Parameterized Algorithms for Computing Pareto Sets

Joshua Marc Könen
Institute of Computer Science, University of Bonn, Germany

Heiko Röglin
Institute of Computer Science, University of Bonn, Germany

Tarek Stuck
Institute of Computer Science, University of Bonn, Germany

Abstract

The problem of computing the set of Pareto-optimal solutions has been studied for a variety of multiobjective optimization problems. For many such problems, algorithms are known that compute the Pareto set in (weak) output-polynomial time. These algorithms are often based on dynamic programming and by weak output-polynomial time, we mean that the running time depends polynomially on the size of the Pareto set but also on the sizes of the Pareto sets of the subproblems that occur in the dynamic program. For some problems, like the multiobjective minimum spanning tree problem, such algorithms are not known to exist and for other problems, like multiobjective versions of many NP-hard problems, such algorithms cannot exist, unless $\mathcal{P} = \mathcal{NP}$.

Dynamic programming over tree decompositions is a common technique in parameterized algorithms. In this paper, we study whether this technique can also be applied to compute Pareto sets of multiobjective optimization problems. We first derive an algorithm to compute the Pareto set for the multicriteria s-t cut problem and show how this result can be applied to a polygon aggregation problem arising in cartography that has recently been introduced by Rottmann et al. (GIScience 2021). We also show how to apply these techniques to also compute the Pareto set of the multiobjective minimum spanning tree problem and for the multiobjective TSP. The running time of our algorithms is $\mathcal{O}(f(w) \cdot \text{poly}(n, p_{\max}))$, where f is some function in the treewidth w, n is the input size, and $p_{\max}$ is an upper bound on the size of the Pareto sets of the subproblems that occur in the dynamic program. Finally, we present an experimental evaluation of computing Pareto sets on real-world instances of polygon aggregation problems. For this matter we devised a task-specific data structure that allows for efficient storage and modification of large sets of Pareto-optimal solutions. Throughout the implementation process, we incorporated several improved strategies and heuristics that significantly reduced both runtime and memory usage, enabling us to solve instances with treewidth of up to 22 within reasonable amount of time. Moreover, we conducted a preprocessing study to compare different tree decompositions in terms of their estimated overall runtime.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases parameterized algorithms, treewidth, multicriteria optimization problems, multicriteria MST, multicriteria TSP, polygon aggregation

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.105

Related Version *Full Version*: `https://arxiv.org/abs/2509.06124`

Supplementary Material *Software*: `https://github.com/Tarek-pub/Bicriteria_Aggregation` [19]
archived at `swh:1:dir:b6eacf189792239e9115cf288be27c6753b8df17`
InteractiveResource: `https://github.com/Tarek-pub/Bicriteria_Aggregation_plotting` [20]

Funding This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 390685813; 459420781 and by the Lamarr Institute for Machine Learning and Artificial Intelligence lamarr-institute.org.

© Joshua Marc Könen, Heiko Röglin, and Tarek Stuck;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Multiobjective optimization problems arise naturally in many contexts. When booking a train ticket, one might be interested in minimizing the travel time, the price, and the number of changes. Similarly when planning a new infrastructure, companies often have to find a compromise between reliability and costs, to name just two illustrative examples. For multiobjective optimization problems there is usually not a single solution that is optimal for all criteria simultaneously, and hence, one has to find a trade-off between the different criteria. Since it is not a priori clear which trade-off is desired, one often studies the set of *Pareto-optimal solutions* (also known as *Pareto set*), where a solution is Pareto-optimal if it is not dominated by any other solution, i.e., there does not exist another solution that is better in at least one criterion and at least as good in all other criteria. The reasoning behind this is that a dominated solution cannot be a reasonable compromise of the different criteria and it makes sense to restrict one's attention to only the Pareto-optimal solutions.

A lot of research in multiobjective optimization deals with algorithms for computing the Pareto set and with studying the size of this set for various optimization problems. If this set is not too large then it can be presented to a (human) decision maker who can select a solution. The Pareto set also has the important property that any monotone function that combines the different objectives into a single objective is optimized by a Pareto-optimal solution. This means, in order to optimize such a function, one could first generate the Pareto set and then pick the best solution from this set.

This approach is only reasonable if there are not too many Pareto-optimal solutions. While in the worst case the Pareto set can be of exponential size for almost all multiobjective optimization problems, it has been observed that the Pareto set is often small in practice (see, e.g., [16, 23]). It has also been shown in the probabilistic model of smoothed analysis that for many problems the expected size of the Pareto set is polynomial if the input is subject to a small amount of random noise [10, 5]. This motivates the development of algorithms for computing the Pareto set that have polynomial running time with respect to the output size.

