Faster S-Metric Calculation by Considering Dominated Hypervolume as Klee's Measure Problem

Nicola Beume and Günter Rudolph

University of Dortmund, Department of Computer Science, Chair of Algorithm Engineering,
44221 Dortmund, Germany,
{nicola.beume,guenter.rudolph}@udo.edu,
http://ls11-www.cs.uni-dortmund.de/

Abstract
The dominated hypervolume (or *S-metric*) is a commonly accepted quality measure for comparing approximations of Pareto fronts generated by multi-objective optimizers. Since optimizers exist, namely evolutionary algorithms, that use the S-metric internally several times per iteration, a faster determination of the S-metric value is of essential importance. This paper describes how to consider the S-metric as a special case of a more general geometrical problem called *Klee's measure problem (KMP)*. For KMP an algorithm exists with run time $O(n \log n + n^{d/2} \log n)$, for n points of $d \geq 3$ dimensions. This complex algorithm is adapted to the special case of calculating the S-metric. Conceptual simplifications of the implementation are concerned that save on a factor of $O(\log n)$ and establish an upper bound of $O(n \log n + n^{d/2})$ for the S-metric calculation, improving the previously known bound of $O(n^{d-1})$.

Key Words
Multi-objective optimization, evolutionary algorithms, performance assessment, hypervolume, S-metric, Klee's measure problem.

1 Introduction

In multi-objective optimization d objective functions $f = (f_1, \ldots, f_d)$ are given with f_i to be minimized. Since these objectives typically are conflicting, we do not search for one optimal solution, but for a set of good compromise solutions. Vectors $\mathbf{a}$ and $\mathbf{b}$ are assumed to be d-dimensional vectors composed of objective values of d minimization problems. These vectors are partially ordered according to the component-wise order. A vector $\mathbf{a}$ *weakly dominates* vector $\mathbf{b}$ ($\mathbf{a} \preceq \mathbf{b}$) if $a_i \leq b_i$ for all $i \in \{1, \ldots, d\}$. If ($\mathbf{a} \preceq \mathbf{b}$) holds and additionally $\mathbf{a} \neq \mathbf{b}$, then $\mathbf{a}$ *dominates* $\mathbf{b}$, denoted ($\mathbf{a} \prec \mathbf{b}$). Distinct points $\mathbf{a}, \mathbf{b}$ are comparable if either $\mathbf{a} \preceq \mathbf{b}$ or $\mathbf{b} \preceq \mathbf{a}$, and incomparable otherwise. A set M is called *non-dominated* if no two elements exist that are comparable to each other according to the dominance relation. Minimal elements of the partially ordered domain of the d objectives are called Pareto optimal. The set of all Pareto optimal objective vectors is called *Pareto front*. For a comprehensive introduction to Pareto optimization with evolutionary algorithms see e.g. Deb [1] or Coello Coello et al. [2].

The purpose of Pareto optimization is to obtain a good approximation of the Pareto front. Approximations are non-dominated sets, whose elements shall be near to members of the Pareto front, be well distributed along the whole Pareto front and shall con-

tain many elements. The quality of the approximation of the Pareto front can be valued by various measures (called *metrics*). Among these metrics, the S-metric developed by Zitzler and Thiele [3] is of exceptional interest. It is considered to be a rather fair measure, because it has nearly optimal properties concerning the outperformance relations which transfer the partial order among vectors to sets of vectors. Considering two sets, the S-metric is the only unary measure that always values the better set higher and a higher value indicates that the set is not worse (cf. Zitzler et al. [4]).

The S-metric valuates a set of non-dominated solutions in the objective space by the hypervolume that is dominated by the set. The dominated hypervolume corresponds to the size of the region of the objective space (bounded by a reference point) which contains solutions being weakly dominated by at least one of the members of the set. The metric value is to be maximized. Each member $\mathbf{y}$ of a set M weakly dominates a region in the objective space shaped like an infinite hypercuboid $h^*(\mathbf{y}) = [y_1, \infty] \times \ldots \times [y_d, \infty]$ (in case the domain is infinite). These hypercuboids become finite by bounding them with a reference point $\mathbf{r}$, which has to be dominated by each member of the set M: $h(\mathbf{y}) = [y_1, r_1] \times \ldots \times [y_d, r_d]$. The S-metric is the hypervolume of the union of the weakly dominated hypercuboids, whereas severally covered regions are counted once. The formal definition is based on the Lebesgue measure Λ:

$$S(M, \mathbf{r}) = \Lambda\left(\left\{\bigcup h(\mathbf{y}) \mid \mathbf{y} \in M\right\}\right). \tag{1}$$

Two algorithms have been developed for calculating the S-metric, namely LebMeasure by Fleischer [5] and HSO described independently by Zitzler [6] and Knowles [7]. These algorithms partition the covered space into many cuboid-shaped regions, whereas HSO is regarded as the better one. In the worst case of HSO, the space is partitioned into $\binom{n+d-2}{d-1}$ cuboids (cf. While et al. [8]), resulting in a run time of $O(n^{d-1})$. Recently While et al. [9] developed heuristics for HSO, which reorder the input such that the worst possible case is avoided. Nevertheless, it is unknown how far this improves the exponential order of the worst case upper bound. Thus $O(n^{d-1})$ has been the best known upper bound of the S-metric which is significantly improved by the algorithm presented here.

