since the original publication by Jensen for someone to undertake the task to generalize the algorithm to be able to handle the case of multiple individuals sharing the same objective value. No other modifications are known.

On the other hand, the core of our algorithm is just the M-list. The way in which reference individuals are chosen depends entirely on the user. The user is free to choose any strategy to select the reference individual. There are probably many data structures more sophisticated than the K-d tree. Some algorithms compute the nearest neighbor in the objective space for their own purposes. This computation can be reused in inserting an individual into the M-list. A notable example of such an algorithm is the DEMO/obj algorithm [4]. Other algorithms perturb the population one individual at a time. In this case the unperturbed individual is likely to be close to the perturbed one, and may serve as the reference individual. One example of such a MOEA is *differential evolution*. We used this approach in our previous work [1].

The M-list itself can be modified to use a different type of dominance, such as the ϵ-dominance. This may be also possible with Jensen's algorithm, but it is not very straightforward.

Lastly, the M-front is a standalone archive which can be used in algorithms such as [24] that store their non-dominated individuals.

E. Parallelization

The `Helper_A` procedure splits the problem into three subproblems which need to be solved in a particular order. Even though divide-and-conquer algorithms are usually easy to parallelize, there is a sequential dependence in `Helper_A`.

The three problems created by `Helper_B`, on the other hand, can be executed in any order. Therefore we suppose that Jensen's algorithm can be easily parallelized.

TABLE II
COMPARISON OF THE JENSEN-FORTIN'S ALGORITHM AND OUR METHOD.

	Jensen-Fortin	**M-front**
Average complexity	$O(N \ln^{M-1}(N))$	$O(M^2 N^{2-\frac{1}{M-1}})$
Best performance on	High N	High M
Main concept	Procedure	Data structure
Computes all fronts	Yes	No
Flexibility	No	Yes
Parallelization	Yes	Yes

TABLE III
EXPERIMENTAL SETUP.

Crossover	exponential
Cr	0.2
F	0.2
initial population size N	50, 125, 250, 500, 1000, 2000, 4000, 8000
number of objectives M	3, 4, 5, 6, 7, 8
number of generations	500
number of runs	10
number of variables	15

Our algorithm performs many insertions and removals each of which locks the M-front. Parallelization within each transaction is possible but because the insertion is a relatively small operation we are not sure if significant speedup can be achieved.

F. Summary

The differences between Jensen-Fortin's algorithm and our method are summarized in Table II.

VI. EXPERIMENTAL RESULTS

A. Experimental setup

In order to test the performance of our algorithm we have implemented the GDE3 (Generalized Differential Evolution) MOEA [3] using three non-dominated sorting methods:

- Our method (M-front)
- Jensen-Fortin's algorithm
- Deb's fast non-dominated sorting

The three algorithms produce identical outputs. Only thing that is different is the speed. We ran the algorithm on a variety of DTLZ1 [25] and WFG9 [5] problems.

We chose WFG9 in particular because it is multi-modal and non-separable and therefore we hope that it resembles a large number of real world problems.

We chose DTLZ1 because its objective functions are relatively steep. This means that the evaluation of the initial randomly initialized population is quite far from the *true Pareto front*. The randomly initialized population has objective values in the ranges of hundreds while the true Pareto front is a simplex within the hyper-box $[0; 0.5]^M$. During the run of the MOEA, the population needs to travel a significant distance. This should test our K-d tree re-balancing mechanism.

The GDE3 algorithm has two main parameters. The crossover operator and the scaling factor F. We have chosen *exponential crossover* and a value of 0.2 for both F and the *crossover probability* Cr. We chose the parameters according to empirical results by Kukkonen [26] and after an informal off-line calibration of the algorithm.

We ran experiments with various *initial population sizes* N and various *numbers of objectives* M. The population sizes start at 50 individuals and increase in an almost geometric progression up to 8000 individuals. We chose such big populations to clearly demonstrate at which population size the asymptotic superiority of Jensen-Fortin's algorithm prevails and our method is outperformed. Large population sizes are especially useful for many objective problems where the number of individuals needed to approximate the Pareto front with a fixed precision grows *exponentially* with the number of objectives [18].

The number of objectives ranges from 3 to 8, while the number of variables is always 15. For each configuration we ran the algorithm for 500 generations, 10 times with different random seeds, and averaged the results. The experimental setup is summarized in Table III.

We implemented all algorithms in C++ and compiled them using the gcc 4.8.1 compiler using the aggressive compiler optimization flag -O2. We ran the experiments on a desktop PC with an Intel Core i7-2600 CPU @ 3.40GHz x 8 processor running the Ubuntu 13.04 operating system with Linux 3.8.0-19.29 kernel. We ran the experiments one at a time, with no other programs running.

B. Comparison with Jensen-Fortin's algorithm

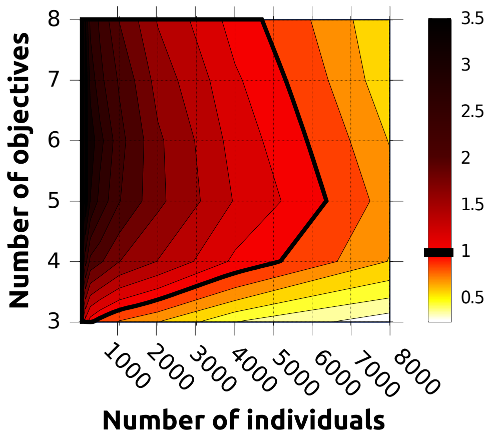
1) Total computation time: First we measured the total wall clock computation time *spent just on non-dominated sorting* using the C++11 `<chrono>` library. Instead of presenting the *absolute* times, which we believe to be more susceptible to change from platform to platform, we present *ratios* of the average time used by Jensen-Fortin's algorithm *divided* by the average time used by our method. Since we have the same number of runs for both algorithms, the presented ratios become:

$$\text{ratio} = \frac{\sum_{k=1}^{10} \mathrm{JF}_k}{\sum_{k=1}^{10} \mathrm{MF}_k}$$