In the literature such output-sensitive algorithms have been developed for different multiobjective optimization problems (e.g., for the multiobjective shortest path problem [12, 17, 29], multiobjective flow problems [15, 24], and the knapsack problem when viewed as a bicriteria optimization problem [25]). There is, however, a small caveat. Since these algorithms are usually based on dynamic programming, they are not output-polynomial in the strict sense but they solve certain subproblems of the given instance and their running times depend not only polynomially on the size of the Pareto set of the entire instance but also polynomially on the sizes of the Pareto sets of the subproblems. While in theory it could be the case that some of the subproblems have Pareto sets of exponential size while the Pareto set of the entire instance is small (see, e.g., [9]), this behavior is neither observed in experiments nor does it occur in probabilistic input models. Hence, algorithms that are output-polynomial in the weak sense that their running time depends polynomially on the sizes of the Pareto sets of all subproblems that occur in the dynamic program are useful and state-of-the-art for many multiobjective optimization problems. For some problems such output-sensitive algorithms are not known to exist. In particular, for the multiobjective spanning tree problem no algorithm is known that computes the Pareto set in output-polynomial time (not even in the weak sense).

Dynamic programming is a common technique in parameterized algorithms. For many graph problems, dynamic programming on tree decompositions is applied to obtain fixed-parameter tractable (FPT) algorithms with respect to the treewidth of the input graph.

In this paper, we explore for the first time the potential of dynamic programming on tree decompositions for computing Pareto sets. First we design an algorithm for the multicriteria s-t cut problem and apply it on a problem from cartography that has recently been introduced by Rottmann et al. [28] and that was the original motivation for our research.

Rottmann et al. study maps with building footprints and present a new model how less detailed maps can be derived from a given map. Their method is based on viewing the problem as a bicriteria optimization problem. It is assumed that the plane containing the building footprints is triangulated and a set of triangles to glue together some of the buildings is to be selected that on the one hand minimizes the total area and on the other hand minimizes the total perimeter. Rottmann et al. present an algorithm that computes the set of extreme nondominated solutions (i.e., the set of solutions that optimize a linear combination of the objectives), which is a subset of the Pareto set. They ask if it is also possible to compute the entire Pareto set in some output-efficient way.

We show how a treewidth-based algorithm can be used to compute the Pareto set for the s-t cut problem. The running time of this algorithm is FPT in the treewidth and output-polynomial in the weak sense, i.e., it is of the form $\mathcal{O}(f(w) \cdot \text{poly}(n, p_{\max}))$ where f is some function in the treewidth w, n denotes the input size, and $p_{\max}$ denotes an upper bound on the size of the Pareto sets of the subproblems that occur in the dynamic program.

As a special case, the results for the multiobjective s-t cut algorithm directly translate to the cartography problem by computing the whole Pareto set in FPT time and being output-polynomial in the weak sense. We experimentally analyze the computation of Pareto sets for this cartography problem on real-world datasets by designing a specialized data structure, developing heuristics, and integrating additional implementation-specific techniques tailored to the task.

1.1 Our Results

We first consider the multiobjective s-t cut problem and its special case, the triangle aggregation problem [28]: Given some polygons P and triangles T, the goal is to find all Pareto-optimal subsets $T' \subseteq T$ minimizing both the total area and perimeter of $P \cup T'$. This problem arises in cartography, where the goal is to compute a less detailed map from a given map. Here the polygons are building footprints and the space between these footprints is triangulated. By including triangles, one can glue together these building footprints, leading to a less detailed version of the map. We show that the multiobjective s-t cut problem implies an algorithm to compute the Pareto set in time $\mathcal{O}(nw \cdot 2^w \cdot p_{\max}^2 \log(p_{\max}))$ for this problem, where w is the treewidth of the triangle adjacency graph.

In the full version of the paper, we show that this method for computing Pareto sets can also be applied for other multiobjective optimization problems, such as the multiobjective minimum spanning tree problem (MST) and the multiobjective traveling salesman problem (TSP). For both, we present algorithms with running time $\mathcal{O}(nw^{\mathcal{O}(w)} \cdot p_{\max}^2 \log^{d-2}(w^{\mathcal{O}(w)} \cdot p_{\max}^2))$ where w is the treewidth of the input graph, n is the number of vertices in the graph, $p_{\max}$ is the size of the largest Pareto set computed in one of the subproblems of the dynamic program, and $d \geq 2$ is the (constant) number of different objectives. This shows that the Pareto set can be efficiently computed for graphs with small treewidth if the Pareto sets of the subproblems are not too large, which is often the case in realistic inputs.

