A Fast Incremental Archive for Multi-objective Optimization

Tobias Glasmachers
Institute for Neural Computation, Ruhr-University Bochum, Germany
tobias.glasmachers@ini.rub.de

Note: A paper with identical content was submitted to PPSN'2016.

Abstract

Maintaining an archive of all non-dominated points is a standard task in multi-objective optimization. A simple solution is to store all evaluated points and to obtain the non-dominated subset in a post-processing step. Alternatively the non-dominated set can be updated on the fly. Surprisingly, there exists little work on the latter setting. While keeping track of many non-dominated points efficiently is easy for two objectives, we propose an efficient algorithm based on a binary space partitioning (BSP) tree for the general case of three or more objectives. Our analysis and our empirical results demonstrate the superiority of the method over the brute-force baseline method.

1 Introduction

Given $m \geq 2$ objective functions $f_1, \ldots, f_m : X \rightarrow \mathbb{R}$, a central theme in multi-objective optimization is to find the Pareto set of optimal compromises: the set of points $x \in X$ that cannot be improved in any objective without getting worse in another one. The cardinality of this set of often huge or even infinite. In a black-box setting it is best approximated by the set of mutually non-dominated query points.

An archive $A$ of non-dominated solutions is a data structure for keeping track of the known non-dominated points $\{x^{(1)}, \ldots, x^{(n)}\} \subset X$ of a multi-objective optimization problem. It can result from a single run of an optimization algorithm, or from many runs of potentially different algorithm. In contrast to other studies (e.g., [4]) we aim to store all non-dominated solutions, not a subset of a-priori bounded size. Hence we are not concerned with the removal of non-dominated points, but rather with handling potentially huge sets.

Depending on the application such an archive can serve different purposes. It can act as a portfolio of solutions accessed by a decision maker, possibly after going through further post-processing steps. It can also act as input to various algorithms, e.g., selection operators of evolutionary algorithms, stopping criteria, and performance assessment and monitoring tools. All of these
algorithms may involve the computation of set quality indicators such as dominated hypervolume, for which the computation of the non-dominated subsets is a pre-processing step.

Some of the above applications require access to the non-dominated set “anytime”, i.e., already during the optimization (online case), while others get along with storing all solutions and extracting the non-dominated subset after the optimization is finished (offline case). The storage of millions of intermediate points does not pose a problem on today’s computers, and even full non-dominated sorting is feasible on huge collections of objective vectors with suitable algorithms, i.e., with $O(n \log(n)^{m-1})$ operations and $O(nm)$ memory [3, 2]. Hence, we consider the offline case a solved problem, at least for moderate values of $m$.

In this paper we focus on the online case where the non-dominated set must be incrementally updated after each objective function evaluation. We aim for a fast procedure that checks whether a candidate point $x$ is dominated by any point in the archive or not. If $x$ is non-dominated then it is added to the archive. In addition, any points in $A$ that happen to be dominated by $x$ are removed. We are not aware of any prior work on solving this task efficiently.

Of course it is always possible to perform a full non-dominated sort after adding a new point, however, this proceeding is computationally wasteful. Even the $O(nm)$ “brute-force” method of comparing the new candidate to each point in the archive scales better. In the sequel we aim to improve on this baseline.

2 Definitions and Notation

**Dominance Order** The objectives are collected in the vector-valued objective function $f : X \rightarrow \mathbb{R}^m$, $f(x) = (f_1(x), \ldots, f_m(x))$. For objective vectors $y, y' \in \mathbb{R}^m$ we define the Pareto dominance relation

\[
\begin{align*}
y \preceq y' & \iff y_k \leq y'_k \text{ for all } k \in \{1, \ldots, m\}, \\
y \prec y' & \iff y \preceq y' \text{ and } y \not= y' .
\end{align*}
\]

This relation defines a partial order on $\mathbb{R}^m$, incomparable values $y, y'$ fulfilling $y \not\preceq y'$ and $y' \not\preceq y$ remain. The relation is pulled back to the search space $X$ by the definition $x \preceq x'$ iff $f(x) \preceq f(x')$.

