Improved Dominance Filtering for Unions and Minkowski Sums of Pareto Sets

Konstantinos Karathanasis
Department of Computer Engineering and Informatics, University of Patras, Greece
PIKEI New Technologies, Patras, Greece

Spyros Kontogiannis
Department of Computer Engineering and Informatics, University of Patras, Greece
Computer Technology Institute and Press "Diophantus", Patras, Greece

Christos Zaroliagis
Department of Computer Engineering and Informatics, University of Patras, Greece
Computer Technology Institute and Press "Diophantus", Patras, Greece

Abstract

A key task in multi-objective optimization is to compute the *Pareto frontier* (a.k.a. *Pareto subset*) P of a given d-dimensional objective space F; that is, a maximal subset $P \subseteq F$ such that every element in P is *non-dominated* (i.e., it is better in at least one criterion, against any other point) within F. This process, called *dominance-filtering*, often involves handling objective spaces derived from either the *union* or the *Minkowski sum* of two given partial objective spaces which are Pareto sets themselves, and constitutes a major bottleneck in several multi-objective optimization techniques. In this work, we introduce three new data structures, ND$^+$-trees, QND$^+$-trees and TND$^+$-trees, which are designed for efficiently indexing non-dominated objective vectors and performing dominance-checks. We also devise three new algorithms that efficiently filter out dominated objective vectors from the union or the Minkowski sum of two Pareto sets. An extensive experimental evaluation on both synthetically generated and real-world data sets reveals that our new algorithms outperform state-of-art techniques for dominance-filtering of unions and Minkowski sums of Pareto sets, and scale well w.r.t. the number of $d \geq 3$ criteria and the sets' sizes.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Multi-Objective Optimization, Multi-Dimensional Data Structures, Pareto Sets, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.59

Related Version *Full Version*: http://arxiv.org/abs/2508.20689

Funding This work was partially supported by the EU I3 Instrument under GA No 101115116 (project AMBITIOUS) and by the University of Patras under GA No 83770 (programme "MEDICUS").

1 Introduction

In multiobjective combinatorial optimization (MOCO) problems, given an implicit description (e.g., via linear constraints) of a *solution space* X and the corresponding *objective space* F with d-dimensional ($d \geq 2$) objective-value vectors of all elements in X, the goal is to compute the *Pareto frontier*, or *Pareto subset*: a maximal subset of F whose elements are non-dominated (i.e., they are better in at least one criterion, against any other point) within F. Many algorithms for MOCO problems, especially when having to work with instances of substantial sizes, rely heavily on the *dominance-filtering* subtask, aiming to efficiently combine (the Pareto frontiers of the objective spaces for) partial solution spaces and filtering out all the dominated objective-value vectors. In this work we focus on two special cases of dominance-filtering, in which the merged objective space F is created as either the *union* $A \cup B$, or the *Minkowski sum*[1] $A \oplus B$ of two Pareto sets A, B. These are the two most frequently used variants by solvers of various MOCO problems, e.g., of *decomposition* techniques for multiobjective integer programming [18], of *Pareto local search* for multiobjective set cover [15], and of *dynamic programming* methods for multiobjective shortest paths (MOSP) [16, 20, 22], multi-objective knapsack [6], multi-objective vehicle routing [19], or multi-objective network design [2, 4]. As manifested in [21], dominance-checking constitutes a major computational burden of most state-of-the-art algorithms for MOSP problems during the identification of new solutions. Hence, the development of efficient data structures and algorithms to handle dominance-filtering in unions and Minkowski sums of Pareto sets is of utmost importance in MOCO problems.

Related Work and Motivation. The literature offers a diverse collection of dominance filtering techniques. For $d = 2$ objectives, some highly efficient algorithms have been developed [9, 12]. For the more challenging case of $d \geq 3$ objectives, some general approaches have been explored [11, 13]. In dynamic settings, where solutions are not known in advance and are revealed gradually, the choice of an indexing data structure plays a crucial role in efficiently updating the Pareto frontier. Several indexing data structures for dominance checking have been proposed in the literature, such as *balanced binary search trees* [17], *ND-trees* [10, 14], and a variant of *k-d trees* [1, 3]. To the best of our knowledge, the most efficient algorithms for dominance-filtering of unions and Minkowski sums of Pareto sets for $d \geq 3$ objectives appear in [13]. These methods utilize *space-partitioning ND-trees* [10, 14], or *divide-and-conquer* strategies. Despite their effectiveness, these methods suffer from an inherent inefficiency that occurs when the input data emerge from real-world scenarios that typically contain *plateaus* (large collections of objective vectors with identical values in one or more dimensions, e.g., tolls in road networks), and/or are correlated (e.g., distance and time in road networks). In such cases, ND-trees turn out to be highly unbalanced, which results in significant time bottlenecks for the elementary operations of removing dominated elements from an ND-tree and of re-balancing the tree.

Our Contribution. This work focuses on dominance-filtering techniques for unions and Minkowski sums of Pareto sets for $d \geq 3$ optimization criteria. As our first contribution, we propose three novel data structures for indexing sets of non-dominated elements, which are custom-tailored to overcome the critical bottlenecks of the algorithms in [13]: (1) **ND$^+$-trees**, which inherit some desirable features of *k-d* trees [1] and ND-trees [10, 14]. (2) **QND$^+$-trees**, which dynamically adapt partitioning techniques when constructing the indexing tree from a given Pareto set, selecting the most suitable splitting method for each case. This ensures a *provably* balanced tree structure, leading to faster dominance-checks while also achieving dimensionality reduction, whenever this is possible. (3) **TND$^+$-trees** which are specially designed for scenarios where large *plateaus* occur that cause severe imbalances, which the TND$^+$-trees mitigate while also achieving dimensionality reduction, whenever this is possible.

Our second contribution concerns three new algorithms for dominance-filtering of unions and/or Minkowski sums of two Pareto sets for $d \geq 3$ optimization criteria: (1) `PlainNDred`, which reduces the problem's dimensionality by lexicographically sorting the elements, and eliminates the need for element removals from the data structure. (2) `PreND`, which constructs

[1] The *Minkowski sum* $A \oplus B$ contains all the component-wise additions of elements in A and B. If $A = \{(3, 5, 4), (5, 2, 1)\}$ and $B = \{(2, 1, 3), (6, 3, 2)\}$, then $A \oplus B = \{(5, 6, 7), (9, 8, 6), (7, 3, 4), (11, 5, 3)\}$.

an initial tree from a subset of the Pareto set, thereby reducing the need for frequent re-balancing, and avoids element removals. (3) `SymND`, which exploits symmetry to compute non-dominated objective vectors, also avoiding element removals. `PlainNDred` and `PreND` are applicable to both the union and the Minkowski sum of two Pareto sets. They can also be applied to a single objective space, as pure dominance-checks, to extract its Pareto frontier. `SymND`, on the other hand, is applicable only to the union of two Pareto sets. All three algorithms are compatible with each of the aforementioned data structures.

Our final contribution is an extensive experimental evaluation to assess the performance of our algorithms and data structures. We consider all nine combinations of a filtering algorithm among `PlainNDred`, `PreND`, and `SymND` with an indexing data structure from ND$^+$-trees, QND$^+$-trees, and TND$^+$-trees. We compare them with the state-of-the-art algorithms in [13] for $d \geq 3$ criteria. For our experimental evaluation, we used real-world data sets, synthetic data sets similar to those in [13], and new synthetic data sets specifically designed to resemble features of real-world instances. Our experimental results reveal that our algorithms are very efficient and scale well w.r.t. both the number of criteria d and the set sizes across all data sets. Notably, they achieve speedups up to 5.9× on real-world data sets and up to 13.2× on synthetic data sets against the best-performing algorithms from [13].