Finally, we experimentally evaluate our algorithm on real-world datasets. We introduce new heuristic pruning techniques, runtime estimates for tree decompositions and their root, memory optimization, multi-threading, and other improvements for reducing runtime and memory usage.

1.2 Related Work

Output-polynomial time algorithms (at least in the weak sense) are known for many problems. Examples include the multiobjective shortest path problem [12, 17, 29], multiobjective flow problems [15, 24], and the knapsack problem when viewed as a bicriteria optimization problem [25]. For the multiobjective spanning tree problem such algorithms are not known and there are only results on the set of extreme nondominated solutions. This is a subset of the Pareto set and it contains all solutions that optimize some linear combination of the different objectives. For the multiobjective spanning tree problem it is known that there are only polynomially many extreme nondominated solutions [14], and efficient algorithms for enumerating them exist [2].

Rottmann et al. [28] introduce the triangle aggregation problem under the objective of map simplification by grouping multiple building footprints together. They show that, similar to the multiobjective minimum spanning tree problem, there exist only a polynomial number of nondominated extreme solutions and they present an algorithm to compute this set in polynomial time. They also provide an extensive experimental study of their algorithms on real-world data sets showing that the model captures nicely the intention to aggregate building footprints to obtain less detailed maps. They put it as an open question whether or not it is possible to also compute the set of Pareto-optimal solutions in an output-efficient way.

Motivated by the observation that for many problems the Pareto set is often small in applications, the number of Pareto-optimal solutions has been studied in the probabilistic framework of smoothed analysis in a sequence of articles [27, 22, 10, 5]. It has been shown for a large class of linear integer optimization problems (which contains in particular the multiobjective MST problem and the multiobjective TSP) that the expected number of Pareto-optimal solutions is polynomially bounded if the coefficients (in our case, the edge weights) are independently perturbed by random noise. Formally, the coefficients of $d-1$ out of the d objectives are assumed to be independent random variables with adversarially chosen distributions with bounded density. It is shown that not only the expected number of Pareto-optimal solutions is bounded polynomially in this setting but also all constant moments, so in particular the expected squared number of Pareto-optimal solutions is also polynomially bounded. Combined with this, our results imply that the Pareto set for the multiobjective MST problem, the multiobjective TSP and the multiobjective s-t cut problem can be computed in expected FPT running time $\mathcal{O}(f(w) \cdot \mathrm{poly}(n))$ in the model of smoothed analysis.

While dynamic programming over tree decompositions is a well-established technique, using it effectively in practice comes with more difficulties than simply minimizing the width of the decomposition. In experiments the runtime of an algorithm can vary largely for different tree decompositions with the same width.

Bodlaender and Fomin [8] introduced the concept of the *f-cost* of a tree decomposition, which sums up a cost function f applied to the bag sizes across all nodes. This framework formalizes the observation that not all nodes contribute equally to runtime or memory, and that practical efficiency can benefit from structurally favorable decompositions.

The f-cost approach was later extended and evaluated experimentally by Abseher et al. [1] using machine learning to select a tree decomposition based on different features such as bag sizes, branching factors, and node depths. With their method they could often select a decomposition that performs well in practice. Kangas et al. [18] used the same model for the problem of counting linear extensions, confirming the value of such estimators, and suggesting the average join-node depth as a good single feature for estimating runtime.

Beyond selection heuristics, tree decompositions have also been studied with respect to memory efficiency. Charwat et al. [11] explored compressed representations of intermediate states using decision diagrams, and Betzler et al. [6] proposed anchor-based compression techniques to reduce hard drive memory usage.

In our case, we not only solve an optimization problem, but compute the entire set of Pareto-optimal solutions. This makes memory usage more sensitive to the structure of join operations and limits the applicability of pointer-based or diagram-compressed representations. To address this, we develop a custom estimation strategy for predicting both runtime and memory usage, based on predicting the number of Pareto-optimal solutions at each node and the runtime impact of join operations. We then select a decomposition based on a combination of both runtime and memory usage. Details of this procedure are provided in the full version of the paper.

2 Preliminaries

Let $[i] := \{1, \ldots, i\}$. We consider an arbitrary optimization problem with a constant number $d \in \mathbb{N}$ of objectives to be minimized. Each (feasible) solution s is mapped to a cost vector $f(s) = (f(s)_1, \ldots, f(s)_d) \in \mathbb{R}^d$ where f is our objective function. A solution s_1 is said to *dominate* another solution s_2 if $f(s_1)_i \leq f(s_2)_i$ for all $i \in [d]$ and there exists some $j \in [d]$ such that $f(s_1)_j < f(s_2)_j$. We write $s_1 \prec s_2$ to denote that s_1 dominates s_2. The set of all non-dominated solutions is called the *Pareto set*. These definitions extend naturally to optimization problems where a subset of objectives is to be maximized instead of minimized. In the following we assume that all objectives are to be minimized, but all arguments and results can be adapted analogously when some (or all) objectives are to be maximized.

