Pareto Sums of Pareto Sets

Demian Hespe
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Sabine Storandt
University of Konstanz, Germany

Carina Truschel
University of Konstanz, Germany

Abstract

In bi-criteria optimization problems, the goal is typically to compute the set of Pareto-optimal solutions. Many algorithms for these types of problems rely on efficient merging or combining of partial solutions and filtering of dominated solutions in the resulting sets. In this paper, we consider the task of computing the Pareto sum of two given Pareto sets A, B of size n. The Pareto sum contains all non-dominated points of the Minkowski sum $M = \{a + b | a \in A, b \in B\}$. Since the Minkowski sum has a size of n^2, but the Pareto sum C can be much smaller, the goal is to compute C without having to compute and store all of M. We present several new algorithms for efficient Pareto sum computation, including an output-sensitive one with a running time of $\mathcal{O}(n \log n + nk)$ and a space consumption of $\mathcal{O}(n + k)$ for $k = |C|$. We also describe suitable engineering techniques to improve the practical running times of our algorithms and provide a comparative experimental study. As one showcase application, we consider preprocessing-based methods for bi-criteria route planning in road networks. Pareto sum computation is a frequent task in the preprocessing phase. We show that using our algorithms with an output-sensitive space consumption allows to tackle larger instances and reduces the preprocessing time compared to algorithms that fully store M.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Randomness, geometry and discrete structures

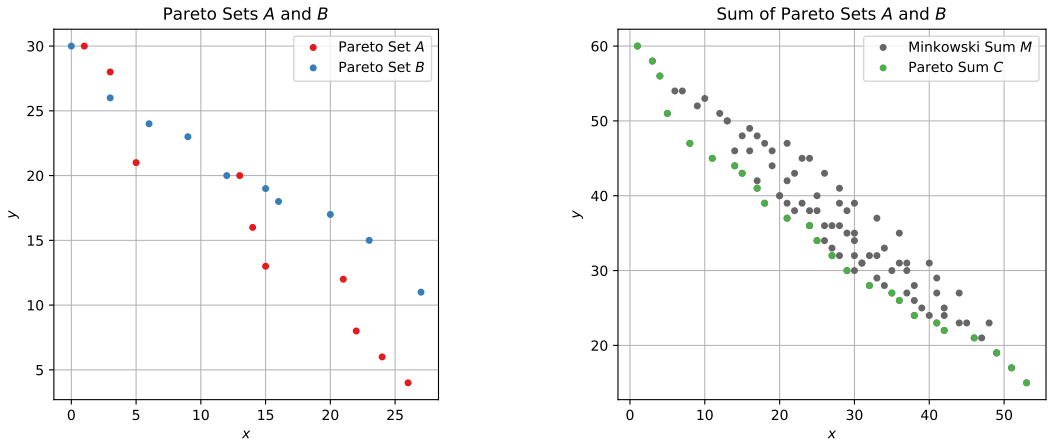
Keywords and phrases Minkowski sum, Skyline, Successive Algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.60

Funding *Peter Sanders*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).
Carina Truschel: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 251654672 – TRR 161.

1 Introduction

Solving multi-objective combinatorial optimization problems demands to find the set of non-dominated solutions, also referred to as skyline, Pareto frontier or Pareto set. To solve problem instances of substantial size, solution approaches often rely on efficient combination and filtering of partial solutions. In particular, non-dominance filtering of unions or Minkowski sums of intermediate Pareto sets occur as a frequent subtasks in optimization algorithms. Examples include decomposition approaches for multi-objective integer programming [16], dynamic programming methods for multi-objective knapsack [7], bi-directional search algorithms for multi-criteria shortest path problems [4], or Pareto local search for multi-objective set cover [14].



	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
B_1	1, 60	3, 58	5, 51	13, 50	14, 46	15, 43	21, 42	22, 38	24, 36	26, 34
B_2	4, 56	6, 54	8, 47	16, 46	17, 42	18, 39	24, 38	25, 34	27, 32	29, 30
B_3	7, 54	9, 52	11, 45	19, 44	20, 40	21, 37	27, 36	28, 32	30, 30	32, 28
B_4	10, 53	12, 51	14, 44	22, 43	23, 39	24, 36	30, 35	31, 31	33, 29	35, 27
B_5	13, 50	15, 48	17, 41	25, 40	26, 36	27, 33	33, 32	34, 28	36, 26	38, 24
B_6	16, 49	18, 47	20, 40	28, 39	29, 35	30, 32	36, 31	37, 27	39, 25	41, 23
B_7	17, 48	19, 46	21, 39	29, 38	30, 34	31, 31	37, 30	38, 26	40, 24	42, 22
B_8	21, 47	23, 45	25, 38	33, 37	34, 33	35, 30	41, 29	42, 35	44, 23	46, 21
B_9	24, 45	26, 43	28, 36	36, 35	37, 31	38, 28	44, 27	45, 23	47, 21	49, 19
B_{10}	28, 41	30, 39	32, 32	40, 31	41, 27	42, 24	48, 23	49, 19	51, 17	53, 15

Figure 1 Example instance with input Pareto sets A, B of size 10. The Minkowski sum has 100 elements. The Pareto sum C consists of 27 elements (marked green in the plot as well as in the matrix representation).

In this paper, we focus on the efficient computation of the filtered Minkowski sum of two-dimensional Pareto sets A, B. The Minkowski sum M is defined as the set of elements derived from pairwise addition of elements in A and B. However, the Minkowski sum often contains many dominated elements. In fact, it was proven in [12] that for A, B of size n, the set of non-dominated elements in M – which we refer to as Pareto sum of A, B – might have a size in $o(n)$. Thus, algorithms that first compute all elements of M and subsequently apply non-dominance filtering might be unnecessarily wasteful as they come with a running time and space consumption in $\Omega(n^2)$. The goal of the paper is to design practical algorithms for Pareto sum computation with output-sensitive space consumption, and to evaluate their performance on realistic inputs. As one particular use case of our methods, we will consider the bi-criteria route planning problem in road networks. There exists a plethora of algorithms to compute the set of Pareto-optimal paths between a given source and a target node in the network, see e.g. [8, 2]. The currently fastest methods rely on preprocessing. In particular, variants of contraction hierarchies (CH) have been proven to be very useful in this context [17, 19]. In a CH, the input graph is augmented with so called shortcut edges that represent sets of Pareto-optimal paths between their end points. The shortcuts store the costs of these paths in the form of Pareto sets. On query time, shortcuts are instrumented to decrease the search space size of a Pareto-Dijkstra run, resulting in significantly faster query times and reduced space consumption. In the preprocessing phase, the shortcuts are inserted incrementally. The base operation is to concatenate two shortcut or original edges $e = \{u, v\}$ and $e' = \{v, w\}$ to form a new shortcut $\{u, w\}$. The Pareto set of the new shortcut is the set of non-dominated elements in the Minkowski sum of the Pareto sets corresponding to e and e'. Thus, the preprocessing time crucially depends on an efficient Pareto sum computation. In [9],

it was discussed that computing the Minkowski sum and filtering all dominated elements in a naive fashion is too time-consuming. Therefore, filtering strategies were proposed that prune dominated elements. However, these strategies are not guaranteed to retrieve the Pareto sum but usually produce a superset thereof. Keeping supersets slows down later stages of the preprocessing as well as query answering. We will propose novel algorithms that allow for fast and exact Pareto sum computation.

