Hypervolume Subset Selection with Small Subsets

Benoît Groz  
Silviu Maniu  
LRI, Université Paris-Sud, Université Paris-Saclay, Orsay, France

https://doi.org/10.1162/evco_a_00235

Abstract

The hypervolume subset selection problem (HSSP) aims at approximating a set of \( n \) multidimensional points in \( \mathbb{R}^d \) with an optimal subset of a given size. The size \( k \) of the subset is a parameter of the problem, and an approximation is considered best when it maximizes the hypervolume indicator. This problem has proved popular in recent years as a procedure for multiobjective evolutionary algorithms. Efficient algorithms are known for planar points (\( d = 2 \)), but there are hardly any results on HSSP in larger dimensions (\( d \geq 3 \)). So far, most algorithms in higher dimensions essentially enumerate all possible subsets to determine the optimal one, and most of the effort has been directed toward improving the efficiency of hypervolume computation. We propose efficient algorithms for the selection problem in dimension 3 when either \( k \) or \( n - k \) is small, and extend our techniques to arbitrary dimensions for \( k \leq 3 \).

Keywords

Hypervolume subset selection, hypervolume contributions, minimal triples in weighted hypergraphs.

1 Introduction and Related Work

1.1 Context: The Hypervolume Subset Selection Problem

The hypervolume indicator of a set of points measures the subspace dominated by the respective set. It is one of the possible measures for the quality of a Pareto set: a set (Pareto or not) dominating another set will have a larger indicator. The hypervolume indicator has thus been used to assess the quality of solutions in multi-objective evolutionary optimization algorithms (Zitzler and Thiele, 1999), and to guide their search toward desirable solutions (see, e.g., Bringmann and Friedrich, 2010 or Kuhn et al., 2016 and references therein). Many such optimizers approximate a Pareto front by iteratively removing the point that contributes least to the hypervolume. This greedy approach is suboptimal (Beume, Naujoks et al., 2009; Bringmann and Friedrich, 2010) but relatively fast as long as hypervolume computations are efficient. Alternatively, the problem we focus on aims at selecting directly the optimal subset of points: the hypervolume subset selection problem (HSSP\( (n,d,k) \), or HSSP for short) takes as input a set \( D \) of \( n \) points in \( \mathbb{R}^d \) together with an integer \( k \), and returns the subset of \( k \) points from \( D \) maximizing the hypervolume indicator. The problem can also be approached symmetrically: for large \( k \) the problem looks for the \( k' = n - k \) points which minimize the decrease of the hypervolume indicator of the remaining points.
1.2 State of the Art

The hypervolume of a set of points can be computed in $O(n \log n)$ when $d \leq 3$, which is optimal (Beume, Fonseca et al., 2009), but computing the hypervolume (which can be viewed as a restricted case of the Klee Measure Problem) is #P-hard for arbitrary $d$ (Bringmann and Friedrich, 2012), with the best algorithms currently running in $\tilde{O}(n^{d/3})$ (Chan, 2013).

Algorithms returning the optimal HSSP solution for arbitrary $n$, $k$, and $d$ systematically, and inefficiently, enumerate subsets. The HSSP problem is NP-hard (Bringmann and Friedrich, 2012), already for $d = 3$ (Bringmann et al., 2017). In the planar case, HSSP can be solved in $O(n \log n + k(n - k))$ through dynamic programming, thanks to monotonicity properties specific to $d = 2$ (Bringmann et al., 2014; Kuhn et al., 2016). After a preliminary sorting step, HSSP($n, 2, k$) also reduces in linear time to a $k$-link shortest path problem on a directed acyclic graph (DAG) having the concave Monge property (Kuhn et al., 2016). The reduction yields algorithms in $O(n \log n + n^2 \cdot O(\sqrt{\log n \log \log n}))$ (Aggarwal et al., 1994) and $O(n \log n + n^2 \cdot \sqrt{k \log n})$ (Schieber, 1998). An $O(nk)$ bound had also been observed in Aggarwal et al. (1994) for such path problems, already using the same monotone matrix search technique as in Kuhn et al. (2016), hence the $O(nk + n \log n)$ algorithm from Kuhn et al. (2016) is also closely related to the $k$-link shortest path reduction. For $d = 3$ an $n^{O(\sqrt{\log n})}$ algorithm was proposed in Bringmann et al. (2017). Bringmann and Friedrich (2010) prove that in dimension $d$ one can remove the $k' = n - k$ points that together contribute least to the hypervolume, and thus solve HSSP($n, 3, k$) in $O(n^{d/2} + n^k)$ as they avoid recomputing from scratch the hypervolume for each of the combinations. That bound is based on the best upper bound for hypervolume computation known in 2010: $O(n^{d/2} \log n)$. Chan (2013) since proposed a faster algorithm in $\tilde{O}(n^{d/3})$ for hypervolume computation, but does not address HSSP, so it is not obvious whether the new algorithm could be adapted for HSSP.

Heuristics have been proposed to avoid the high complexity costs of exact HSSP algorithms. In particular, incremental and decremental greedy strategies have been suggested (Bradstreet et al., 2007). The incremental approach starts with an empty candidate solution, then appends iteratively to the solution the point which adds the largest hypervolume to the solution. The decremental approach starts with the whole input as a candidate solution, then iteratively removes the point that contributes least to the hypervolume. After $k' = n - k$ iterations, the decremental approach can be arbitrarily far from the optimal solution HSSP($n, k$) (Bringmann and Friedrich, 2010), whereas after $k$ iterations the incremental approach is within a factor $(1 - 1/e)$ of the optimum HSSP($n, d, k$) because the hypervolume is a monotone submodular function (Ulrich and Thiele, 2012; Nemhauser et al., 1978). The incremental greedy approach, denoted in the following as gHSSP, is surveyed in Guerreiro et al. (2016), where optimized algorithms for $d = 2$ and $d = 3$ are introduced. Bringmann et al. (2017) have recently proposed an efficient polynomial-time approximation scheme for HSSP in dimension $d$, with running time $O(n e^{-d} (\log n + k + 2^O(e^{-2} \log 1/e'))).

An important subproblem in incremental and decremental HSSP algorithms is the computation of the hypervolume contribution of a point or a set of points. Emmerich and Fonseca (2011) establish tight bounds on the computation of hypervolume contributions when $d = 2$ or $d = 3$. They investigate two flavors of the problem: OneContribution

1 In the following, the notation $\tilde{O}(f(n))$ will be used for asymptotic complexities when we omit polylogarithmic factors in $n$. 

---

612 Evolutionary Computation Volume 27, Number 4
Hypervolume Subset Selection with Small Subsets
denotes the problem of computing the contribution of a given query point in $D$, whereas $\text{AllContributions}$ computes the contribution of every point in $D$. They show that $\text{OneContribution}$ is $\Theta(n)$ for $d = 2$ and $\Theta(n \log n)$ for $d = 3$, whereas $\text{AllContributions}$ is $\Theta(n \log n)$ for $d = 2$ and $d = 3$. The algorithms for computing contributions imply an $O(n \log n)$ algorithm for $\text{HSSP}(n, d, n - 1)$ when $d \leq 3$, whereas $\text{HSSP}(n, d, 1)$ is trivially in $O(n)$ for any $d$. When the dimension $d$ is unbounded, computing contributions is #P-hard, and checking if a given point has the minimal contribution is NP-hard (Bringmann and Friedrich, 2012).

Guerreiro et al. (2016) also investigate joint contributions $H(p, S)$ of a point $p$ with respect to a set $S$, defined as the volume dominated by $p$ but not by the set $S$. As we shall see, their definition differs from ours. This is because their paper investigates algorithms for $\text{gHSSP}$. In this setting, points with the highest contribution are added iteratively, so the contributions are computed with respect to a different set after each iteration. Our perspective is slightly different as we compute contributions in a given (static) set.

We next summarize known results on the worst-case complexity of HSSP:

**Theorem 1 (State of the art for HSSP):** We can solve for any constant $d$

1. $\text{HSSP}(n, 2, k)$ in $O(n \log n + k(n - k))$
2. $\text{HSSP}(n, 2, k)$ in $O(n \log n + n^2 \log \log n)$
3. $\text{HSSP}(n, 2, k)$ in $O(n \log n + n \sqrt{k \log n})$
4. $\text{HSSP}(n, 3, k)$ in $O(\sqrt{k})$
5. $\text{HSSP}(n, d, n - k)$ in $O(n^{d/2} \log n + n^k)$

Unlike Kuhn et al. (2016), the original article of Bringmann et al. (2014) only claims and achieves $O(n \log n + kn)$ running time, but we show that a straightforward fix tightens this bound to $O(n \log n + k(n - k))$.