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two sets of Pareto-optimal solutions. We define their combined Pareto set as $\mathcal{P}_1 + \mathcal{P}_2 := \{p \in \mathcal{P}_1 \cup \mathcal{P}_2 \mid p \text{ is Pareto-optimal in } \mathcal{P}_1 \cup \mathcal{P}_2\}$, their union as $S^{\mathcal{P}_1,\mathcal{P}_2} := \{p_1 \cup p_2 \mid p_1 \in \mathcal{P}_1, p_2 \in \mathcal{P}_2\}$, which contains all pairwise unions of solutions from $\mathcal{P}_1$ and $\mathcal{P}_2$, and the Pareto set of $S^{\mathcal{P}_1,\mathcal{P}_2}$ as $\mathcal{P}_1 \oplus \mathcal{P}_2 := \{s \in S^{\mathcal{P}_1,\mathcal{P}_2} \mid s \text{ is Pareto-optimal in } S^{\mathcal{P}_1,\mathcal{P}_2}\}$.

2.1 Treewidth

Our algorithms rely on dynamic programming over a *tree decomposition*. The concept of tree decomposition and *treewidth*, introduced by Robertson and Seymour [26], provide a measure of how similar a graph is to a tree. Many $\mathcal{NP}$-hard problems become tractable on graphs with small treewidth. Computing the optimal treewidth is $\mathcal{NP}$-hard [3], but it is FPT with respect to the treewidth [7], and a tree decomposition of width at most $2 \cdot w(G)$ can be computed in linear time using approximation algorithms [21]. Hence, we assume that our algorithm starts from such an approximate decomposition if no optimal one is available. In the following, we briefly introduce the notation used for tree decompositions.

▶ **Definition 1** (Tree decomposition). *A tree decomposition of a graph $G = (V, E)$ is a pair $\mathcal{T} = (\mathbb{T} = (\{t_1, \ldots, t_m\}, E), \{X_t\}_{t \in V(\mathbb{T})})$ where $\mathbb{T}$ is a tree and each node $t \in V(\mathbb{T})$ is associated with a bag $X_t \subseteq V$, such that the following conditions hold:*

- *$\bigcup_{t \in V(\mathbb{T})} X_t = V(G)$, i.e., every vertex of G appears in at least one bag,*
- *$\forall \{u, v\} \in E(G)\ \exists t \in V(\mathbb{T}) : u \in X_t \wedge v \in X_t$, i.e., for every edge $\{u, v\} \in E(G)$ the vertices u and v must appear together in at least one bag,*
- *$T_u = \{t \in V(\mathbb{T}) \mid u \in X_t\}$ is a connected subtree of $\mathbb{T}$ for every $u \in V(G)$.*

The *width* of a tree decomposition is $w(\mathcal{T}) := \max_{t \in V(\mathbb{T})} |X_t| - 1$, and the *treewidth* $w(G)$ is the smallest possible *width* of a tree decomposition. We denote with V_t the union of vertices that were introduced at t or some child of t. In the following, we write w for the width of the tree decomposition used by our algorithm, which may be larger than $w(G)$ if an approximation is used.

We assume we have a *nice tree decomposition*, which is a binary tree decomposition rooted at some node $r \in V(\mathbb{T})$ with $X_r = \emptyset$ and with four specific node types:

- **Leaf node**: Let t be a node with no child nodes. Then t is a *Leaf node* iff $X_t = \emptyset$.
- **Introduce node**: Let $t_{\text{parent}}, t_{\text{child}}$ be two nodes in $V(\mathbb{T})$, where t_{parent} has exactly one child t_{child}. Then t_{parent} is an *Introduce node* iff $X_{t_{\text{parent}}} = X_{t_{\text{child}}} \cup \{v\}$ for some $v \in V(G)$.
- **Forget node**: Let $t_{\text{parent}}, t_{\text{child}}$ be two nodes in $V(\mathbb{T})$, where t_{parent} has exactly one child t_{child}. Then t_{parent} is a *Forget node* iff $X_{t_{\text{parent}}} \cup \{v\} = X_{t_{\text{child}}}$ for some $v \in V(G)$.
- **Join node**: Let $t_{\text{parent}}, t^1_{\text{child}}, t^2_{\text{child}}$ be three nodes in $V(\mathbb{T})$, where t_{parent} has exactly two children t^1_{child} and t^2_{child}. Then t_{parent} is a *Join node* iff $X_{t_{\text{parent}}} = X_{t^1_{\text{child}}} = X_{t^2_{\text{child}}}$.