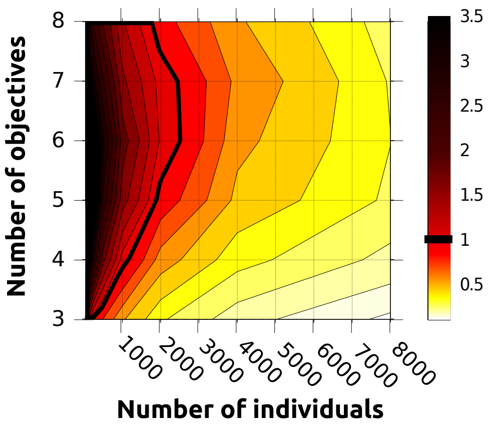
where $\mathrm{JF}_1, \ldots, \mathrm{JF}_{10}$ are the times taken by Jensen-Fortin's algorithm and $\mathrm{MF}_1, \ldots, \mathrm{MF}_{10}$ are the times taken by the M-front method. All results are summarized in Table IV. All results have been tested for significance using the Wilcoxon signed rank test on the significance level 0.05. A number in boldface means that our method *significantly* outperformed the competitor, while a number in italics means that our method has been *significantly* outperformed for that particular configuration. A number without boldface or italics means that the results were not significantly different. We use the same notation in Table V, where we compare our results with fast non-dominated sorting. Numbers greater than one mean that our method is faster in this instance. For the sake of perspective, we provide the average times used by Jensen-Fortin's algorithm for the *most complex* and *least* complex problem setup in Table VI.

TABLE IV
RATIO OF AVERAGES $\frac{\text{JENSEN-FORTIN}}{\text{M-FRONT}}$.

	WFG9								DTLZ1							
M	N = 50	125	250	500	1000	2000	4000	8000	N = 50	125	250	500	1000	2000	4000	8000
	Total non-dominated sorting time (ratios)															
8	**3.13**	**3.50**	**2.79**	**2.54**	**2.05**	**1.57**	**1.10**	*0.56*	**1.96**	**2.97**	**2.53**	**2.03**	**1.40**	*0.90*	*0.57*	*0.29*
7	**2.79**	**3.77**	**3.08**	**2.67**	**2.28**	**1.72**	**1.21**	*0.59*	**2.20**	**3.97**	**3.27**	**2.61**	**1.64**	**1.11**	*0.65*	*0.33*
6	**2.79**	**3.61**	**3.36**	**2.98**	**2.41**	**1.86**	**1.29**	*0.68*	**2.32**	**5.08**	**4.23**	**3.23**	**1.94**	**1.15**	*0.59*	*0.34*
5	**2.96**	**3.78**	**3.17**	**2.78**	**2.33**	**1.87**	**1.36**	*0.76*	**2.44**	**5.30**	**4.40**	**2.78**	**1.69**	*0.94*	*0.52*	*0.32*
4	**2.27**	**2.70**	**2.23**	**2.10**	**1.82**	**1.58**	**1.14**	*0.68*	**1.91**	**3.93**	**2.89**	**1.89**	**1.10**	*0.62*	*0.37*	*0.23*
3	*1.00*	**1.29**	1.09	0.96	*0.82*	*0.69*	*0.44*	*0.28*	1.11	**1.48**	1.01	*0.82*	*0.57*	*0.35*	*0.21*	*0.13*
	Non-dominated sorting time in the last generation (ratios)															
8	**2.38**	**3.68**	**2.87**	**2.60**	**2.17**	**1.68**	**1.14**	*0.59*	**2.15**	**3.66**	**2.24**	**2.30**	**1.47**	*0.89*	*0.62*	*0.32*
7	**3.30**	**4.61**	**3.04**	**2.72**	**2.33**	**1.78**	**1.24**	*0.61*	**2.07**	**4.27**	**3.36**	**2.67**	**1.78**	1.04	*0.67*	*0.30*
6	**2.15**	**4.07**	**3.45**	**3.08**	**2.45**	**1.89**	**1.33**	*0.70*	**2.17**	**4.28**	**4.57**	**3.18**	**1.91**	**1.16**	*0.58*	*0.34*
5	**2.64**	**3.42**	**3.10**	**2.67**	**2.31**	**1.87**	**1.41**	*0.79*	**2.01**	**4.03**	**4.00**	**2.70**	**1.96**	**1.19**	*0.74*	*0.40*
4	**2.94**	**3.01**	**2.21**	**2.01**	**1.79**	**1.69**	**1.27**	*0.79*	**1.65**	**3.05**	**3.11**	**2.58**	**1.88**	**1.26**	*0.81*	*0.51*
3	0.77	**1.37**	1.04	1.04	*0.89*	*0.86*	*0.62*	*0.48*	0.90	**1.58**	1.26	**1.32**	1.01	*0.82*	*0.54*	*0.42*



(a) WFG9



(b) DTLZ1

Fig. 17. Ratio of average total non-dominated sorting time: $\frac{\text{Jensen-Fortin}}{\text{M-front}}$.

The numbers in parentheses are the standard deviations across the 10 runs.

We can see that for 3 objectives the Jensen-Fortin's method is faster in almost all instances. For 4 objectives *and higher* our method catches up and outperforms the Jensen-Fortin's algorithm for all population sizes up to 4000 individuals. There the asymptotic superiority of Jensen-Fortin's algorithm becomes apparent.

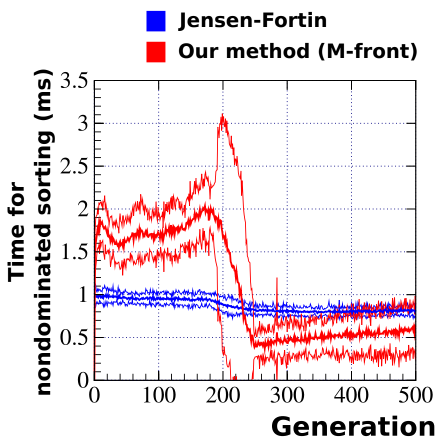
The smaller the population the better our method performs in comparison to Jensen-Fortin's algorithm. However, this trend breaks down for 50 individuals. This is probably due to the fixed cost that the K-d tree carries along.