2 Preliminaries

Let $[n] = \{1, 2, \ldots, n\}$, $\forall n \in \mathbb{Z}^+$. In the following, small letters denote scalars, boldfaced small letters denote vectors, and capital letters denote sets. For any element or point $\mathbf{p} \in \mathbb{R}^d$, let $\mathbf{p}[i]$ denote the value of its i-th coordinate, for each $i \in [d]$. We consider multi-objective minimization problems with $d \geq 2$ objective functions:

$$\begin{array}{ll} \text{minimize} & \mathbf{f}(\mathbf{x}) = (\ \mathbf{f}(\mathbf{x})[1] = f_1(\mathbf{x}),\ \mathbf{f}(\mathbf{x})[2] = f_2(\mathbf{x}),\ \ldots\ ,\ \mathbf{f}(\mathbf{x})[d] = f_d(\mathbf{x})\ ) \\ \text{s.t.} & \mathbf{x} \in X \end{array}$$

X is the *solution space*, i.e., the set of feasible solutions for the instance at hand. $F = \mathbf{f}(X) = \left\{\mathbf{p} \in \mathbb{R}^d : \exists \mathbf{x} \in X, \mathbf{p} = \mathbf{f}(\mathbf{x})\right\}$ is the corresponding *objective space*, with all d-dimensional vectors that appear as *objective-value vectors* for at least one feasible solution from X. We refer to these objective vectors simply as *(data) points* and focus on $F \subseteq \mathbb{R}^d$, since all dominance checks are conducted among the points of F. For $\mathbf{p}, \mathbf{p}' \in \mathbb{R}^d$, we say that $\mathbf{p}$ *dominates* $\mathbf{p}'$, denoted as $\mathbf{p} < \mathbf{p}'$, if $\mathbf{p} \neq \mathbf{p}'$ and $\mathbf{p}[i] \leq \mathbf{p}'[i]$, $\forall i \in [d]$. For $F \subseteq \mathbb{R}^d$, its *Pareto frontier* (*subset* or *skyline*) is the maximal subset $P \subseteq F$ of points which are not dominated by any other point in F. If $P = F$, then F itself is also called a *Pareto set*. For $A, B \subset \mathbb{R}^d$, their *Minkowski sum* is $A \oplus B = \{\mathbf{a} + \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}$. Given two Pareto sets $A, B \subset \mathbb{R}^d$, their *Pareto union* is the Pareto frontier of $A \cup B$, and their *Pareto sum* is the Pareto frontier of $A \oplus B$. Given $F \subset \mathbb{R}^d$, the *dominance-filtering* problem aims at filtering out all points in F which are dominated by other points in F, so as to construct its Pareto frontier.

3 Algorithmic Background

A generic approach for dominance-filtering is to process the points of $F = \{\mathbf{p_1}, \ldots, \mathbf{p_n}\}$ sequentially, and keep updating a subset P, which will eventually be the Pareto frontier of F, as follows. For each new point $\mathbf{p_i} \in F$: Compare $\mathbf{p_i}$ (sequentially) with each point $\mathbf{p_j} \in P$ $(j < i)$. If $\mathbf{p_j} < \mathbf{p_i}$ then reject $\mathbf{p_i}$ (it is dominated by some point in P) and proceed with the next point of F. Otherwise, if $\mathbf{p_i} < \mathbf{p_j}$, then remove $\mathbf{p_j}$ from P (it is dominated by $\mathbf{p_i}$); if there is no other point in P to compare with, append $\mathbf{p_i}$ to P; otherwise, proceed with a comparison of $\mathbf{p_i}$ with the next point in P. The efficiency of the data structure used to maintain the current subset P and perform the previously mentioned dominance-checks is critical for the performance of this incremental approach. A well-suited data structure for this task is the ND-tree [10], which we discuss subsequently.

3.1 ND-trees

An ND-tree is a typical rooted c-ary tree T, in which a distinct node $r = root(T)$, of degree at most c, is the *root node*, all nodes of degree 1 (except possibly for the root) are its *leaf nodes*, and the remaining nodes of degree from 2 up to $c+1$ are its *internal nodes*. The ND-trees are *leaf-oriented*, meaning that all data points are stored exclusively in leaf nodes. Each leaf node can store up to m points. The parameters c and m must satisfy the condition $c \leq m+1$ [14]. Each node v stores a lower-bounding vector $\mathbf{lb_v}$ and an upper-bounding vector $\mathbf{ub_v}$ for all the data points stored in leaves of the subtree T_v of T rooted at v. Specifically, for each point $\mathbf{p}$ stored in T_v, it holds that $\forall i \in [d], \mathbf{lb_v}[i] \leq \mathbf{p}[i] \leq \mathbf{ub_v}[i]$. An example of an ND-tree can be found in Figure 1.

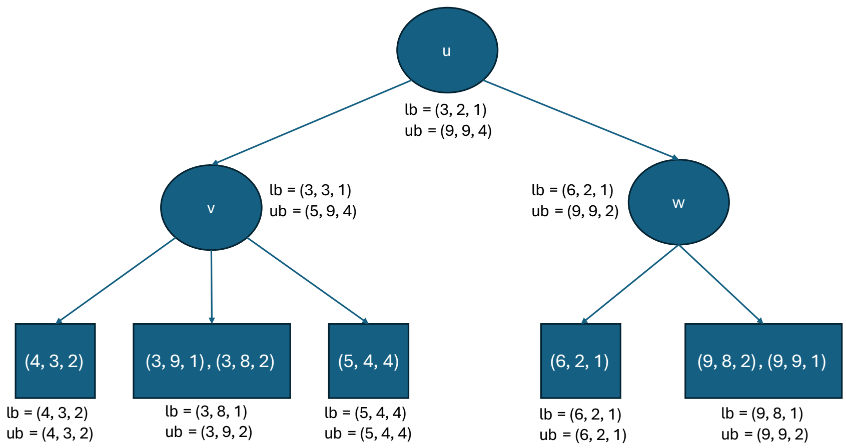

Figure 1 ND-tree containing 3-dimensional points.

The lower and upper-bounding vectors are typically used to determine, as early as possible, if a new data point $\mathbf{p}$ is dominated by any data point already in the tree. For instance, if $\exists i \in [d] : \mathbf{p}[i] < \mathbf{lb_v}[i]$, then $\mathbf{p}$ is not dominated by any data point stored in T_v, and we do not have to explicitly verify this with all of them (of course, it might still be the case that $\mathbf{p}$ dominates some of these data points). If $\mathbf{p} < \mathbf{lb_v}$ then $\mathbf{p}$ dominates all the data points stored in T_v. Finally, if $\mathbf{p} > \mathbf{ub_v}$, then all the data points stored in leaves of T_v dominate $\mathbf{p}$. An ND-tree T supports the following operations.

- `NonDomPrune`$(\mathbf{p}, T)$: This operation effectively utilizes the bounding vectors to perform two tasks, a *dominance-check* for a point $\mathbf{p}$ to decide whether it is dominated by any data point in T, and a *pruning* of T to remove all its data points that are dominated by $\mathbf{p}$. If $\mathbf{p}$ is not dominated by any point in T, `NonDomPrune` returns True; otherwise, it returns False.
- `Insert`$(\mathbf{p}, v)$: This operation inserts a new point $\mathbf{p}$ into a leaf of T_v as follows. If v is a non-leaf node, then a child node w of v is selected with minimum distance from $\mathbf{p}$. The distance of $\mathbf{p}$ from any node u is the Euclidean distance between $\mathbf{p}$ and $\frac{\mathbf{lb_u}+\mathbf{ub_u}}{2}$ (the center of the bounding box containing all data points stored T_u). Consequently, $\mathbf{p}$ is

recursively requested to be inserted in T_w. For a leaf node v, if it stores less than m data points, then $\mathbf{p}$ is simply appended to its list of stored points; otherwise, v is converted into an internal node with c children, and the pending $m+1$ data points are distributed evenly among them. Throughout the insertion process, the bounds of all affected nodes are updated accordingly.

- `SPNDBuild`(P): This operation, introduced in [14], aims to handle situations in which repeated insertions into an ND-tree might eventually lead to an unbalanced tree structure. It takes as input a Pareto set P (e.g., with all the data points stored in an unbalanced ND-tree) and builds from scratch a *perfectly balanced* ND-tree from it, as no pruning is ever required, in which the bounding areas defined by the upper and lower bounds also *are non-overlapping.*

3.2 ND-Tree based Algorithms for Dominance Filtering

The following dominance-filtering algorithms, proposed in [13], are all based on ND-trees and are, to our knowledge, the state-of-the-art techniques for $d \geq 3$ criteria.