### 1.3 Our Contributions

Our contributions are the following: (1) we propose algorithms to solve HSSP efficiently when $k = 2$ or $k = 3$ – or, if $d = 3$, any small $k$, (2) we propose algorithms to solve more efficiently HSSP when $d = 3$ and $k' = n - k$ is small, and (3) we unify and improve existing algorithms for $d = 2$, and propose the first optimal implementation for $\text{gHSSP}$ with $d = 2$. Furthermore, while obtaining the above results, we introduce algorithms that we believe to be of independent interest to: (4) compute efficiently the contribution of any subset of points to the hypervolume when $d = 3$, and (5) compute the minimal weight $k$-set in a weighted hypergraph. Finally, we implement the algorithms in this article and evaluate them on various synthetic datasets. The source code is available at https://gitlri.lri.fr/groz/hssp-hypervolume-contributions. The experiments show that, especially for large input sizes, our algorithms outperform previous solutions, often by an order of magnitude. These empirical results match the theoretical ones. We also discuss how large the input must be so that the algorithms are efficient in practice, as compared to the naive ones.

To summarize, using the algorithms presented here, along with minor changes in some cases, allows us to obtain the following complexity results:
Theorem 2 (New Bounds): Let $h = \lfloor k/3 \rfloor$, $r = k - 3h$. We can solve for any constant $d$:

- $HSSP(n, d, 2)$ in $O(n \log^d n)$
- $HSSP(n, 3, n - 2)$ in $O(n \log n)$
- $HSSP(n, 3, n - 3)$ in $O(n^{3/2})$
- $HSSP(n, 3, n - k)$ in $O(n^{2.38h + r})$

1.4 Techniques Involved in Our Algorithms

To improve on the state of the art, our algorithms solve the subset selection problem without enumerating subsets. Our algorithms rely on extreme-point queries to solve HSSP for small $k$, and to optimize the greedy heuristic from Bradstreet et al. (2007) and Guerreiro et al. (2016). These extreme-point queries can be viewed as an extension of the so-called “upper envelope” trick discussed in Section 3 and used by Bringmann et al. (2014) to solve HSSP when $k = 2$. The technique saves a factor of $n$ for $d = 3$ or $k = 2$.

To compute (joint) contributions in dimension 3, we maintain dynamically multiple layers of $2^d$ skylines while sweeping along a third dimension. This generalizes the algorithm of Emmerich and Fonseca (2011) that computes the total hypervolume and the individual contribution of every point in dimension $d = 3$. The original algorithm of Emmerich and Fonseca (2011) shows that contributions can be computed in optimal $O(n \log n)$ using a traditional sweeping plane technique for $3d$ skylines (Kung et al., 1975). We show that joint contributions (involving at most $k$ points) can be computed in optimal $O(n \log n)$ by maintaining layers of skylines.

Finally, we reduce HSSP with small $k' = n - k$ into a minimal independent set problem on a weighted hypergraph recording the joint contributions.

1.5 Outline

In Section 2 we introduce our notations and definitions. Section 3 deals with the problem in dimension $d = 2$. Section 4 presents our algorithms for small values of $k$, and Section 5 presents our algorithms for large values of $k$. Finally, Section 6 is devoted to the experimental evaluation of our algorithms.

2 Notations and Definitions

We adopt a RAM model with unit cost operations. Let $D = \{p_1, \ldots, p_n\}$ a set of $n$ distinct points in $\mathbb{R}^d$, where $d$ is a fixed constant.

Definition 1: Given two points $p, p' \in D$ we say that $p'$ dominates $p$ and write $p \prec p'$ if $\forall h \leq d$, $p.h \leq p'.h$. The skyline of $D$ is the set of maximal elements for this partial order, hence $D$ is a skyline iff no point of $D$ is dominated by another point of $D$.

A set of points ($d = 2$) and its skyline.
Without loss of generality we assume that $D$ is a skyline: Kirkpatrick and Seidel (1985) show the skyline can be computed in $O(n \log^{d-2} s)$, where $s$ is the cardinality of the skyline. For each $h \in 1 \ldots d$, the $h$th coordinate of point $p$ is denoted by $p.h$.

**Definition 2:** The hypervolume $I_{\text{hyp}}(V)$ of a set of points $V = \{v_1, \ldots, v_k\} \subseteq \mathbb{R}^d$ is the Lebesgue measure of the set $\{p \in \mathbb{R}^d \mid \exists v \in V, 0 \prec p \prec v\}$.

Some alternative definitions in the literature measure the hypervolume with respect to a reference point. Our definition assumes a reference point at the origin $(0, \ldots, 0)$.

![Diagram of HSSP algorithm and definitions]

Abusing notations, we denote by $HSSP(D, k)$ (1) the problem of computing an optimal set, (2) the solution to the problem, and (3) the hypervolume for this solution. We also denote by $HSSP(n, d, k)$ the worst case complexity of solving $HSSP(D, k)$ with a set $D$ of $n$ points in dimension $d$. Even when our algorithms are presented as returning the hypervolume of the solution instead of the solution itself, the subset of points achieving this hypervolume can also be returned without additional effort.

We assume points are in a general configuration, that is, do not admit ties on any coordinate.

**Definition 3:** The hypervolume contribution of a set of points $V \subseteq D$ is defined as $\text{CON}(V) = I_{\text{hyp}}(D) - I_{\text{hyp}}(D \setminus V)$.

The definitions of hypervolume and contributions are illustrated in Figure 1 for two-dimensional points. We define the joint contribution of a set of points $V$ as the subspace that becomes non-dominated only when all of the points of $V$ are removed. In other words, it is the subspace of points dominated by all points of $V$ but dominated by no other point from $D$.

**Definition 4:** The joint contribution of $V$ is the hypervolume of $\bigcap_{v \in V} \text{Dom}(v) \setminus \text{Dom}(D \setminus V)$, where $\text{Dom}(X)$ denotes the subspace $\{p \mid \exists x \in X, 0 \prec p \prec x\}$.

Finally, we denote by $\langle v, v' \rangle$ the dot product of points $v$ and $v'$.

### 3 Extreme Point Queries, and HSSP in Dimension 2

#### 3.1 Data Structure

The following lemma summarizes some well-known results from the literature about extreme point queries. Our HSSP algorithms will make use of these data structures.

---

2This traditional assumption in computational geometry helps simplify arguments, but does not restrict our algorithms. Indeed, ties are not an issue because we can perturb slightly the input so that the algorithms deal with ties (Edelsbrunner and Mücke, 1990).
Lemma 1: Let \( L = \{v_1, \ldots, v_i\} \) denote \( i \) points in \( \mathbb{R}^d \). For \( d = 2 \) and \( d = 3 \), we can preprocess \( L \) for queries that take as input \( v' \in \mathbb{R}^d \) and return \( \arg \max_{j \leq i} \langle v_j, v' \rangle \). After a preprocessing in \( O(i \log i) \) we can answer any query in \( O(\log i) \).

Proof: The case \( d = 2 \) is a reformulation of the envelope argument in Bringmann et al. (2014). For \( d = 2 \) the point we are looking for lies on the convex hull of \( L \). During preprocessing we compute the hull of \( L \) in \( O(i \log i) \). The optimal vector for any \( v' \) can then be obtained in \( O(\log i) \) through binary search, as illustrated in Example 1. For \( d = 3 \) we build in \( O(i \log i) \) a Dobkin-Kirkpatrick hierarchy (Kirkpatrick, 1983) on the hull of \( L \). The algorithm for computing extreme point queries is detailed, among others, in O’Rourke (1998).

Remark 1: Some of our algorithms will use a dynamic version of the data structure for \( d = 2 \), where points can be added to the set \( L \) in addition to extreme point queries. It is well-known that an insertion will have a logarithmic cost. Furthermore, the amortized cost of an insertion will be constant in the particular case where the inserted point is larger than all points of \( L \) on coordinate 1.

We also observe that efficient algorithms for extreme point queries can also be obtained for larger dimensions \( d \geq 4 \) (Eppstein, 2016; Agarwal and Matousek, 1993; Agarwal and Erickson, 1998), but these complex algorithms do not appear to be practical.