The following section illustrates the relationship between the dominated hypervolume or S-metric and KMP. In Section 3, the main ideas of the fastest known algorithm for KMP are described and the simplified, adapted algorithm is presented. Section 4 explains lower and upper bounds for KMP and the S-metric calculation with the mentioned algorithms. Finally, the last section summarizes the main results and gives hints on the application of the algorithms and topics of future research.

2 Conversion of S-Metric to KMP

Klee's measure problem (KMP) (Klee [10]) has originally been formulated as calculating the size of the union of a set of n real-valued intervals. Generalized to d dimensions, the intervals become d-dimensional axis-parallel hypercuboids (Bentley [11]).

Beume [12] describes the trivial conversion of the dominated hypervolume to KMP as the dominated region of each point of a non-dominated set is a hypercuboid.[1] To transform a non-dominated set to a valid input set of KMP, each objective vector is replaced by its weakly dominated cuboid. Speaking in perception of intervals, the objective vector provides the lower bounds of the d-dimensional intervals and the reference point the upper ones. Independent of its dimension, a hypercuboid is completely defined by providing two corners on a space diagonal. Here, we define the 'lower left' (the former objective vector) and the 'upper right' (the former reference point) corners.

For KMP, the cuboids may be positioned arbitrarily. For the considered special case of calculating the S-metric, some properties of the set of cuboids can directly be derived from the definitions above:

- All cuboids have the same upper bounds (upper right corner), namely the coordinates of the reference point.
- No cuboid is completely contained within others since the lower bounds stem from non-dominated points. We assume that the input set does not contain copies of points.

Next, we transfer the vocabulary of relations of partially ordered points into terms of geometry. A point *covers* a region if it weakly dominates its lower boundary, thus the region is completely contained in the cuboid induced by the point. A point *partially covers* a region if its induced cuboid intersects the region. The point may weakly dominate the region's lower bound, be incomparable to it, or be dominated by it while dominating its upper bound. Beware that during the algorithm, points of different dimension have to be considered. The non-dominated points are d-dimensional and the regions in the orthogonal partition tree are $(d-1)$-dimensional as described in the following section. For the definitions above only the first $(d-1)$ components of a point are considered.

3 S-Metric Algorithm adopted from KMP Algorithm

3.1 Basic Concept and Decisive Especialness

Overmars and Yap [13] developed a sweep-line algorithm that uses a specific data structure to calculate a $(d-1)$-dimensional volume and performs a sweep along the remaining dimension to get the d-dimensional measure. For the partitioning of the $(d-1)$-dimensional space into regions, a data structure called *orthogonal partition tree* is used, that is a binary space partition tree whose splitting lines are extensions of the axis-parallel cuboids. An example of a non-dominated set is pictured in Figure 1.

The significant idea of Overmars and Yap's algorithm is to not partition the space into empty and covered regions, but stopping the partitioning as soon as a region contains a *trellis*. In a trellis, the cuboids that intersect the region, cover it completely in each of the $(d-1)$ dimensions except one. An example of this structure is shown in Figure 2. A cuboid that does not cover the ith dimension completely is called an i-*pile*. For each dimension i, the 1-dimensional KMP of the projection of the i-piles on the

[1] For convenience we will omit the prefix 'hyper' and talk of 'volume' and 'cuboid' in arbitrary dimension.

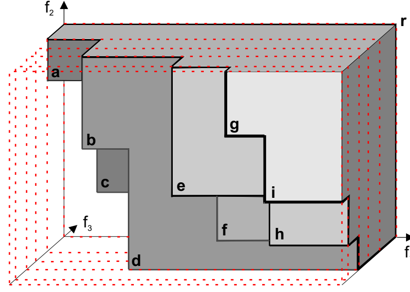

Fig. 1. The figure displays a non-dominated set of nine 3-dimensional points. The dominated volume is bounded by the reference point $\mathbf{r}$. Along each dth coordinate, the d-dimensional space is cut into $(d-1)$-dimensional slices that are stored in the orthogonal partition tree. In the example, the 2-dimensional slices are shown by the dashed lines. The d-dimensional volume is calculated by computing the $(d-1)$-dimensional volume with the help of the orthogonal partition tree and sweeping along the slices in dimension d.