- `PlainND`: This algorithm employs `NonDomPrune` and `Insert` to compute either the Pareto union or the Pareto sum of two Pareto sets A and B. It begins with an empty ND-tree T, and processes sequentially the points in F (either $A \cup B$, or $A \oplus B$). For each point $\mathbf{p} \in F$, it calls `NonDomPrune`$(\mathbf{p}, T)$. If False is returned, $\mathbf{p}$ is discarded. Otherwise, it executes `Insert`$(\mathbf{p}, T)$ to store $\mathbf{p}$ in T. After having processed all points in F, the points eventually stored in the leaves of T constitute the Pareto frontier of F.
- `PlainSPND`: This algorithm is similar to `PlainND`, but it periodically takes the Pareto set P of data items in the current ND-tree, it then destroys the tree, and consequently calls `SPNDBuild`(P) to create a new, balanced ND-tree. This periodic tree reconstruction can significantly improve the efficiency of intermediate calls to the `NonDomPrune` and `Insert` operations, due to limitations in the imbalance of the evolving ND-tree, while the tree reconstruction cost is amortized among consecutive insertion and pruning operations.
- `PruneSPND`: This algorithm is custom-tailored for computing the Pareto union of two Pareto sets A and B. It exploits the fact that points in A may only be dominated by points in B, and vice versa. Therefore, for the larger of the two sets (say, A) it calls `SPNDBuild`(A) to build a balanced ND-tree T. Subsequently, for each point $\mathbf{p}$ in the smaller set (say, B), it calls `NonDomPrune`$(\mathbf{p}, T)$ to check if $\mathbf{p}$ is dominated and to remove from T all points dominated by $\mathbf{p}$. If $\mathbf{p}$ is dominated, it is removed from B. After having processed all points in B, T contains all points of A which are not dominated by any point in B, and (eventually) B has only retained those points which are not dominated by any point in A. Their union constitutes the Pareto union.

4 New Data Structures for Dominance-Filtering

We present here our new data structures, ND$^+$-trees, QND$^+$-trees and TND$^+$-trees, designed to boost the efficiency of dominance-filtering. Detailed description of their operations, pseudocode, and the proofs of theorems can be found in the full version of the paper.

4.1 Overview of ND$^+$-trees

An ND$^+$-tree T is a leaf-oriented *binary* tree, with each leaf node storing up to m data points. As in k-d trees [1], every node v is associated with a level-dependent dimension $v.dim = 1 + (v.level \textbf{ mod } d)$, a subset S_v of data points (to be stored in the leaves of T_v) and the median value $v.q$ among the coordinates of all the points of S_v in dimension $v.dim$. Each node v also maintains a lower-bounding vector $\mathbf{lb_v}$ (i.e., the coordinate-wise minimum) of all points in S_v, similarly to ND-trees [10]. However, we avoid storing also the upper-bounding vectors, since our experimental evaluation showed that maintaining them is not beneficial for the operations on ND$^+$-trees. Moreover, if v is internal node, then S_v is partitioned in two distinct subsets, one per child of v: the left child gets all points in S_v with $\mathbf{p}[v.dim] < v.q$ and the right child gets all the remaining points of S_v. An example of an ND$^+$-tree is shown in Figure 2. For each ND$^+$-tree T, the following elementary operations are supported:

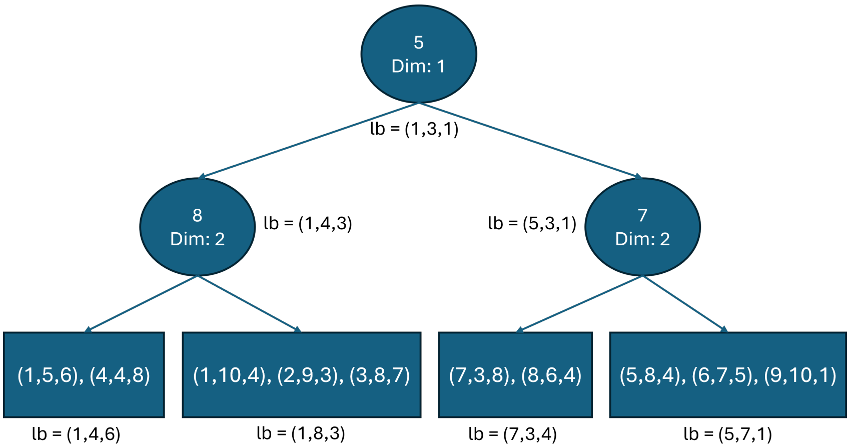

Figure 2 Example of an ND$^+$-tree containing 3-dimensional points with $m = 3$.

- `BuildND`$^+$(P, ℓ, d) constructs an ND$^+$-(sub)tree at level ℓ (an entire tree, when $\ell = 0$) containing all the data points of a Pareto set P.
- `ComputeBoundsND`$^+$(r) computes the lower-bounding vectors for all nodes of T_r.
- `WidenBoundsND`$^+$($v, \mathbf{p}$) updates the lower-bounding vector of node v, if necessary, due to a (previous) addition of a new point $\mathbf{p}$ in T_v.
- `InsertND`$^+$($v, \ell, \mathbf{p}$) inserts a new point $\mathbf{p}$ into the subtree T_v rooted at the level-ℓ node v.
- `DominatedND`$^+$($v, \ell, \mathbf{p}$) decides whether a new point $\mathbf{p}$ is dominated by any other point in the subtree T_v rooted at the level-ℓ node v.

In the remaining part of this section we provide some theoretical guarantees on the complexities of these elementary operations, when each leaf of the tree stores at most m d-dimensional points, for arbitrary constants $m, d \in O(1)$.

▶ **Theorem 1.** *Given a Pareto set of n points,* `BuildND`$^+$ *constructs an ND$^+$-tree, with $N = O(n)$ nodes in $O(n \log n)$ time when all splits of a point set produce constant fractions for both parts, and in $O(n^2)$ time otherwise.*

▶ **Theorem 2.** *Given an ND$^+$-tree with N nodes, n points and height h, the following bounds hold for its elementary operations: (i)* `ComputeBoundsND`$^+$ *takes $O(nmd) = O(n)$ time; (ii)* `InsertND`$^+$ *takes $O(hd + m) = O(h)$ time; (iii)* `DominatedND`$^+$ *takes $O(dn) = O(n)$ time.*

4.2 Overview of QND$^+$-trees

For a Pareto set P with its points having *distinct values* per dimension, `BuildND`$^+$ constructs a *balanced* ND$^+$-tree in *quasilinear* time (cf. Theorem 1). However, objective spaces F emerging from real-world scenarios (and their Pareto subsets) rarely adhere to such a strong property. Instead, it is common for large subsets of objective vectors to possess identical values in certain dimensions, e.g., for tolls in road networks. When a subset of S_v constitutes a large *plateau* around $v.q$ (i.e., those data points have the same value $v.q$ in $v.dim$) for some internal node v, the resulting partition of S_v may be (possibly heavily) uneven. If this pattern occurs frequently at intermediate nodes, then the resulting ND$^+$-tree will be heavily unbalanced, leading to *quadratic* construction time and also *linear* (instead of logarithmic) time for insertion of new points. This may happen even if initially a balanced tree is constructed, due to subsequent insertions of points that constitute a plateau in F.

To tackle these worst-case performances of the ND$^+$-trees, we introduce in this section an alternative data structure, the QND$^+$-trees (*Quartile ND$^+$-trees*). In a nutshell, the QND$^+$-trees are almost identical to the ND$^+$-trees, the only difference being that they perform a more careful bipartition of the data set S_v associated with an internal node v. Specifically, if there is a large plateau (more than one fourth) in dimension $v.dim$ around the splitting value $v.q$, then this plateau is entirely assigned to v's right child, with the remaining points being assigned (irrespectively of their values in dimension $v.dim$) to its left child. Moreover, when checking for dominance-checks in the subtree rooted at v's right child, it is no longer necessary to consider $v.dim$, achieving dimensionality reduction. If no such plateau is discovered in S_v, then the split is done as in an ND$^+$-tree.