The reader should note that the initial population sizes do not correspond *exactly* to the size of the domination sorting problem being solved. The GDE3 algorithm produces a population that has somewhere between N and $2N$ individuals, which needs to be trimmed to N individuals. Therefore the size of the problem being solved is slightly bigger.

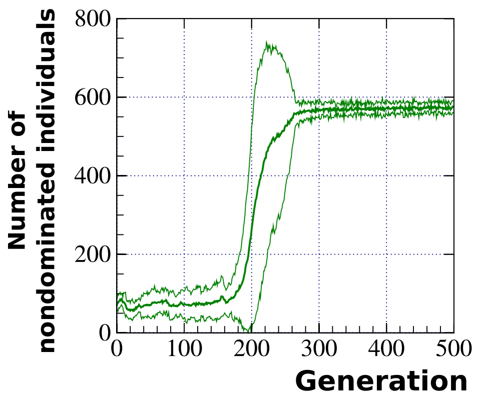
We can interpolate our experimental results so that the pattern is more visible. In Fig. 17 we can see a contour plot which interpolates our experimental results. Population sizes in our experiments grow exponentially, but Fig. 17 tries to give us a better understanding of the overall pattern. The isocurve for 1, that is the hypothetical curve on which the two algorithms perform with the same speed, is shown in bold black.

When we look at the results for DTLZ1 in Table IV we see a similar pattern. The asymptotic superiority of Jensen-Fortin's method is immediately visible. Moreover we see that for large populations the results are more favorable for Jensen-Fortin's method. Already for 2000 individuals the performance of the two algorithms are tied, while for 4000 and 8000 individuals Jensen-Fortin's algorithm is faster. On the other hand, for small population sizes and high number of objectives, our method is faster.

2) Computation time in the last generation: On first sight it may seem strange that the results for 3 objectives are so unfavorable for our method. We shall now examine this situation in detail.



(a) Decrease in the average computational time of our method around generation 200.



(b) Increase in the average number of non-dominated individuals around generation 200.

Fig. 18. Influence of overnondomination on computational speed of our method for DTLZ1 - 500 individuals - 3 objectives. All series are surrounded by a band of ± 2 standard deviations.

Let us look at the computational time spent on non-dominated sorting *in each generation*. In Fig. 18a we can see the averaged computational times for the DTLZ1 problem with 3 objectives and 500 individuals. Our method is significantly slower, but around generation 200 it accelerates and becomes faster than Jensen-Fortin's algorithm. The reason for this behavior is that the proportion of non-dominated individuals is relatively small in the first 200 generations. We can see this in the lower part of Fig. 18b. When the number of non-dominated individuals is small, our algorithm needs to resort to using the algorithm in Fig. 4 to determine additional fronts. Once the number of non-dominated individuals is greater than the initial population size, 500 individuals in this case, the GDE3 algorithm has enough non-dominated individuals in the M-front and does not need to invoke the auxiliary algorithm (Fig. 4) to compute an additional non-dominated front.

We can examine how strong this effect is, by comparing the average computational times *only in the last generation*. The intuition is that by the last generation the population has almost converged and the proportion of the non-dominated individuals is high.

TABLE VI
AVERAGE ABSOLUTE TIME TAKEN BY JENSEN-FORTIN'S ALGORITHM FOR THE MOST COMPLEX AND LEAST COMPLEX PROBLEM SETUP (MILLISECONDS).

	Total	
	$M = 3, N = 50$	$M = 8, N = 8000$
WFG9	52.7061 (0.818)	615 008 (11619.6)
DTLZ1	26.5054 (0.383)	259 788 (6472.1)
	In the last generation	
WFG9	0.1024 (0.0045)	1279.94 (36.14)
DTLZ1	0.0522 (0.0039)	581.20 (18.98)

These results are summarized in the bottom of Table IV. By comparing the data for the last generation to the total data we can see that the most significant differences are visible for 3 objectives. The reason is that for 3 objectives the overnondomination phenomenon is not yet present.

C. Comparison with fast non-dominated sorting

Here we present only the results for the WFG9 algorithm in order to save space, since the core of our experimental section is the comparison with Jensen-Fortin's algorithm. The results for DTLZ1 were slightly worse, but similar to those for WFG9. This can be inferred from looking at Table IV. We choose the same methodology of presenting our data as when comparing with Jensen-Fortin's algorithm.

In Table V we see that our algorithm outperforms the fast non-dominated sorting in *all* problem instances. With few exceptions the relative performance of our algorithm *increases with population size* and *decreases with number of objectives*. This is in accordance with our estimation of average computational complexity. To quantify the importance of overnondomination, we also present the comparison of computational times for the last generation. By comparing these results with the total times in Table V, we can see that the biggest difference is for 3 objectives. This is consistent with our previous analysis.

The fast non-dominated sorting algorithm performs *domination comparisons* between pairs of individuals. Our algorithm also performs *domination comparisons* when it compares the inserted individual to the individuals in the reference sets. We can count the number of these comparisons which are executed during the entire run of the optimizer. This way we get a measure which is *independent* from the underlying hardware and programming language, since all implementations should perform exactly the same steps.

We can see the comparison in terms of domination comparisons in the bottom part of Table V. We can see that these results are strongly correlated to the results in terms of computational time. This favors the hypothesis that a major part of the computational time is consumed by domination comparisons.

It is also interesting to see that the results in terms of domination comparisons are more favorable for our algorithm than the results in terms of computational time. In other words our algorithm does not reach its full potential. For example

TABLE V
RATIO OF AVERAGES $\frac{\text{Fast non-dominated sorting}}{\text{M-front}}$ FOR THE WFG9 PROBLEM.