i^{th} coordinate axis is solved. Thereby the exact position of the cuboids is neglected. Let K_i denote the value of the 1-dimensional KMP of the i-piles, and L_i denote the size of the region in dimension i, respectively. The contained volume of the region is calculated by the inclusion-exclusion principle (cf. Overmars and Yap [13]) in constant time, assuming d is a constant:

$$\sum_{1 \leq a \leq d-1} (-1)^{a+1} \left(\sum_{1 \leq j_1 < \ldots < j_a \leq d-1} \left(\prod_{1 \leq i \leq a} K_{j_i} \prod_{l \in \{j_1, \ldots, j_a\} \wedge l \neq j_i} L_l \right) \right) . \quad (2)$$

For clarification, we consider a 3-dimensional KMP with 2-dimensional volume in the regions. Then the volume is calculated as: $L_1 K_2 + L_2 K_1 - K_1 K_2$.

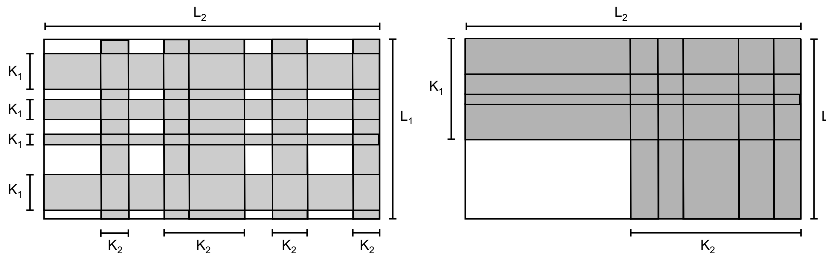

Fig. 2. The left figure shows an example of a trellis for the general KMP. The structure on the right arises for the specific problem of calculation the S-metric, whenever the condition of a trellis is fulfilled.

3.2 Progression of the Algorithm

Overmars and Yap [13] describe two variants of their algorithm. On the one hand the orthogonal partition tree is build up completely in a preprocessing step and the sweep is performed afterwards, inserting beginning cuboids into the data structure and removing enclosed ones. On the other hand, the data structure is build on the fly by splitting the current node if necessary. By recursing on the left child before the right, the partition tree is traversed in pre-order and the sweep is simulated whenever a leaf node is reached. This technique refers back to Overmars and Edelsbrunner [14] and is called *streaming*. The orthogonal partition tree requires $O(n^{d/2})$ storage, whereas the streaming variant works with linear space as only one node is considered at one time. Thus it is to be preferred, easier to implement, and even more efficient because some special cases can be handled easier. Here, the algorithm based on the streaming variant and adapted to the S-metric calculation (cf. Algorithm 1) is described in detail with remarks to differences to the original one by Overmars and Yap [13].

The main procedure of the algorithm has the following parameters.

`double[][] region` The current region is represented by a two-dimensional array containing the vectors of the lower bounds and the upper bounds.

`list points` Points whose induced cuboids partially or completely cover `region` are stored in a list `points`.

`int split` The dimension at which `region` is cut to generate two child regions is called `split`.

`double cover` The value of the dth coordinate of the first cuboid that covers the parent node's region is stored in `cover`.

Inputs of the algorithm are a set of non-dominated points and a reference point, thus the cuboids are represented implicitly. The reference point $\mathbf{r}$, the initial size n of the input set, and the dimension d are assumed to be known globally. Before the main procedure `volumeOY` starts, the list of points is sorted ascending according to the dth component of the vectors. This sorting will be maintained stable in all recursive calls of `volumeOY`. The procedure is initially called with the whole $(d-1)$-dimensional space as `region`, the non-dominated input set as `points`, `split`$= 1$ and `cover` as the dth coordinate of the reference point $\mathbf{r}$. A small example with $n = 9$ points in $d = 3$ dimensions is pictured in Figure 3.

The algorithm recursively splits the region, whereas the two resulting regions correspond to the children nodes within the binary tree. The splitting ends when the region contains a trellis, thus a leaf node is reached and the volume can be calculated. The procedure `volumeOY` consists of three parts. First it is checked if a cuboid covers `region`. If the remaining cuboids form a trellis, their hypervolume is calculated. Otherwise the region is further partitioned and the volume is calculated in recursive calls.