We consider the following partitioning strategies: `Median Partitioning (MP)`, also used in ND$^+$-trees, assigns points of S_v with values in $v.dim$ less than $v.q$ to v's left child, and the remaining points of S_v to v's right child; `Quartile Partitioning (QP)` assigns points of S_v with value $v.q$ in dimension $v.dim$ to v's right child, and all other points to v's left child. The following elementary operations are supported for QND$^+$-trees:

- `BuildQND`$^+(P, \ell, d)$ constructs a QND$^+$-tree with the points of Pareto set P, in an analogous manner with `BuildND`$^+(P, \ell, d)$. The only difference is that, before splitting the point set S_v associated to an internal node v, it first computes the quartiles of S_v in dimension $v.dim$ and then applies `QP` when $Q_1 = Q_2$ (this implies that the size of the plateau is at least $|S_v|/4$), and `MP` otherwise. After completing the tree construction, it calls `ComputeBoundsND`$^+$ to compute all the lower bounding vectors.
- `InsertQND`$^+(v, \ell, \mathbf{p})$ inserts a new point $\mathbf{p}$ in T_v, similarly to the corresponding operation on ND$^+$-trees. It first updates $\mathbf{lb_v}$ using `WidenBoundsND`$^+$ and then recursively calls itself for the appropriate child of v (if this is an internal node), depending on the existence (or not) of a plateau in $v.dim$, or else (for a leaf node) it either stores $\mathbf{p}$ in v's data list, otherwise (when there is no free space in v's list) it changes v into an internal node and redistributes evenly all the pending data points among its two children, executing also `ComputeBoundsND`$^+$ to update their lower bounding vectors.
- `DominatedQND`$^+(v, \ell, \mathbf{p}, D)$, an adaptation of `DominatedND`$^+$ on QND$^+$-trees, checks whether a new point $\mathbf{p}$ is dominated by any other point in T_v, taking also into account if there are any dimensions to ignore (due to existence of plateaus) during its recursive calls.

▶ **Theorem 3.** *Given a Pareto set of n points, `BuildQND`$^+$ constructs a QND$^+$-tree with $N = O(n)$ nodes and height $O(\log n + d) = O(\log n)$ in $O(n(\log n + d)) = O(n \log n)$ time.*

▶ **Theorem 4.** *Given a QND$^+$-tree with N nodes and height h, containing n points, then the following time bounds hold for its elementary operations: (i) `InsertQND`$^+$ takes $O(hd) = O(h)$ time; (ii) `DominatedQND`$^+$ takes $O(dn) = O(n)$ time.*

4.3 Overview of TND$^+$-trees

TND$^+$-trees (*Ternary ND$^+$-trees*) are designed to exploit both tree balance and dimensionality reduction due to the existence of plateaus. Contrary to the other trees, TND$^+$-trees are not strictly binary. In particular, whenever a large plateau is discovered within S_v in dimension $v.dim$ of an internal node v, three children are created: The left child is associated with points $\mathbf{p} \in S_v$ with $\mathbf{p}[v.dim] < v.q$, the right child with points $\mathbf{p} \in S_v$ with $\mathbf{p}[v.dim] > v.q$, and the middle child with all the points which constitute the plateau (i.e., $\mathbf{p}[v.dim] = v.q$). Again, for the middle child we also exploit the resulting dimensionality reduction. We will refer to this plateau-based partitioning strategy as a `TriPartitioning (TP)`. If no plateau is detected, then the standard `Median Partitioning (MP)` strategy is applied. The following elementary operations are supported for TND$^+$-trees:

- `BuildTND`$^+(P, \ell, d)$ constructs a TND$^+$-tree for the points of a Pareto set in a fashion analogous to that of `BuildQND`$^+$, applying `(TP)` whenever $Q_1 = Q_2$ or $Q_2 = Q_3$ for the quartiles of S_v in dimension $v.dim$, ensuring that at least 25% of S_v are assigned to the middle child where we also benefit from dimensionality reduction. After completing the tree construction, `ComputeBoundsTND`$^+$ is executed, a slightly modified version of `ComputeBoundsND`$^+$ that also takes into consideration the middle child, to compute the lower-bounding vectors of each node in the tree.
- `InsertTND`$^+(v, \ell, \mathbf{p})$ inserts $\mathbf{p}$ to the TND$^+$-(sub)tree T_v, resembling `InsertQND`$^+$. It first updates v's lower-bounding vector using `WidenBoundsND`$^+$. If v is an internal node, then a recursive call of the method is executed to insert $\mathbf{p}$ to the appropriate child of v, taking also into account whether v possesses a middle child. If v is a leaf node that stores less than m points, then it simply stores $\mathbf{p}$ at v's list of points, otherwise v is converted into an internal node and the $m + 1$ now pending points (including $\mathbf{p}$) are redistributed among its (either two or three, depending on the presence of a plateau) children, using `BuildTND`$^+$. Upon completion, `ComputeBoundsTND`$^+$ is executed to compute the lower bounds of the newly created children.
- `DominatedTND`$^+(v, \ell, \mathbf{p}, D)$ determines whether $\mathbf{p}$ is dominated by any other point in the TND$^+$-(sub)tree T_v.

▶ **Theorem 5.** *Given a Pareto set of n points, `BuildTND`$^+$ constructs an TND$^+$-tree with $N = O(n)$ nodes and height $O(\log n + d) = O(\log n)$ in $O(n(\log n + d)) = O(n \log n)$ time.*

▶ **Theorem 6.** *Given a TND$^+$-tree with n nodes and height h, then the following time bounds hold for its elementary operations: (i) `InsertTND`$^+$ takes $O(hd) = O(h)$ time; (ii) `DominatedTND`$^+$ takes $O(dn) = O(n)$ time.*

4.4 Comparison of the New Data Structures

To illustrate the differences between the three data structures, and to demonstrate how QND$^+$- and TND$^+$-trees produce more balanced structures than ND$^+$-trees in the presence of plateaus, consider building ND$^+$-, QND$^+$-, and TND$^+$-trees from the point set $S = \{(1, 10, 2),\ (2, 9, 6),\ (2, 8, 7),\ (2, 12, 0),\ (2, 7, 8),\ (2, 11, 1),\ (4, 7, 4),\ (5, 7, 3),\ (6, 7, 2),\ (7, 6, 1),\ (8, 6, 0)\}$. Assume that each leaf node can store up to $m = 4$ points.

- **ND$^+$-tree:** Median Partitioning (MP) is first applied in the first dimension, yielding the median value 2. This assigns $(1, 10, 2)$ to the left subtree and all remaining points to the right, giving $L = \{(1, 10, 2)\}$ and $R = \{(7, 6, 1),\ (8, 6, 0),\ (2, 7, 8),\ (4, 7, 4),\ (5, 7, 3),\ (6, 7, 2),\ (2, 8, 7),\ (2, 9, 6),\ (2, 11, 1),\ (2, 12, 0)\}$. MP is then applied to R in the second dimension, with median 7, producing $RL = \{(7, 6, 1),\ (8, 6, 0)\}$ and $RR = \{(2, 12, 0),$

$(2, 11, 1)$, $(6, 7, 2)$, $(5, 7, 3)$, $(4, 7, 4)$, $(2, 9, 6)$, $(2, 8, 7)$, $(2, 7, 8)\}$. Since RR exceeds the leaf size m, it is split once more using MP in the third dimension, where the median is 4, giving $RRL = \{(2, 12, 0), (2, 11, 1), (6, 7, 2), (5, 7, 3)\}$ and $RRR = \{(4, 7, 4), (2, 9, 6), (2, 8, 7), (2, 7, 8)\}$. All resulting subtrees contain at most m points, completing the tree (see Figure 3).

