A Fast Incremental BSP Tree Archive for Non-dominated Points

Tobias Glasmachers

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

Abstract. Maintaining an archive of all non-dominated points is a standard task in multi-objective optimization. Sometimes it is sufficient 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. 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, as well as graceful scaling to large numbers of objectives.

1 Introduction

Given $m \geq 2$ objective functions $f_1, \ldots, f_m : X \to \mathbb{R}$, 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 is often huge or even infinite. In a black-box setting it is best described 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.

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 multi-objective optimization algorithms (MOEAs), 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, called dynamic problem in [16]), while others get along with storing all solutions and extracting the non-dominated subset after the optimization is finished (offline or static 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. Hence, we consider the offline case a solved problem, at least for moderate values of m.

In this paper we are interested in archives with the following properties:

- Online updates of the set of non-dominated points shall be efficient. In particular, it shall be feasible to process long sequences of candidate solutions one by one, i.e., as soon as they are proposed and evaluated by an iterative optimization algorithm.
- The archive shall not contain dominated points, even if these points were non-dominated at an earlier stage. I.e., points shall be removed as soon as they are dominated by a newly inserted point.
- The full set of non-dominated points shall be stored, not an approximation set of a-priori bounded cardinality.
- Ideally, the algorithm should scale well not only to large archives, but also to a large number of objectives.

These prerequisites imply the following processing steps. First it is checked 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. All of these steps should be as efficient as possible. At the very least, they should be faster than the $\mathcal{O}(nm)$ linear "brute force" search through the archive.

In this paper we propose an algorithm achieving this goal. It is based on a binary space partitioning tree (BSP tree). Its performance is demonstrated empirically, for large archives (up to $n = 2^{18}$ non-dominated points) and large numbers of objectives (up to $m = 50$), as well as analytically in the form of asymptotic (lower and upper) runtime bounds. We provide C++ source code for the proposed archive.[1] It is based on the implementation from the (non-public) code base of the black box optimization competition (BBComp)[2], where it is applied for online monitoring of the optimization progress.

In the remainder of this paper we first define the problem and fix our notation. After reviewing related work we present the proposed archiving algorithm. We derive asymptotic lower and upper bounds on its runtime and assess its practical performance empirically on a variety of tasks.

2 Definitions and Notation

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

$$\begin{aligned} y \preceq y' \quad &\Leftrightarrow \quad y_k \leq y'_k \text{ for all } k \in \{1, \ldots, m\} \ , \\ y \prec y' \quad &\Leftrightarrow \quad y \preceq y' \text{ and } y \neq y' \ . \end{aligned}$$

[1] http://www.ini.rub.de/PEOPLE/glasmtbl/code/ParetoArchive/
[2] http://bbcomp.ini.rub.de

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 $Y = \{f(x) \,|\, x \in X\} = f(X) \subset \mathbb{R}^m$ 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

$$Y^* = \left\{ y \in Y \;\middle|\; \nexists\, y' \in Y : y' \prec y \right\} .$$

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

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, represented by a minimal complete 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 Related Work

Fixed Memory Approximations. The growth of the set of non-dominated points is in general unbounded. Still, many MOEAs use their (fixed size) population as a rough approximation of the Pareto front. Even if an external archive is employed, computer memory is finite, which sets limits to the approach of archiving an unbounded number of non-dominated points. Therefore limited memory archives (e.g., based on clustering) were subject of intense study [3, 11]. Their main disadvantage is the limited precision of their Pareto front representation, which can even lead to oscillatory behavior when used for optimization.

Memory Consumption. Nowadays even commodity PCs are equipped with gigabytes of RAM, enabling the storage of millions of objective vectors in memory.[3] This makes it feasible for MOEAs to maintain all non-dominated points in the population, as proposed by Krause et al. [9]. Although the algorithm is limited to $m = 2$ objectives, it demonstrates successfully that even on standard hardware memory is no longer a limiting factor for archiving of all known non-dominated points.

[3] This is usually sufficient for the needs of evolutionary optimization. In data base query problems larger sets must be processed. Hence in some applications memory consumption is still a concern.