The dth coordinate of the first covering point is saved as `coverNew` and the corresponding index in `points` as `coverIndex`. The volume is increased by the region's complete $(d-1)$-dimensional volume multiplied with the distance of `coverNew` to `cover`. Since the list `points` is sorted according to the dth coordinate component, the points behind `coverIndex` do not add volume. These and the point itself are discarded in the remainder of this call of the procedure by considering `points` only

```
coverNew = cover; coverIndex=1; allPiles = true; bound = -1

/*  is the region completely covered?  */
while (coverNew == cover  &&  coverIndex!= points.length) do
    if covers(points[coverIndex], region) then
        coverNew = points[coverIndex][d]
        volume += getMeasure(region) * (cover - coverNew)
    else  coverIndex++
if coverIndex==1 then  return

/*  do the cuboids form a trellis?  */
for i=1 to coverIndex-1 do
    if !isPile(points[i], region) then  allPiles = false
if allPiles then
    /*  calculate volume by sweeping along dimension d  */
    i = 1;  for j=1 to d-1 do  trellis[j] = r[j]
    repeat
        current = points[i][d]
        repeat
            pile = getPile(points[i], region)
            if points[i][pile] < trellis[pile] then  trellis[pile] = points[i][pile]
            i++
            if i<coverIndex-1 then  next = points[i][d]  else next = coverNew
        until current != next
        volume += measure(trellis, region) * (next - current)
    until next == coverNew
else
    /*  split region in two children regions  */
    repeat
        intersect = ∅; nonIntersect = ∅
        for i=1 to coverIndex-1 do
            intersection = intersects(points[i], region, split)
            if intersection == 1 then  add(points[i][split], intersect)
            if intersection == 0 then  add(points[i][split], nonIntersect)
        if intersect ≠ ∅ then  bound = median(intersect)
        else if nonIntersect.length > √n then  bound = median(nonIntersect)
        else  split++
    until bound != -1

    /*  recurse on the two children regions  */
    regionC = region; regionC[1][split] = bound; pointsC = ∅
    for i to coverIndex-1 do
        if partCovers(points[i], regionC) then  move(points[i], pointsC)
    if pointsC ≠ ∅ then  volumeOY(regionC, pointsC, split, coverNew)
    reinsert(pointsC, points);
    regionC = region; regionC[0][split] = bound; pointsC = ∅
    for i to coverIndex-1 do
        if partCovers(points[i], regionC) then  move(points[i], pointsC)
    if pointsC ≠ ∅ then  volumeOY(regionC, pointsC, split, coverNew)
    reinsert(pointsC, points)
```

Algorithm 1: volumeOY(region, points, split, cover)

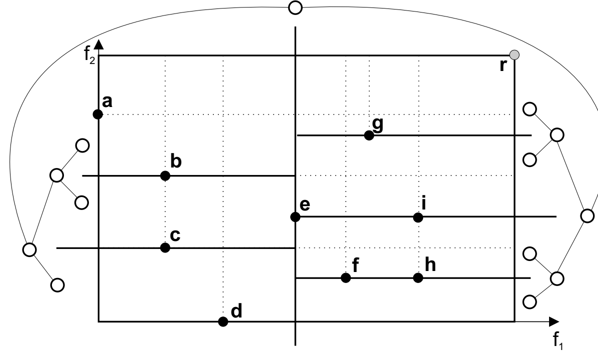

Fig. 3. Illustration of the 2-dimensional orthogonal partition tree for a 3-dimensional KMP. The non-dominated set of Figure 1 is projected on the first two dimension. The lines show the partitioning of the 2-dimensional space, which is upper bounded by the gray reference point **r** and lower bounded by the contained points. The dotted lines adumbrate their induced dominated cuboids. The orthogonal partition tree is depicted, whereas the nodes are placed alongside their associated region. The sweep is performed along the third dimension f_3.

to index (`coverIndex`-1). The rear points are still required on higher levels of recursion. If `coverIndex`$=1$, `volumeOY` is aborted because no points are left. In the original description, only covering cuboids are removed. We added here, that covered cuboids are also discarded. This is a valid supplement to the algorithm for the general KMP, too.

In the second part of `volumeOY`, it is checked if the induced cuboids form a trellis. If so, the sweeping along the dth dimension is performed to calculate the contained volume. The points with the first d^{th} coordinate (equal values may occur) are considered and $(d-1)$ 1-dimensional KMP are solved for them. The $(d-1)$-dimensional volume is calculated by the inclusion-exclusion-principle according to Eq. 2 and multiplied with the distance to the next d^{th} coordinate. This is done for all consecutive d-boundaries and the last distance in dimension d is calculated as difference to `coverNew`.

To solve a 1-dimensional KMP on piles, Overmars and Yap invoked a segment tree to calculate the union of the 1-dimensional intervals. For the special case of the S-metric calculation, this can be done significantly faster and segment trees are not necessary. In case the cuboids fulfill the condition of a trellis, they actually form an even simpler structure. An example is shown in Figure 2 (right). Since each cuboid extends to the reference point in each dimension, no upper bounds of cuboids are contained inside of the current region. A region may only contain lower boundaries and the remainder of the region is covered from thereon. Thus, only the minimal i^{th} coordinate of the i-piles has to be identified. The result of the 1-dimensional KMP is the difference of this value to the region's upper bound, respectively in each dimension. The minimal values are stored in a $(d-1)$-dimensional array called `trellis`. Cuboids that become active during the sweep procedure are checked if they undercut the current values in trellis. Then `trellis` is updated in constant time by just one comparison. The update of

the originally applied segment tree takes time $O(\log n)$. This factor is saved on by this adapted algorithm.