	Total non-dominated sorting time (ratios)								Non-dominated sorting time in the last generation (ratios)							
M	N = 50	125	250	500	1000	2000	4000	8000	N = 50	125	250	500	1000	2000	4000	8000
8	**1.30**	**2.06**	**2.38**	**3.23**	**3.81**	**4.11**	**4.17**	**3.54**	0.97	**1.99**	**2.34**	**3.26**	**3.92**	**4.29**	**4.25**	**3.52**
7	**1.23**	**2.37**	**2.80**	**3.65**	**4.75**	**5.35**	**5.49**	**4.53**	**1.43**	**2.70**	**2.66**	**3.64**	**4.77**	**5.43**	**5.58**	**4.54**
6	**1.33**	**2.51**	**3.43**	**4.82**	**6.19**	**7.40**	**7.94**	**7.09**	1.03	**2.66**	**3.38**	**4.89**	**6.22**	**7.52**	**8.15**	**7.11**
5	**1.59**	**3.18**	**4.11**	**6.05**	**8.61**	**11.36**	**13.69**	**13.28**	**1.37**	**2.80**	**4.00**	**5.78**	**8.57**	**11.40**	**13.92**	**13.66**
4	**1.69**	**3.60**	**5.04**	**8.65**	**13.64**	**20.53**	**26.39**	**28.10**	**2.11**	**3.81**	**4.94**	**8.26**	**13.58**	**22.22**	**29.45**	**32.35**
3	**1.78**	**5.11**	**7.94**	**13.56**	**21.89**	**33.49**	**37.80**	**43.58**	1.38	**5.64**	**7.80**	**15.22**	**24.80**	**42.93**	**55.14**	**75.79**
	Total number of domination comparisons (ratios)								Number of domination comparisons in the last generation (ratios)							
8	**3.8**	**5.1**	**5.6**	**5.9**	**6.4**	**6.8**	**7.2**	**7.6**	**4.3**	**5.5**	**5.9**	**6.2**	**6.7**	**7.1**	**7.3**	**7.7**
7	**4.6**	**6.4**	**7.2**	**7.9**	**8.5**	**9.1**	**9.6**	**10.1**	**5.1**	**6.9**	**7.8**	**8.3**	**8.6**	**9.2**	**9.7**	**10.3**
6	**6.0**	**8.7**	**10.0**	**11.0**	**12.0**	**13.1**	**14.3**	**15.8**	**6.3**	**9.3**	**10.4**	**11.2**	**12.2**	**13.3**	**14.6**	**16.1**
5	**8.2**	**12.3**	**15.0**	**17.6**	**20.2**	**23.1**	**26.3**	**29.6**	**8.6**	**12.3**	**16.2**	**17.7**	**20.6**	**23.8**	**27.6**	**31.5**
4	**11.4**	**20.5**	**29.8**	**37.2**	**44.9**	**51.0**	**57.9**	**63.9**	**11.8**	**20.3**	**32.3**	**40.2**	**50.6**	**62.2**	**74.7**	**89.9**
3	**22.4**	**46.4**	**68.8**	**80.3**	**89.9**	**90.1**	**89.5**	**88.7**	**25.0**	**50.7**	**88.3**	**125.2**	**178.9**	**240.8**	**341.8**	**469.6**

with 3 objectives and 1000 individuals our algorithm requires **89.9** times *fewer* domination comparisons, but overall it is only **21.89** times faster. This is caused by all the additional data structures that our algorithm needs to maintain. This includes primarily the K-d tree, but also the sorted lists, the reference sets, and the hash-table. The results for the number of dominance comparisons can be improved using *exact* nearest neighbor computation within the M-front.

To see the importance of overnondomination in our algorithm we provide the ratios of domination comparisons needed in the last generation in the bottom right part of Table V. Again the differences between the *total* numbers of domination comparisons and domination comparisons in the *last* generation are most pronounced for 3 objectives, where the overnondomination is weakest.

D. Confirmation of theoretical results

In the chapter on computational complexity, we tried to model the population in an M-front using random vectors and then use the techniques from the theory of probability to model the computational cost of operations on the M-front. During this modeling, we have made a number of assumptions on the distribution of the individuals in the population. These assumptions are relatively simple in order to make the proofs of our theorems simple. Now we shall confront this model to the experimental data.

We have shown that an average insertion into the M-front needs $O(MN^{1-\frac{1}{M-1}})$ domination comparisons where N is the number of individuals in the M-front. Assuming overnondomination, there are roughly as many individuals in the M-front as is the initial population size. We can try to substitute the initial population size in our experimental data for N.

The number of insertions in one generation is exactly the same as the initial population size. Therefore the computational cost is roughly $O(MN^{2-\frac{1}{M-1}})$ domination comparisons, where N is the initial population size.

TABLE VII
FITTING OF CURVE $f(N) = \alpha N^{\beta}$ TO EXPERIMENTAL DATA:
NUMBER OF DOMINATION COMPARISONS IN THE LAST GENERATION (WFG9).

M	β theoretical	β estimate	α estimate
8	1.857	1.902	1.170
7	1.833	1.890	0.978
6	1.800	1.848	0.923
5	1.750	1.768	0.958
4	1.667	1.625	1.122
3	1.500	1.442	1.095

For a fixed M this gives the power function:

$$f(N) = \alpha N^{\beta}. \tag{10}$$

We tried to fit the general curve given by (10) to our experimental results on the WFG9 problem using the least squares method. In almost all cases the approximation was appropriate. First let us look at the data for the *number of domination comparisons in the last generation* which is presented in Table VII.

We can see that the estimated β coefficients are not very far from their theoretical counterparts. There is a slight tendency for the experimental coefficients to be higher. On the other hand the (less important) α coefficients seem to be more or less constant with respect to M. We believe that this is due to the fact that we used the *maximum* metric in our theoretical computation, but in experiments we used the more strict *Manhattan* metric.

Next we present the comparison in terms of actual wall clock time. The comparison of fitted and theoretical coefficients for the *elapsed time in the last generation* is in Table VIII.