Example 1: We illustrate the procedure for extreme point queries when \( d = 2 \):
3.2 Warmup: Analysis of HSSP Algorithms in Dimension 2

We summarize the analysis from Kuhn et al. (2016) and unify its presentation with that of Bringmann et al. (2014) to show that both approaches share the same structure and that both yield the same $O(n \log n + k(n - k))$ bound. Unlike Kuhn et al. (2016), the original article of Bringmann et al. (2014) only claims and achieves $O(n \log n + kn)$ running time, but we show that a straightforward fix tightens this bound to $O(n \log n + k(n - k))$.

Lemma 2: The algorithms Bringmann et al. (2014) and Kuhn et al. (2016) for HSSP($n, 2, k$) of share a similar structure. Both run in $O(n \log n + k(n - k))$.

Proof: The two papers essentially solve the same problem, but the formulation of Kuhn et al. (2016) aims at minimizing the contribution of the points that are not selected, whereas the formulation of Bringmann et al. (2014) aims at maximizing the hypervolume of the points selected. We briefly recall the algorithms to justify our claim. Algorithm 1 left) presents the dynamic program as formulated in Kuhn et al. (2016). Assume points of $D$ are ordered on first coordinate: $p_0 < p_1 < \cdots < p_n < p_{n+1}$ where $p_0$ and $p_{n+1}$ are two dummy points systematically selected. For any $i \leq j$, let $c_{i,j}$ denote the contribution of $\{p_\alpha | i < \alpha < j\}$. We define $P(l, j)$ as follows: we first restrict the set of input points to $\{p_0, \ldots, p_j\}$, and select optimally $l + 1$ points including $p_0$ and $p_j$ from those in view of maximizing their hypervolume. Then $P(l, j) = \min \{|\text{CON}(S) | S \subseteq \{p_0, \ldots, p_{j-1}\}, |S| = j - l\}$ denotes the contribution of the unselected points within the restricted input. Finally, the solution to HSSP($n, 2, k$) is obtained by minimizing $P(k, j) + c_{i,n+1}$ over $j \in \{k, \ldots, n\}$.

To evaluate the $P(l, j)$ values, the algorithm first computes and records in $O(n)$ all $c_{i,j}$ and $c_{i,n+1}$. From these values we can evaluate in constant time any $c_{i,j}$. The crux of the proof in Kuhn et al. (2016) is that for a fixed $l$ the relevant values of $(P(l - 1, i) + c_{i,j})$ form a totally monotone matrix in $\mathbb{R}^{(n-k+1) \times (n-k+1)}$. At lines 4,5 we must evaluate the minimum of this matrix on every column $j$. The total monotonicity property (which means there does not exist a $2 \times 2$ submatrix whose row minima are in the top right and bottom left corners) allows the computation of all these minima in $O(n - k)$, which
yields an overall complexity of $O(k(n-k))$ for the algorithm, after an initial sorting step of $O(n \log n)$.

Algorithm 1 (right) from Bringmann et al. (2014) follows the same steps but computes $Q(l, j) = \max \{I_{\text{hyp}}(S) \mid S \subseteq \{p_1, \ldots, p_j\}, j \in S, |S| = l\}$. Instead of exploiting an algorithm for minima in a monotone matrix, the algorithm of Bringmann et al. (2014) maintains the convex envelope of the set of planar points $\{(p_i, 1, Q_{i,j}) \mid i \in \{l-1, \ldots, j-1\}\}$ for increasing values of $j$. In short, the rationale is that $Q(l-1, i) + p_j \cdot 2 \times \langle p_j, 1 - p_1 \rangle > (Q(l-1, j') + p_j \cdot 2 \times \langle p_j, 1 - p_1 \rangle)$ if and only if $p_j \cdot 2 \times \langle p_j, 1 - p_1 \rangle > (Q(l-1, j') - Q(l-1, i))$ so we obtain the optimal value of $i$ by comparing the slopes on the envelope with $p_j \cdot 2$. We compute all $Q(l, j)$ in $O(n-k)$ through a simple linear walk on the envelope. The total cost of maintaining the envelope and searching the optimal value on it is $O(n-k)$ for each value of $l$, so that the complexity of the whole algorithms is $O(k(n-k))$ once the input is sorted.

\[ \square \]

Remark 2: We observe that line 5 in the outline of Bringmann et al. (2014) can be interpreted as an extreme point query: we are looking for the index $i$ that maximizes $\langle v_i, w \rangle$ where $v_i = (-p_i, 1, 1)$, and $w = (Q(l-1, i) + p_j \cdot 2 \times p_j, 1)$. We could use the procedure from Lemma 1 to compute the optimal index $i$, but this would raise the cost to $O(n-k) \log(n-k)$ for lines 4,5. To achieve $O(n-k)$, the algorithm from Bringmann et al. (2014) performs a linear walk on the envelope instead of the binary searches of Lemma 1, which allows amortization of the cost of each search.

3.3 Greedy Approximation Algorithm

Guerreiro et al. (2016) propose an algorithm that computes the greedy incremental solution $gHSSP(n, k, d = 2)$ in $O(kn + n \log n)$. They observe that this is no better than the exact algorithm for $HSSP(n, k, d = 2)$ but claim it may be faster and easier to implement. We show that $gHSSP$ can be solved even faster asymptotically, though our approach exploits complex data structures which are not easy to implement and therefore are mostly of theoretical interest.

Theorem 3: We can compute $gHSSP(n, k, d = 2)$ in $O(n \log n)$.

Proof: We first build a three-dimensional extreme point data structure $V$ for the set of points \( \{v(p) \mid p \in D, v(p) = (p.x, p.y, p.x \times p.y)\} \). After this preprocessing in $O(n \log n)$, we iteratively add to the candidate solution $S$ the point from $D$ which increases most its hypervolume.

The algorithm maintains the candidate solution $S$ as a doubly-linked list sorted by increasing $x$—as we assume the input is a skyline, this implies they are also sorted by decreasing $y$. The algorithm also maintains the top $|S| + 1$ candidates for the next point that should be added in $S$, together with the hypervolume they would add. Those top $|S| + 1$ candidates are ordered according to the contribution. To support efficient insertions and deletions, the top candidates are stored in an AVL (or red-black) tree $T$.

The steps for maintaining $S$ and $T$ using $V$ are detailed in Algorithm 2. For the first iteration we initialize $S$ with the point $p$ having the largest hypervolume. We also add to $S$ sentinels $l = (0, +\infty)$ and $(+\infty, 0)$ to simplify the implementation. Point $p$ can be computed as $\arg \max \{v(p) \mid p \in D\}$ in $O(\log n)$, or alternatively through a linear scan of $D$. $T$ never has more than $k$ points, so each iteration can add and remove points to $T$ in $O(\log k)$ (lines 10, 12, 14). Similarly, insertions into $S$ can be performed in $O(\log k)$ (line 16). By Lemma 1, the complexity of procedure $\text{MAXHYPVINCREMENT}$ is $O(\log n)$ using the data structure for extreme point queries. Consequently, each
Algorithm 2: gHSSP($D, k, 2$): greedy incremental hssp with $d = 2$.

// $S$: doubly-linked list of points, sorted by increasing $x$ (* hence decreasing $y$ *)

// $T$: AVL tree of (points, volume), sorted by increasing volume

1: build a data structure $V$ for extreme point query on \{($v(p)$ | $p \in D, v(p) = (p.x, p.y, p.x \times p.y)$).

2: $p \leftarrow$ point with maximal hypervolume in $D$

3: $c \leftarrow p.x \times p.y$ (* the contribution of $p$ *)

4: $S \leftarrow \{l, p, r\}$ (* where $l$ and $r$ are the sentinels *)

5: $\text{vol} \leftarrow 0$ (* volume of the candidate solution *)

6: $T \leftarrow \{(p, c)\}$

7: $i \leftarrow 1$

8: while $i < k$ do

9: $p_1, p_2 \leftarrow$ previous and next point around $p$ in $S$

10: remove $(p, c)$ from $T$

11: $(p', c') \leftarrow \text{MAXHYPVINCREMENT} (p_1, p)$

12: add $(p', c')$ to $T$

13: $(p', c') \leftarrow \text{MAXHYPVINCREMENT} (p, p_2)$

14: add $(p', c')$ to $T$

15: $(p, c) \leftarrow$ maximal element of $T$

16: add $p$ to $S$

17: $\text{vol} \leftarrow \text{vol} + c$

18: return $S$, vol

19: procedure $\text{MAXHYPVINCREMENT} (q_1, q_2)$ (* point between $q_1$ and $q_2$ that most increments volume *)

20: $w \leftarrow (\neg q_2.y, \neg q_1.x, 1)$

21: $p \leftarrow \text{arg max}$ \{($v(p), w)$ | $p \in D$\} (* extreme point query using $V$ *)

22: $c \leftarrow (p.x - q_1.x) \times (p.y - q_2.y)$ (* $c = \langle v(p), w \rangle + q_1.x \times q_2.y$ *)

23: return $(p, c)$
iteration through line 8 has cost $O(\log n)$ and therefore the whole algorithm, including preprocessing, runs in $O(n \log n)$. □

The technique does not appear to generalize for dimension 3, because the contribution of a point in dimension 3 may depend on an arbitrary number of points instead of just 2 neighbors, so it is not clear how extreme point queries could help to identify efficiently the point that must be added in each iteration.