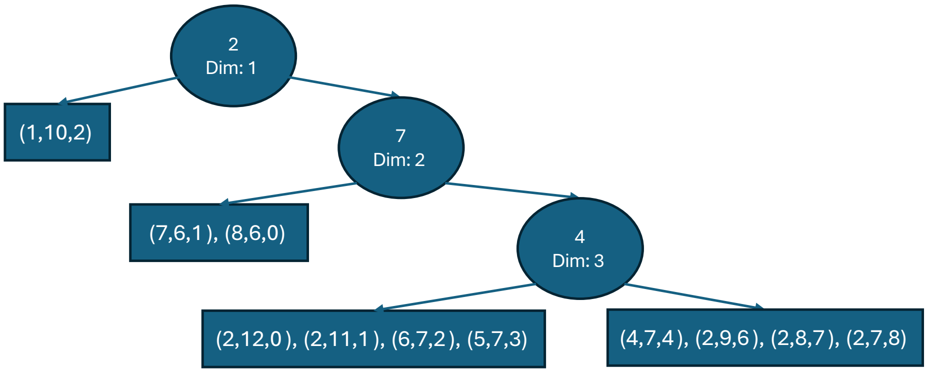

Figure 3 ND$^+$-tree containing 3-dimensional points with $m = 4$.

- **QND$^+$-tree:** Since $Q_1 = Q_2 = 2$ in the first dimension, Quartile Partitioning (QP) is used, yielding $L = \{(7, 6, 1), (8, 6, 0), (4, 7, 4), (5, 7, 3), (6, 7, 2), (1, 10, 2)\}$ and $R = \{(2, 7, 8), (2, 8, 7), (2, 9, 6), (2, 11, 1), (2, 12, 0)\}$. In L, $Q_1 = 6 \neq 7 = Q_2$ in the second dimension, so MP is applied, splitting it into $LL = \{(7, 6, 1), (8, 6, 0)\}$ and $LR = \{(4, 7, 4), (5, 7, 3), (6, 7, 2), (1, 10, 2)\}$, both of which satisfy the leaf size constraint. In R, $Q_1 = 8 \neq 9 = Q_2$ in the second dimension, so MP is again applied, giving $RL = \{(2, 8, 7), (2, 7, 8)\}$ and $RR = \{(2, 9, 6), (2, 11, 1), (2, 12, 0)\}$, each also within the allowed limit. The tree is thus complete (see Figure 4).

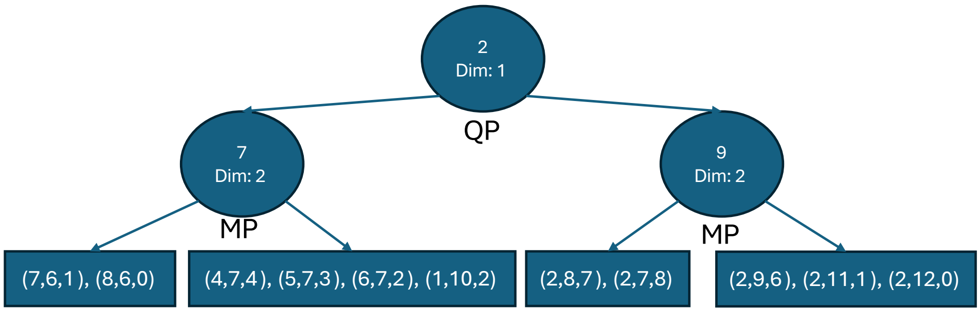

Figure 4 QND$^+$-tree containing 3-dimensional points with $m = 4$.

- **TND$^+$-tree:** $Q_1 = Q_2 = 2$ in the first dimension, so TriPartitioning (TP) is applied, producing $L = \{(1, 10, 2)\}$, $M = \{(2, 7, 8), (2, 8, 7), (2, 9, 6), (2, 11, 1), (2, 12, 0)\}$, and $R = \{(7, 6, 1), (8, 6, 0), (4, 7, 4), (5, 7, 3), (6, 7, 2)\}$. L needs no further processing. In M, no plateau is present in the second dimension, so MP is used with median 9, resulting in $ML = \{(2, 7, 8), (2, 8, 7)\}$ and $MR = \{(2, 9, 6), (2, 11, 1), (2, 12, 0)\}$. In R, a plateau is found: $Q_2 = 7 = Q_3$, so TP is applied again, yielding $RL = \{(7, 6, 1), (8, 6, 0)\}$ and $RM = \{(4, 7, 4), (5, 7, 3), (6, 7, 2)\}$. All resulting subtrees respect the leaf size constraint, completing the tree (see Figure 5).

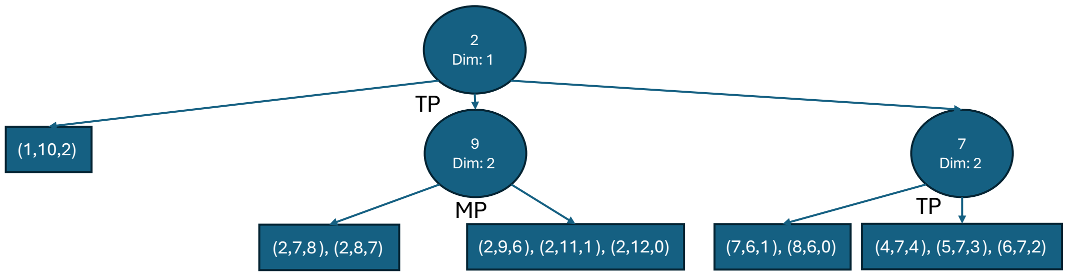

Figure 5 TND$^+$-tree containing 3-dimensional points with $m = 4$.

5 Overview of New Algorithms for Pareto Unions and Sums

We now present our new algorithms for computing Pareto unions or Pareto sums of Pareto sets. These algorithms are designed to work with any of the three data structures of Section 4. For simplicity, all algorithms are presented w.r.t. ND$^+$-trees – details and missing proofs can be found in the full version. In our experimental evaluation (cf. Section 6) we consider all possible combinations of algorithms and data structures.

5.1 PlainNDred

As noted in [13], the main computational burden in `PlainND`, `PlainSPND` and `PruneSPND` is the execution of `NonDomPrune` operations (cf. Section 3). The most demanding task is the removal of all dominated points from the tree, as new points are inserted to it. The main idea behind the `reduced PlainND` algorithm (`PlainNDred` in short) is to avoid this costly *pruning* task of the evolving tree, by ensuring that any point that is inserted to the tree is actually a member of the required Pareto frontier of F. To achieve this, `PlainNDred` first lexicographically sorts F, in quasilinear time. Then, to efficiently manage *dominance-checks*, it processes the data points in that order and uses one of the new indexing structures (ND$^+$-trees in `PlainNDred`, QND$^+$-trees in `PlainQNDred`, and TND$^+$-trees in `PlainTNDred`) to store only the non-dominated ones of F so far. In particular, for each point $\mathbf{p}$ in the lexicographic order, the algorithm must only check if it is dominated by any point in the tree, since $\mathbf{p}$ cannot dominate any of the preceding points in that order, as shown next.

▶ **Lemma 7.** *Let $S = (\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n})$ be a lexicographic order of a set $F \subset \mathbb{R}^d$ of n points. Then the following non-dominance property holds: $\forall 1 \leq i < j \leq n$, $\mathbf{p_j}$ cannot dominate $\mathbf{p_i}$.*

If $\mathbf{p}$ is not dominated by any point already in the tree, the algorithm inserts it. Moreover, *dominance-checks* can safely ignore dimension 1, since $\mathbf{p}[1] \geq \mathbf{q}[1]$ for any point preceding $\mathbf{p}$ in the lexicographic order. Therefore, `PlainNDred` needs only to check the remaining $d - 1$ dimensions. As a result, the algorithm builds a tree considering only the last $d-1$ dimensions of the points in F. We denote as $\mathbf{p_{red}}$ the projection of $\mathbf{p}$ on the last $d - 1$ dimensions. The following statement demonstrates time complexity of `PlainNDred`.