1.1 Related Work

Non-dominance filtering in point sets is a well-studied task in computational geometry [6], also referred to as skyline or maxima computation. There exist output-sensitive algorithms for the two-dimensional case as e.g. the one proposed by Kirkpatrick and Seidel [11] with a running time of $\mathcal{O}(N \log k)$ where N denotes the size of the point set and k the size of the skyline. The basic idea is to first partition the input points into k sets of size $\approx N/k$ with non-overlapping ranges with respect to their x-coordinates. Then, the sets are processed individually in sorted order. As k is typically not known beforehand, a more intricate version of the algorithm allows to achieve the same asymptotic running time by starting with a coarse partition and refining it on demand as soon as a certain number of non-dominated points are identified. With a worst-case running time of $\mathcal{O}(N \log N)$ and close-to-linear running time for small k, this algorithm seems to be well-suited for Pareto sum computation. However, in our application we have $N = |M|$ where M is the Minkowski sum of the input sets A, B; and any approach that relies on access to M as a whole is bound to a running time and space consumption in $\Omega(n^2)$.

Using the interpretation of input elements as two-dimensional points, Pareto sum computation can also be reduced to computing the Minkowski sum of the orthogonal hulls of A and B (where both sets are augmented with a dummy point based on the maximum coordinate values in the respective set). The Minkowski sum of two convex polygons can be computed in linear time [15]. For non-convex inputs P, Q, the polygons are first decomposed into convex subpolygons $P_1, \ldots, P_s$ and $Q_1, \ldots, Q_t$. Then, the linear time algorithm is applied to all pairs P_i, Q_j, and finally the union of all partial results is computed. The running time depends on the applied decomposition technique and the number and complexity of the resulting subpolygons [1]. However, if P and Q are orthogonal convex hulls of size n, their convex decomposition cannot contain fewer than n subpolygons, and thus the approach needs to compute the union of $\Theta(n^2)$ partial solutions.

Recently, new algorithms for Pareto sum computation with the potential to achieve subquadratic running time and space consumption were proposed in [12]. The so called NonDomDC algorithm exploits the structure of the matrix that represents the Minkowski sum (see Figure 1). In particular, it makes use of the fact that columns in the matrix are Pareto sets themselves. Assuming the Pareto sum P_i of elements occurring in the first i columns is known, P_{i+1} can be computed by merging P_i and column $i+1$ and pruning dominated elements in $\mathcal{O}(|P_i| + n)$ time. Thus, with $P := \max_{i=1}^{n} |P_i|$ denoting the maximum size of an intermediate solution, the total running time is in $\mathcal{O}(Pn)$ and the space consumption is in $\mathcal{O}(n + P)$. However, this does not constitute an output-sensitive algorithm as the size of the intermediate Pareto sum can be significantly larger than the final result size. So even for small k, the algorithm might have cubic running time and quadratic space consumption. However, their experimental study demonstrates good performance in practice. Similar methods, as the box-based method proposed in [10], were shown to be outperformed.

Table 1 Running time and space consumption of different algorithms for Pareto sum computation. The input size is denoted by n and the output size by k.

algorithm		running time	space	
NonDomDC	(ND)	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	[12]
Kirkpatrick-Seidel	(KS)	$\mathcal{O}(n^2 \log k)$	$\Theta(n^2)$	[11]
Brute Force	(BF)	$\mathcal{O}(n^4)$	$\mathcal{O}(n+k)$	4.1
Binary Search	(BS)	$\mathcal{O}(n^3 \log n)$	$\mathcal{O}(n+k)$	4.2
Sort & Compare	(SC)	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n+k)$	4.3
Successive Binary Search	(SBS)	$\mathcal{O}(nk \log n)$	$\mathcal{O}(n+k)$	5.1
Successive Sweep Search	(SSS)	$\mathcal{O}(n \log n + nk)$	$\mathcal{O}(n+k)$	5.2

1.2 Contribution

In this paper, we consider the problem of efficient Pareto sum computation in theory and practice. First, we present an algorithm that has the ability to identify a subset of the Pareto sum C in linear time. This algorithm can be used as a preprocessing step for all other approaches for Pareto sum computation. Additionally, we show that for certain kinds of inputs, the algorithm already returns whole set C. Then, we present and thoroughly analyze several algorithms for Pareto sum computation with a special focus on achieving an output-sensitive space consumption. Table 1 provides an overview of the characteristics of our proposed algorithms as well as existing baseline approaches. In an extensive experimental study, we compare their scalability. We consider randomly generated data as well as real inputs that stem from bi-criteria route planning instances. For both input types alike, our output-sensitive successive sweep search proves to be the most efficient algorithm. This aligns well with our theoretical analysis, as it turns out, especially for large input sizes, that the Pareto sum C contains only a small fraction of the elements in the Minkowski sum.

2 Problem Definition

In this section, we formally define the notion of a Pareto sum and provide notation used throughout the paper.

▶ **Definition 1** (Domination). *Given two points $p, p' \in \mathbb{R}^2$, we say that p dominates p', or $p \prec p'$, if $p \neq p'$ and $p.x \leq p'.x$ as well as $p.y \leq p'.y$.*

▶ **Definition 2** (Pareto set). *A set $S \subset \mathbb{R}^2$ is a Pareto set if no point in S dominates another point in S, that is $\nexists s, s' \in S$ with $s \prec s'$.*

We always assume that Pareto sets are sorted in lexicographic order. We use S_i to refer to the element with rank i in set S.

▶ **Definition 3** (Minkowski sum). *Given two Pareto sets $A, B \subset \mathbb{R}^2$, their Minkowski sum $M = A \oplus B$ is defined as $M := \{a + b |\ a \in A, b \in B\}$.*

In a slight abuse of notation, we will use M to refer to the set of elements in the Minkowski sum as well as the matrix where $M_{ij} = A_i + B_j$.

▶ **Definition 4** (Pareto sum). *Let $A, B \subset \mathbb{R}^2$ be two Pareto sets of size n and let $M = A \oplus B$ denote their Minkowski sum. Then the Pareto sum C of A, B is defined as the set of all non-dominated points in M.*

Figure 1 illustrates the concepts of Minkowski and Pareto sums. Throughout the paper, we will use k to denote the size of C.