4 HSSP Beyond Dimension 2: Small $k$

Additional notations. Let $S \subset \{1, \ldots, d\}$. We denote by $\bar{S}$ the set $\{1, \ldots, d\} \setminus S$. We define the partial order $<_S$ on points as follows: $v <_S v'$ iff $S$ is the set of coordinates on which $v$ is greater than $v'$, and $\bar{S}$ the coordinates on which $v'$ is greater. For any $V \subseteq D$, we define $\text{convexhull}^S(V)$ as the upper envelope for the set $\{(\prod_{h \in S} p.h, \prod_{h \in \bar{S}} p.h) \mid p \in V\}$ of planar points. We define

$$\text{BestVal}^S(p', V) = \max_{p \in V \mid p <_S p'} \mathcal{I}_{\text{hyp}}(p, p').$$

4.1 General Overview

The general idea is to compute for each point $p' \in D$ the point which together with $p'$ yields the highest hypervolume. This is very similar to what the first iteration of $g$HSSP computes, except that, in our case, the input points do not lie in a $2d$ space. To generalize the extreme-point query technique with $p'$ in some arbitrary dimension, we compute for each subset $S$ of dimensions the best point $\text{BestVal}^S(p', D)$ within the corresponding orthant around $p'$: in dimension 2 there are two such orthants, which are explored in lines 11 and 13 of Algorithm 2 (there are 4 orthants, actually, but only 2 are relevant as $D$ is a skyline). In general, there will be $2^d$ orthants ($2^d - 2$ relevant ones), which is constant for fixed $d$.

The point of fixing an orthant with $S$ is that it allows to replace Equation 1 below with Equation 2, thereby eliminating the min operator. We will then use extreme point queries to compute efficiently $\text{BestVal}^S(p', D)$, which solves the HSSP problem with $k = 2$, according to Equation 3.

Computing the hypervolume. The hypervolume covered by a pair of points $p, p'$ is

$$\mathcal{I}_{\text{hyp}}(p, p') = \prod_{h=1}^{d} p.h + \prod_{h=1}^{d} p'.h - \prod_{h \in \bar{S}} \min(p.h, p'.h).$$

If we fix the order of $p$ and $p'$ on all coordinates, this simplifies into:

for all $p <_S p'$: $\mathcal{I}_{\text{hyp}}(p, p') = \prod_{h=1}^{d} p.h + \prod_{h=1}^{d} p'.h - \prod_{h \in S} \prod_{h \in \bar{S}} p.h h.$

To exploit this equation we will use the following property:

$$\text{HSSP}(D, 2) = \max_{p, p' \in D} \mathcal{I}_{\text{hyp}}(p, p') = \max_S \max_{p' \in D} \text{BestVal}^S(p', D).$$

The next Lemma shows how extreme point queries, along with hypervolume contributions, can help to compute operator $\text{BestVal}$ defined in Section 4, and thereby help solve HSSP when $k = 2$:

**Lemma 3:** Let $L = \{p_1, \ldots, p_i\} \subseteq D$ and $S \subseteq \{1, \ldots, d\}$. After a preprocessing of $L$ in $O(i \log i)$, we can compute $\text{BestVal}^S(p', L)$ in $O(\log i)$ (through a single extreme point query).
Hypervolume Subset Selection with Small Subsets

Proof: For all $p \in \mathbb{R}^d$, we define 3 vectors $w(p) = (\prod_{h \in S} p.h, \prod_{h \neq S} p.h)$, $v(p) = (-\prod_{h \in S} p.h, 1)$ and $c(p) = \prod_{h \in S} p.h$. For any $p' \in \{ p \in D \mid \forall p_j \in L, p_j \prec_S p \}$ and $p_j \in L$, we have by Equation 2: $I_{hyp}(p_j, p') = c(p') + (v(p'), w(p_j))$. By definition, $\text{BestVal}^S(p', L) = \max_{p \in L} I_{hyp}(p, p')$. Hence $\text{BestVal}^S(p', L) = \prod_{h \neq S} p'.h + \max_{j=1} \langle v(p'), w(p_j) \rangle$. In other words, $\text{BestVal}^S(p', L)$ translates into an extreme point query over $\text{convhull}^S(L)$. Furthermore, we can compute in constant time $c(p')$, $v(p')$, $w(p_j)$ as $d$ is fixed. The result then follows from Lemma 1. \[\square\]

If $L$ is the set of points $\{ p \in D \mid p \prec_S p' \}$, i.e., the orthant defined by $S$, the Lemma states that a single extreme point query returns $\text{BestVal}^S(p', L)$. Preprocessing the orthants around every point $p'$, for extreme point queries, is infeasible. Instead, we maintain a $d$-dimensional range tree: a hierarchical space-partitioning data structure that stores a set of points in each node (this is further explained in Section 4.2). We maintain one extreme point data structure on each set in the range tree. For any point $p'$ and $S \subseteq \{1, \ldots, d\}$, the range tree allows in $O(\log^d n)$ the identification of a list of sets whose union is the orthant we want. We perform one extreme point query on each set in this union, and return the best among the results.

4.2 Algorithm for $k = 2, d > 2$

The strategy for $k = 2$ is summarized in Algorithm 3. We next explain notations and estimate the complexity. We fix some $S \subseteq \{1, \ldots, d\}$. There are $2^d$ such sets but we assume fixed $d$ all along our computation that follows so this is a constant factor. We store the points of $D$ in a $d$-dimensional range tree $t$ (Bentley, 1979; Chazelle, 1990a,b) according to coordinates 1, $\ldots, d$.

**Range tree.** We first build a binary search tree $t_1$ on points sorted on coordinate 1. Each node of $t_1$ is associated with a canonical set of points. The canonical set associated to each leaf contains a single point, and the canonical set in each internal node is the union

**Algorithm 3: HSSP(D, 2) (* for arbitrary but fixed d *).**

1. compute the $d$-dimensional range tree $t$ on $D$

2. for all $S \subseteq \{1, \ldots, d\}$ do

3. for all nodes $u$ in $t$ do

4. preprocess $u$ for BestVal queries with Lemma 3 (* record convhull$^S(u) *$

5. for all $p \in D$ do

6. compute Bucket$\prec_S(p)$

7. for $V \in \text{Bucket}_{\prec_S(p)}$ do

8. compute BestVal$^S(p, V)$

9. return $\max_S \max_p \max_{V} \text{BestVal}^S(p, V)$

**Proof:** For all $p \in \mathbb{R}^d$, we define 3 vectors $w(p) = (\prod_{h \in S} p.h, \prod_{h \neq S} p.h)$, $v(p) = (-\prod_{h \in S} p.h, 1)$ and $c(p) = \prod_{h \in S} p.h$. For any $p' \in \{ p \in D \mid \forall p_j \in L, p_j \prec_S p \}$ and $p_j \in L$, we have by Equation 2: $I_{hyp}(p_j, p') = c(p') + (v(p'), w(p_j))$. By definition, $\text{BestVal}^S(p', L) = \max_{p \in L} I_{hyp}(p, p')$. Hence $\text{BestVal}^S(p', L) = \prod_{h \neq S} p'.h + \max_{j=1} \langle v(p'), w(p_j) \rangle$. In other words, $\text{BestVal}^S(p', L)$ translates into an extreme point query over $\text{convhull}^S(L)$. Furthermore, we can compute in constant time $c(p')$, $v(p')$, $w(p_j)$ as $d$ is fixed. The result then follows from Lemma 1. \[\square\]
of the sets in its left and right children. In the following, we consider the canonical set of each node as being the node itself. We recursively attach to each node \( u \) of the tree \( t_h (1 \leq h \leq d - 1) \) another binary search tree \( t_{h+1}(u) \). When \( h = d \), we do not attach any further binary tree so we qualify the \( t_d \) subtrees as terminal. The tree \( t_{h+1}(u) \) records the points contained in \( u \), sorted on coordinate \( h + 1 \). For each point \( p \) let \( X \) denote the set of nodes \( u_i \) in the terminal subtrees of the range tree \( t \)—such that \( u_i \subseteq \{ p' \in D \mid p' <_S p \} \). We denote by Bucket\(_{<_S}(p)\) the set \( \{u_1, \ldots, u_\alpha\} \) of maximal nodes in \( X \). Here, maximality refers to the ordering of nodes according to their level in the tree (and therefore implies maximality for containment). By construction, \( \bigcup u_i = \{ p' \in D \mid p' <_S p \} \). The range tree can be computed in \( O(n \log^{d-1} n) \) and allows to answer range queries efficiently: in our case, it returns Bucket\(_{<_S}(p)\) in \( O(\log^{d-1} n + \alpha) \) (Chazelle, 1990a, 1990b), where \( \alpha \in O(\log^d n) \) is the number of nodes in Bucket\(_{<_S}(p)\).