**Pareto front and Pareto set** Let $Y = \{f(x) \mid x \in X\} = f(X) \subset \mathbb{R}^m$ denote the image of the objective function (also called the attainable objective space). The Pareto front is defined as the set of objective values that are optimal w.r.t. Pareto dominance, i.e., the set of non-dominated objective vectors

\[
Y^* = \left\{ y \in Y \mid \nexists y' \in Y : y' \prec y \right\} .
\]

The Pareto set is $X^* = f^{-1}(Y^*)$. 

2
Non-dominated points In the sequel we will deal with objective vectors, not with actual search points. In this paper we restrict ourselves to maintaining only a single search point for each non-dominated objective vector, although it is possible to observe any number of search points of equal quality. In other words, we aim to approximate the Pareto front, not necessarily the full Pareto set.

For our purposes, let $y^{(1)}, \ldots, y^{(n)} \in \mathbb{R}^m$ denote the set of all known non-dominated objective vectors stored in the archive at some point, with $n$ denoting the current cardinality of the archive. The objective vector of the new candidate $x$ is denoted by $y = f(x) \in \mathbb{R}^m$.

3 Algorithms

We first present a trivial baseline method and a highly efficient alternative for the special case of $m = 2$ objectives. Then we turn to our core contribution, a rather efficient method for the general case of $m \geq 3$ objectives.

3.1 Baseline Method

The naive “brute force” baseline method stores all non-dominated points in a flat (linear memory) vector or (double linked) list. The insert operation loops over the archive. It compares $y$ to each $y^{(k)}$ w.r.t. dominance ($O(m)$ operations, with four possible outcomes: $y = y^{(k)}$, $y \prec y^{(k)}$, $y^{(k)} \prec y$, or $y$ and $y^{(k)}$ are incomparable). If $y^{(k)} \preceq y$ then the procedure returns immediately. If $y \prec y^{(k)}$ then $y^{(k)}$ is removed. Each removal operation takes constant time (independent of $n$) by overwriting the dominated point with the last point in the data structure ($O(1)$ operations for the list, $O(m)$ operations for the vector by overwriting $y^{(k)}$, which is then removed). Finally $y$ is appended to the end of the list or vector (amortized constant time for vectors with a doubling strategy, but even a possible $O(nm)$ relocation of the memory block does affect the analysis). Hence the overall runtime of the brute force method is $O(nm)$ operations. It is significantly faster (in expectation over a presumed random order of the archive) only if $y$ is dominated by more than $O(1)$ points from the archive. Advantages of this algorithm are the trivial implementation and, in case of a linear memory array, optimal use of the processor cache. The method can be considered an online variant of Deb’s fast non-dominated sorting algorithm [1], however, restricted to the first non-dominance rank.

3.2 Special Case of Two Objectives

For the bi-objective case the Pareto front is well known to obey a special structure that can be exploited for our purpose: we keep the archive sorted w.r.t. the first objective $f_1$ in ascending order, which automatically keeps it sorted in descending order w.r.t. to the second objective $f_2$. Given $y$ we search for the
indices

\[ \ell = \arg \max \left\{ j \mid y_1(j) \leq y_1 \right\} \quad \text{and} \quad r = \arg \min \left\{ j \mid y_1(j) \geq y_1 \right\} \]

of \( y \)'s potential "left" and "right" neighbors on the front. If the \( \arg \max \) is undefined because the set is empty then \( y \) is non-dominated and we set \( r = 0 \) for further processing. If \( y \) is weakly dominated by any archived point then it is also weakly dominated by \( y_\ell \), which is the case exactly if \( y_2(\ell) \leq y_2 \). Hence once \( \ell \) is found the check is fast. If \( y \) happens to dominate any archived points, then these are of the form \( \{ y(r), y(r+1), \ldots, y(r+s) \} \). Hence, given \( r \) it is easy to identify the dominated points, which are then removed from the archive.