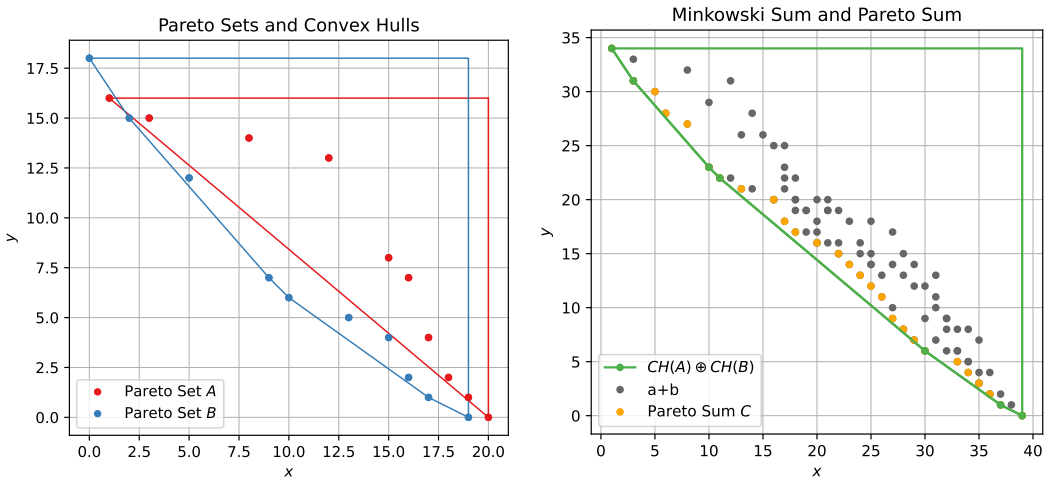


Figure 2 Left: Pareto sets A, B together with their convex hulls. Right: The Minkowski sum $CH(A) \oplus CH(B)$ of the convex hulls encloses all pairwise vector additions $a + b$ with $a \in CH(A)$ and $b \in CH(B)$. The respective vertices (green points) are a subset of the Pareto Sum C.

3 Minkowski Sum of Convex Hulls

In this section, we present an algorithm that for given Pareto sets A, B computes part of their Pareto sum in linear time. Thus, this algorithm can be used as an efficient preprocessing method before applying other (more costly) techniques.

For two convex polygons $P, Q \in \mathbb{R}^2$, their Minkowski sum $P \oplus Q$ is a convex polygon with at most $|P| + |Q|$ vertices and these vertices can be computed in linear time [15]. Let now A, B be sorted Pareto sets augmented with dummy points (x, y) where $x := \max_{s \in S} s.x$ and $y := \max_{s \in S} s.y$ for $S = A$ and $S = B$, respectively. We use $CH(A)$ and $CH(B)$ to refer to the convex hulls of these two sets. The following observation captures the connection between these convex hulls and the Pareto sum.

▶ **Observation 5.** *The vertices of the Minkowski sum $CH(A) \oplus CH(B)$ are a subset of the Pareto sum of A and B (excluding the dummy point sum).*

For a sorted Pareto set, its convex hull can be computed in linear time using Andrew's algorithm [3]. Then, using the linear time Minkowski sum algorithm on the two convex hulls and extracting the respective polygon vertices, we obtain a subset of the Pareto sum C, see Figure 2. If both A, B are convex, then this procedure already returns all of C. We thus get the following corollary.

▶ **Corollary 6.** *The Pareto sum of two convex, sorted Pareto sets can be computed in $\mathcal{O}(n)$.*

For non-convex A, B, we might only get part of the Pareto sum. However, as this step only takes linear time (assuming the Pareto sets are presorted), it can always be used as an initial step before applying other algorithms. We will discuss below in more detail how the knowledge of $C' \subset C$ can be exploited to decrease the practical running time of several of the algorithms we propose.

4 Base Algorithms

In this section, we discuss three simple base algorithms for Pareto sum computation along with engineering concepts for their acceleration and space consumption reduction. The algorithms all proceed by checking for each element $p \in M$ whether there exists $p' \in M$ that dominates p. If there is no such p', the point p is added to the Pareto sum C. The only difference between the algorithms is the implementation of the dominance check.

4.1 Brute Force (BF)

The easiest way to check for a point $p \in M$ whether it is non-dominated is by pairwise comparison to all other elements in M. This dominance check takes $\mathcal{O}(|M|)$ time per point, accumulating to a total time of $\mathcal{O}(|M|^2) = \mathcal{O}(n^4)$. As the elements M_{ij} can be computed on demand, the space consumption is linear in the input size n and the output size k.

▶ **Corollary 7.** *The BF algorithm runs in $\mathcal{O}(n^4)$ time using $\mathcal{O}(n+k)$ space.*

4.2 Binary Search (BS)

To decrease the time needed for the dominance check, we take the structure of M into account. Based on the assumption that A and B are sorted and that M_{ij} is defined as $A_i + B_j$, we have the property that each column (and each row) of the matrix M forms a sorted Pareto set on its own. Thus, if we want to check whether column M_j contains an element dominating p, we simply have to find the entry M_{ij} with the largest index i such that $M_{ij}.x \leq p.x$ as well as the entry $M_{i'j}$ with the smallest index i' such that $M_{i'j}.y \leq p.y$. Then all elements in M_j with a row index in $[i', i]$ dominate p (or are equal to p). Accordingly, the dominance check for M_j boils down to evaluating whether $i' \leq i$ holds. These two indices can each be identified via a binary search over the respective coordinates in the column. Hence the dominance check time per column is in $\mathcal{O}(\log n)$, resulting in a total time of $\mathcal{O}(n \log n)$ per element in M. Entries of M that need to be accessed can be computed on demand.

▶ **Corollary 8.** *BS runs in $\mathcal{O}(n^3 \log n)$ time using $\mathcal{O}(n+k)$ space.*

To reduce the practical running time of the BS algorithm, we propose the following engineering techniques.

Pruning. Whenever we identify a non-dominated point p and add it to C, we can also compute all entries in M dominated by p in time $\mathcal{O}(n \log n)$, again with the help of two binary searches per column. For those points, dominance does not need to be checked again. However, if we simply store a flag for each entry in M whether it needs to be further considered or not, the space consumption increases to n^2. Instead we can store for each column the set of intervals of dominated points in an interval tree. The number of intervals per column is upper bounded by k. Then, for a point $p = M_{ij}$ we can query the interval tree in time $\mathcal{O}(\log k)$ to see whether the point lies in a dominated region. Intervals can also be added or merged within the same time. However, the space consumption would still increase to $\mathcal{O}(nk)$. To keep the space consumption linear, one might only want to store a constant number of intervals per column (e.g. only the largest one) and then merge or replace intervals if possible or needed. If pruning is applied, we can also disregard fully dominated columns in the binary searches of the remaining elements.