Offline Case. Efficient algorithms are known for the offline case of obtaining the Pareto optimal subset from a set of points, i.e., at the end of an optimization run. This problem was first addressed in [10]. Even full non-dominated sorting (delivering not only the Pareto optimal subset, but a partitioning into disjoint fronts) can be achieved in only $\mathcal{O}(n \log(n)^{m-1})$ operations and $\mathcal{O}(nm)$ memory [7, 4]. However, computing the full Pareto front from scratch after every update is wasteful, and hence specialized updating algorithms are needed for the online case.

Skylines. A closely related but slightly different problem is found in data base queries: so-called "skyline queries" ask for a skyline of records, which is just a different terminology for the non-dominated (Pareto optimal) subset with respect to a specified subset of fields. While the offline case is the most important one, online algorithms with good anytime performance are of relevance. The requirements differ from the problem under study in a decisive point: in the data base setting the full set of records is available from the start. The process is limited only by computational and memory constraints, but not by the sequential nature of proposing points in an iterative optimizer. Efficient algorithms (e.g., based on nearest neighbor queries) exist for the skyline problem [8, 14].

Search Trees. Tree data structures of various sorts are attractive for tackling our problem. They offer a natural problem decomposition, i.e., adding a single point usually affects only a small neighborhood of the current front, possibly represented by a small sub-tree. The dominance decision tree (DDT) data structure was proposed specifically for the problem at hand [16]. It works well if the fraction of non-dominated points and hence the archive is small [16]. Dominated and non-dominated trees [3] represent a different approach. None of these methods achieve a significant reduction of the computational complexity of the problem. Quad trees can reduce the computational effort for large archives [13]. A clear disadvantage of quad trees generalized to $m > 2$ objectives is that partitioning a space cell results in 2^m sub-cells, which limits the approach to small numbers m of objectives. A different approach is based on modeling X^* instead of Y^* [12] with a binary decision diagram. This can work for some problems, however, it cannot be expected to be efficient in general.

4 Algorithms

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

4.1 Baseline Method

The naive "brute force" baseline method stores all non-dominated points in a flat linear memory vector (dynamically extensible array) or linked list. The

insert operation loops over the archive. It compares the candidate vector y to each stored $y^{(k)}$ w.r.t. dominance. This takes $\mathcal{O}(m)$ operations per point in the archive. If $y^{(k)} \preceq y$ then the point does not need to be archived and the procedure stops. If $y \prec y^{(k)}$ then $y^{(k)}$ is removed. In the list, removal takes $O(1)$ operations. In the vector we overwrite the dominated point with the last point, which is then removed ($\mathcal{O}(m)$ operations). 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 $\mathcal{O}(nm)$ relocation of the memory block does not affect the analysis). Hence, in the worst case the brute force method performs $\Theta(nm)$ operations. It is significantly faster (in expectation over a presumed random order of the archive) only if y is dominated by more than $\mathcal{O}(1)$ 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.

4.2 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

$$\ell = \arg\max \left\{ j \,\middle|\, y_1^{(j)} \leq y_1 \right\} \qquad \text{and} \qquad r = \arg\min \left\{ j \,\middle|\, y_1^{(j)} \geq y_1 \right\}$$

of y's potential "left" and "right" neighbors on the front. If the $\arg\min$ 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 ℓ is found the check is fast. If y happens to dominate any archived points, then these are of the form $\{y^{(r)}, y^{(r+1)}, \ldots, y^{(r+s)}\}$, for some 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 ℓ and r and the insert operation require $\mathcal{O}(\log(n))$ 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 $\mathcal{O}(\log(n))$, which is far better than the $\mathcal{O}(n)$ 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 effort of n iterative updates equals (up to a constant) that of the offline algorithm [10].

4.3 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 which speed-up is achievable. However,

it should be possible to surpass the linear complexity of the brute-force baseline. In the following we present an algorithm that achieves this goal.

We propose to store the archived non-dominated points in a binary space partitioning (BSP) tree. Among the different variants a simple k-d tree [1] seem to be best suited. This choice was briefly discussed and quickly dismissed in [3] and [16]. Nonetheless we propose an archiving algorithm based on a k-d tree, for the following reason. We expect most new non-dominated points to dominate only a local patch of the current front. Therefore it should be possible to limit dominance comparisons to few archived points close to the candidate point. Space partitioning is a natural approach to exploiting this locality.