**Naïve implementation.** At each node \( u \) in the tree, we compute the upper envelope convexhull\(_S^\uparrow(u)\). These can all be computed in \( O(n \log^{d+1} n) \), because the envelope at node \( u \) is computed in \( O(|u| \log |u|) \), and each point of \( D \) appears in \( O(\log^d n) \) nodes. Finally, for each \( p \in D \) we first compute Bucket\(_{<_S}(p)\), then evaluate BestVal\(_S(V)\) on each \( V \) in this list using the envelope data structure. The cost for each \( p \) is one binary search per node \( u_i \), hence a total of \( O(n \log^{d+1} n) \) over all \( p \) for line 8. Using this naïve implementation, HSSP\((n, d, 2)\) is solved in \( O(n \log^{d+1} n) \) for any fixed \( d \).

We next detail two optimizations that save a logarithmic factor each from the naïve implementation of Algorithm 3. We first adopt a sweeping hyperplane technique to lower the dimension of the range trees from \( d \) to \( d - 1 \). In the sweeping hyperplane approach we do not distinguish the preprocessing and queries but rather maintain incrementally the extreme point data structure between queries. The second optimization is specific to \( d = 3 \), and exploits the properties of 3d skylines.

**Sweeping hyperplane.** We can cut one logarithmic factor from Algorithm 3 using a sweeping hyperplane approach: at line 5 we enumerate \( p \) in increasing order on coordinate \( d \). We maintain dynamically the extreme point data structures so that they only consider points from previous iterations: as those points have larger coordinate \( d \), we only have to deal with the remaining \( d - 1 \) lower dimensions in the range tree.

We now take \( S \subseteq \{1, \ldots, d - 1\} \), and maintain a \((d - 1)\)-dimensional range tree on \( D \). The data structures for BestVal queries are initially empty (we drop the preprocessing phase of lines 3-4): we initially consider all points in the range tree as “inactive.” Each iteration first computes Bucket\(_{<_S}(p)\), then evaluates BestVal\(_S(V)\) on all \( V \in\) Bucket\(_{<_S}(p)\). Point \( p \) is then “activated” in the tree: for each node \( v \) containing \( p \), we update convexhull\(_S(v)\). To perform the update we first locate in \( O(\log n) \) the position of \( (\prod_{h \in S} p_h, \prod_{h \in S} p_h) \) on the convex hull. Then we update the hull in amortized constant time. There are \( O(\log^{d-1} n) \) nodes affected by each update or query, hence the overall cost for maintaining the structures and computing BestVal\(_S(p, V)\) is \( O(n \log^d n) \).

**Exploiting the properties of 3d skylines.** For \( d = 3 \), we can exploit our assumption that input points form a skyline to lower the dimension of the range tree, and thus save an additional logarithmic factor. In a skyline of dimension \( 3 \), whenever \( p' .2 < p .2 \) and \( p' .3 < p .3 \) we know that necessarily \( p' .1 > p .1 \). There are only 3 relevant sets \( S \): all pairs in \( \{1, \ldots, d\} \). Fix for instance \( S = \{2, 3\} \). We only need to make sure that \( p \) is smaller than points of \( V \) on coordinates 2 and 3 for the extreme point queries at line 8 of Algorithm 3. The sweeping hyperplane approach will guarantee that \( p \) is smaller on coordinate 3, so we only need a 1-dimensional range tree over coordinate 1.
Let us observe that we could exploit fractional cascading (Chazelle and Guibas, 1986) to lower the cost of both preprocessing and BestVal queries to $O(n \log^d n)$ instead of the sweeping hyperplane optimization.

**Theorem 4:** We can solve HSSP($n, d, 2$) in $O(n \log^d n)$ for any fixed $d$, and HSSP($n, d = 3, 2$) in $O(n \log^2 n)$.

Using the inclusion/exclusion formula to compute the hypervolume of a set of points (Azarm and Wu, 2001), we could adapt the technique of Theorem 4 to solve HSSP($n, d, 3$) or solve HSSP($n, d = 3, k$) but the algorithms would not be practical due to large constants and due to their reliance on extreme point queries in dimension 4.

### 5 HSSP Beyond Dimension 2: Large $k$

In this section we address the problem of computing HSSP($n, d = 3, k'$) for large values of $k'$: $k' = n - 2$ or $n - 3$. For all $k$, we call $k$-joint contribution the joint contribution of any set $V$ of cardinality $k$, and define $E_k$ as the set of $k$-tuples with (non-empty) joint contribution. Finally, all complexity results in this section assume that we can implement associative arrays with constant-time insertions and lookups; an assumption that is not unreasonable in practice, where we can use well-designed hash maps.

#### 5.1 Computing Joint Contributions

We first generalize a result from Emmerich and Fonseca (2011):

**Theorem 5:** Let $D$ be a set of $n$ three-dimensional points. There are $O(k^2 n)$ sets of $j \leq k$ points from $D$ having a $j$-joint contribution. These sets can be listed in $O(k^2 n + n \log n)$, which is optimal if we represent independently each joint contribution as a pair (hypervolume, dominating points). Alternatively, if we allow for a compact representation of dominating points, the complexity becomes $O(k^2 n + n \log n)$, which is optimal.

**Proof (outline):** We focus on the $O(k^2 n + n \log n)$ data structure with implicit representation of the points. For any sequence of points $p_1, \ldots, p_j$, in decreasing order by some coordinate, if $p_1, \ldots, p_j$ has joint contribution $v_j > 0$, then $p_1, \ldots, p_{j-1}$ also has joint contribution $v_{j-1} > 0$. Our compact data structure thus records the sequences $p_1, \ldots, p_j$ and $v_1, \ldots, v_j$ to avoid repeating the list of points that are prefixes of $p_1 \ldots, p_j$. We can easily list contributions with their explicit list of dominating points by expanding this compact representation, at the cost of an additional $k$ factor.

Our algorithm adapts the dimension sweep algorithm of Emmerich and Fonseca (2011), which in turn is closely related to the 3d skyline algorithm of Kung et al. (1975). The dimension sweep algorithm considers points by decreasing $z$ order and maintains (using AVL trees) the skyline of their projection on the $xy$-plane while new points are considered. The dimension sweep algorithm essentially removes from the skyline the portion dominated by $p$, and "inserts" $p$ as a substitute. Our generalized algorithm instead maintains $k$ layers of skylines, and moves the dominated portion from one skyline to the next while substituting the portion with $p$ (or the following skyline portion). The evolution of those skyline layers is illustrated in Example 2 together with Figure 2. Maintaining each skyline layer in an AVL-tree (or rather in a red-black tree for our implementation) allows our algorithm to identify quickly for each new point the contributions whose area must be updated, and those who must be transferred to the next layer. The running time and implementation details are discussed in the online supplement available at http://www.mitpressjournals.org/doi/suppl/10.1162/evco_a_00235.
To prove the lower bound $k^2n$, we exhibit a set $D$ having $\Omega(k^2n)$ joint contributions, while the $n \log n$ lower bound was proved in Emmerich and Fonseca (2011). The proofs are detailed in the online supplement available at http://www.mitpressjournals.org/doi/suppl/10.1162/evco_a_00235.

Our HSSP algorithms for large $k$ will exploit the result above about $k$-joint contributions. We believe that contribution queries may in general prove as useful as joint contribution queries, where a $(j)$-contribution query takes as input $j$ points and returns their (non-necessarily joint) contribution. For instance, the HSSP algorithm of Bringmann and Friedrich (2010) for large dimensions is essentially about avoiding to recompute the hypervolume while performing all $k$-contribution queries. While most subsets $S \subseteq D$ do not have joint contributions, each has some non-null contribution. Consequently, precomputing all of these appears impractical. Of course, we could compute the contribution of $S$ in $O(n \log n)$ as the difference between $I_{\text{hyp}}(D)$ and $I_{\text{hyp}}(D \setminus S)$. But we can do better if we preprocess $D$:

**Theorem 6:** Let $D$ be a set of $n$ three-dimensional points. We can preprocess $D$ in $O(k^2n + n \log n)$ to support $(j)$-contribution queries in $O(j^2)$ for all $j \leq k$.