If the cuboids do not form a trellis, the region is split in two and the algorithm proceeds on the two emerged regions. The partitioning aspires that no points are contained inside of a region. To this end, the dimension that is cut by the splitting hyperplane is to be determined. As the cuboids are axis-parallel, the ith coordinate of a point induces a so-called i*-boundary* that is a hyperplane which cuts through the ith coordinate axis and is parallel to all others. The sub-procedure `intersects` detects those points that induce a `split`-boundary inside of the region. Points that additionally induce an i-boundary with $i <$`split` are stored in a list `intersect`, the others in `nonIntersect`. By recursion, the region will be split along each of the `split`-boundaries of the points in `intersect`. In each call of `volumeOY`, the median of these `split`-boundaries is chosen as the splitting hyperplane. This choice takes time $O(coverIndex)$. If `intersect` is empty, but `nonIntersect` contains more than $\sqrt{n}$ `split`-boundaries the region is split along the median of these.[2] If `intersect` is empty and there are not more than $\sqrt{n}$ `split`-boundaries in `nonIntersect`, `split` is increased and the search for the splitting line is tried again, beginning with the sub-procedure `intersects`.

In the example of Figure 3, the space is split once along the median 1-boundary. Afterwards, each region contains not more than $\sqrt{9} = 3$ 1-boundaries and `split` is increased. The left region is split along the median 2-boundary of those points that establish a 1-boundary within the region. Concerning the left child region, the point $\mathbf{d}$ is a 1-pile and no further partitioning is required. The right child region is split again because the point $\mathbf{d}$ is located inside of it.

Knowing the splitting line, the left child region is defined accordingly. Points that partially cover the child's region are sent down to recursion, together with the child region itself, the split value, and the value of `coverNew` of the current region. Afterwards, the points are reunited with the list `points` and the recursion on the right child's region is performed analogously.

Note that points are never copied, but moved from `points` to other lists if necessary. Thus, recursion does not cause any increase of storage, since each point is stored at only one place. Invoking pointers to the elements in `points` would also be possible as their amount of storage is marginal. All lists of points are sorted, since this is done in the pre-processing step. Whenever a list is to be reunited with `points`, this can be done in linear time, whereas the sorting is maintained.

In Overmars and Yap's algorithm, the lists `intersect` and `nonIntersect` are considered as sets, thus without copies. Here, we do not reject copies for reasons of efficiency. The median can be chosen in linear time, whereas the rejection of copies requires time $O(n \log n)$. The search for copies could be afforded in the original algorithm since a sorting is done anyway which requires $O(n \log n)$ and enables the deletion of copies in linear time.

Details on the implementation of the sub-procedures invoked during `volumeOY` (Algorithm 1) are described in Appendix A.

[2] The list `intersect` is especially empty for `split=1`. Thus, the points are partitioned into subsets of size $O(\sqrt{n})$ during the beginning phase.

4 Runtime Analysis

4.1 Lower Bounds

Klee's measure problem has a lower bound of $\Omega(n \log n)$ for $d \geq 1$ shown by Fredman and Weide [15]. The S-metric has a complexity of $\Theta(n)$ in case of $d = 1$ as only the minimal element has to be determined. As the S-metric is a quality measure for results of a multi-objective optimization process, its definition only makes sense for $d \geq 2$. Obviously, calculating the S-metric is not harder than solving KMP as it is a special case of it, though it is unknown if it is significantly easier. It remains an open question, if the S-metric has a lower bound which is smaller than the one of Klee's measure problem.

4.2 Upper Bounds

In case of $d = 2$, the S-metric can be calculated in time $O(n \log n)$. The input set is sorted according to one objective. Afterwards, the covered area can be divided into rectangles bounded by the neighboring point in one dimension and by the reference point in the other dimension. The S-metric value of a set $M = \{\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)}\}$ can be calculated as:

$$S(M, \mathbf{r}) = (r_1 - y_1^{(1)})(r_2 - y_2^{(1)}) + \sum_{i=2}^{n} (r_1 - y_1^{(i)})(y_2^{(i-1)} - y_2^{(i)}) \tag{3}$$

For $d \geq 3$, the algorithm of Overmars and Yap is applicable, which provides an upper bound of $O(n \log n + n^{d/2} \log n)$ for the calculation of the S-metric. This adapted algorithm (Algorithm 1) for computing the S-metric has a run time of $O(n \log n + n^{d/2})$. The factor $\log n$ is saved on omitting the segment trees and calculating the one-dimensional KMPs of a trellis in constant time as described afore.