Here we see that the β coefficients are slightly underestimated for *all* values of M. Nevertheless, the theoretical and estimated coefficients follow a similar pattern as we increase the number of objectives.

TABLE VIII
FITTING OF CURVE $f(N) = \alpha N^{\beta}$ TO EXPERIMENTAL DATA: TIME ELAPSED IN THE LAST GENERATION (MICROSECONDS) (WFG9).

M	β theoretical	β estimate	α estimate
8	1.857	1.818	0.111
7	1.833	1.832	0.078
6	1.800	1.704	0.145
5	1.750	1.607	0.196
4	1.667	1.503	0.226
3	1.500	1.304	0.500

VII. CONCLUSION

We have presented a new method to decrease the cost of non-dominated sorting. The main idea is to use a special data structure which holds and *updates* the knowledge of non-dominated individuals during the run of a MOEA. We have provided one such data structure which we call the M-front.

Jensen-Fortin's algorithm is the latest, and in terms of computational complexity, fastest algorithm. We have demonstrated that although the M-front does not provide an improvement in terms of average computational complexity, it is faster than Jensen-Fortin's algorithm on some problems for a broad range of population sizes and numbers of objectives. For very big populations the asymptotic superiority of Jensen-Fortin's algorithm eventually prevails but our algorithm can perform faster up to a fairly big population size (approximately 4000 individuals). Besides that, our algorithm scales very well with the number of objectives. This is because our algorithm uses the fact that a large proportion of the population tends to be non-dominated in such instances to its advantage.

All performance results aside, an important advantage of our method is that the non-dominated individuals are *known at all times*. If one individual changes, the change in the non-dominated front is immediately recorded. The core of our algorithm is the M-front which can be reused as a cost-effective data structure for MOEAs which archive their non-dominated individuals.

In future we should explore different implementations of the fast archive. An implementation using *segment trees* [19] seems to be a promising candidate. Also, replacing the K-d tree in the M-front with a more sophisticated nearest neighbor data structure may bring further improvement.

ACKNOWLEDGMENT

The work of Martin Drozdík has been supported by the Monbukagakusho scholarship of the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

APPENDIX A
PROOF OF THEOREM 2

Proof: Suppose that we have two individuals $a, b \in RF_{Pf}$, whose objective values are:

$$Y_a = (y_{a,1}, y_{a,2}, \dots, y_{a,M-1}, Pf(y_{a,1}, y_{a,2}, \dots, y_{a,M-1}))$$
$$Y_b = (y_{b,1}, y_{b,2}, \dots, y_{b,M-1}, Pf(y_{b,1}, y_{b,2}, \dots, y_{b,M-1}))$$

For a to dominate b it is necessary that $y_{a,i} \leq y_{b,i}$ for all i and $y_{a,j} < y_{b,j}$ for at least one j, for $i,\ j \in [\![1; M]\!]$. However, if there is such a $j < M$ we have

$$y_{a,M} = Pf(y_{a,1}, \dots, y_{a,M-1}) > Pf(y_{b,1}, \dots, y_{b,M-1}) = y_{b,M}$$

and if there is not a such $j < M$, meaning $y_{a,i} = y_{b,i}$ for all $i < M$, it implies

$$y_{a,M} = Pf(y_{a,1}, \dots, y_{a,M-1}) = y_{b,M}.$$

Hence a cannot dominate b. This ends the proof. ■

APPENDIX B
PROOF OF THEOREM 3

We shall need the following three lemmas in our proof.

Lemma 1 (First order statistic)**.** *Let $x_1, \dots, x_n$ be i.i.d. random variables and F be the cumulative distribution function (cdf) of these variables. Then the cdf of the random variable*

$$x_{\min} := \min(x_1, \dots, x_n) \tag{11}$$

is given by:

$$F_{\min}(x) = 1 - (1 - F(x))^n \ . \tag{12}$$

The $x_{\min}$ from (11) is called a *first order statistic*. More information on order statistics can be found in [27].

Lemma 2 (Expected distance of closest point)**.** *Let $X_1, \dots, X_N$ be i.i.d. random* vectors *with uniform distribution on* $[0; 1]^M$. *Let $X_{\min}$ denote the random vector which is closest to the center $c = (0.5, 0.5, \dots, 0.5)$ of the hyper-box* $[0; 1]^M$ *with respect to the* maximum metric *d. Then, the expected distance of $X_{\min}$ from the center c,*

$$D_{M,N} := E[d(X_{\min}, c)] \ , \tag{13}$$

can be expressed in a closed form as:

$$D_{M,N} = \frac{N}{2} B(N, 1 + \frac{1}{M}) \ , \tag{14}$$

where $B(x, y) := \int_0^1 t^{x-1}(1-t)^{y-1} dt$ is the beta function.

Proof: Let us first construct the cumulative distribution function (cdf) F_1 for the distance $d(X_{\rm i}, c)$ of an arbitrary X_i from c. By definition, the value of a cdf in $x \in \mathbb{R}$ is the probability that the random variable is smaller than x. The set of all vectors whose distance to c is less than x with respect to the maximum measure forms a cube with edge length of $2x$. Then, since $X_i \sim U[0; 1]^M$, the probability of X_i falling into such a cube is equal to the volume of the cube, namely $(2x)^M$. That is, $F_1(x) = (2x)^M$ for $x \in [0; 0.5]$.

Then, since the distance of the closest individual to c is in fact the minimum of the distances, we have from (12) that the cdf of $d(X_{\min}, c)$ is

$$F_N(x) = 1 - (1 - (2x)^M)^N \quad \text{for any } x \in [0; 0.5].$$

Since $d(X_{\min}, c)$ is nonnegative, its expectation is computed by

$$\begin{aligned} D_{M,N} &= \int_0^{1/2} (1 - F_N(x))dx = \int_0^{1/2} (1-(2x)^M)^N dx \\ &= \frac{1}{2M}\int_0^1 (1-t)^N t^{\frac{1}{M}-1} dt = \frac{1}{2M} B(N+1, \frac{1}{M}) \\ &= \frac{N}{2} B(N, \frac{1}{M}+1) \ . \end{aligned}$$

This completes the proof. ■

The asymptotic properties of this expected distance are summarized in the following theorem:

Lemma 3 (Asymptotic properties of $D_{M,N}$). *Let $D_{M,N}$ be the expected distance from* (13). *Then for any $M \geq 1$:*

$$\lim_{N\to\infty} N^{\frac{1}{M}} D_{M,N} = \frac{1}{2}\Gamma(1+\frac{1}{M}) \ . \tag{15}$$

Here Γ is the Gamma function.