Self-balancing trees such as AVL-trees and red-black-trees are suitable data structures for performing these operators quickly. The search for \( \ell \) and \( r \) and the insert operation require \( O(\log(n)) \) operations each, and so does the removal of a point. In an amortized analysis there can be at most one removal per insert operation, hence the overall complexity remains as low as \( O(\log(n)) \), which is far better than the \( O(n) \) brute-force method (note that here \( m \) does not really enter the complexity since it is fixed to the constant \( m = 2 \)). With this archive, the cumulated cost of \( n \) iterative updates is not more costly than a full non-dominated sort post-processing step by sweeping objectives.

3.3 General Case of \( m \geq 3 \) Objectives

For more than two objectives non-dominated sets obey far less structure than in the bi-objective case. Hence it is unclear whether logarithmic runtime is achievable. However, it should be possible to surpass the linear complexity of the brute-force baseline algorithm. In the following we propose an algorithm that achieves this goal.

We propose to store the archived non-dominated points in a binary space partitioning (BSP) tree. Each interior node of the tree is a data structure keeping track of its left and right child nodes \( \ell \) and \( r \), an objective index \( j \in \{1, \ldots, m\} \), and a threshold \( \theta \in \mathbb{R} \). For a node \( N \) we refer to these data fields with the dot-notation, i.e., \( N.r \) refers to the right child node of \( N \). With \( p(N) \) we refer to the parent node, i.e., \( p(N.\ell) = N \) and \( p(N.r) = N \), while \( p(R) \) is undefined. We write \( p^k(N) = p(p(N)) \), and \( p^k(N) \) for the \( k \)-th ancestor of \( N \). The objective space is partitioned as follows. Let \( \Delta(N) \) denote the subspace represented by the node \( N \). Let \( R \) denote the root node, then we have \( \Delta(R) = \mathbb{R}^m \). We recursively define \( \Delta(N.\ell) = \{ y \in \Delta(N) \mid y_{N,j} < N.\theta \} \) and \( \Delta(N.r) = \{ y \in \Delta(N) \mid y_{N,j} \geq N.\theta \} \). Each leaf node holds a set \( P \) of objective vectors limited in cardinality by a predefined bucket size \( B \in \mathbb{N} \). In addition, each node (interior or leaf) keeps track of the number \( n \) of objective vectors in the subspace it represents.

The empty archive is represented by the root node \( R \) being a leaf node holding no objective vectors: \( R.n = 0 \) and \( R.P = \emptyset \). Newly generated objective vectors \( y \) are added one by one to the archive with the \texttt{Process} algorithm laid out in algorithm [1].
Algorithm 1: Process($y$)

\[
s \leftarrow \text{CheckDominance}(R, y, 0, \emptyset)
\]

if $s \geq 0$ then
    $N \leftarrow R$
    while $N$ is interior node do
        $N.n \leftarrow N.n + 1$
        if $y_{N,j} < N.\theta$ then $N \leftarrow N.\ell$ else $N \leftarrow N.r$
    end
    $P \leftarrow N.P \cup \{y\}$
    if $|P| \leq B$ then $N.P \leftarrow P$; $N.n \leftarrow |P|$ else
        make $N$ an interior node and create new leaf nodes as children
        select $N.j$ and $N.\theta$ (see text)
        $N.\ell.P \leftarrow \{p \in P \mid p_{N,j} < N.\theta\}$; $N.\ell.n \leftarrow |N.\ell.P|
        $N.r.P \leftarrow \{p \in P \mid p_{N,j} \geq N.\theta\}$; $N.r.n \leftarrow |N.r.P|
    end
end