From any tree decomposition of width w, a nice tree decomposition of the same width with $\mathcal{O}(|V(G)| \cdot w)$ nodes can be computed in polynomial time [13].

The algorithm works as follows: Given an instance $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^d$, it first computes a nice tree decomposition $(\mathbb{T}, \{X_t\}_{t \in V(\mathbb{T})})$. In each leaf node the Pareto sets are initialized as empty sets. Then, for each node t, depending on its type, the algorithm recursively computes the set of Pareto-optimal solutions for each subproblem at this node. We distinguish between the functions `computeLeafNode`, `computeIntroduceNode`, and `computeForgetNode`, corresponding to their respective node types, and show correctness assuming that the Pareto sets for all child nodes are correctly computed.

3 Multiobjective s-t cut problem

We consider the problem of computing all Pareto-optimal s-t cuts in a graph. Given a graph $G' = (V \cup \{s, t\}, E')$ with $n = |V|$ and a cost function $c' : E' \to \mathbb{R}^d$, a cut corresponds to a subset $S \subseteq V$. The cost of a cut S is defined as the sum of costs of all edges crossing from $S \cup \{s\}$ to $(V \setminus S) \cup \{t\}$. Let $\delta(S) := \{\{u, v\} \in E' \mid u \in S \cup \{s\}, v \in (V \setminus S) \cup \{t\}\}$ be the set of edges cut by S. The cost of S is given by $\sum_{e \in \delta(S)} c'(e)$. We now wish to compute all s-t cuts that are Pareto-optimal. Without loss of generality, we assume $\{s, t\} \notin E'$, since such an edge only introduces a constant shift in cost to every Pareto-optimal solution.

We denote $\delta(A, B) := \{\{u, v\} \in E' \mid u \in A, v \in B\}$ for $A, B \subseteq V(G')$ as the set of edges between vertices in A and B that are being cut. We define $c : 2^{V(G')} \times 2^{V(G')} \to \mathbb{R}^d$ with $c(A, B) = \sum_{e \in \delta(A,B)} c'(e)$ for any subsets $A, B \subseteq V(G')$. For simplicity, for a single vertex $p \in V(G')$ and subset $A \subseteq V(G')$, we write $c(p, A)$ instead of $c(\{p\}, A)$.

Let $G = G'[V]$ be the subgraph induced by V. Given a nice tree decomposition $(\mathbb{T}, \{X_t\}_{t \in V(\mathbb{T})})$ for G, we compute for each node $t \in V(\mathbb{T})$ and subset $S \subseteq X_t$ the Pareto set $\mathcal{P}^S_t$ consisting of all Pareto-optimal solutions $p \subseteq V_t$ with $p \cap X_t = S$. We refer to such a subset S as a *selection*. Every such solution p has cost $c'(\delta(p)) = c(p, V \setminus p) + c(p, t) + c(V \setminus p, s)$. We refer to such an instance by a pair (t, S).

▶ **Theorem 2.** *For an arbitrary s-t cut instance $(G' = (V \cup \{s, t\}, E'), c')$ with cost function $c' : E \to \mathbb{R}^d$, we can compute the set of Pareto-optimal s-t cuts in time $\mathcal{O}(nw \cdot 2^w \cdot p^2_{\max} \log^{\max\{d-2,1\}}(p^2_{\max}))$ if a nice tree decomposition of graph $G = G'[V]$ with width w is provided.*

In the following we describe how to compute the Pareto set for each node type. For each instance (t, S), the set of Pareto-optimal solutions is stored in an entry $D[t, S]$. For simplicity, we assume that $D[t, S]$ contains the actual corresponding Pareto set. Since the algorithm only depends on the cost vectors and selection S, in the actual implementation we will only save their cost vectors and reconstruct each solution by additionally storing pointers to the solutions from which it originated. Since we assume d as a constant, the encoding length of every Pareto-optimal solution is constant as well. Proofs of the following lemmas are provided in the full version of the paper.

`computeLeafNode`(t, S): For a leaf node t, we have only one solution $D[t, \emptyset] = \{\emptyset\}$.