▶ **Theorem 8.** *For an n-point set F that is either the Minkowski sum or the union of two Pareto sets, the time complexity of `PlainNDred` algorithm is $O(n^2(d - 1)) = O(n^2)$.*

Proof. The points of F are first lexicographically sorted, in time $O(n \log n)$. Then, for each point, we perform a dominance-check against the previously processed points that belong to the tree. Each pairwise dominance-check takes time $O(d - 1)$ since only the last $d - 1$ dimensions matter. Even if all points in F are non-dominated and no pruning occurs, each point is compared to all previously processed points. Therefore, the algorithm makes at most $(d - 1)\frac{n(n-1)}{2} \in O((d - 1)n^2)$ comparisons, for all dominance-checks. ◀

5.2 PreND

The consecutive insertions into the tree by `PlainNDred` are likely to gradually unbalance it, thereby diminishing the efficiency of subsequent dominance-checks and insertions. To address this challenge, one option would be to periodically rebuild the tree, as is done by `PlainSPND` in [13]. However, building the entire tree from scratch is not a trivial task. To avoid that, we could leverage once more the lexicographic order of the point set $F = (\mathbf{p_1}, \dots, \mathbf{p_n})$ to compute first an initial subset P of the Pareto frontier, which is then used to construct a *balanced* ND$^+$-tree that will be large enough so that the subsequent insertions of the remaining non-dominated points, again examined in lexicographic order, will not be able to unbalance it severely. This is exactly the main idea of the `presorted ND` algorithm (`PreND` in short).

To compute this subset of the Pareto frontier, we deploy the `ParetoSubset` algorithm. It starts with the initialization of a vector $\mathbf{y}$ of length d, corresponding to the number of dimensions, with each dimension assigned the value ∞, and then makes a single pass over the data points, in lexicographic order, using $\mathbf{y}$ to keep track of the smallest values seen in each dimension, up to the current point $\mathbf{p_i}$. For the next point in order, $\mathbf{p_{i+1}}$, if there exists a dimension $j \in [d]$ such that $\mathbf{p_{i+1}}[j] < \mathbf{y}[j]$, then $\mathbf{p_{i+1}}$ is not dominated by any preceding point. Moreover, due to the lexicographic order, $\mathbf{p_{i+1}}$ cannot be dominated by any subsequent point $\mathbf{p_k} : k \geq i+2$ (cf. Lemma 7). Therefore, $\mathbf{p_{i+1}}$ is certainly a non-dominated point in F and is appended to P, and $\mathbf{y}$ is updated to always keep the smallest value seen so far, per dimension. Otherwise, when $\mathbf{y} \leq \mathbf{p_{i+1}}$, $\mathbf{p_{i+1}}$ is appended to another subset Q, for further examination, during the second processing phase. Note that, since all points are processed in lexicographic order, Q remains lexicographically sorted. Observe also that, for $d = 2$, `ParetoSubset` already computes the entire Pareto frontier of F.

The `PreND` algorithm initially calls `ParetoSubset` to get the sets P and Q. It then calls `BuildND`$^+$ to construct an initial ND$^+$-tree with the points of P. Subsequently, for each point $\mathbf{q} \in Q$, it calls `DominatedND`$^+$ to determine if $\mathbf{q}$ is dominated by any point of the tree. If it is dominated, then it is discarded. Otherwise, it is appended to P and inserted to the ND$^+$-tree by calling `InsertND`$^+$. After having processed all points in Q, P is the required Pareto frontier of F. Note that `PreND` can be used for constructing the Pareto union or the Pareto sum of two Pareto sets, but also for identifying the Pareto frontier of a single set.

▶ **Theorem 9.** *For an n-point set F that is either the Minkowski sum or the union of two Pareto sets, the time complexity of `PreND` is $O(n^2)$.*

Proof. `ParetoSubset` first lexicographically sorts the set of points, in $O(n \log n)$ time. Then, it iterates through all n points and for each point determines in $O(d)$ time whether it belongs to P or Q. Thus, `ParetoSubset` runs in $O(n \log n + nd) = O(n \log n)$ time. Next, an ND$^+$-tree is constructed from the points in P. In the worst case, this step takes $O(n^2)$ time ($O(n \log n)$ for QND$^+$ and TND$^+$-trees). After constructing the tree, for each point in Q, we perform a dominance-check against the points already in the tree. In the worst case, where no pruning occurs and every point is non-dominated, each dominance-check involves comparing the point with all previously processed points. Since the pairwise dominance-checks are executed in time $O(d)$, and we perform this check for at most $\frac{n(n-1)}{2}$ pairs of points, the total time complexity for the dominance-checks is $O(dn^2)$. Hence, the overall time complexity of the `PreND` algorithm is $O(dn^2) = O(n^2)$. ◀

5.3 SymND

Especially for the union of two Pareto sets A, B, recall that points of one set may be dominated only by points of the other set. Thus, applying some sort of symmetric dominance-filtering could be extremely efficient. This is exactly what the `symmetric ND` algorithm (`SymND` in short) does. It constructs first an ND$^+$-tree using the points of A and then executes `DominatedND`$^+$ for each point in B, to remove from B all those points which are dominated by a point in A. Then, it constructs another ND$^+$-tree, using only the remaining points in B, and executes `DominatedND`$^+$ for each point in A, to remove those points which are dominated by a point in B. In the end, both surviving subsets of A and B contain only non-dominated points, and their union constitutes the Pareto union of A and B.

▶ **Theorem 10.** *Given two Pareto sets A and B, the time complexity of `SymND` to compute their Pareto union is $O(|A| \cdot |B|)$.*

Proof. Let $|A| = n_1 \leq |B| = n_2$. Assume also that the size of their Pareto union (to be computed) is k. First, an ND$^+$-tree is constructed using the points of the smaller set A, which takes time $O(dn_1^2)$ in the worst case for ND$^+$-trees, and $O(n_1 \log n_1)$ for QND$^+$ and TND$^+$-trees. Then, for each point in set B, the `DominatedND`$^+$ method is applied. Since there are n_2 points in set B, and each `DominatedND`$^+$ operation takes time $O(dn_1)$ in the worst case (when each point of B is compared to all points in the tree), the total complexity of this step is $O(dn_1n_2)$.

Consequently, we construct a second ND$^+$-tree using the remaining (at most $\hat{n}_2 = \min\{k, n_2\}$) points from set B. This tree construction requires in worst case time $O(d\hat{n}_2^2)$ time for ND$^+$-trees and $O(\hat{n}_2 \log \hat{n}_2)$ for QND$^+$ and TND$^+$-trees. Next, the `DominatedND`$^+$ method is applied for each point in set A, removing any dominated points from A. As there are n_1 points in set A, the total complexity of this second phase is $O(dn_1\hat{n}_2)$. Therefore, the overall time complexity of the `SymND` algorithm is $O(dn_1^2 + dn_1n_2 + d\hat{n}_2^2 + dn_1\hat{n}_2) = O(dn_1(n_2 + \min\{k, n_2\}))$ with ND$^+$-trees and $O(dn_1 \log n_1 + dn_1n_2 + d\hat{n}_2 \log \hat{n}_2 + dn_1\hat{n}_2) = O(dn_1n_2)$ with QND$^+$-trees and TND$^+$-trees. ◀

6 Experimental Evaluation

In our experimental evaluation, we implemented all nine combinations of our proposed indexing data structures and dominance-filtering algorithms. We distinguish each combination with an appropriate naming as follows: For each algorithm, its short name is used to indicate an implementation with ND$^+$-trees, and variants with QND$^+$-trees and TND$^+$-trees are indicated by the appearance in the short name of the substrings "QND" and "TND", respectively. For example, `PlainNDred` indicates the implementation of `reduced PlainND` with ND$^+$-trees, `PreQND` indicates the implementation of `presorted ND` with QND$^+$-trees, and `SymTND` indicates the implementation of `symmetric ND` with TND$^+$-trees.