3.3.1 Processing a Candidate Point.

Process calls the recursive CheckDominance algorithm (algorithm 2), which returns the number of points in the archive dominated by $y$, or $-1$ if $y$ is dominated by at least one point in the archive. The procedure also removes all points dominated by $y$. If $y$ is non-dominated (i.e., the return value is non-negative), then $y$ is inserted into the tree by descending into the leaf node $N$ fulfilling $y \in \Delta(N)$. If the insertion would exceed the bucket size ($|P| > B$, with $P = N.P \cup \{y\}$), then the node is split. To this end we need to select an objective index $j$ and a threshold $\theta$. The only hard constraint on the choice of $j$ is that $V_j = \{y_j \mid y \in P\}$ must contain at least two values, so that splitting the space in between yields two leaf nodes holding at most $B$ objective vectors. However, for reasons that will become clear in the next section we aim to select different objectives if possible when descending the tree. For $j \in \{1, \ldots, m\}$ we define the distance $d_j = \min\{p^k(N), j = j \mid k \in N\}$ of the node $N$ in its chain of ancestors from the next node splitting at objective $j$. We set $d_j = \infty$ if the root is reached. Before observing objective $j$. We chose $j \in \arg \max_j \{d_j \mid |V_j| > 1\}$. For the selection of $\theta$ we have to take into account that multiple objective vectors may agree in their $j$-th component. We select $\theta$ as the midpoint of two values from $V_j$ so that the split balances the leaves as well as possible. For even $B$ (and hence an uneven number of objective vectors) we prefer more points in the left leaf.

3.3.2 Recursive Check of the Dominance Relation.

The CheckDominance algorithm is the core of our method. It takes a node $N$, the candidate point $y$, and two index sets $B$ and $W$ as input. It returns
Algorithm 2: CheckDominance($N, y, B, W$)

$k \leftarrow 0$

if $N$ is leaf node then
    foreach $p \in N.P$ do
        if $p \preceq y$ then return $-1$
        if $y \prec p$ then
            $N.P \leftarrow N.P \setminus \{p\}$
            $N.n \leftarrow N.n - 1$
            $k \leftarrow k + 1$
        end
    end
end

if $N$ is interior node then
    if $y.N.j < N.\theta$ then $B' \leftarrow B \cup \{N.j\}$; $W' \leftarrow W$
else $B' \leftarrow B$; $W' \leftarrow W \cup \{N.j\}$
    if $|W'| = m$ then return $-1$
    else if $|B'| = m$ then $k \leftarrow k + N.r.n$; $N.r.n \leftarrow 0$
    else
        if $W' = \emptyset \lor B = \emptyset$ then
            $s \leftarrow$ CheckDominance($N.\ell, y, B, W'$)
            if $s < 0$ then return $-1$
            $k \leftarrow k + s$
        end
        if $B' = \emptyset \lor W = \emptyset$ then
            $s \leftarrow$ CheckDominance($N.r, y, B', W$)
            if $s < 0$ then return $-1$
            $k \leftarrow k + s$
        end
    end
end

$N.n \leftarrow N.n - k$

if $N.\ell.n > 0$ and $N.r.n = 0$ then overwrite $N$ with $N.\ell$
if $N.\ell.n = 0$ and $N.r.n > 0$ then overwrite $N$ with $N.r$
end

return $k$
the number of points in the subtree strictly dominated by $y$, and $-1$ if $y$ is weakly dominated by any point in the space cell represented by the subtree. The algorithm furthermore removes all dominated points from the subtree.