`computeIntroduceNode`(t, S): Assume we have $X_t = X_{t'} \cup \{v\}$. If $v \notin S$, then $D[t, S] = D[t', S]$. If $v \in S$, then v is only adjacent to vertices in $X_{t'}$ and $V \setminus V_t$. If $X_{t'}$ is fixed, placing v on the same side of the cut as s only changes the cost of every solution by a fixed amount. Therefore we have $D[t, S] = D[t', S \setminus \{v\}]$, and their costs change by $c(v, X_{t'} \cup (V \setminus V_t) \cup \{t\}) - 2c(S \setminus \{v\}, v) - c(s, v)$.

▶ **Lemma 3.** *If node t introduces some vertex v and has a child t' for which $D[t', S]$ has been computed, then* `computeIntroduceNode`(t, S) *computes $D[t, S]$ in time $\mathcal{O}(p_{\max})$ for any partition S.*

`computeForgetNode`(t, S): Assume we have $X_t = X_{t'} \setminus \{v\}$. To obtain $D[t, S]$, we compute the union of the sets $D[t', S]$ and $D[t', S \cup \{v\}]$ and then remove all dominated solutions.

▶ **Lemma 4.** *If node t forgets some vertex v and has a child t' for which all possible sets $D[t', S']$ have been computed, then* `computeForgetNode`(t, S) *computes $D[t, S]$ in time $\mathcal{O}(p_{\max} \log^{d-2}(p_{\max}))$.*

For the correctness of `computeJoinNode`(t, S), the following lemma will be useful:

▶ **Lemma 5.** *For each Pareto-optimal solution $p \subseteq V$ and any node $t \in V(\mathbb{T})$, the solution $p \cap V_t$ is Pareto-optimal in the instance $(t, X_t \cap p)$.*

`computeJoinNode`$(t, \mathcal{S})$: Assume we have $X_t = X_{t_1} = X_{t_2}$. Since X_t separates $V_{t_1} \setminus X_t, V_{t_2} \setminus X_t$ and $V \setminus V_t$, we can combine every solution from $D[t_1, S]$ with every solution from $D[t_2, S]$ and then remove all dominated ones.

▶ **Lemma 6.** *If node $t \in V(\mathbb{T})$ is a join node with children $t_1, t_2 \in V(\mathbb{T})$ and sets $D[t_1, S]$ and $D[t_2, S]$ have been correctly computed, then* `computeJoinNode`(t, S) *computes $D[t, S]$ in time $\mathcal{O}(p_{\max}^2 \log^{\max\{d-2,1\}}(p_{\max}^2))$.*

Proof of Theorem 2. The correctness follows directly from the previous lemmas for the different node types, and from $D[r, \emptyset]$ being the Pareto set for the entire instance.

The runtime is dominated by the join nodes. Our nice tree decomposition contains $\mathcal{O}(nw)$ nodes, and for each node t the number of subsets $S \subseteq X_t$ is bounded by 2^{w+1}. Computing $D[t, S]$ at a join node takes time $\mathcal{O}(p_{\max}^2 \log^{\max\{d-2,1\}}(p_{\max}^2))$. Hence, the total running time is bounded by $\mathcal{O}(nw \cdot 2^w \cdot p_{\max}^2 \log^{\max\{d-2,1\}}(p_{\max}^2))$. ◀

The technique of computing the Pareto set for the s-t cut problem can also be applied to other multiobjective optimization problems as well. To illustrate this, we show how to compute the Pareto set for the multiobjective MST problem and the multiobjective TSP problem. Proofs of the following theorems are provided in the full version of the paper.

▶ **Theorem 7.** *For an arbitrary multiobjective spanning tree instance $(G = (V, E), c)$ with $d \geq 2$ objectives and cost function $c : E \to \mathbb{R}^d$, we can compute the set of Pareto-optimal spanning trees in time $\mathcal{O}(n(2w)^{\mathcal{O}(w)} \cdot p_{\max}^2 \log^{\max\{d-2,1\}}((2w)^{\mathcal{O}(w)} \cdot p_{\max}^2))$, if a nice tree decomposition of graph G with width w is provided.*

▶ **Theorem 8.** *For an arbitrary multiobjective traveling salesman instance $(G = (V, E), c)$ with $d \geq 2$ objectives and cost function $c : E \to \mathbb{R}^d$, we can compute the set of all Pareto-optimal tours in time $\mathcal{O}(n(3w + 3)^{\mathcal{O}(w)} \cdot p_{\max}^2 \cdot \log^{\max\{d-2,1\}}((3w + 3)^{\mathcal{O}(w)} \cdot p_{\max}^2))$, if a nice tree decomposition of graph G with width w is provided.*