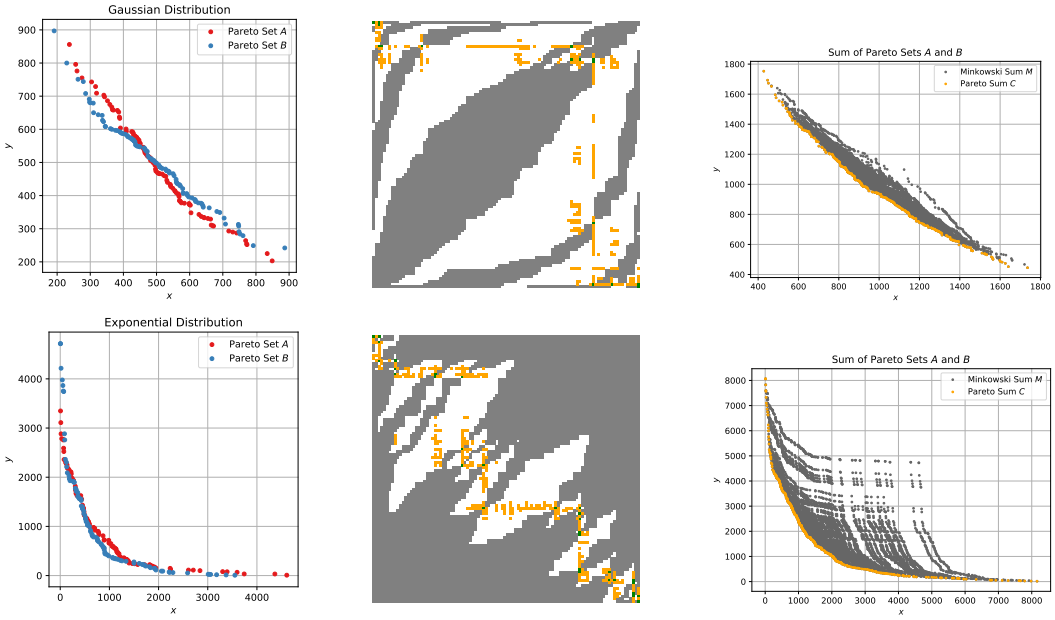


Figure 3 Input Pareto sets following different kinds of distributions (left) and schematic depiction of the corresponding Minkowski matrix M (middle). Green dots indicate entries in M that are points on the Minkowski sum of the convex hulls of A and B, black dots indicate entries that are dominated by the green ones, and orange dots encode the remaining elements of the Pareto sum which then together dominate the white dots. In the right images, the Minkowski sum and the Pareto sum are illustrated based on point coordinates.

Priority Binary Search (PBS). As soon as dominance checks might be avoided for some of the elements based on the pruning techniques described above, the order in which the points are considered impacts the running time. Identifying points that dominate many other points early on can significantly reduce the total number of checks. For that purpose, we will use the preprocessing step described in Section 3 to get an initial set of points $C' \subset C$. We can directly exclude any points dominated by the points in C'. Furthermore, we conjecture that points in the same rows or columns as the points of C' occupy in M are likely to be also part of C. Thus, we give priority to these points in our search. Figure 3 shows some visual support for this hypothesis.

4.3 Sort & Compare (SC)

Given a sorted set of points, extracting the set of non-dominated points can be accomplished in constant time per point. The smallest element is always added to C. For each other element in sorted order, we check whether it is dominated by (or equal to) the currently last element in C. If that is not the case, the element is added to C.

Computing and sorting M as a whole takes time $\mathcal{O}(n^2 \log n)$ and requires quadratic space. But we can exploit the structure of M to improve the space consumption to linear as follows: We use a min-heap data structure and initialize it with the first row of M. Each element in the heap remembers its position in M. When we extract the min element M_{ij} from the heap and C is empty so far, we add the element to C. Otherwise, we compare M_{ij} to the element added to C last. If M_{ij} is not dominated, we also add it to C. In any case, we add its column successor M_{i+1j} to the heap (as long as $i < n$). As each column is a sorted Pareto

set in itself, we know that M_{i+1j} has to have larger x-value than M_{ij}. Thus, we extract the elements from the heap exactly according to their global lexicographic order. As the heap never contains more than n elements, its space consumption is in $\mathcal{O}(n)$ and the heap operations take $\mathcal{O}(\log n)$ per round.

In conclusion, the heap-based variant has the same asymptotic running time as the one where we fully compute and sort M, but a significantly reduced space consumption.

▶ **Corollary 9.** *SC runs in $\mathcal{O}(n^2 \log n)$ time using $\mathcal{O}(n+k)$ space.*

5 Output-Sensitive Algorithms

If the Pareto sum C contains (almost) all elements of the Minkowski sum M, a quadratic running time is needed already to report C. In this case, the running time of SC is asymptotically optimal up to logarithmic factors. However, in case C is small, subquadratic running times might be possible. We will present two output-sensitive algorithms in this section that have a running time asymptotically faster than SC for $k \in o(n)$ or $k \in o(n \log n)$, respectively. Both algorithms detect the elements in C successively. This is a well-established paradigm for output-sensitive skyline computation, see e.g. [13, 18]. However, known algorithms rely on the explicit availability of the point set to construct an efficient search data structure. Based on the following lemma, we will design successive algorithms that do not need access to M as a whole.

▶ **Lemma 10.** *Let A, B be two Pareto sets and $c, c' \in C$ two elements of their Pareto sum with $c.x < c'.x$. Then the lexicographically smallest element m in M that dominates $(c'.x - \varepsilon, c.y - \varepsilon)$ for $\varepsilon > 0$ is also part of C (if such an element exists).*

Proof. We first argue that for any $m \in M$, the smallest point $p \in M$ dominating m (or being equal to m) is part of the Pareto sum C. Assume otherwise for contradiction. Then there is a point $p' \in C$ that dominates p and thus also m. But in this case p' is smaller than p which contradicts the choice of p as smallest element to dominate m.

Now, if there is any point $m \in M$ that dominates the dummy point $(c'.x - \varepsilon, c.y - \varepsilon)$ the above argumentation applies. ◀

Clearly, M_{11} and M_{nn} are always part of the Pareto sum, as those are the points with smallest global x-value and y-value, respectively. All other elements in C must have an x-value in the open interval $(M_{11}.x, M_{nn}.x)$. Thus, if we have an oracle that returns the smallest point $m \in M$ (with respect to lexicographic ordering) in a given range $[x_{\min}, x_{\max}) \times [y_{\min}, y_{\max})$, we can compute C based on Lemma 10 as follows. We initialize $C = M_{11}, M_{nn}$ and $x_{\min} = M_{11}.x, x_{\max} = M_{nn}.x, y_{\min} = M_{nn}.y, y_{\max} = M_{11}.y$. Then we query the oracle to get the smallest point m in the respective range. If such a point does not exist, we abort. Otherwise we add the point m to C and set $x_{\min} = m.x, y_{\max} = m.y$ before repeating the process. Figure 4 illustrates the core concept.

Thus, the algorithm discovers the points in C one-by-one in increasing order of their x-values (except for M_{nn} which is known from the start) using k calls to the range-minimum oracle. A naive oracle implementation would be to check all points in M for containment in the range and to keep track of the minimum among them. Then each call to the oracle costs $\mathcal{O}(n^2)$ and the overall running time of the successive algorithm would be $\mathcal{O}(n^2 k)$. Next, we describe how to implement the oracle more efficiently.

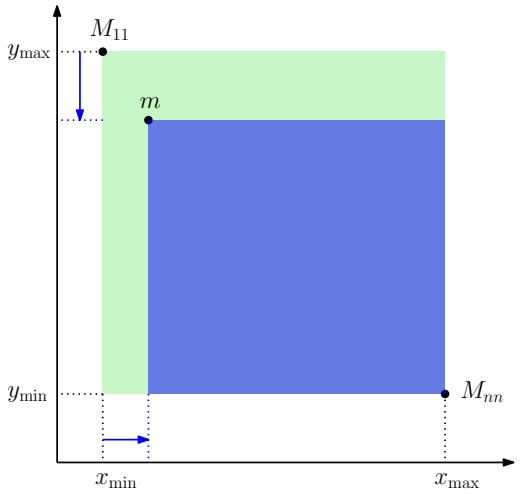


Figure 4 Initial search range (green rectangle) spanned by M_{11} and M_{nn}. The range-minimum element m then leads to a reduction of $y_{\max}$ and an increase of $x_{\min}$ (blue arrows) which tightens the search range for the next element of C.