**Proof:** We first observe that any contribution $\text{CON}(S)$ is the sum of the joint contributions of all $S' \subseteq S$. We first apply Theorem 5 to record all $j$-joint contributions on $D$ with $j \leq k$ in a data structure $h$ grouping the joint contributions according to their upper corner in the $xy$-plane. After this preprocessing, whenever we are given a set $S \subseteq D$, $|S| = j \leq k$, we identify all $S'' \subseteq S$ having joint contribution in $O(j^2)$ and deduce the result. To achieve $O(j^2)$ instead, we precompute sums of joint-contributions within each group during preprocessing, as detailed in the online supplement available at http://www.mitpressjournals.org/doi/suppl/10.1162/evco_a_00235.

**Example 2:** Figure 2 illustrates the algorithm for computing contributions from Theorem 5 on a sample of 5 points. The figure features 5 iterations, from left to right, corresponding to the introduction of points $p_1$ to $p_5$. For each iteration, the figure illustrates...
Hypervolume Subset Selection with Small Subsets

the projection on the \(xy\) plane of the space dominated by each set of point. Below each iteration we also chose to detail the information recorded by the algorithm about the contributions whose upper right corner is (4,2).

The contribution is computed as a sum of vertical slices: when the projection in the \(xy\) plane of the contribution shrinks because it is split by a new point, a new slice is introduced. Variable \(a\) represents the area in the \(xy\) plane for the current slice of the contribution, while \(z\) records the start of this slice, and \(\text{vol}_{\text{above}}\) records the cumulative volume of the slices above the current slice. The algorithm completes the computation of \(p_1\)'s contribution during the fourth iteration (while considering point \(p_4\)), then the object representing this contribution is moved to the next skyline layer as it now records the joint contribution of \(p_1\) and \(p_4\). After the fifth iteration, the variables \(\text{up_points}\) and \(\text{vol}_\text{contributions}\) have recorded the contribution of \(p_1\) and the joint contribution of \(p_1, p_4\): respectively 14 and 2. While this example focused on the evolution of the object (cell)—depicted with a star—recording the contributions with upper corner at (4,2), our algorithm records in similar objects the contributions having other upper corners.

The algorithm of Theorem 5 can build in \(O(k^2 n + n \log n)\) a map recording joint contributions. This supports \((j-)\)joint contributions \((j \leq k)\) in \(O(j)\), which is optimal as we have to read the \(j\) input points anyway. This algorithm is satisfactory for our purposes—we only care about small values of \(k\) to solve HSSP—but the \(k^2 n\) term may still appear excessive, so we observe that we can do better if we do not record the list of contributions in any “simple” representation, but only build a data structure able to answer contribution queries.

**Remark 3:** We can support \(j\)-joint contribution queries \((j \leq k)\) in \(O(\log n + j)\) after a preprocessing in \(O(kn + n \log n)\).

For this, we only record the upper corner of each joint contribution in the \(xy\)-plane, and rely on top-\(k\) orthogonal range reporting data structures to represent the dominating points, combining the algorithm from Rahul and Tao (2015, Theorem 2), which does not analyze the time complexity of the preprocessing, with the running-time bounds from Afshani and Tsakalidis (2014, Theorem 2). This would mostly be a theoretical result, though, because we are not aware of any implementation of this top-\(k\) ORR structure, let alone an efficient one.

### 5.2 From Weighted Hypergraphs to HSSP

Let \(\omega\) denote the degree of matrix multiplication, and \(\omega^*\) denote the value \(\max(\omega, 2 + o(1))\) as in Czumaj and Lingas (2009). We next show that we can compute efficiently a minimal-weight independent triple in sparse graphs, where \(|E| \in O(n)\), in \(O(n^{2\omega^*/(\omega^*+1)})\) \(\in O(n^{1.41})\). In fact, we only use the result to prove Lemma 5; hence, a simpler analysis establishing an \(n^{3/2}\) bound would suffice in our application.

A vertex-weighted graph \((V, E, W)\) is defined as an undirected graph together with a weight function on vertices: \(V\) is the set of vertices, \(E\) the edges and \(W : V \to \mathbb{R}\) is the weight function.

**Lemma 4:** Let \(G(V, E, W)\) a vertex-weighted graph with \(m \in O(n)\) edges. We can compute the minimal independent triple in \(G\) in \(O(n^{2\omega^*/(\omega^*+1)})\).
Proof (Outline): We adapt the results of Czumaj and Lingas (2009) to independent sets. In line with Czumaj and Lingas (2009) and previous results (Alon et al., 1997) that deal with searching triangles in sparse graphs, we adopt the approach of distinguishing heavy and light vertices based on their degree, and develop our proof on a case analysis based on the number of heavy vertices in the triple. Our algorithm computing the minimal-weight independent set is a bit more involved than the triangle algorithm from Czumaj and Lingas (2009) and Alon et al. (1997): we sort vertices and their adjacency lists, and we observe that in graphs with minimal degree \( n - \delta \) any set of \( 2\delta \) vertices and more contains a triangle. The observation is used to restrict our searches to the first few items in each list. We develop the full proof in the online supplement.

The following lemma allows computation of the minimal weight triple in sparse weighted graphs, where the weight function is defined on vertices, edges and hyperedges. This differs from Lemma 4 where the weights are on vertices only. Finding minimal weight triangles (connected triples in a graph) in edge-weighted graphs requires \( \Omega(n^{2.5 - \epsilon}) \) in general (Williams, 2014), but we can exploit the sparsity of our weight function. Our algorithm (left for the online supplement) exploits a case analysis based on the distinction between heavy and light vertices similar to the one for Lemma 4.

Lemma 5: Let \( G(V, E, W) \) represent a graph with weights \( W \) defined on vertices and \( O(n) \) edges and hyperedges (triples). We can compute in \( O(n^{3/2}) \) the triple \( X \in \binom{V}{3} \) that minimizes the total weight \( \sum_{X' \subseteq X} W(X') \).

Theorem 7: We can solve

1. \( \text{HSSP}(n, 3, n - 2) \) in \( O(n \log n) \).
2. \( \text{HSSP}(n, 3, n - k) \) in \( O(n^\omega \ell^r) \) where \( \ell = \lfloor k/3 \rfloor \) and \( r = k - 3\ell \).
3. \( \text{HSSP}(n, 3, n - 3) \) in \( O(n^{3/2}) \).

Proof:

1. Recall that we denote by \( E_k \) the set of \( k \)-tuples having non-empty joint contribution (which we generally identify with the associative array mapping the tuple to its contribution). We first compute all individual and pairwise-joint contributions, in \( O(n \log n) \) according to Theorem 5. We then sort individual contributions in increasing order and take the smallest two. If the pair is not in \( E_2 \), we return them. Otherwise we compute for all points \( p \) the smallest point \( p' \) such that \( (p, p') \notin E_2 \) and then compute the best such pair over all \( p \). The naive nested loop search over the sorted list will run in \( O(n) \) since all iterations, except the first, on each point can be charged to a distinct pair in \( E_2 \). We finally compare this pair with the smallest contribution obtained by a pair from \( E_2 \).

2. Algorithm 4 presents our strategy for computing \( \text{HSSP}(n, 3, n - k) \) with the help of Theorem 5. We partition \( \binom{V}{3} \) into 2 types of sets: in Case (i) we compute the maximal hypervolume obtained by removing a set of \( k \) points without any pairwise contributions. In Case (ii) we compute the maximal hypervolume obtained by removing a set in which there is at least one joint contribution (hence at least one pairwise contribution). This second case is handled recursively by fixing a pair \( (p_1, p_2) \) of points with pairwise contributions, and recursing to find the remaining \( k - 2 \) points. When computing the best remaining \( k - 2 \) points we must not forget the joint combinations involving some of these points with \( p_1 \) and/or \( p_2 \). For instance assume we pick a pair \( p_1, p_2 \) at line 7, and assume \( (p_4, p_5) \) have a...
Algorithm 4: HSSP(D, n - k) (* for d = 3 *)

1: compute all i-joint contributions $E_i$ with $i \leq k$.