In addition, we implemented nine algorithms of [13], which constitute, to the best of our knowledge, the state-of-the-art dominance-filtering algorithms for MOCO problems with $d \geq 3$ dimensions. Apart from the algorithms `PlainSPND` and `PruneSPND` which were discussed in Section 3, several more algorithms were provided in [13]: `NonDomDC` explores divide-and-conquer strategies that partition the initial set of solutions into smaller subsets to reduce unnecessary comparisons; `FilterX2` and `FilterSym` are bidirectional filters that are also based on divide-and-conquer techniques; `BatchedSPND` that utilizes SPND-trees; `LimMem` provides a memory-efficient alternative for scenarios where memory availability is limited; finally, `Doubling(Filter)` and `Doubling(Tree)` are adaptations of `FilterSym` and

`PruneSPND`, respectively, which are custom-tailored for Minkowski sums. All implemented algorithms are listed in Table 1, where their applicability on the specific dominance-filtering variant (union and/or Minkowski sum) is also mentioned.

Table 1 Algorithms that were implemented and tested in our experimental evaluation.

Reference: [13]		Reference: this work	
Algorithm	**Usage**	**Algorithm**	**Usage**
FilterX2	Union	SymND	Union
FilterSym	Union	SymQND	Union
BatchedSPND	Minkowski sum	SymTND	Union
Doubling(Filter)	Minkowski sum	PreND	Union & Minkowski sum
Doubling(Tree)	Minkowski sum	PreQND	Union & Minkowski sum
LimMem	Minkowski sum	PreTND	Union & Minkowski sum
NonDomDC	Union & Minkowski sum	PlainNDred	Union & Minkowski sum
PlainSPND	Union & Minkowski sum	PlainQNDred	Union & Minkowski sum
PruneSPND	Union	PlainTNDred	Union & Minkowski sum

6.1 Data Sets

To evaluate the performance of our algorithms, we used both real-world and synthetic data sets. Note that real-world data sets with three or more objectives ($d \geq 3$) are very rare. Since our main goal is to test scalability with dimensionality, we have also used two families of synthetic data sets with up to $d = 10$ objectives: the randomly constructed data sets of [13], and some new, carefully generated synthetic data sets that resemble some crucial features of real-world instances for MOCO problems. Below, we provide a brief overview of all these data sets; more details can be found in the full version of the paper. The **RW** sets are based on the New York City road network [5], and are equipped with two cost metrics: travel time and distance. We extend them to higher dimensions according to well established augmentation techniques: for $d = 3$, we adopted [17] and introduced a third objective which is related to hazardous material transportation [7]. For $d = 5$, we adopted [8] and added a fourth objective which is a random integer from 1 to 100, and a fifth objective which is a random integer from 1 up to the number of graph edges. The **URS** sets were synthetically generated according to the procedure described in [13] (sampling points uniformly in d-dimensional space and then projecting them into the unit sphere to ensure that these are Pareto sets). Apart from these “baseline” RW and URS data sets, we also considered a few extensions towards incorporating some typical features of real-world instances. The **RWP** and **URSP** sets try to model realistic instances with repeated or flat objective values (e.g., tolls for road networks), by introducing plateaus in some objectives of the RW and URS sets, respectively. The **RWC** and **URSC** sets simulate interdependencies between objectives (also encountered quite often in real-world instances), by introducing correlations between some objectives. Finally, the **URSPC** set combines both correlation and plateau features in a single Pareto set. The size of data sets given as input to all algorithms varies from 10K to 1M points, while the number of objectives (dimensions) d varies from 3 to 10 (3, 5 for RW, 5 for RWP and RWC, 4, 6, 8, 10 for URS, URSP, URSC and 5, 6, 8, 10 for URSPC).

6.2 Overview of Experimental Results

We provide here an overview of our experimental results. All details can be found in the full version of the paper.

Comparison with the state-of-the-art. Tables 2 and 3 compare our algorithms with the best-performing algorithms from [13] for the Pareto sum and the Pareto union of two Pareto sets, respectively. We compute the speedup factor per (algorithm, data set) pair, as follows: for each (dimensionality, input size) pair, we identify the fastest algorithm of [13]. We then compute the ratio of its runtime to that of our own algorithm. In the tables we report the minimum and maximum speedups observed across all (dimensionality, input size) pairs.

Table 2 Min and Max speedups for each algorithm for the *Pareto Sum* operation.

Algorithm	RW		RWP		RWC		URS		URSP		URSC		URSPC	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
PreND	1.1	3.8	2.9	4.4	1.4	2.6	1.8	5.2	1.8	4.9	1.7	4.7	2.9	7.7
PreQND	1.0	3.7	2.8	4.3	1.4	2.5	1.7	5.6	1.8	5.3	1.7	5.2	2.9	8.2
PreTND	1.0	3.7	2.8	4.2	1.4	2.5	1.7	5.5	1.8	5.2	1.7	5.0	2.9	7.8
PlainNDred	1.1	3.7	2.6	4.5	1.4	2.5	2.9	10.8	2.4	7.0	1.7	6.2	2.9	12.2
PlainQNDred	1.1	3.6	2.6	4.3	1.4	2.6	2.9	11.8	2.4	9.2	1.7	7.5	3.0	13.2
PlainTNDred	1.1	3.6	2.5	4.4	1.4	2.6	3.0	8.6	2.4	8.8	1.7	7.2	2.9	12.7

Table 3 Min and Max speedups for each algorithm for the *Pareto Union* operation.

Algorithm	RW		RWP		RWC		URS		URSP		URSC		URSPC	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
PreND	1.9	5.5	2.0	4.2	1.7	4.1	1.5	5.2	1.7	3.6	0.4	2.6	0.5	1.5
PreQND	2.1	5.9	2.0	4.9	1.7	4.5	1.6	5.8	1.8	4.0	0.5	2.9	0.5	1.7
PreTND	2.0	5.7	1.7	4.6	1.6	4.3	1.6	5.7	1.8	3.9	0.5	2.8	0.5	1.6
SymND	1.4	3.0	1.4	3.0	1.1	2.9	1.7	4.8	1.3	3.7	1.4	3.6	1.0	3.5
SymQND	1.5	3.2	1.4	3.4	1.1	3.1	1.7	5.4	1.4	4.4	1.6	3.9	1.0	3.7
SymTND	1.4	3.1	1.3	3.1	1.1	2.9	1.6	5.2	1.3	3.9	1.4	3.5	1.0	3.4
PlainNDred	2.1	5.3	1.7	2.6	1.7	3.1	1.7	6.3	1.8	3.8	0.5	2.2	0.5	1.6
PlainQNDred	2.2	5.8	2.3	3.0	2.0	3.6	1.9	6.8	1.9	4.3	0.6	2.4	0.5	1.6
PlainTNDred	2.2	5.6	2.0	2.7	2.0	3.3	1.8	6.7	1.8	4.1	0.6	2.3	0.5	1.7

For Pareto sums (Table 2), `PreND`, `PlainNDred` and their variants exhibit nearly identical performance, consistently outperforming all other algorithms on RW/RWC/RWP, with speedups reaching up to 4.5×. For the synthetic data sets, all our algorithms significantly outperform the algorithms of [13], with `PlainNDred` and its variants achieving the largest speedups, up to 11.8× for URS, 9.2× for URSP, 7.5× for URSC, and 13.2× for URSPC.

For Pareto unions (Table 3), the variants of `PreND` achieve the greatest speedup on RW/RWP/RWC. On URS/URSP, all our algorithms show similar speedup ranges. On URSC/URSPC, the variants of `SymND` emerge as the top performers. Note that, although the variants of `PreND` and `PlainNDred` seem to occasionally be slower than `PruneSPND` on URSC and URSPC, they are faster or identical in average in most cases. `SymND` and it variants outperform `PruneSPND` for all data sets, with speedups of up to 3.9×.

Regarding the three data structures, for the Pareto union of two Pareto sets, QND$^+$-trees and TND$^+$-trees outperform ND$^+$-trees across almost all data sets. However, for the Pareto sum, ND$^+$-trees generally perform better than QND$^+$-trees and TND$^+$-trees on the real-world data sets. In contrast, for the synthetic data sets, QND$^+$-trees are typically the most efficient, followed by TND$^+$-trees, with ND$^+$-trees trailing behind.