4 Experiments

In this section we present the experimental evaluation of our proposed algorithm for a special case of the multiobjective s-t cut problem: the bicriteria polygon aggregation problem, introduced by Rottmann et al. [28]. An instance (T, P) consists of a set $P = \{p_1, \ldots, p_m\}$ of polygons and a set $T = \{t_1, \ldots, t_n\}$ of triangles. For any polygon $s \in T \cup P$, let $A(s) > 0$ denote its area and $P(s) > 0$ its perimeter. Given a set of triangles $S \subseteq T$, we define $A(S)$ as the total area of $S \cup P$ and $P(S)$ as the total perimeter of $S \cup P$. The goal is to compute the Pareto set $\mathcal{P}$ of subsets $S \subseteq T$ minimizing both $A(S)$ and $P(S)$.

As with many multiobjective optimization problems, the set of Pareto-optimal solutions for the bicriteria polygon aggregation problem can be of exponential size. In fact, one can more generally show that the bicriteria polygon aggregation problem can be seen as a generalization of the bicriteria knapsack problem. The statement on the size of the Pareto set follows as a corollary. The proof of the following theorem is provided in the full version of the paper.

▶ **Theorem 9.** *For every instance I of the bicriteria knapsack problem, there exists an instance I' of the bicriteria triangle aggregation problem such that there is a one-to-one correspondence between the Pareto-optimal solutions in I and I'.*

Our implementation follows the methodology outlined in Section 3, is written in Java, and was evaluated on the datasets used by Rottmann et al.[28], which consist of triangle aggregation problems derived from various cities and villages in Germany. The source code is available at `https://github.com/Tarek-pub/Bicriteria_Aggregation`.

We use the s-t cut construction from Rottmann et al. [28] to generating each instance. Additionally, we remove vertices corresponding to polygons from the graph G, resulting in a more compact, yet equivalent graph structure. To reduce complexity, weights (i.e. area and perimeter) are rounded to one decimal place (perimeter in decimals), and only unique-weight solutions are retained. Tree decompositions are computed using the JDrasil framework [4]. Several observations during development led to significant improvements in runtime and memory usage, which we integrated iteratively into the implementation and quantify through an analysis on the Ahrem dataset (Table 1). Experiments were run using 16 threads of an Intel Core i9-13900 with 100GB RAM. Multiple tree decompositions were generated for exactly 1 hour using 16 threads.

Further implementation details are provided in the full version of the paper, including an extended description of optimizations and a complete list of datasets with corresponding runtimes and memory usage.

As noted in Section 3, it suffices to store only the overall cost of a solution, rather than the full set of triangles included. The actual solutions can be reconstructed later by maintaining, additionally to the cost, pointers to the solutions from which the current one originated.

Table 1 Performance comparison of different algorithmic improvements on the dataset Ahrem.
*: These values are extrapolated based on sampled subproblems and scaled proportionally to estimate full runtimes (more details are provided in the full version of the paper).
**: For the versions before choosing a good root or tree decomposition, we chose the median rated root / tree decomposition in this table for a fair comparison.

Algorithm improvement	**Runtime [h]**	**Storage usage [GiB]**
Outsourcing & Pruning**	13767.8*	3707
Choosing a good root**	8571.6* (−38%)	3991 (+8%)
Choosing from multiple tree decompositions	7129.6* (−17%)	3647 (−9%)
Join node heuristic	162.0 (−98%)	3647 (−0%)
Join-forget nodes	8.2 (−95%)	1499 (−59%)
Introduce-join-forget nodes	7.0 (−15%)	122 (−92%)
Summary	**−99.95%**	**−96.71%**

We briefly describe the data structure used in our implementation, as well as a rough description of the algorithm adaptations. Each solution p is represented as a weighted pair $(A(p), P(p))$ for area and perimeter.

4.1 Outsourcing and Pruning

Initially, the implementation was unable to solve even small datasets, such as Osterloh (treewidth 14), due to RAM exhaustion reaching up to 100GB. To solve this problem, we implemented an efficient outsourcing strategy that stores all currently unneeded solutions onto a hard drive, keeping only the necessary solutions in RAM. Each solution is stored as a 16-byte entry in a dedicated *origin-pointer* file. Each DP table $D[t, S]$ is stored in a separate *surface-pointer* file, containing solution IDs and their weights. These files are kept on disk and only loaded into RAM when needed. This data structure allowed solving Osterloh without exceeding RAM limits, but other datasets remained unsolvable due to hard drive usage growing up to 2TB. To free up disk space, we remove obsolete surface-pointer files and, when necessary, run a slow in-place pruning step to clean the origin-pointer file. This involves recursively marking only reachable entries by following their pointers and compacting the file accordingly. Pruning is only triggered when space is critically low, based on conservative estimates of future storage use. These strategies prevented further storage issues, but many datasets were still unsolvable due to excessive runtime, particularly in join nodes. For example, the runtime required to solve the Ahrem dataset at this stage was estimated at approximately 13,768 hours, or one and a half year worth of CPU.