// record the results as a map: $c_i(s) \in \mathbb{R}$ is the contribution of $s \in D^i$ ($s \in c_i$.keys)

2: **return** $f(D, k, E_1, \ldots, E_k)$

3: **procedure** $f(D, k, c_1, \ldots, c_k)$

(* Case (i): none of $p_1, p_2, \ldots, p_k$ have joint contributions *)

4: build the vertex-weighted graph $G = (V, E, W)$:

\[ V = D, E = V^2 \setminus c_2, \text{ and } W(v) = c_1(v) \text{ for all } v \in V. \]

5: $x \leftarrow k$-clique of minimal weight in $G$

(* Case (ii): there exist some joint contributions among $p_1, p_2, \ldots, p_k$ *)

6: if $k \geq 2$ then

7: for all $(p_1, p_2) \in E_2$ do

8: $(c'_{-1}, c'_0) \leftarrow (\emptyset, \emptyset)$ // irrelevant, but prevent indexes getting out of bound

9: $x_0 \leftarrow c_1(p_1) + c_1(p_2) + c_2(\{p_1, p_2\})$ // summing contributions of $p_1, p_2$

10: for all $i \in 1..k$ do

11: $c'_i \leftarrow c_i$

12: for all $s \in c_i$.keys do

13: $(c'_{i-2}, c'_{i-1}, c'_i) \leftarrow \text{UC}(i, s, \{p_1, p_2\}, c'_{i-2}, c'_{i-1}, c'_i)$

14: $x \leftarrow \min(x, x_0 + f(D \setminus \{p_1, p_2\}, k - 2, c'_1, \ldots, c'_{k-2}))$

15: **return** $x$

16: **procedure** $\text{UC}(i, s, \{p_1, p_2\}, c'_{i-2}, c'_{i-1}, c'_i)$ // updates contributions to account for $S$

17: if $|s \cap \{p_1, p_2\}| = 2$ then

18: $c'_{i-2}(s \setminus \{p_1, p_2\}) \leftarrow c'_{i-2}(s \setminus \{p_1, p_2\}) + c_i(s)$ // and insert if absent in $c'_{i-2}$

19: Delete $s$ from $c'_i$

20: else if $|s \cap \{p_1, p_2\}| = 1$ then

21: $c'_{i-1}(s \setminus \{p_1, p_2\}) \leftarrow c'_{i-1}(s \setminus \{p_1, p_2\}) + c_i(s)$ // and insert if absent in $c'_{i-1}$

22: Delete $s$ from $c'_i$

23: **return** $(c'_{i-2}, c'_{i-1}, c'_i)$
joint contribution stored in $c_2$ and similarly for $(p_1, p_4, p_5)$ in $c_3$; when recursing at line 12 we add the joint contribution $v$ of $p_1, p_4, p_5$ to the joint contribution of $p_4, p_5$. More accurately, if the pair $\{p_4, p_5\}$ was not in $c_2$ then it is inserted in $c_2$ with value $v$, and if the pair was in $c_2$ then we add $v$ to its value. Of course we proceed similarly not only with pairs $\{p_4, p_5\}$ but with every set of $i \leq k - 2$ points, as detailed in lines 8 to 11.

The complexity of Algorithm 4 matches our bounds:

- Case (i) can be solved with the complexity $U(n, k)$ using fast matrix multiplication techniques, where $U(n, k)$ is defined in Czumaj and Lingas (2009, Theorem 9), and satisfies $U(n, k) \in O(n^{\omega_\ell + r})$ where $\ell = \lfloor k/3 \rfloor$ and $r = k - 3\ell$ (hence $k = 3\ell + r$).

- Case (ii) involves $O(n)$ recursive calls because $|E_2| \in O(n)$. And for the same reason the cost of lines 8 to 14 is $O(n)$. Let $T(n, k)$ denote the complexity of HSSP($n, d = 3, n - k$). We then have the following equation, in which $U(n, k)$ is the dominant term: $T(n, k) \leq U(n, k) + O(n) \times (T(n - 2, k - 2) + O(kn))$.

3. The general idea is to replace the recursion of lines 6 to 12 in Algorithm 4 with a case analysis based on the number of joint contributions. Theorem 5 bounds the number of $i$-joint contributions ($i \in \{1, 2, 3\}$) to $O(n)$. The result follows immediately from Lemma 5 with weights defined as the joint contributions.

6 Experiments

All implementations discussed in this section are available at https://gitlri.lri.fr/groz/hssp-hypervolume-contributions. Experiments were run on a PowerEdge R430 machine with 96 GB RAM and CPU: E5-2640 v4 2.4 GHz, 640 kB Cache L1, 2.5 MB Cache L2, and 25 MB Cache L3. Experiments were restricted to a single core. The OS was Ubuntu 16.04. Compilation instructions were g++ -std=c++14 -O3 in gcc 5.4.0. We estimate the memory consumption through the unix top command every 100 ms. These memory estimates are exact up to 0.1 GB; for this reason, some of the corresponding curves (like Figure 3, right) have an irregular shape.

We considered 3 distributions of points: convex, concave and linear, as detailed in Guerreiro et al. (2016) (see also Emmerich and Fonseca, 2011). Distribution convex samples points on the positive orthant of a sphere centered at $(0, \ldots, 0)$, concave samples on the negative orthant of a sphere centered at $(1, \ldots, 1)$ and linear samples points such that $\sum_i x_i = 1$. The 3 distributions above are symmetric (the distribution of coordinates are the same on each dimension), but we also considered distributions that do not display this symmetry. For instance, we also considered a tweaked version of distribution concave where the origin of the sphere is shifted before selecting the orthant around $(1, \ldots, 1)$. Our results did not vary significantly between repeated samples, but also not between the distributions above (neither did those in Guerreiro et al., 2016), so we opted to sample from linear for our experiments.

In dimension 2: HSSP($n, 2, k$). The algorithm from Bringmann et al. (2014) was originally implemented in Java. Our implementation, extreme2d, is a re-implementation of this algorithm in C++, with a few modifications. Independently from our work, a C++ implementation of the initial algorithm is present in the machine-learning library.

\footnote{We assume constant $k$, and constant time associative arrays.}
Hypervolume Subset Selection with Small Subsets

Figure 3: Running time of HSSP programs for $n = 10,000$, $d = 2$.

Shark. We refer to this implementation as shark. We implemented the smawk algorithm from Kuhn et al. (2016). The main differences between extreme2d and shark are the following:

- **extreme2d** assumes the input is a Pareto front with points in general position (no ties), whereas shark is slightly more general as it does not make such assumptions.

- **extreme2d** is drastically faster for small $k' = n - k$ because it performs iterations over data of size $O(n - k)$ instead of $O(n)$ for shark and the original description of the algorithm in Bringmann et al. (2014).

- **extreme2d** defines convex hull objects and performs extreme point queries on these whereas shark adopts the original layout from Bringmann et al. (2014), namely an algorithm that repeatedly computes the maximum of linear functions. (The operations performed are conceptually equivalent, but we believe the convex hull approach simplifies the presentation.)

- **extreme2d** is templated and can therefore accommodate various data types, whereas shark assumes point coordinates are double.

To justify our skyline assumption, we observe that skylines can be computed very fast in 2D, with a choice of algorithms in $O(n \log n)$. It is therefore arguably safer to first prune the input by computing the skyline before we tackle the HSSP problem: in fact, the skyline computation in shark featured a minor bug when we retrieved the code.

Figure 3 compares the running time of those 3 programs on a 2D skyline of $n = 10,000$ points, with varying values of $k$, for the convex distribution. In fact, the performance of each algorithm did not vary between the 3 distributions we considered, hence the type of distribution does not seem relevant here. The figure shows that extreme2d consistently outperforms smawk; this is probably due to the fact that the dynamic program approach of smawk on monotone matrices is much more intricate than the convex hulls of extreme2d, and therefore might hide higher constant factors. When the number of selected points is very large, that is, the case $n - k$, we observed that shark performs very poorly, which is consistent with our expectations. For small values, shark remains faster than smawk, but is surprisingly slower than extreme2d. This is most
likely due to the implementation, and not the theoretical performance: shark and extreme2d apply the same high-level algorithm. Finally, both shark and smawk appear to select \( k \) points faster than \( n - k \) when \( k \) is very small, and the reverse when \( k \) is closer to \( n/2 \). This slight edge of the \( n - k \) variant when \( k \) is close to \( n/2 \) may be the result of implementation and design choices.