Proof: Note that

$$B(N, \frac{1}{M}+1) = \frac{\Gamma(N)\Gamma(\frac{1}{M}+1)}{\Gamma(N+\frac{1}{M}+1)} \ .$$

Using the asymptotic property of the Gamma function, derived from Stirling's formula:

$$\lim_{n\to\infty} \frac{\Gamma(n+x)}{n^x \Gamma(n)} = 1, \quad \text{for any } x \in \mathbb{R},$$

we have

$$\begin{aligned} N^{\frac{1}{M}} D_{M,N} &= \frac{N^{\frac{1}{M}+1}}{2} B(N, \frac{1}{M}+1) \\ &= \frac{N^{\frac{1}{M}+1}\Gamma(N)}{\Gamma(N+\frac{1}{M}+1)} \frac{\Gamma(\frac{1}{M}+1)}{2} \\ &\xrightarrow{N\to\infty} \frac{\Gamma(\frac{1}{M}+1)}{2} \ . \end{aligned}$$

This ends the proof. ■

Now we are ready to prove Theorem 3.

Proof of Theorem 3: A random individual $a \in RF_{Pf}$ belongs to a reference set induced by ref and new iff the following condition is satisfied for some $i \in \{1, 2, \ldots, M\}$:

$$y_{a,i} \in [y_{\texttt{new},i}; y_{\texttt{ref},i}] \text{ or } y_{a,i} \in [y_{\texttt{ref},i}; y_{\texttt{new},i}]. \tag{16}$$

Define sets

$$\begin{aligned} &A(x) := \{y \in [0;1]^{M-1} \mid \exists i \in [\![1; M-1]\!],\ y_i \in [0.5-x; 0.5+x]\} \\ &B(x) := [Pf(0.5+x, \ldots, 0.5+x); Pf(0.5-x, \ldots, 0.5-x)] \end{aligned}$$

and a random variable $\delta = d_{M-1}(Y_{\texttt{ref}}, Y_{\texttt{new}})$. Since Pf is strictly decreasing w.r.t. each element and $Y_{\texttt{new}} = (0.5, \ldots, 0.5, Pf(0.5, \ldots, 0.5))$, the reference area induced by $Y_{\texttt{ref}}$ is a subset of $A(\delta) \times B(\delta)$. Therefore, letting $\check{C}_{M,N}$ denote the number of individuals whose first $M-1$ components exist in $A(\delta)$ and $\hat{C}_{M,N}$ denote the number of individuals whose last component exists in $B(\delta)$, we have $C_{M,N} \leq \check{C}_{M,N} + \hat{C}_{M,N}$. Since the inequality inherits when the expectation is taken, we find

$$E[C_{M,N}] \leq E[\check{C}_{M,N}] + E[\hat{C}_{M,N}] \ . \tag{17}$$

Hence, it suffices to show the right-hand side is bounded by a desired order.

First, we consider $E[\check{C}_{M,N}]$. Let $\mathbb{I}\{X\}$ be the indicator which is 1 if the event X happens and 0 otherwise. For the simplicity of notation, we let Z_a be the first $M-1$ components of $a \in RF_{Pf}$. Then,

$$\begin{aligned} E[\check{C}_{M,N}] &= E\big[\textstyle\sum_{i=1}^N \mathbb{I}\{Z_i \in A(\delta)\}\big] \\ &= \textstyle\sum_{i=1}^N E\left[\mathbb{I}\{Z_i \in A(\delta)\}\right] \ . \end{aligned}$$

Remember that $\texttt{ref} = \operatorname{argmin}_{i\in[\![1;N]\!]} d_{M-1}(Z_i, Z_{\texttt{new}})$ and $\delta = d_{M-1}(Z_{\texttt{ref}}, Z_{\texttt{new}}) = \min_{i\in[\![1;N]\!]} d_{M-1}(Z_i, Z_{\texttt{new}})$. The inside of the summation is then

$$\begin{aligned} &E[\mathbb{I}\{Z_i \in A(\delta)\}] \\ &= E[\mathbb{I}\{i = \texttt{ref}\} + \mathbb{I}\{i \neq \texttt{ref}\}\mathbb{I}\{Z_i \in A(\delta)\}] \\ &= 1/N + E[\mathbb{I}\{i \neq \texttt{ref}\}\mathbb{I}\{Z_i \in A(\delta)\}] \\ &= 1/N + E[\mathbb{I}\{i \neq \texttt{ref}\}\mathbb{I}\{Z_i \in A(\min_{i\in[\![1;N]\!]} d_{M-1}(Z_i, Z_{\texttt{new}}))\}] \\ &= 1/N + E[\mathbb{I}\{i \neq \texttt{ref}\}\mathbb{I}\{Z_i \in A(\min_{j\in[\![1;N]\!]\setminus\{i\}} d_{M-1}(Z_j, Z_{\texttt{new}}))\}] \\ &\leq 1/N + \Pr[Z_i \in A(\min_{j\in[\![1;N]\!]\setminus\{i\}} d_{M-1}(Z_j, Z_{\texttt{new}}))] \ . \end{aligned} \tag{18}$$