When faced with a leaf node it operates similar to the brute force algorithm. However, for interior nodes it can do better. To this end, note that the set $\Delta(N) \subset \mathbb{R}^m$ can be written in the form

$$\Delta(N) = \Delta_1(N) \times \cdots \times \Delta_m(N)$$

where $\Delta_j(N) \subset \mathbb{R}$ is the projection of $\Delta(N)$ to the $j$-th objective. Since the reals are totally ordered, we distinguish the cases $y_j < \Delta_j(N)$, $y_j \in \Delta_j(N)$, and $y_j > \Delta_j(N)$. Note that $y_j < \Delta_j(N)$ for all $j$ implies that $y$ dominates the whole space cell $\Delta(N)$, similarly $y_j > \Delta_j(N)$ implies that $y$ is dominated by any point in $\Delta(N)$. If there exist $i,j$ so that $y_i < \Delta_i(N)$ and $y_j > \Delta_j(N)$ then $y$ is incomparable to all points in $\Delta(N)$.

The algorithm descends the tree by recursively invoking itself on the left and right child nodes, but only if necessary. The recursion is necessary only if the comparisons represented by the recursive calls up the call stack do not determine the dominance relation between $y$ and $\Delta(N)$ yet. At the root node we know that it holds $y_j \in \Delta_j(R)$ for all $j \in \{1, \ldots, m\}$. Hence we have either $y_{R.j} \in \Delta_{R.\ell}$ or $y_{R.j} \in \Delta_{R.r}$, and hence either $y_{R.j} > \Delta_{R.\ell}$ or $y_{R.j} < \Delta_{R.r}$. The sets $B$ and $W$ keep track of the objectives in which $y$ is better or worse than $\Delta(N)$. The two sets are apparently disjoint. If they are non-empty at the same time then the candidate point and the space cell are incomparable, hence the recursion can be stopped. If $W$ equals the full set $\{1, \ldots, m\}$ then $y$ is dominated by the space cell $N$. The mere existence of the node guarantees that it contains at least one point, so we can conclude that $y$ is dominated. If on the other hand $B = \{1, \ldots, m\}$ then all points in $N$ are dominated and hence removed. If the algorithm finds one of its child nodes empty after the recursion then it recovers a binary tree by replacing the current node with the remaining child. No action is required if both child nodes are empty since this implies $N.n = 0$, and the node will be removed further up in the tree.

### 3.3.3 Balancing the Tree.

In contrast to the bi-objective case it is unclear how to balance the tree at low computational cost. This is not a severe problem for objective vectors drawn i.i.d. from a fixed distribution. This situation is fulfilled in good enough approximation when performing many optimization runs. However, for a single (potentially long) run of an optimizer we can expect a systematic shift from low-quality early objective vectors towards better and better solutions over time. Hence most points proposed late during the run will tend to end up in the left child of a node, the split point of which was determined early on. We counter this effect by introducing a balancing mechanism as follows. If the quotient $\frac{N.\ell.n}{N.r.n}$ raises above a threshold $z$ or falls below $\frac{1}{z}$ then the smaller child node is removed, the larger one replaces its parent, and the points represented by the
smaller node are inserted. Although this process is computationally costly, it can pay off in the long run in case of highly unbalanced trees.

4 Analysis

In this section we analyze on the complexity of the BSP tree based archive algorithm. We start with the storage requirements. When storing \( n \) non-dominated points, in the worst case there are \( n \) distinct leaf nodes, and hence \( n - 1 \) interior nodes in the tree, requiring \( \mathcal{O}(n) \) memory in addition to the unavoidable requirement of \( \mathcal{O}(n m) \) for storing the non-dominated objective vectors. Hence the added memory footprint due to the BSP tree is unproblematic. In our implementation the overhead of a tree node is 56 bytes on a 64bit system, which makes it feasible to store millions of points in RAM.