Let R denote the root node. Each interior node of the tree is a data structure keeping track of its left and right child nodes ℓ 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 $p^2(N) = p(p(N))$, 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. We have $\Delta(R) = \mathbb{R}^m$. We recursively define $\Delta(N.\ell) = \{y \in \Delta(N) \,|\, y_{N.j} < N.\theta\}$ and $\Delta(N.r) = \{y \in \Delta(N) \,|\, y_{N.j} \geq N.\theta\}$ for given j and θ.

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.

Newly generated objective vectors y are added one by one to the archive with the `Process` algorithm laid out in algorithm 1.

Algorithm 1: Process(y)

```
s ← CheckDominance(R, y, 0, ∅)
if s ≥ 0 then
    N ← R
    while N is interior node do
        N.n ← N.n + 1
        if y_{N.j} < N.θ then N ← N.ℓ else N ← N.r
    end
    P ← N.P ∪ {y}
    if |P| ≤ B then N.P ← P; N.n ← |P|
    else
        make N an interior node and create new leaf nodes as children
        select N.j and N.θ (see text)
        N.ℓ.P ← {p ∈ P | p_{N.j} < N.θ}; N.ℓ.n ← |N.ℓ.P|
        N.r.P ← {p ∈ P | p_{N.j} ≥ N.θ}; N.r.n ← |N.r.P|
    end
end
```

Algorithm 2: CheckDominance(N, y, B, W)

```
k ← 0
if N is leaf node then
    foreach p ∈ N.P do
        if p ⪯ y then return −1
        if y ≺ p then
            N.P ← N.P \ {p}
            N.n ← N.n − 1
            k ← k + 1
        end
    end
end
if N is interior node then
    if y_{N.j} < N.θ then B′ ← B ∪ {N.j}; W′ ← W
    else B′ ← B; W′ ← W ∪ {N.j}
    if |W′| = m then return −1
    else if |B′| = m then k ← k + N.r.n; N.r.n ← 0
    else
        if W′ = ∅ ∨ B = ∅ then
            s ← CheckDominance(N.ℓ, y, B, W′)
            if s < 0 then return −1
            k ← k + s
        end
        if B′ = ∅ ∨ W = ∅ then
            s ← CheckDominance(N.r, y, B′, W)
            if s < 0 then return −1
            k ← k + s
        end
    end
    N.n ← N.n − k
    if N.ℓ.n > 0 and N.r.n = 0 then overwrite N with N.ℓ
    if N.ℓ.n = 0 and N.r.n > 0 then overwrite N with N.r
end
return k
```

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 θ. The only hard constraint on the choice of j is that $V_j = \{y_j \,|\, y \in P\}$ 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 prefer to select different objectives if possible when descending the tree. For

$j \in \{1, \ldots, m\}$ we define the distance $d_j = \min\{k \in \mathbb{N} \,|\, p^k(N).j = j\}$ 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 node is reached before observing objective j. We chose $j \in \arg\max_j \left\{d_j \,\middle|\, |V_j| > 1\right\}$. For the selection of θ we have to take into account that multiple objective vectors may agree in their j-th component. We select θ 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.

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 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 are 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.

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 short 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 $\frac{N.\ell.n}{N.r.n}$ rises 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.

5 Analysis

In this section we analyze 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 $\mathcal{O}(nm)$ 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 of depth $\log_2(n)$. Although optimistic, this assumption is not too unrealistic: note that the depth of a random tree is typically of order $2\log_2(n)$ [15].

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 dominating 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 $k \in \{0, \dots, m\}$ 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. *Under the conditions stated above, 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 $1/2$ 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

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

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. *Under the conditions stated above, assume that each node splits the space along an objective j drawn uniformly at random from $\{1, \dots, m\}$. Then we have* $T \in o(nm)$.

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

$$\begin{aligned} Q_d(0) &= \frac{1}{2} Q_{d-1}(0) \\ Q_d(k) &= \frac{1}{2} \left[Q_{d-1}(k-1) \cdot \frac{m-k+1}{m} + Q_{d-1}(k) \cdot \frac{m+k}{m} \right] \end{aligned}$$

We obtain $\lim_{d \to \infty} Q_d(m) = 1$, hence 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 $\mathcal{O}(m)$ operations in general, we arrive at $T \in o(nm)$. ∎

6 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.