5.1 Successive Binary Search (SBS)

In the BS approach described in Section 4.2, we use two binary searches per column of M to check for a query point $m \in M$ whether an element dominating m exists in that column. We can use the same concept to implement a range-minimum oracle: For each column, we identify via binary searches the first position f_x with an x-coordinate larger or equal to $x_{\min}$, the last position l_x with an x-coordinate smaller than $x_{\max}$, the first position f_y with a y-coordinate smaller than $y_{\max}$, and the last position l_y with a y-coordinate larger or equal to $y_{\min}$. If $[f_x, l_x] \cap [f_y, l_y] \neq \emptyset$, we return $\max(f_x, f_y)$. The entry at that position is the smallest point dominating m in the column. Keeping track of the smallest returned point over all columns provides the desired result in $\mathcal{O}(n \log n)$ per oracle call.

▶ **Corollary 11.** *Successive BS runs in $\mathcal{O}(nk \log n)$ time using $\mathcal{O}(n + k)$ space.*

For standard BS we can use the Minkowski sum of the convex hulls of A and B to already identify a subset C' with size $k' \leq k$ of the Pareto sum C without the need of binary searches. We can apply the same initialization here and then simply use the successive algorithm independently in each of the $k' - 1$ ranges induced by any two consecutive points in C' (in sorted order). By that, the number of oracle calls increases to at most $k + k' - 1 < 2k$ as in each of the $k - 1$ ranges the respective last oracle call will return no point. This does not affect the asymptotic running time, though. But it allows to conduct up to $k' - 1$ oracle calls in parallel.

To further foster parallelization, we can weaken the oracle requirement to always return the smallest point in a given range to the requirement to return any non-dominated point in the range. The new point then splits the previous range into two subranges that can be queried independently. With that, the oracle might be called up to $k + 2k' - 2 < 3k$ times as now in each of the $k' - 1$ ranges the call to its leftmost induced subrange and the call to its rightmost induced subrange will return no point. Again, the asymptotic running time remains unaffected. To implement the weaker oracle, we propose the so called Cascading BS (CBS) algorithm. Again, we consider the columns one after each other. But now, as soon as we find a column entry p in the given query range, we update the range immediately and then search for a point dominating p in the remaining columns. If we find such a point,

we immediately update again. Note that it can never happen that a point p' in an already visited column dominates p, as then we would have selected one point from said column to tighten the query range and there can never be two points in one column dominating one another. Thus, after we considered all columns, we can safely add the current point p to C.

For an example of the difference between SBS and CBS, consider the matrix in Figure 1 and assume the current search range is $[14, 53) \times [15, 44)$. SBS discovers the element $A_6 + B_1 = (15, 43)$ next as this is the point with smallest x-value that dominates the dummy point $(53 - \varepsilon, 44 - \varepsilon)$. CBS, however, first detects the point $A_1 + B_{10} = (28, 41)$ as it already dominates the dummy point. It then proceeds by trying to find an element that dominates $(28, 41)$. It thus detects $A_3 + B_5 = (17, 41)$ next and tightens the search range accordingly. As this is a Pareto sum point, no further dominating elements are found in the remaining columns and $(17, 41)$ is returned and added to C. The search range is then split into $[14, 17) \times [41, 44)$ and $[17, 53) \times [15, 41)$, which can be processed independently.

5.2 Successive Sweep Search (SSS)

To improve the oracle time of SBS, we observe that the binary searches in the columns are somewhat redundant. If M was fully available, we could apply fractional cascading [5] to the column vectors. This would reduce the running time to compute the positions f_x, l_x, f_y, l_y in all columns from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$. Thus, the k oracle calls cost $\mathcal{O}(nk)$. Unfortunately, computing M and the data structure for fractional cascading requires space and time in $\Theta(n^2)$. Fortunately, we can also achieve linear oracle time without the need to access M as a whole. Based on the structure of M, we know that for an entry M_{ij} all elements M_{st} with $s \geq i$ and $t \geq j$ have a larger x-coordinate than M_{ij} but a smaller y-coordinate. Vice versa, all elements in M_{st} with $s \leq i$ and $t \leq j$ have a smaller x-coordinate than M_{ij} but a larger y-coordinate. This implies, for example, that the position of f_x in some column cannot be larger than the position of f_x in the neighboring column to its left. Similar relationships hold for the positions of l_x, f_y and l_y in neighboring columns. Accordingly, we can find the respective column values by a single left-to-right sweep, where the search path forms a monotone staircase structure and is thus bounded in length by $2n$.

Even better, we can have a single unified sweep to find the range-minimum m in linear time: We start at M_{n1}, that is, the last entry of the first column. Whenever we enter a new column j, we apply upwards linear search in that column until we reach an entry M_{ij} where either $M_{ij}.x > x_{\min}$ and $M_{i-1j}.x \leq x_{\min}$ or where $M_{ij}.y < y_{\max}$ and $M_{i-1j}.y \geq y_{\max}$. Thus, we get $i = \max(f_x, f_y)$; except if the entry we start from already has a too small x-value or a too large y-value or both which means that the column contains no point in the query range. In the former case, we check whether M_{ij} is contained in the range. If the check is passed, M_{ij} is a valid candidate for the range-minimum m. We keep track of the smallest viable candidate over the course of the algorithm. We then go from element M_{ij} to its right neighbor M_{ij+1} and proceed with the new column as described above. After processing the last column, we return the current m as the range-minimum element.

▶ **Lemma 12.** *The sweep algorithm computes the smallest $m \in M$ in a given range in $\mathcal{O}(n)$.*

Proof. To prove correctness, we need to argue that for a column j entered at row i and exited at row $i' \leq i$, the range-minimum m can not be an entry M_{i^*j} with $i^* < i'$ or $i^* > i$.

Clearly, checking elements in column j with a row index smaller than i' cannot give us any viable candidates, as either their respective x-value is too small or their y-value is too large by definition. Hence we only need to consider $i^* > i$. If M_{ij} is in the query range, then the entries in column j with $i^* > i$ cannot constitute the range-minimum as they all have an x-value larger than that of M_{ij}. If M_{ij} is not in the query range, we have the following cases:

- $M_{ij}.x > x_{\max}$. But then $M_{i^*j}.x > x_{\max}$ holds as well.
- $M_{ij}.x < x_{\min}$ or $M_{ij}.y > y_{\max}$. This case only occurs if $i = n$. Thus there is no $i^* > i$.
- $M_{ij}.y < y_{\min}$. But then $M_{i^*j}.y < y_{\min}$ holds as well.

Accordingly, if column j contains the range minimum it needs to be an element with a row index in $[i', i]$. If $M_{i'j}$ is in the range, it is clearly the best candidate in column j for the range-minimum m. If $M_{i'j}$ is not in the query range, then the same applies to all entries in the same column with larger row index as argued above. Thus, it is sufficient to check $M_{i'j}$ for each column j. The running time is determined by the number of elements in M that are accessed. As the interval of elements checked for each column only overlaps with the intervals of all columns to its left in a single row index, at most $2n$ elements in M are considered in total. ◀

Based on this sweep search (SS) oracle, we now get a successive algorithm with better running time than SBS.

▶ **Corollary 13.** *Successive SS runs in $\mathcal{O}(n \log n + nk)$ using $\mathcal{O}(n + k)$ space.*

In fact, if $k \in o(\log n)$, the running time is dominated by the initial sorting step of the elements in A, B. For $k \in o(n)$, we achieve a subquadratic running time.