Exploring the impact of tree height on algorithmic performances. The balance of a tree is crucial for the performance of our algorithms. When the tree is well-balanced, pruning mechanisms at each level can reduce more efficiently the search space and limit the number of candidate points that may dominate a new one. To evaluate this in practice, we conducted an experiment using a URSP set A consisting of $n = 100,000$ points, with a plateau of size $n/2$ around the median in a random dimension. We constructed an ND$^+$, a QND$^+$ and a TND$^+$ tree using their `Build` methods on A. For each tree, we computed the average height, the balance indicator $BI = \max(\text{height}) - \min(\text{height})$, and the number of dominance-checks required per point from a second set B when queried against the respective trees.

Table 4 Comparison of the novel data structures, w.r.t. average tree height, balance indicator (BI), and number of dominance-checks, for $100K$ points.

Tree	4 Dimensions			6 Dimensions			8 Dimensions			10 Dimensions		
	Height	BI	Checks	Height	BI	Checks	Height	BI	Checks	Height	BI	Checks
ND$^+$	16	10	56	15	7	243	14	4	511	14	2	537
QND$^+$	13	0	68	13	0	230	13	0	431	13	0	423
TND$^+$	13	1	38	13	1	137	13	1	371	13	1	428

The results, summarized in Table 4, show that QND$^+$ and TND$^+$ trees are consistently more balanced than ND$^+$ trees and also exhibit shorter heights. Although ND$^+$ trees are only slightly taller on average, their significantly higher balance indicator values reveal a more skewed structure. Consequently, except in the case of 4 dimensions, the ND$^+$ trees required more dominance-checks than the other two variants. It is important to note that, as highlighted in our complexity analysis (in the full version of the paper), the worst-case scenario may require a point to be compared against all nodes in the tree. However, the experimental results indicate that in practice, the number of dominance-checks is substantially lower. This highlights the practical efficiency of our tree structures and the effectiveness of their lower-bounding mechanisms.

7 Conclusions and Future Work

We introduced three new data structures and three efficient algorithms for computing the Pareto unions and Pareto sums of Pareto sets, for which we provided a theoretical analysis for their worst-case performances and conducted a thorough experimental evaluation against state-of-art techniques (for $d \geq 3$) from [13], on several real-world and synthetically generated data sets. In all instances and dominance-filtering scenarios all of our algorithms consistently outperformed each algorithm in [13] (except for the case of Pareto unions on URSC and URSPC data sets, in which only `SymND` and its variants outperformed the best algorithm in [13]). Future work will focus on enhancing the performance of `PreND` by developing an alternative method to `ParetoSubset`, so as to precompute larger subsets of the Pareto frontier without significantly increasing computational costs.

References

1 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. doi:10.1145/361002.361007.

2 Anthony Chen, Juyoung Kim, Seungjae Lee, and Youngchan Kim. Stochastic multi-objective models for network design problem. *Expert Syst. Appl.*, 37(2):1608–1619, 2010. doi:10.1016/J.ESWA.2009.06.048.

3 Wei-Mei Chen, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. Maxima-finding algorithms for multidimensional samples: A two-phase approach. *Comput. Geom.*, 45(1-2):33–53, 2012. doi:10.1016/J.COMGEO.2011.08.001.

4. Mina Dehghani, Vahab Vahdat, Maghsoud Amiri, Elaheh Rabiei, and Seyedmohammad Salehi. A multi-objective optimization model for a reliable generalized flow network design. *Comput. Ind. Eng.*, 138, 2019. doi:10.1016/J.CIE.2019.106074.
5. Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The shortest path problem: Ninth DIMACS implementation challenge*, volume 74. American Mathematical Soc., 2009.
6. Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectr.*, 22(4):425–460, 2000. doi:10.1007/S002910000046.
7. Erhan Erkut, Stevanus A. Tjandra, and Vedat Verter. Chapter 9 hazardous materials transportation. In Cynthia Barnhart and Gilbert Laporte, editors, *Transportation*, volume 14 of *Handbooks in Operations Research and Management Science*, pages 539–621. Elsevier, 2007. doi:10.1016/S0927-0507(06)14009-8.
8. Carlos Hernández, William Yeoh, Jorge A. Baier, Ariel Felner, Oren Salzman, Han Zhang, Shao-Hung Chan, and Sven Koenig. Multi-objective search via lazy and efficient dominance checks. In Edith Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 7223–7230. International Joint Conferences on Artificial Intelligence Organization, August 2023. Main Track. doi:10.24963/ijcai.2023/850.
9. Demian Hespe, Peter Sanders, Sabine Storandt, and Carina Truschel. Pareto sums of pareto sets. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 60:1–60:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ESA.2023.60.
10. Andrzej Jaszkiewicz and Thibaut Lust. Nd-tree-based update: A fast algorithm for the dynamic nondominance problem. *IEEE Trans. Evol. Comput.*, 22(5):778–791, 2018. doi:10.1109/TEVC.2018.2799684.
11. Antoine Kerberenes, Daniel Vanderpooten, and Jean Michel Vanpeperstraete. Computing efficiently the nondominated subset of a set sum. *Int. Trans. Oper. Res.*, 30(6):3455–3478, 2023. doi:10.1111/ITOR.13191.
12. David G. Kirkpatrick and Raimund Seidel. Output-size sensitive algorithms for finding maximal vectors. In Joseph O'Rourke, editor, *Proceedings of the First Annual Symposium on Computational Geometry, Baltimore, Maryland, USA, June 5-7, 1985*, pages 89–96. ACM, 1985. doi:10.1145/323233.323246.
13. Kathrin Klamroth, Bruno Lang, and Michael Stiglmayr. Efficient dominance filtering for unions and minkowski sums of non-dominated sets. *Comput. Oper. Res.*, 163:106506, 2024. doi:10.1016/J.COR.2023.106506.
14. Bruno Lang. Space-partitioned nd-trees for the dynamic nondominance problem. *IEEE Trans. Evol. Comput.*, 26(5):1004–1014, 2022. doi:10.1109/TEVC.2022.3145631.
15. Thibaut Lust and Daniel Tuyttens. Variable and large neighborhood search to solve the multiobjective set covering problem. *Journal of Heuristics*, 20:165–188, 2014. doi:10.1007/s10732-013-9236-8.
16. Francisco Javier Pulido, Lawrence Mandow, and José-Luis Pérez-de-la-Cruz. Dimensionality reduction in multiobjective shortest path search. *Comput. Oper. Res.*, 64:60–70, 2015. doi:10.1016/J.COR.2015.05.007.
17. Zhongqiang Ren, Richard Zhan, Sivakumar Rathinam, Maxim Likhachev, and Howie Choset. Enhanced multi-objective A* using balanced binary search trees. In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 162–170. AAAI Press, 2022. doi:10.1609/SOCS.V15I1.21764.
18. Britta Schulze, Kathrin Klamroth, and Michael Stiglmayr. Multi-objective unconstrained combinatorial optimization: a polynomial bound on the number of extreme supported solutions. *Journal of Global Optimization*, 74:495–522, 2019. doi:10.1007/s10898-019-00745-6.

19 Fei Tan, Zheng-Yi Chai, and Ya-Lun Li. Multi-objective evolutionary algorithm for vehicle routing problem with time window under uncertainty. *Evol. Intell.*, 16(2):493–508, 2023. doi:10.1007/S12065-021-00672-0.

20 George Tsaggouris and Christos D. Zaroliagis. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. *Theory Comput. Syst.*, 45(1):162–186, 2009. doi:10.1007/S00224-007-9096-4.

21 Carlos Hernández Ulloa, Han Zhang, Sven Koenig, Ariel Felner, and Oren Salzman. Efficient set dominance checks in multi-objective shortest-path algorithms via vectorized operations. In Ariel Felner and Jiaoyang Li, editors, *Seventeenth International Symposium on Combinatorial Search, SOCS 2024, Kananaskis, Alberta, Canada, June 6-8, 2024*, pages 208–212. AAAI Press, 2024. doi:10.1609/SOCS.V17I1.31560.

22 Arthur Warburton. Approximation of pareto optima in multiple-objective, shortest-path problems. *Oper. Res.*, 35(1):70–79, 1987. doi:10.1287/OPRE.35.1.70.