For the last inequality we used $E[\mathbb{I}\{X\}\mathbb{I}\{Y\}] \leq E[\mathbb{I}\{X\}]E[\mathbb{I}\{Y\}]$ and $E[\mathbb{I}\{X\}] = \Pr[X]$. Note that on the right-most side Z_i is independent of $\min_{j\in[\![1;N]\!]\setminus\{i\}} d_{M-1}(Z_j, Z_{\texttt{new}})$. Since Z_i is uniformly distributed in $[0;1]^{M-1}$, given

$$\delta_i = \min_{j\in[\![1;N]\!]\setminus\{i\}} d_{M-1}(Y_j, Y_{\texttt{new}})$$

the above probability is the proportion of the volume of $A(\delta_i)$ to $[0;1]^{M-1}$, which is bounded above by $2(M-1)\delta_i$. Then,

$$E[\check{C}_{M,N}] \leq 1 + 2(M-1)\textstyle\sum_{i=1}^N E[\delta_i] \ . \tag{19}$$

Next, we consider $E[\hat{C}_{M,N}]$. For the simplicity of notation, we let W_a be the last component of $a \in RF_{Pf}$. Then, By definition

$$\begin{aligned} E[\hat{C}_{M,N}] &= E\big[\textstyle\sum_{i=1}^N \mathbb{I}\{W_i \in B(\delta)\}\big] \\ &= \textstyle\sum_{i=1}^N E\left[\mathbb{I}\{W_i \in B(\delta)\}\right] \ . \end{aligned}$$

Analogously to (18), we have

$$E\left[\mathbb{I}\{W_i \in B(\delta)\}\right] = 1/N + \Pr[W_i \in B(\delta_i)] \ .$$

Note that on the right-most side W_i is independent of δ_i. Given δ_i, the above probability reads

$$\begin{aligned} &\Pr[W_i \in B(\delta_i) \mid \delta_i] \\ &= \textstyle\int_{Pf(0.5+\delta_i,\ldots,0.5+\delta_i)}^{Pf(0.5-\delta_i,\ldots,0.5-\delta_i)} f_{Pf}(w)dw \\ &\leq \textstyle\int_{Pf(0.5+\delta_i,\ldots,0.5+\delta_i)}^{Pf(0.5-\delta_i,\ldots,0.5-\delta_i)} S dw \\ &= S|Pf(0.5-\delta_i, \ldots, 0.5-\delta_i) - Pf(0.5+\delta_i, \ldots, 0.5+\delta_i)| \\ &\leq 2SL\delta_i. \end{aligned}$$

Taking the expectation over δ_i and plugging it into the above inequalities we have

$$E[\hat{C}_{M,N}] \leq 1 + 2SL\textstyle\sum_{i=1}^N E[\delta_i] \ . \tag{20}$$

With (17), (19) and (20) we have

$$E[C_{M,N}] \leq 2 + 2((M-1) + SL)\sum_{i=1}^{N} E[\delta_i] \ . \quad (21)$$

Note that $E[\delta_i]$ is nothing but $D_{M-1,N-1}$ in Lemma 2. Hence, we find

$$E[C_{M,N}] \leq 2 + 2((M-1) + SL)ND_{M-1,N-1} \ . \quad (22)$$

For the limit of $N \to \infty$, using Theorem 3 we obtain

$$\begin{aligned} &\lim_{N\to\infty} \frac{E[C_{M,N}]}{N^{1-\frac{1}{M-1}}} \\ &\leq \lim_{N\to\infty} \frac{2 + 2((M-1) + SL)ND_{M-1,N-1}}{N^{1-\frac{1}{M-1}}} \\ &= ((M-1) + SL)\Gamma(1 + \frac{1}{M-1}) \in O(M) \ . \end{aligned}$$

Therefore, we find $E[C_{M,N}] \in O(MN^{1-\frac{1}{M-1}})$. ■

References

[1] M. Drozdik, H. Aguirre, and K. Tanaka, "Attempt to Reduce the Computational Complexity in Multi-objective Differential Evolution algorithms," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO 13. New York, NY, USA: ACM, 2013, pp. 599–606.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm NSGA-II," *IEEE Transactions on evolutionary computation*, vol. 6, no. 2, 2002.

[3] S. Kukkonen and J. Lampinen, "GDE3: The Third Evolution Step of Generalized Differential Evolution," in *IEEE Congress on Evolutionary Computation*, 2005, pp. 443–450.

[4] T. Robič and B. Filipič, "DEMO: Differential Evolution for Multiobjective Optimization," *Proc. Intl. conf. on Evolutionary Multi-criterion optimization (EMO 2005), Springer, LNCS 3410*, pp. 520–533, 2005.

[5] S. Huband, P. Hingston, L. Barone, and L. While, "A Review of Multiobjective Test Problems and a Scalable Test Problem Toolkit," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 5, pp. 477–506, Oct 2006.

[6] H. T. Kung, F. Luccio, and F. P. Preparata, "On Finding the Maxima of a Set of Vectors," *Journal of the ACM*, vol. 22, pp. 469–476, 1975.

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

[8] F.-A. Fortin, S. Grenier, and M. Parizeau, "Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO 13. New York, NY, USA: ACM, 2013, pp. 615–622.

[9] K. McClymont and E. Keedwell, "Deductive Sort and Climbing Sort: New Methods for Non-dominated Sorting," *Evol. Comput.*, vol. 20, no. 1, pp. 1–26, Mar 2012.

[10] O. Schütze, "A New Data Structure for the Nondominance Problem in Multi-objective Optimization," in *Proceedings of the 2nd international conference on Evolutionary multi-criterion optimization*, ser. EMO03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 509–518.

[11] J. E. Fieldsend, R. M. Everson, and S. Singh, "Using Unconstrained Elite Archives for Multiobjective Optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 7, no. 3, pp. 305–323, June 2003.

[12] D. Hadka and P. Reed, "Borg: An Auto-adaptive Many-objective Evolutionary Computing Framework," *Evol. Comput.*, vol. 21, no. 2, pp. 231–259, 2013.

[13] E. Zitzler, "Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications," Ph.D. dissertation, Comput. Eng. Netw. Lab. Swiss Federal Instit. Technol. (ETH), Zurich, Switzerland, 1999.