The two variants—the classical one and the streaming technique (cf. Section 3.2)—of the algorithm by Overmars and Yap [13] have the same run time. Actually the same operations are done, though in different order. They describe the analysis for the variant which completely builds the orthogonal partition tree before the sweep. The pre-processive sorting requires $O(n \log n)$. It is shown that a cuboid is stored in $O(n^{(d-2)/2})$ leaves of the partition tree as the partitioning ensures that this is an upper bound for the number of partially covered regions.[3] The contained volume within these leaves has to be updated when the cuboid is inserted or removed from the orthogonal partition tree during the sweep. Thus, over all cuboids there are $O(n^{(d-2)/2} \cdot n) = O(n^{d/2})$ updates. Updating means computing the measure in the trellis which takes time $O(\log n)$ with the help of the segment trees. The adapted algorithm (Algorithm 1) computes an update in constant time. Thus, the original algorithm has a run time of $O(n \log n + n^{d/2} \log n)$ and the adapted one only $O(n \log n + n^{d/2})$.

[3] Details of the proof are described in Appendix B.

5 Summary and Outlook

Klee's measure problem (KMP) is characterized as the hypervolume of intersecting axis-parallel hypercuboids. It is similar to the S-metric that is defined as the dominated space of a non-dominated set. Since the dominated regions of points actually are axis-parallel hypercuboids, algorithms for KMP can be applied almost directly. The hypercuboids form a certain structure that makes the S-metric easier to calculate than the general KMP. The fastest known algorithm for KMP from Overmars and Yap has been adapted to that special case resulting in an upper bound of $O(n \log n + n^{d/2})$ for the S-metric calculation. The algorithm performs a partitioning of the space and the calculation of the hypervolume within the cells allows for faster computation due to the special configuration of the hypercuboids. The description of the original algorithm is rather complex, the deduced algorithm is completely presented in pseudo code and requires only fundamental data structures.

In the scope of multi-objective optimization, the S-metric is not only used as a quality measure but also as a component of evolutionary multi-objective optimization algorithms (EMOA). The S-Metric Selection EMOA (SMS-EMOA) by Emmerich et al. [16, 17] integrates the maximization of the population's S-metric value into the EMOA to guide it during the optimization process. The run time of this algorithm is $O(\mu \log \mu + \mu^{(d/2)+1})$ based on μ S-metric calculations per generation to determine the following population, with μ denoting the population size. Other EMOA invoke an approximation of the S-metric such as Zitzler and Künzli's IBEA (Indicator-based evolutionary algorithm) [18] and the ESP (Evolution Strategy with Probabilistic mutation) developed by Huband et al. [19]. A topic of future research is the question whether the run time of the SMS-EMOA can be further reduced by an efficient update of the information of the hypervolume in consecutive iterations.

Studies on test data of differently structured non-dominated sets are to be accomplished providing numerical comparison of the CPU time of hypervolume algorithms. Additionally, it is planned to design an approximation algorithm for the S-metric based on the algorithm of Overmars and Yap and the adapted one presented here.

A Details on Sub-procedures of the Adapted Algorithm

Details of the sub-procedures invoked by `volumeOY` are described in the following. A mathematical description of the used variables are given next to a possible solution of implementation, which is maybe not optimally efficient but easy to understand. Beware the sequence of the sub-procedures within `volumeOY`. Conditions that are assured by previous sub-procedures can be assumed without repeated checks.

The procedure `partCovers` determines the points that partially cover the considered child region. It is called by `partCovers(points[i], regionC)`, with $i = 1$ to `coverIndex-1` and the currently considered child region. The resulting set `pointsC` is defined as:

$$pointsC = \{points[i] \mid \forall j : \ points[i][j] < region[1][j]\} \tag{4}$$

for *j=1* **to** *d-1* **do**
  ⌊ **if** *points[i][j]* $>=$ *region[1][j]* **then** return false
return true

Algorithm 2: partCovers(points[i], region)

The variables `coverNew` and `coverIndex` are calculated with the information provided by the procedure `covers` which decides whether a point covers the region. It is called by `covers(points[i], region)`, with $i = 1$ to $|$`points`$|$ and the current region.

$$coverNew = \min_{i \in \{1,...,|points|\}} \{points[i][d] \mid \forall j : points[i][j] \leq region[0][j] \; ; \; cover\} \tag{5}$$

The index of the minimizing argument is named `coverIndex`.

for *j=1* **to** *d-1* **do**
  ⌊ **if** *points[i][j]* $>$ *region[0][j]* **then** return false
return true

Algorithm 3: covers(points[i], region)