The analysis of the runtime complexity is more involved. Since archiving small numbers of points is uncritical, we focus on the case of large \( n \), which is well described by an average case amortized analysis. For the analysis we drop the rather heuristic balancing mechanism. For simplicity we set the bucket size to \( B = 1 \) and assume a perfectly balanced tree depth \( \log_2(n) \) (while the depth of a random tree is typically of order \( 2 \log_2(n) \)). The strongest technical assumption we make for the analysis is that objective vectors are sampled i.i.d. from a static distribution. Let \( P \) denote a probability distribution on \( \mathbb{R}^m \) so that for two random objective vectors \( a, b \sim P \) the events \( a \preceq b \) and \( \exists j \in \{1, \ldots, m\} : a_j = b_j \) have probability zero. We consider a BSP archive constructed by inserting \( n \) points sampled i.i.d. from \( P \). Then we are interested in bounding the expected runtime \( T \) required for processing a candidate point \( y \sim P \).

For a node \( N \) representing the space cell \( \Delta(N) \) we define its order w.r.t. the candidate point \( y \) as \( k = |\{j : y_j \notin \Delta_j(N)\}| \). We call a node comparable to \( y \) if its space cell contains at least one comparable point (w.r.t. the Pareto dominance relation). All incomparable cells are skipped by the algorithm, hence the runtime is proportional to the number of comparable nodes. A node is incomparable to \( y \) if there exist \( j_1 \) and \( j_2 \) such that \( y_{j_1} < \Delta_{j_1}(N) \) and \( y_{j_2} > \Delta_{j_2}(N) \). Furthermore, cells dominated by \( y \) (\( y_j < \Delta_j(N) \) for all \( j \)) don’t need to be visited, actually, encountering such a cell stops the algorithm immediately. Similarly, nodes dominated by \( y \) can be ignored in an amortized analysis since on average only one point can be removed per insertion, and the cost of a removal is as low as \( \mathcal{O}(\log(n)) \). Hence all nodes of order \( m \) can be ignored for the analysis.

In the following we denote the probability for a random node at depth \( d \) below the root node (the root has depth 0) to have order \( k \in \{0, \ldots, m\} \) with \( Q_d(k) \). We have \( Q_0(0) = 1 \) and \( Q_d(k) = 0 \) for \( k > d \).

The following theorem provides a lower bound on the runtime in the best case, namely when each split of the BSP tree induces the same chance to yield an incomparable child node.

**Theorem 1.** Assume that when traversing from root to leaf no two space splits are along the same objective. Then we have \( T \in \Omega(n^{\log_2(3/2)}) \).
Proof. The prerequisite implies \( m \geq d \) and hence \( m \geq \log_2(n) \). Since objective vectors are sampled i.i.d., the probability of the candidate point to be covered by the left or right sub-tree is 50% at each node. This corresponds to a 50% chance to increment the order \( k \) when descending an edge of the tree, hence \( k \) follows a binomial distribution. We obtain \( Q_d(k) = 2^{-d} \cdot \binom{d}{k} \) for \( k \leq d \). For given \( k \), the chance of a node to be comparable to the candidate point is \( \min\{1, 2^{1-k}\} \), since for this to happen all \( k \) decisions (descending left or right in the tree) must coincide. Hence among the \( 2^d \) nodes at depth \( d \) an expected number of

\[
1 + \sum_{k=1}^{d} 2^{1-k} \cdot \binom{d}{k} = 2^{d \log_2(3/2)+1} - 1
\]

nodes is comparable to the candidate point. The statement follows by summing over all depths \( d \leq \log_2(n) \).

Under the milder (and actually pessimistic) assumption of random split objectives we obtain a sub-linear upper bound.

**Theorem 2.** Assume that each node splits the space long an objective \( j \) drawn uniformly at random from \( \{1, \ldots, m\} \). Then we have \( T \in o(nm) \).

Proof. In this case \( Q \) is described by the following recursive formulas for \( d > 0 \):

\[
Q_d(0) = \frac{1}{2}Q_{d-1}(0)
\]

\[
Q_d(k) = \frac{1}{2} \left[ Q_{d-1}(k-1) \cdot \frac{m-k+1}{m} + Q_{d-1}(k) \cdot \frac{m+k}{m} \right]
\]

We obtain \( \lim_{d \to \infty} Q_d(m) = 1 \), hence a only \( o(n) \) nodes are of order at most \( m-1 \), and only a subset of these must be visited. Since processing a leaf node requires \( O(m) \) operations in general, we arrive at \( T \in o(nm) \). □

5 Experimental Evaluation

All three archives were implemented efficiently in C++. Here we investigate how fast the archives operate in practice, how their runtimes scale to large \( n \) and \( m \), and how their practical performance relates to our analysis.