In terms of memory, all programs required at most 1 GB of RAM in Figure 3. For larger values of \( n \), shark becomes impractical with large values of \( k \) (for \( n = 100,000 \) and \( k = 99,900 \), shark requires 6 min and 70 GB RAM, whereas extreme2d returns in 0.2 seconds using 0.1 GB). Figure 4 shows that for \( k = n/2 \), smawk and extreme2d use half the memory of shark, even though smawk is a bit slower than shark.

A similar implementation and comparison of the two algorithms has already been performed in Kuhn et al. (2016), but without applying our fix. Our experiments from Figures 3 and 4 complement their results. In particular, in our experiments featuring the minor fix, the approach of Bringmann et al. (2014) outperforms the approach of Kuhn et al. (2016) even for large values of \( k \).

**Extreme point queries in range trees for \( k = 2, d \geq 3 \).** We compared two possible implementations for our approach based on range trees, and for \( k = 2 \) and \( d = 3 \). In our setting with \( k = 2 \), the algorithms from the literature all perform at best as well as the naive nested loop algorithm, denoted naive_pair, which computes for each pair of points their dominated hypervolume. We therefore consider naive_pair as the baseline to which we compare our two programs.

Our first implementation follows the approach described in algorithm 3, having complexity \( O(n \log^3 n) \): a range tree data structure, with convex hulls to support extreme point queries, is first built, containing all input points, then, for each input point, the best match is computed. We denote this implementation static_tree. Our second implementation, called dynamic_tree, implements the \( O(n \log^d n) \) algorithm described in Theorem 4, using the incremental approach of Theorem 4, adding points to the hull iteratively. The implementations of the above two algorithms, optimizing the case \( d = 3 \) by exploiting the properties of 3d skylines, are denoted respectively as static_tree_3d and dynamic_tree_3d.

While experimenting, we observed that the range trees spent most time recursing on small instances. To speedup our programs, we therefore chose to break the recursion as soon as the number of points in a node drops below some threshold \( \text{trunc} \). This way,
we revert to the naive nested loop algorithm whenever the number of points is small enough. In other words, when the number of points is below the threshold, the node becomes a leaf of the range tree data structure, and the points associated to that leaf are stored as a set. Given a point \( p \), the queries asking for the best match of \( p \) within such a leaf will iterate over all points in that set instead of performing extreme point queries in hulls. This truncation curbs the depth and size of our tree structure, and thereby speeds up the programs in practice, while maintaining the same theoretical asymptotic complexity—assuming the \( \text{trunc} \) parameter is a constant.

Figure 5 illustrates the impact of our truncation optimization. The optimal truncation size appears to be around a few hundreds, though it varies slightly depending on the program and the input size. We therefore fixed the value of \( \text{trunc} \) at 200 for the next experiment in Figure 6.

Figure 6 shows that our programs outperform the naive approach when the number of input points exceeds a few thousand. It also shows that our optimizations, lowering the dimensionality of the range trees, pay off for large values of \( n \). Our program
Figure 7: Running time and memory of HSSP programs for $k = 2, d = 4$, varying number of points $n$.

Figure 8: Running time and memory to compute hypervolume contributions of all $k$-sets for $d = 3$, varying $n$.

dynamic_tree_3d combining both optimizations is clearly the fastest for large inputs, in addition to being more space-efficient than the other range tree programs.

As expected from the theoretical analysis, the range tree approach gets less competitive as the dimension increases. Figure 7 shows that already for $d = 4$, the range tree approach only starts to outperform the naive approach for a very large number of points, while it requires a very large amount of memory (for the static version at least).

**Hypervolume contributions.** Figure 8 shows the performance of the implementation of the algorithm to compute all $k$-joint contributions, discussed in Theorem 5. Figure 8 is consistent with the theoretical bound of $O(k^2n + n \log n)$: the running time is almost linear in $n$. For $k = 1$ our implementation is essentially the algorithm from Emmerich and Fonseca (2011), except that our implementation stores one cell per $k$-joint contributions, whereas the original algorithm from Emmerich and Fonseca (2011) splits each cell into a list of rectangular boxes. This is a relatively minor difference. In Figure 8 we observe that for $k = 1$ our implementation is a bit slower than the shark library’s
Figure 9: Running time and memory to compute hypervolume contributions of all $k$-sets for $d = 3$, varying $k$.

implementation of the original algorithm from Emmerich and Fonseca (2011). This is because our program supports arbitrary $k \geq 1$ and therefore performs many useless operations when $k = 1$.

Figure 9 shows that the running time is consistent with the theoretical $O(k^2 n + n \log n)$ bound. The decreasing slope when $k \gg n/2$ can be explained by fact that the number of cells dominated jointly by $k$ points decreases with $k$, for large $k$ (the extreme case being $k = n$; there is a single cell jointly dominated by all points).

**Minimal pair and triangle for hypervolume contributions.** We also evaluated the performance of our algorithms for HSSP($n, 3, n-2$) and HSSP($n, 3, n-3$): those algorithms identify the pair (respectively triple) of points with minimal contribution. As there are no other algorithms to compare with, we implemented the naïve nested loop implementation, which computes the pair (respectively triple) of points with minimal contribution by iterating over all possible pairs (resp. triple) of points. The first step in each of these algorithms is to compute a weighted hypergraph representing all $k$-joint contributions for $k \leq 3$. The running time measured in Figure 10 includes the construction of this graph. The pair of points with minimal contribution can be computed with negligible overhead once the hypergraph has been computed. In other words, the running time for computing the hypergraph roughly coincides with the curve for computing the minimal pair with our algorithm. The hypergraph is built in two steps: we first use the algorithm from Theorem 6 then convert the result into a suitable graph representation. In our implementation, the two steps have roughly the same cost.

We observe that our programs behave as expected, relative to their theoretical complexity, and can be drastically faster than the naïve approach. The gap between the naïve implementation and our algorithm can reach several orders of magnitude, and is markedly higher than for HSSP($n, d, 2$). One reason why the naïve approach performs so poorly in HSSP($n, 3, n-2$) is because for each pair of points, we must query the graph of contributions, which triggers cache misses, whereas for HSSP($n, d, 2$) the

---

4We adopted the **shark** implementation as a benchmark rather than the original implementation from Emmerich and Fonseca (2011) because the latter assumes multiple parameters are hardcoded, such as the number of input points.
hypervolume of a pair of point can be computed directly from that pair, which is much more cache-friendly.

7 Discussion

Our experiments show that all the algorithms behave in line with theory, and do not really present unexpected patterns. Above all, the experiments show how well the algorithms scale with respect to the number of points, and how they compare to the state of the art. We did not implement the greedy heuristic for $d = 2$ in $O(n \log n)$ from Theorem 3, because an efficient implementation of the Dobkin-Kirkpatrick hierarchy for extreme point queries seems out of the scope of this article. We believe the exact algorithm extrem2d is a better choice than the heuristic anyway, except perhaps when the number of points is very large.

8 Conclusion and Open Questions

We have proposed algorithms to select efficiently the $k$ points in a dataset that best represent a Pareto front in terms of dominated hypervolume. Our algorithms are designed to compute efficiently an optimal solution in the “easy” cases, that is, selecting either a very few points, or all but a very few points. The experiments show that our algorithms behave in line with their theoretical bounds, without large hidden constants, and can deal with large inputs that would not be practical for existing approaches. The practical relevance of our HSSP algorithms on general datasets when $d \geq 3$ is not clear, as our experiments show the algorithms do not fare so well on small datasets, and are restricted to very small numbers of points $k$.

We believe that the major contributions of our work are about highlighting some interesting properties of the problem HSSP($n, d, k$) with respect to small and large $k$. First, our work underscores the relevance of extreme point queries for HSSP, as these queries (1) yield the fastest implementation of HSSP($d = 2$) in practice, according to our
experiments, (2) improve the asymptotic quadratic complexity of the greedy heuristic to an optimal $O(n \log n)$, and (3) are the only alternative (proposed so far) to enumerating all $k$-subsets when solving HSSP($k$, $d > 2$) with small values of $k$. And second, while designing our HSSP algorithms, we generalize several results from the literature. In particular, besides some rather theoretical result about minimal-weight triples in weighted hypergraphs, we show that in dimension 3 the algorithms computing the individual contribution of each point can be generalized to compute the joint contribution of sets with $k$ points.

Acknowledgments

We would like to thank D. Eppstein for leading to the solution of Lemma 1 by answering our questions on a forum (Eppstein, 2016). We would also like to thank S. Chester and M. Soholm for introducing the HSSP problem to us, and K. Bringmann for sharing with us his ongoing work on this topic.

References