6.1 Analytic Problems

A first series of tests was performed with sequences of objective vectors following controlled, analytic distributions. These tests allow us to clearly disentangle effects caused by different fractions of non-dominated points and non-stationarity due to improvement of solutions over time.

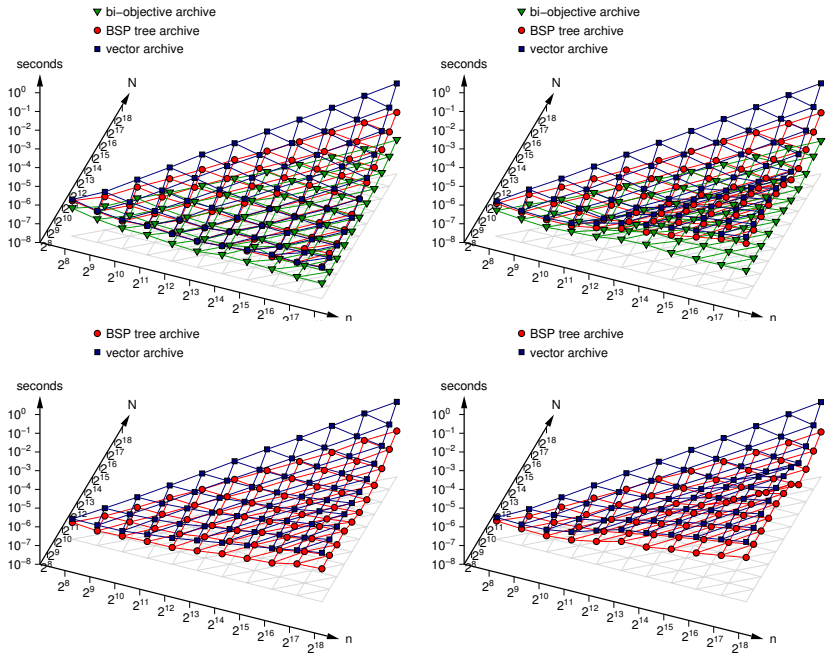

Fig. 1. Processing time per objective vector for $m = 2$ objectives (top) and $m = 3$ objectives (bottom), for a static distribution ($c = 1$, left) and solutions improving over time ($c = 1.1$, right), for systematically varied numbers of overall points ($n = N + D$) and non-dominated points (N) in the range 2^8 to 2^{18}.

We constructed archives from sequences of D dominated ($a > 0$) and N non-dominated ($a = 0$) normally distributed objective vectors according to

$$y^{(k)} \sim \mathcal{N}\left(\frac{a(N+D)}{k}\mathbf{1}, \mathbb{I} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\right) \quad ,$$

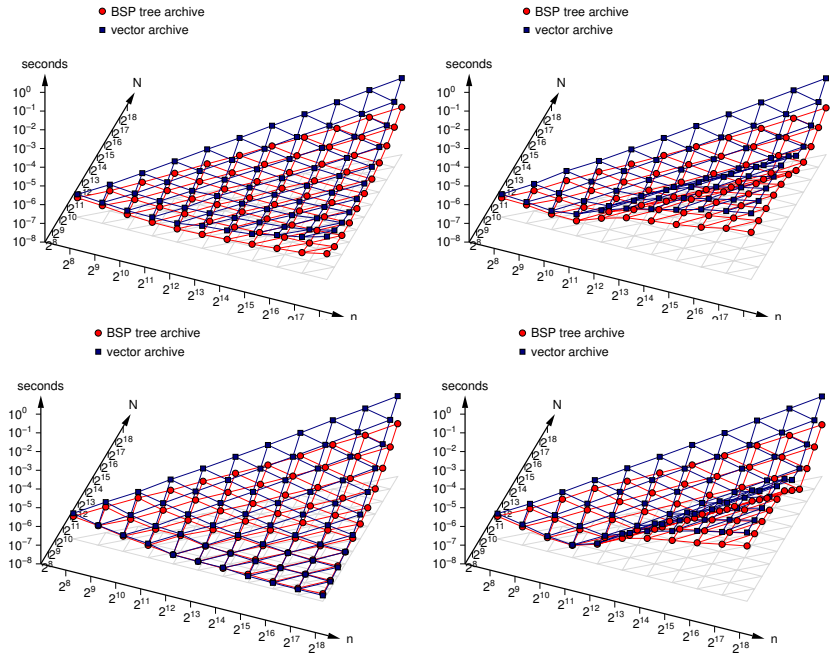

Fig. 2. Processing time per objective vector for $m = 5$ objectives (top) and $m = 10$ objectives (bottom), for a static distribution ($c = 1$, left) and solutions improving over time ($c = 1.1$, right), for systematically varied numbers of overall points ($n = N + D$) and non-dominated points (N) in the range 2^8 to 2^{18}.