The sub-procedure `intersects` detects whether a points `split`-boundary is a candidate for the splitting line that partitions the region in two child regions. It is called for all points in the data structure `points` with index $i = 1...$`coverIndex`-1. At the beginning of the procedure, it is checked whether the point's `split`-boundary is contained inside the region. It is sufficient to test whether the splitting boundary is greater than the region's lower bound. The algorithm already assured that the cuboid partially covers the region, thus it is not necessary to check if the boundary is higher than the region's boundary. The `split`-boundaries of points that induce an i-boundary with $i <$`split` are stored in the list `intersect` and the others in `nonIntersect`.

$$\begin{aligned} intersect = \{ \; &points[i][split] \;\mid\; region[0][split] < points[i][split] \;\wedge \\ &\exists j \in \{1, .., split - 1\} : \; points[i][j] > region[0][j] \; \} \end{aligned} \tag{6}$$

$$\begin{aligned} nonIntersect = \{ \; &point[i][split] \;\mid\; region[0][split] < points[i][split] \;\wedge \\ &\forall j \in \{1, .., split - 1\} : \; points[i][j] <= region[0][j] \; \} \end{aligned} \tag{7}$$

if *region[0][split]* $\geq$ *points[i][split]* **then** return -1
for *j=1* **to** *split-1* **do**
  ⌊ **if** *points[i][j]* $>$ *region[0][j]* **then** return 1
return 0

Algorithm 4: intersects(points[i], region, split)

Recalls that a cuboid is a pile w.r.t. `region` if is covers the region completely in each dimension but one. The procedure `checkPile` returns the sole dimension that is not completely covered if the cuboid induced by the point is a pile. Otherwise `checkPile` returns -1 as an indicator for failure.

$$pile = \begin{cases} j, \text{ if } \exists! j \in \{1, ..., d-1\} : \; points[i][j] > region[0][j] \\ -1, \text{ otherwise} \end{cases} \tag{8}$$

```
pile = -1
for j=1 to d-1 do
    if points[i][j] > region[0][j] then
        if pile != -1 then return -1
    pile = j
return pile
```

Algorithm 5: checkPile(points[i], region)

The $(d-1)$-dimensional volume formed by a trellis, is calculated by the sub-procedure `measure`, called with the current region and the array `trellis` which stores at index i the minimal i^{th} coordinate of the i-piles. The volume is calculated by the inclusion-exclusion principle according to Eq. 2. Each summand is composed of $(d-1)$ factors corresponding to the $(d-1)$ dimensions. The ith factor is either the size of the region in dimension i or the value of a 1-dimensional KMP of the contained i-piles. The sign of a summand depends on the number of KMP factors. The formula (Eq. 2) contains $q = \sum_{k=1}^{d-1} \binom{d-1}{k}$ summands, whereas q accords to the variations of k KMP values out of $(d-1)$ factors. A possible implementation applies an index vector to determine whether a factor is to be a 1-dimensional KMP of the i-piles or the size of the region in dimension i. In the array `indicator` of length $(d-1)$, `indicator[i]=1` corresponds to the ith KMP and `indicator[i]=0` to the size of the region in dimension i. This way, all possible variations can be processed by assigning the indicator vector the binary presentation of the numbers from 1 to q.

```
for i=1 to d-1 do indicator[i]=1
numberSummands = integerValue(indicator)
for i=1 to numberSummands do
    indicator = binaryValue(i)
    oneCounter = 0
    for j=1 to d-1 do
        if indicator[i] == 1 then
            summand += region[1][j] - trellis[j]
            oneCounter++
        else summand += region[1][j] - region[0][j]
    if oneCounter mod 2 == 0 then
        volume -= summand
    else volume += summand
```

Algorithm 6: measure(trellis, region)

B Details on the Proof of the Run Time's Upper Bound

The proof of the upper bound of the run time (cf. Section 4.2) is based on the number of partially covered regions per cuboid. Here, the explanation that this number does not exceed $O(\sqrt{n}^{d-2})$ is given. Recall that a cuboid partially covers a region if an i-boundary cuts through the region. This characteristic is illustrated in Figure 4. A region that is generated by a splitting through dimension i is termed an *i-partition*. There are

$O(\sqrt{n}^{i-1})$ $(i-1)$-partitions. The i-boundary of a cuboid intersects an $(i-1)$-partition at most once and thereby cuts through one of its i-partition. Thus an i-boundary intersects $O(\sqrt{n}^{i-1})$ i-partitions. When the partitioning is done concerning the remaining $(d-1-i)$ dimensions, each i-partition is subdivided into $O(\sqrt{n}^{d-1-i})$ $(d-1)$-partitions. The cuboid's i-boundary cuts $O(\sqrt{n}^{i-1}) \cdot O(\sqrt{n}^{d-1-i}) = O(\sqrt{n}^{d-2})$ $(d-1)$-partitions, which corresponds to the number of leafs that contain the cuboid.