For acceleration of sweep search in practice, we observe that if we enter a column at row i and confirm for some value $i' < i$ that $M_{i'j}$ is still feasible with respect to $x_{\min}$ and $y_{\max}$, we do not have to check intermediate rows to get the correct range-minimum by virtue of Lemma 12. Similarly, if we have not found any M_{ij} in column j with $M_{ij}.x \geq x_{\min}$ and $M_{ij}.y < y_{\max}$ and the same inequalities apply to some $M_{ij'}$ with $j' > j$, we do not need to check intermediate columns for range-minimum candidates. Thus, in both cases we can introduce a skip threshold $\Delta > 1$ and check for $i' = i - \Delta$ or $j' = j + \Delta$, respectively, whether the necessary conditions apply. If that is the case we skip intermediate rows or columns and then try to skip ahead again. If skipping is no longer possible, we simply fall back to linear search. Accordingly, in the worst case, we check at most one superfluous element for each row and column. This does not increase the asymptotic running time of the oracle but might reduce its running time in practice if skipping is successful.

Furthermore, similar to CBS, we also propose Cascading Sweep Search (CSS). Here again, whenever we found a temporary range-minimum candidate m we immediately tighten the search range to enforce that further candidates need to dominate the current m to be considered. The sweep search then also guarantees to return an element of the Pareto sum C. The Minkowski hull preprocessing and the split of search intervals to foster parallelization as described for CBS can be applied here as well.

6 Experimental Evaluation

We implemented the seven algorithms for Pareto sum computation listed in Table 1 in C++: The two existing approaches, namely the Kirkpatrick-Seidel algorithm (KS) [11] and NonDomDC (ND) [12], the three base algorithms (BF, BS, SC), and the two successive algorithms (SBS, SSS). For KS, we implemented the simpler (and thus faster) variant, where the output size k (used for partitioning) is given as an input. We simply compute k with one of our other algorithms and then feed the result into KS. For ND, we actually implemented two variants described in [12]: In the first variant (described in more detail in Section 1.1), one always merges the current result with the next column (Sequential ND, SND). In the second variant, columns are combined in a MergeSort like fashion (Doubling ND, DND). As benchmark data we use randomly generated inputs as well as real inputs. Both types of data sets are described in more detail below. All experiments were conducted on a single core of a 3.4 GHz AMD Ryzen Threadripper 1950X 16-core processor with 126 GB of RAM.

6.1 Results for Generated Data

To generate Pareto sets, we take two random samples of n unique values from a given range. The first sample is sorted increasingly and represents the x-coordinates within the Pareto set. The second sequence is sorted decreasingly and represents the y-coordinates. We consider uniform, Gaussian and exponentially distributed samples over the range $[0, n]$. In addition, we investigate A and B where the respective x-coordinates are drawn from vastly different intervals, namely with upper bounds $\sqrt{n}$ and n^2. We call this a shifted distribution. Figures 3 and 5 show example instances for each input type. Running times are always averaged over 100 generated instances per tested value of n.

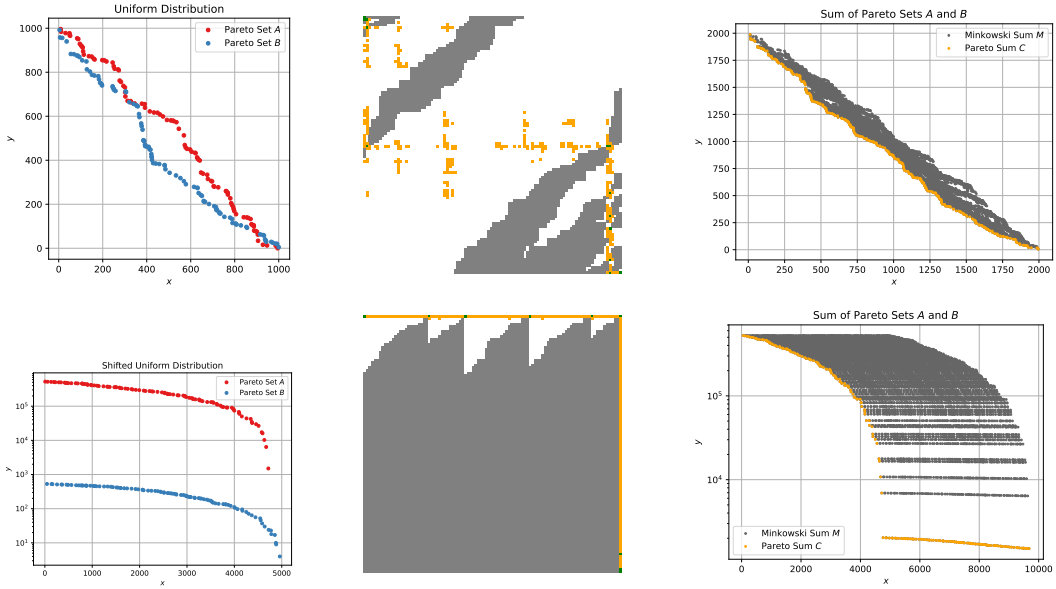


Figure 5 Example instances for uniform and shifted uniform point distributions For the latter, note the logscale of the y-axis. The color coding is the same as in Figure 3.

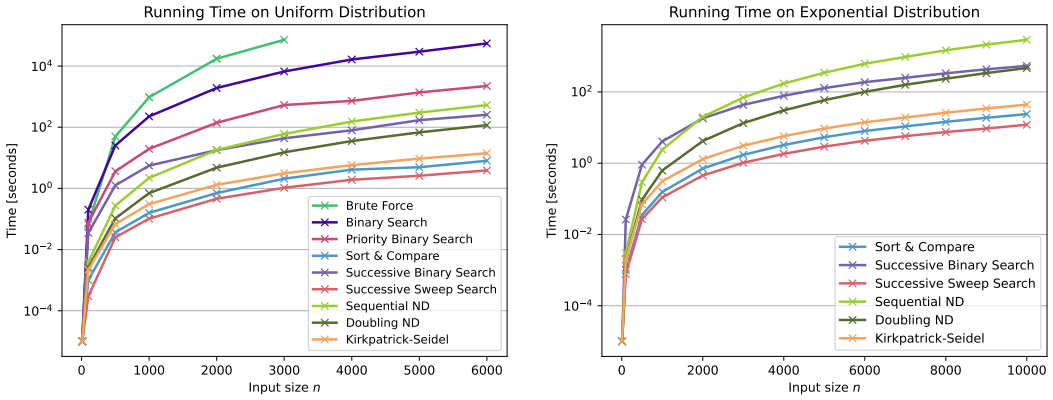


Figure 6 Average running times of all algorithms on uniform distributions (left) and of the top six algorithms on exponential distributions (right). Note the logscale of the y-axis.