where $\mathbb{I} \in \mathbb{R}^{m \times m}$ 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 subspace orthogonal to the $\mathbf{1}$ vector. The parameter $a \geq 0$ controls the systematic improvement of points over time.[4] At position k of the sequence 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.

The bucket size B and the tree re-balancing threshold z are tuning parameters of the algorithm. We propose the settings $B = 20$ and $z = 6$ as default values since they gave robust results across problems with varying characteristics in preliminary experiments. The relatively large bucket B size yields well balanced trees. Therefore a high value of the threshold z is affordable, because re-balancing is rarely needed.

Figures 1 and 2 display the average processing times of the different archives over sequences with varying m, $n = N + D$, N, and c. It is no surprise that

[4] For $a > 0$ the values $a = 0.1$, $a = 0.2$, $a = 0.5$, and $a = 1$ were used with $m = 2$, $m = 3$, $m = 5$, and $m = 10$ objectives, respectively.

for $m = 2$ the specialized bi-objective archive performs clearly best. For $m \geq 3$ the BSP tree is in all cases superior to the baseline. The vector archive is only competitive for $N \ll n$, i.e., as long as the number of non-dominated points in the archive remains small. Unsurprisingly, this is also the domain where the systematic improvement of points over time has a significant effect on archive performance. The overall effect is similar for all archive types, and in comparison to the static case the BSP archive can even increase its advantage over the linear memory archive.

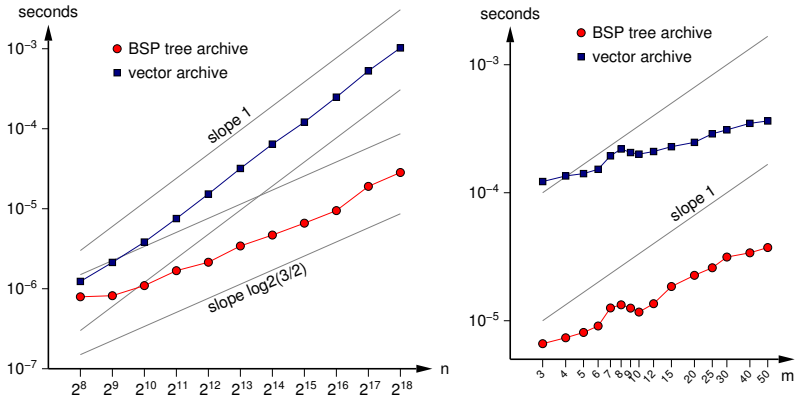

Fig. 3. Empirical scaling w.r.t. n (left) and m (right). The gray lines in the background of the log-log-plots indicate the exponents α and β of the hypothetical scaling laws $T(n) \in \Theta(n^\alpha)$ and $T(m) \in \Theta(m^\beta)$, respectively.

Figure 3 (left) 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 order $n^{\log_2(3/2)} \approx n^{0.585}$ from theorem 1. In contrast, the vector-based archive scales perfectly linear in n.

Figure 3 (right) investigates the scaling to large numbers of objectives. It plots the runtime per processing step for an archive consisting of 2^{15} non-dominated points. In contrast to Theorem 1, in practice the curve is of course not flat. The algorithm still scales gracefully to large numbers of objectives. For the range $3 \leq m \leq 50$ we observe sub-linear scaling. The baseline method (surprisingly) exhibits even slightly better scaling (while taking 10 times more time in absolute terms).

6.2 MOEA Runs

A second series of tests was performed with objective vectors generated by state-of-the-art MOEAs on established benchmark functions. We used two variants of

the multi-objective covariance matrix adaptation evolution strategy (MO-CMA-ES), namely with generational and with steady-state selection [5, 17]. We applied the implementations found in the Shark library,[5] version 3.1 [6]. The population size was set to 100, all parameters were left at their defaults. Sequences of objective vectors were generated by running the optimizers on the scalable benchmark problems DTLZ1 to DTLZ4 [2], with 30 variables, 3 objectives, and a budget of 200,000 function evaluations. The results are summarized in table 1.