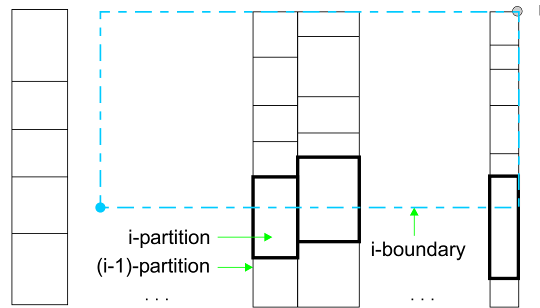

Fig. 4. Illustration of the number of intersected regions. The dashed lines adumbrate the induced dominated hypercuboid. The columns show $(i-1)$-partitions with their contained i-partitions. The bold i-partitions are intersected by the i-boundary of the hypercuboid.

Acknowledgments

This work was partly supported by the *Deutsche Forschungsgemeinschaft (DFG)* as part of the *Collaborative Research Center 'Computational Intelligence' (SFB 531)*.

References

1. Deb, K.: Multi-Objective Optimization using Evolutionary Algorithms. Wiley, Chichester, UK (2001)
2. Coello Coello, C.A., Van Veldhuizen, D.A., Lamont, G.B.: Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers, New York (2002)
3. Zitzler, E., Thiele, L.: Multiobjective Optimization Using Evolutionary Algorithms—A Comparative Case Study. In Eiben, A.E., et al., eds.: Parallel Problem Solving from Nature (PPSN V). LNCS 1498, Springer, Berlin (1998) 292–301
4. Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., da Fonseca, V.G.: Performance assessment of multiobjective optimizers: An analysis and review. IEEE Transactions on Evolutionary Computation **7**(2) (2003) 117–132
5. Fleischer, M.: The Measure of Pareto Optima. Applications to Multi-objective Metaheuristics. In Fonseca, C.M., et al., eds.: Evolutionary Multi-Criterion Optimization, 2nd Int'l Conf. (EMO 2003). LNCS 2632, Springer, Berlin (2003) 519–533

6. Zitzler, E.: Hypervolume Metric Calculation. Computer Engineering and Networks Laboratory (TIK), Zürich (2001) ftp://ftp.tik.ee.ethz.ch/pub/people/zitzler/hypervol.c.
7. Knowles, J.D.: Local Search and Hybrid Evolutionary Algorithms for Pareto Optimization. PhD thesis, University of Reading, Reading, UK (2002)
8. While, L., Hingston, P., Barone, L., Huband, S.: A faster algorithm for calculating hypervolume. IEEE Transactions on Evolutionary Computation **10**(1) (2006) 29–38
9. While, L., Bradstreet, L., Barone, L., Hingston, P.: Heuristics for Optimising the Calculation of Hypervolume for Multi-objective Optimisation Problems. In McKay, B., et al., eds.: Proc. of the 2005 Congress on Evolutionary Computation (CEC 2005), Edinburgh. Volume 3., IEEE Press, Piscataway NJ (2005) 2225–2232
10. Klee, V.: Can the measure of $\bigcup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? In: American Mathematical Monthly. Volume 84. (1977) 284–285
11. Bentley, J.L.: Algorithms for Klee's rectangle problem. Unpublished notes, Department of Computer Science, CMU (1977)
12. Beume, N.: Hypervolumen-basierte Selektion in einem evolutionären Algorithmus zur Mehrzieloptimierung. Diploma thesis (March, 2006), University of Dortmund (2006)
13. Overmars, M.H., Yap, C.K.: New upper bounds in Klee's measure problem. SIAM Journal on Computing **20**(6) (1991) 1034–1045
14. Edelsbrunner, H., Overmars, M.H.: Batched dynamic solutions to decomposable searching problems. Journal of Algorithms **6**(4) (1985) 515–542
15. Fredman, M.L., Weide, B.: The complexity of computing the measure of $\bigcup[a_i, b_i]$. In: Communications of the ACM. Volume 21. (1978) 540–544
16. Emmerich, M., Beume, N., Naujoks, B.: An EMO algorithm using the hypervolume measure as selection criterion. In Coello Coello, C.A., et al., eds.: Evolutionary Multi-Criterion Optimization: 3rd Int'l Conf. (EMO 2005), Springer, Berlin (2005) 62–76
17. Beume, N., Naujoks, B., Emmerich, M.: SMS-EMOA: Multiobjective selection based on dominated hypervolume. European Journal of Operational Research (2006) (In print).
18. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In Yao, X., et al., eds.: 8th Int'l Conf. on Parallel Problem Solving from Nature (PPSN VIII). LNCS 1498, Springer, Berlin (2004) 832–842
19. Huband, S., Hingston, P., While, L., Barone, L.: An evolution strategy with probabilistic mutation for multi-objective optimisation. In: Proc. of the Congress on Evolutionary Computation (CEC 2003), Canberra, Australia. Volume 4., IEEE Press, Piscataway NJ (2003) 2284–2291