[14] K. Tagawa, H. Shimizu, and H. Nakamura, "Indicator-based Differential Evolution Using Exclusive Hypervolume Approximation and Parallelization for Multi-core Processors," in *GECCO*, ser. GECCO 11. New York, NY, USA: ACM, 2011, pp. 657–664.

[15] L. While, L. Bradstreet, and L. Barone, "A Fast Way of Calculating Exact Hypervolumes," *Evolutionary Computation, IEEE Transactions on*, vol. 16, no. 1, pp. 86–95, Feb 2012.

[16] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution - A Practical Approach to Global Optimization*. Berlin, Germany: Springer, 2005.

[17] M. Farina and P. Amato, "On the Optimal Solution Definition for Many-criteria Optimization Problems," in *Fuzzy Information Processing Society, 2002. Proceedings. NAFIPS. 2002 Annual Meeting of the North American*, 2002, pp. 233–238.

[18] H. E. Aguirre and K. Tanaka, "Working Principles, Behavior, and Performance of MOEAs on MNK-landscapes ," *European Journal of Operational Research* , vol. 181, no. 3, pp. 1670–1690, 2007.

[19] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, 2000.

[20] C. Toth, J. O'Rourke, and J. Goodman, *Handbook of Discrete and Computational Geometry, Second Edition*, ser. Discrete and Combinatorial Mathematics Series. Taylor & Francis, 2004.

[21] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[22] S. Kukkonen and K. Deb, "A Fast and Effective Method for Pruning of Non-dominated Solutions in Many-Objective Problems," in *Parallel Problem Solving from Nature - PPSN IX*. Springer Berlin Heidelberg, 2006, vol. 4193, ch. Lecture Notes in Computer Science, pp. 553–562. [Online]. Available: {http://dx.doi.org/10.1007/11844297_56}

[23] M. Drozdik, "Data structures," https://bitbucket.org/martin_drozdik/datastructures, Oct 2014, source code.

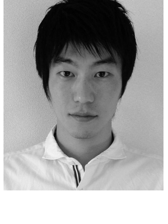
[24] M. Wagner and F. Neumann, "A Fast Approximation-guided Evolutionary Multi-objective Algorithm," in *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, ser. GECCO '13. New York, NY, USA: ACM, 2013, pp. 687–694.

[25] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable Test Problems for Evolutionary Multi-Objective Optimization," in *Evolutionary Multiobjective Optimization*. Springer Berlin Heidelberg, 2005, pp. 105–145.

[26] S. Kukkonen, "Generalized Differential Evolution for Global Multi-Objective Optimization with Constraints," *Acta Universitatis Lappeenrantaensis*, 2012.

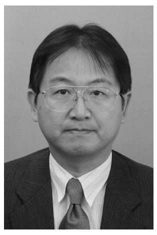
[27] H. A. David and H. N. Nagaraja, *Basic Distribution Theory*. John Wiley & Sons, Inc., 2005, pp. 9–32.

Martin Drozdík received his bachelors and masters degree in applied mathematics from the Comenius University in Bratislava, Slovakia, in 2008 and 2010, respectively. During his masters studies he spent two semesters, studying mathematics and economics, at the University of Pisa in Italy, thanks to the Erasmus student exchange program. Currently he is working towards his PhD at the Interdisciplinary Graduate School of Science and Technology, Shinshu University in Nagano, Japan. His research interests are differential evolution, evolutionary computation, multi-objective optimization and programming in C++.



Youhei Akimoto received his academic degrees in computer science (bachelor) and in computational intelligence and systems science (master and Ph.D.) from Tokyo Institute of Technology in 2007, 2008, and 2011, respectively. From April 2010 to March 2011, he was also a JSPS research fellow. From April 2011 he worked for INRIA-Saclay as a postdoctoral fellow for two years. Since April 2013, he works at Shinshu University as an assistant professor. His research interests include mathematical optimization especially on continuous domain, function-value free optimization, and randomized search heuristics.

Hernán Aguirre received his Engineer degree in computer systems from Escuela Politécnica Nacional, Ecuador, in 1992, and the M.S. and Ph.D. degrees from Shinshu University, Japan, in 2000 and 2003, respectively. Currently, he is an associate professor at Shinshu University. His research interests include evolutionary computation, multidisciplinary design optimization, and sustainability. He has written over 130 international journal and conference research papers on evolutionary algorithms, focusing on the working principles of single-, multi-, and many-objective (any-objective) evolutionary optimizers, landscape analysis, and epistasis. He collaborates actively with industry and with the Japan Aerospace Exploration Agency (JAXA) on the development and real world application of many-objective evolutionary algorithms. He is a member of IEICE and IPSJ.



Kiyoshi Tanaka received his B.S and M.S. degrees in Electrical Engineering and Operations Research from National Defense Academy, Yokosuka, Japan, in 1984 and 1989, respectively. In 1992, he received Dr. Eng. degree from Keio University, Tokyo, Japan. In 1995, he joined the Department of Electrical and Electronic Engineering, Faculty of Engineering, Shinshu University, Nagano, Japan, where he is currently a full professor. He is the associate dean of Faculty of Engineering as well as the director of Shinshu University International Center (SUIC). His research interests include evolutionary computation, multi-objective optimization, smart grid, image and video processing, human visual perception, information hiding, and their applications. He is the project leader of JSPS Strategic Overseas Visits Program for Accelerating Brain Circulation entitled “Global Research on the Framework of Evolutionary Solution Search to Accelerate Innovation” started from 2013. He received IEVC2010 Best Paper Award from IIEEJ, iFAN2010 Best Paper Award from SICE, GECCO2011 Best Paper Award from ACM-SIGEVO, ISPACS2011 Best Paper Award from IEEE, Excellent Journal Paper Award from IIEEJ two times, in 2012 and in 2014, and Best Journal Paper Award from JSEC in 2012. He is a member of IEEE, IEICE, IPSJ, JSEC and IIEEJ. He is the editor in chief of Journal of the Institute of Image Electronics Engineers Japan as well as IIEEJ Transactions on Image Electronics and Visual Computing.