MOEA	problem	N	k-d tree	vector	speed-up
generational MO-CMA-ES	DTLZ1	104,191	10.70	167.78	15.7
steady state MO-CMA-ES	DTLZ1	91,022	9.58	166.49	17.4
generational MO-CMA-ES	DTLZ2	42,153	8.15	64.71	8.0
steady state MO-CMA-ES	DTLZ2	53,814	10.67	79.58	7.4
generational MO-CMA-ES	DTLZ3	7,895	4.13	32.93	8.0
steady state MO-CMA-ES	DTLZ3	61,860	4.59	72.25	15.7
generational MO-CMA-ES	DTLZ4	11,621	2.75	13.31	4.8
steady state MO-CMA-ES	DTLZ4	23,097	3.86	27.00	7.0

Table 1. The table lists the number N of non-dominated points at the end of the optimization run, the average runtimes (in 10^{-6} seconds) for processing a single point with the BSP tree archive and the vector archive, as well as the speed-up factor of the BSP tree archive over the vector archive.

In all cases the BSP tree archive was considerably faster than the baseline. It outperformed the vector-based archive roughly by a factor of 10. The exact speed-up correlates with the number of non-dominated points. This is in line with our analysis, as well with the results for analytically controlled distributions of objective vectors. The results indicate that the tree-based archive also works well for realistic sequences of objective vectors produced by MOEAs. It demonstrates its strengths in particular if the set of non-dominated points grows large.

7 Conclusion

We have presented an algorithm for updating an archive of Pareto optimal objective vectors processed iteratively in a sequence. The data structure is based on a k-d tree for binary space partitioning. We present asymptotic lower and upper bounds on the runtime under a number of technical assumptions. We obtain runtime sub-linear in the archive size n. We demonstrate empirically that the method performs well for medium to large scale archives, where updating performance matters most. The archive is applicable to problems with arbitrary number m of objectives, including the many-objective case $m \gg 3$. Overall, these properties make the proposed algorithm suitable for online processing of non-dominated sets, e.g., in evolutionary multi-objective optimization.

[5] http://shark-ml.org

References

1. M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry*. Springer, 2000.
2. K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. In *Congress on Evolutionary Computation (CEC 2002)*, pages 825–830. IEEE, 2002.
3. J. E. Fieldsend, R. M. Everson, and S. Singh. Using Unconstrained Elite Archives for Multi-Objective Optimization. *IEEE Transactions on Evolutionary Computation*, 7(3):305–323, 2003.
4. 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.
5. C. Igel, N. Hansen, and S. Roth. Covariance matrix adaptation for multi-objective optimization. *Evolutionary Computation*, 15(1):1–28, 2007.
6. C. Igel, V. Heidrich-Meisner, and T. Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.
7. 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.
8. D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.
9. O. Krause, T. Glasmachers, N. Hansen, and C. Igel. Unbounded Population MO-CMA-ES for the Bi-Objective BBOB Test Suite. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2016.
10. H.-T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.
11. M. López-Ibáñez, J. D Knowles, and M. Laumanns. On Sequential Online Archiving of Objective Vectors. In *International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 46–60. Springer, 2011.
12. M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. Symbolic archive representation for a fast nondominance test. In *International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 111–125. Springer, 2007.
13. S. Mostaghim, J. Teich, and A. Tyagi. Comparison of data structures for storing Pareto-sets in MOEAs. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC)*, volume 1, pages 843–848. IEEE, 2002.
14. D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467–478. ACM, 2003.
15. J. M. Robson. The height of binary search trees. *Australian Computer Journal*, 11:151–153, 1979.
16. O. Schütze. A New Data Structure for the Nondominance Problem in Multi-objective Optimization. In *International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 509–518. Springer, 2003.
17. T. Voß, N. Hansen, and C. Igel. Improved step size adaptation for the MO-CMA-ES. In *12th annual conference on Genetic and Evolutionary Computation (GECCO)*, pages 487–494. ACM, 2010.