We constructed archives from sequences of \( D \) dominated (\( d = \frac{1}{100} \)) and \( N \) non-dominated (\( d = 0 \)) normally distributed objective vectors according to

\[
y^{(k)} \sim \mathcal{N} \left( \frac{d(N+D)}{k} \mathbf{1}, \mathbf{I} - \frac{1}{m} \mathbf{1} \mathbf{1}^T \right),
\]

where \( \mathbf{I} \in \mathbb{R}^{m \times m} \) denotes the identity matrix and \( \mathbf{1} = (1, \ldots, 1)^T \in \mathbb{R}^m \) is the vector of all ones. The distribution has unit variance in the space orthogonal

[http://www.ini.rub.de/PEOPLE/glasmtbl/code/ParetoArchive/](http://www.ini.rub.de/PEOPLE/glasmtbl/code/ParetoArchive/)
Figure 1: Processing time per objective vector for $m = 2$ objectives (top) and $m = 3$ objectives (bottom), for a static distribution ($c = 1$, left) and solutions improving over time ($c = 1.1$, right), for systematically varied numbers of overall points ($n = N + D$) and non-dominated points ($N$) in the range $2^8$ to $2^{18}$. 
to $1$. The parameter $d \geq 0$ controls the systematic improvement of points over time. At position $k$ a dominated point was sampled with probability $c \frac{D'}{N' + D'}$, where $D'$ and $N'$ denote the number of remaining dominated and non-dominated points to be placed into the sequence. Hence for $c > 1$ there is a preference for observing more dominated points early on in the sequence, while for $c = 1$ there is not.

We obtained robust results with a bucket size of $B = 20$ and a threshold of $z = 6$ for tree balancing. The large bucket size yields well balanced trees, and the high threshold avoids explicit re-balancing most of the time.

Figure 1 displays the average processing times of the different archives over sequences with varying $m$, $n = N + D$, $N$, and $c$. It is no surprise that for $m = 2$ the specialized bi-objective archive performs clearly best. For $m = 3$ the BSP tree is in all cases superior to the baseline. Figure 2 shows the empirical scaling of the archives for a setting close to the preconditions of the theoretical analysis: $m = 3$, $D = 0$, $c = 1$. The actual scaling is very close to the lower bound of $n \log_2 \left( \frac{3}{2} \right) \approx n^{0.585}$ from theorem 1. The algorithm scales very gracefully to large numbers of objectives. For the range $3 \leq m \leq 50$ we observe sub-linear scaling (not shown).

6 Conclusion

We have presented the first iterative algorithm for updating an archive of Pareto optimal points with runtime sub-linear in the archive size $n$ for the general case of $m \geq 3$ objectives. The method is shown to consistently outperforms the baseline for medium to large scale archives. Its runtime is shown empirically to scale roughly like $n \log_2(3/2)$, which coincides with our lower bound.

References

[1] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In Parallel Problem Solving from Nature (PPSN) VI, pages 849–858. Springer, 2000.

[2] F.-A. Fortin, S. Grenier, and M. Parizeau. Generalizing the Improved Run-Time Complexity Algorithm for Non-Dominated Sorting. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2013.
[3] M. T. Jensen. Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation, 7*(5):503–515, 2003.

[4] M. López-Ibáñez, J. D Knowles, and M. Laumanns. On Sequential Online Archiving of Objective Vectors. In *Evolutionary Multi-objective Optimization (EMO)*, pages 46–60. Springer, 2011.