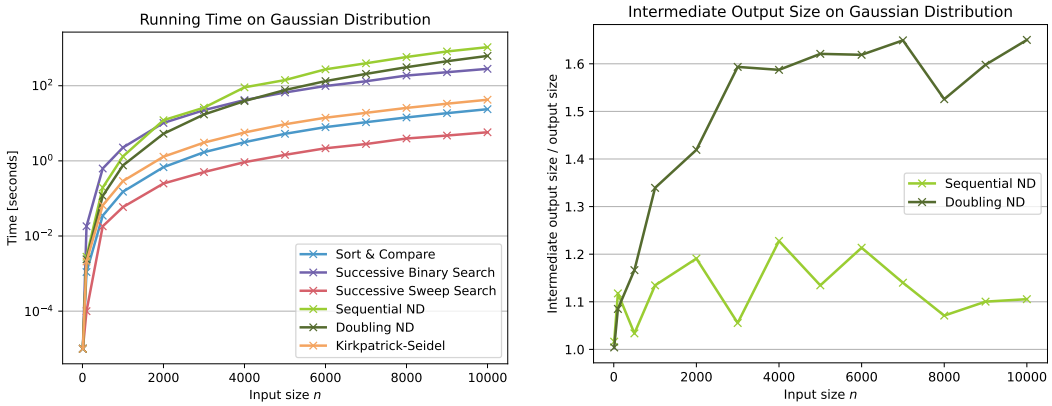


Figure 7 Left: Average running times of selected algorithms on Gaussian distribution. Right: Intermediate output size of the Sequential ND and Doubling ND algorithms.

Figure 6, left, shows the running times of all algorithms on uniformly distributed instances. In line with our theoretical analysis, the Brute Fore approach is by far the slowest. Instances larger than $n = 3000$ were not tested as those already took over an hour. The engineered Binary Search algorithm (PBS) is faster than BS by an order of magnitude but not as fast as the other competitors. Note that we used the variant here that guarantees output-sensitive space consumption by storing the borders of at most on block of dominated elements per column. In compliance with the experimental results in [12], we see that DND is faster than SND. However, they are both slower than the Kirkpatrick-Seidel and the Sort & Compare algorithm, which exhibit very similar running times. SSS is faster than the second best algorithm, Sort & Compare, by a factor of 2-5. On exponential distributions the results are similar, see Figure 6, right. But the ND variants perform slightly worse. On Gaussian distributions, the ND variants are about two orders of magnitude slower than SSS and even scale worse than SBS, see Figure 7, left. The reason for this behavior is investigated in Figure 7, right, which depicts the intermediate solution sizes of the ND algorithms. On uniform and exponential distributions, the space overhead is less than 1%. However, on Gaussian distributions, SND and DND require 20% and 65% more space than the output (and our output-sensitive algorithms), respectively. Also note that even if intermediate sizes are not much larger than the final size, it might be that the maximum intermediate size is reached early and persists close to that value, thereby increasing the running times of the individual column merge steps. Thus, the ND algorithms are both very sensitive to the distribution of the input points and the position of the Pareto sum points within the matrix. In contrast, the performance of our sweep algorithms depends primarily on the size of the output, which was within $4n$ across all tested instances and distributions.

On uniform, Gaussian and exponential distributions, our engineered SSS variant with Δ-skipping had little impact. On shifted distributions, however, this concept proved to be very effective due to the Pareto sum points being mostly located either in the first few rows or the last few columns of the matrix (see Figure 5), and Pareto sum sizes being small in general. Using $\Delta = \sqrt{n}$, we achieved speeds-ups of two orders of magnitude over all other approaches on instances with $n = 10000$.

Table 2 Experimental results for BCH computation on three road networks of different size. The table shows the input graph sizes, the number of edges in the augmented graph (original + shortcuts), the number of non-trivial Pareto sum computations, as well as the running times for conducting these computations with four different algorithms. The final row shows the preprocessing time spent on operations other than Pareto sum computation.

	ROAD1	ROAD2	ROAD3
#Nodes	349479	1246440	3835238
#Edges	720363	2612260	8037228
#BCH-edges	1325259	4981957	15703653
#PS computations	899390	4424857	48844050
Kirkpatrick-Seidel	32.09 s	1234.15 s	>24 h
Sort & Compare	13.45 s	587.77 s	47374.67 s
Doubling ND	19.03 s	454.42 s	37447.61 s
Successive Sweep Search	4.83 s	129.62 s	9668.33 s
Additional preprocessing	7.23 s	54.66 s	1237.58 s

6.2 Results for Real Data

As a real-world application of Pareto sum computation, we consider bi-criteria route planning in road networks. Here, given an input graph $G(V, E)$ and costs $c_1, c_2 : E \to \mathbb{R}^+$, the goal is to either compute all Pareto-optimal paths with respect to c_1, c_2 between two nodes $s, t \in V$, or the path optimal with respect to one cost while not exceeding a budget on the other (also known as the constrained shortest path problem). To accelerate query answering, a bi-criteria contraction hierarchy (BCH) data structure can be used. In the preprocessing phase of a BCH, the input graph is augmented with additional edges, also called shortcuts. The shortcut insertion is guided by a node permutation $\pi : V \to \{1, \dots, n\}$. For nodes $u, w \in V$, a shortcut $\{u, w\}$ is inserted if and only if there exists a simple path from u to w on which no node has a higher π value than $\max(\pi(u), \pi(w))$. The shortcut represents all simple paths p between u and w with that property. For each Pareto-optimal p, the respective cost tuple $(c_1(p), c_2(p))$ should be assigned to the shortcut. To compute these Pareto sets for all shortcuts in an efficient manner, a bottom-up approach is used. Let u be the inner node on a path p from u to w with maximum π-value. If the Pareto sets A and B of the shortcuts $\{u, v\}$ and $\{v, w\}$ are known, respectively, the Pareto set of $\{u, w\}$ is the Pareto sum C of A and B. If there are multiple paths p, the final Pareto set of $\{u, w\}$ is formed by the non-dominated elements of the union of all these Pareto sums. The non-dominated union of two Pareto sets can be computed in linear time by merging the presorted sets to obtain the sorted union and then applying the simple non-dominance check as described in the SC approach. In the final BCH, queries can be answered with a bi-directional Pareto-Dijkstra run that relaxes shortcut edges instead of many original edges whenever possible. This significantly reduces the search space and allows to answer queries orders of magnitude faster [17, 9].

In our experiments, we use test graphs extracted from OpenStreetMap with Euclidean distance and positive height difference as edge costs (in compliance with [17]). Based on our results on generated data, we use the best four algorithms (KS, SC, DND and SSS) for Pareto sum computation. Note that Pareto sets A and B do not necessarily have the same size here, but all proposed Pareto sum computation algorithms can be easily adapted. Table 2 shows the characteristics of the three road network instances we considered in our experiments and the outcomes. The number of Pareto sum computations in the preprocessing phase of the BCH reported in the table excludes trivial inputs where either A or B has size 1. We

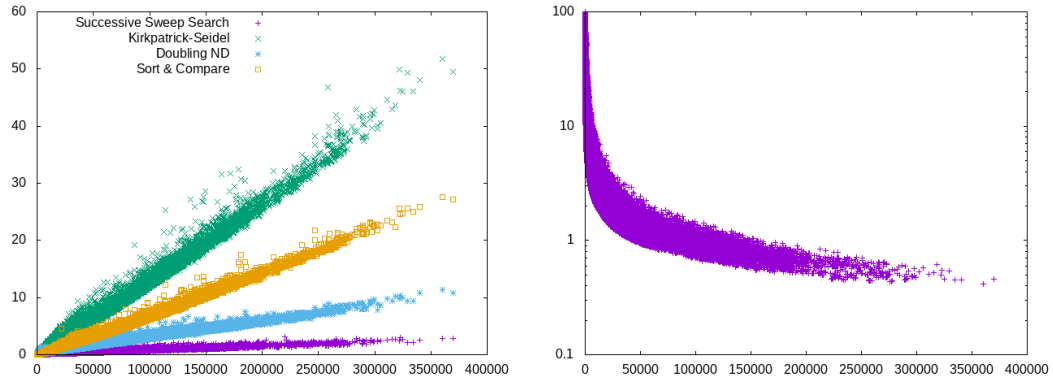


Figure 8 Detailed results for the ROAD2 instance. Left: Running times in seconds per Pareto sum computation in dependency of the size of the Minkowski sum. Right: Pareto sum size as percentage (logscale) of the size of the Minkowski sum.

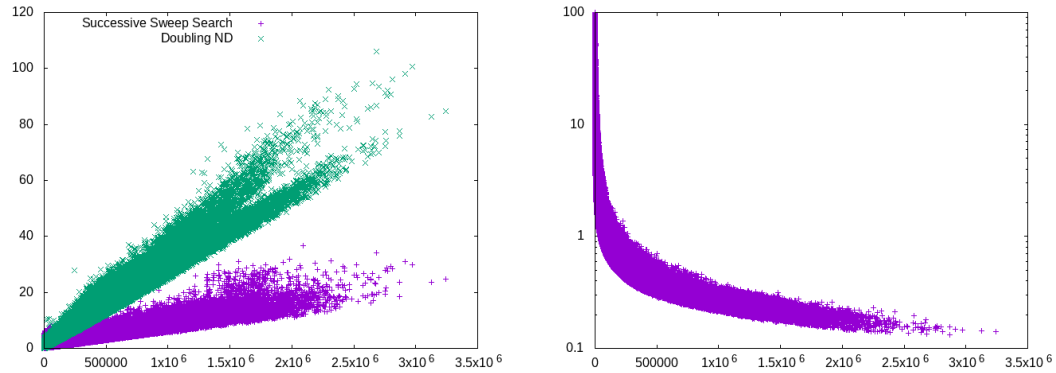


Figure 9 Detailed results for the ROAD3 instance. Left: Running times in seconds per Pareto sum computation in dependency of the size of the Minkowski sum. Right: Pareto sum size as percentage (logscale) of the size of the Minkowski sum.

observe that the time spent on non-trivial Pareto sum computations dominates the overall preprocessing time, especially on larger networks. There are significant differences in running time between the algorithms we tested, though. On all instances, SSS is the fastest approach. It is roughly an order of magnitude faster than the KS algorithm which fully computes and stores M. With KS, we could not compute a BCH data structure within a day on our largest instance with about 4 million nodes. Interestingly, in contrast to the experiments on generated data, DND outperforms SC. Figure 8 shows the running times for all individual Pareto sum computations as well as the size of the respective results for the ROAD2 instance. We observe that the larger the Minkowski sum M, the smaller the relative output size. This explains why SSS consistently outperforms the other approaches, especially on larger inputs. Figure 9 shows results for the two best algorithms, DND and SSS, on ROAD3. Here, $|M|$ was up to $3 \cdot 10^6$ and the percentage of elements in the Pareto sum C even approached 0.1. This is very beneficial for the SSS algorithm as the smaller the output size the fewer range minimum oracle calls are needed.

Furthermore, we used DND and SSS in query answering to combine Pareto sets in the bi-directional Pareto-Dijkstra run whenever the forward and the backward search meet. On the ROAD3 instance, a speed-up of up to 5 over DND was achieved when using SSS.

7 Conclusions and Future Work

We introduced scalable algorithms for Pareto sum computation which avoid the computation of the whole Minkowski sum. Our successive sweep search algorithm was shown to perform best across all instances, generated or real, while guaranteeing an output-sensitive space consumption. One direction for future work is to carefully parallelize all discussed algorithms. We also implemented and tested the cascading sweep search variant, which enables parallel successive search, and observed that the sequential running time matches that of successive sweep search while splitting the search ranges in many subranges which could be processed in parallel. Furthermore, even in a parallel implementation, the sweep search algorithm keeps its output-sensitive space consumption. Another direction for future work is the consideration of higher-dimensional input points. While some algorithms are easily generalizable, novel range minimum oracles need to be designed for the successive algorithms to work.

References

1 Pankaj K Agarwal, Eyal Flato, and Dan Halperin. Polygon decomposition for efficient construction of minkowski sums. *Computational Geometry*, 21(1-2):39–61, 2002.

2 Saman Ahmadi, Guido Tack, Daniel D Harabor, and Philip Kilby. Bi-objective search with bi-directional a*. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, pages 142–144, 2021.

3 Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. In *Information Processing Letters*, volume 9.5, pages 216–219. Elsevier, 1979.

4 Christian Artigues, Marie-José Huguet, Fallou Gueye, Frédéric Schettini, and Laurent Dezou. State-based accelerations and bidirectional search for bi-objective multi-modal shortest paths. *Transportation Research Part C: Emerging Technologies*, 27:233–259, 2013.

5 Bernard Chazelle and Leonidas J Guibas. Fractional cascading: A data structuring technique with geometric applications. In *Automata, Languages and Programming: 12th Colloquium Nafplion, Greece, July 15–19, 1985*, pages 90–100. Springer, 2005.

6 Wei-Mei Chen, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. Maxima-finding algorithms for multidimensional samples: A two-phase approach. *Computational Geometry*, 45(1-2):33–53, 2012.

7 Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-spektrum*, 22:425–460, 2000.

8 Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014. Proceedings 13*, pages 111–122. Springer, 2014.

9 Stefan Funke and Sabine Storandt. Personalized route planning in road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2015.

10 Antoine Kerbérénès, Daniel Vanderpooten, and Jean-Michel Vanpeperstraete. Computing efficiently the nondominated subset of a set sum. *International Transactions in Operational Research*, 2022.

11 David G Kirkpatrick and Raimund Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of the First Annual Symposium on Computational Geometry*, pages 89–96, 1985.

12 Kathrin Klamroth, Bruno Lang, and Michael Stiglmayr. Efficient dominance filtering for unions and minkowski sums of non-dominated sets. *Available at SSRN 4308273*, 2022.

13 Jinfei Liu, Li Xiong, and Xiaofeng Xu. Faster output-sensitive skyline computation algorithm. *Information Processing Letters*, 114(12):710–713, 2014.

14 Thibaut Lust and Daniel Tuyttens. Variable and large neighborhood search to solve the multiobjective set covering problem. *Journal of Heuristics*, 20:165–188, 2014.

15 de Berg Mark, Cheong Otfried, van Kreveld Marc, and Overmars Mark. *Computational geometry algorithms and applications*. Spinger, 2008.

16 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(3):495–522, 2019.

17 Sabine Storandt. Route planning for bicycles—exact constrained shortest paths made practical via contraction hierarchy. In *Twenty-second international conference on automated planning and scheduling*, 2012.

18 Chih-Chiang Yu, Wing-Kai Hon, and Biing-Feng Wang. Improved data structures for the orthogonal range successor problem. *Computational Geometry*, 44(3):148–159, 2011.

19 Han Zhang, Oren Salzman, Ariel Felner, TK Satish Kumar, Carlos Hernández Ulloa, and Sven Koenig. Efficient multi-query bi-objective search via contraction hierarchies. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, 2023.