4.2 Choosing an appropriate tree decomposition

When generating multiple tree decompositions for the same graph, we observed that both the decomposition structure and the choice of the root node can significantly affect performance. Similar to Abseher et al. [1], we therefore generate multiple tree decompositions for the same graph and select one that is expected to yield good performance. However, our approach differs in two important aspects: First, since we consider a concrete algorithm rather than a general class of dynamic programming approaches, we can simulate its execution to approximate actual performance. Second, we do not only estimate runtime but also account for memory consumption, selecting the decomposition that minimizes a weighted combination of both. To this end, each decomposition is traversed in postorder, and we estimate the number of solutions at each node to predict resource usage.

Earlier versions of our approach focused solely on minimizing runtime, which did not always lead to reduced memory usage (see Table 1). Only in the final version did we optimize both runtime and memory simultaneously. Incorporating this performance estimator into the algorithm led to a significant improvement in runtime. For the Ahrem dataset we estimated a runtime decrease to around 7,130 hours, a 48% decrease in the estimated runtime.

In an experiment with 100 decompositions for the instance Osterloh, our estimator selected the decomposition that ranked first in both runtime and storage. The correlation between the predicted score and actual performance was 0.91 for runtime and 0.97 for storage. We obtained similarly strong results when estimating performance for different root node choices, again observing a high correlation between predicted and actual runtime and memory usage. These results suggest that our estimator reliably selects either the best tree decomposition or at least one that still performs well with regards to runtime and memory consumption.

4.3 Join node algorithm

Let t be a join node with children t_1 and t_2. For each subset $S \subseteq X_t$, we compute $\mathcal{P}_t^S = \mathcal{P}_{t_1} \oplus \mathcal{P}_{t_2}$ by combining all solutions $p_1 \in \mathcal{P}_{t_1}^S$ with all solutions $p_2 \in \mathcal{P}_{t_2}^S$ and subsequently remove all dominated solutions. To do this efficiently, we employ a lexicographical ordering of the input lists and process the combinations using a min-heap. Assume $|\mathcal{P}_{t_1}^S| \leq |\mathcal{P}_{t_2}^S|$. We initialize a min-heap with one heap node for each $p_1 \in \mathcal{P}_{t_1}^S$, each pointing to the first entry in $\mathcal{P}_{t_2}^S$. We then repeatedly extract the root of the min-heap, process the corresponding solution pair (p_1, p_2), and increment its pointer to reference the next solution $p_2' \in \mathcal{P}_{t_2}^S$ that follows p_2 in lexicographical order. If such a solution p_2' exists, we update the heap accordingly, otherwise, we remove the heap node from the data structure. This process continues until the heap is empty, ensuring that every valid solution pair has been considered. This min-heap mechanism ensures that all candidate pairs are enumerated in lexicographical order, which enables an efficient check for Pareto-optimality.

4.4 Optimizing join node computations

As with many dynamic programming algorithms over a nice tree decomposition, join nodes are the main computational bottleneck. At each join node t we needed to compute $\mathcal{P} = \mathcal{P}_1 \oplus \mathcal{P}_2$ for two Pareto sets $\mathcal{P}_1, \mathcal{P}_2$ exactly $2^{|X_t|}$ many times. Computing $\mathcal{P}_1 \oplus \mathcal{P}_2$ for $d \geq 2$ can be done in $\mathcal{O}(p_{\max}^2 \log^{\max\{d-2,1\}}(p_{\max}^2))$. In the worst case, $\mathcal{P}_1 \oplus \mathcal{P}_2$ can contain up to $|\mathcal{P}_1| \cdot |\mathcal{P}_2|$ many solutions if every combination is Pareto-optimal. By considering every combination $(p_1, p_2) \in \mathcal{P}_1 \times \mathcal{P}_2$ and filtering out all dominated ones, this becomes very time-consuming. However, in practice we observed only a roughly linear growth in size relative to the larger input Pareto set. This discrepancy indicated that many combinations were computed that would ultimately be discarded. To circumvent this problem, we utilize a heuristic pruning strategy to efficiently exclude many non-Pareto-optimal combinations at the same time.

We construct a heuristic set $\mathcal{H}$ for $\mathcal{P}$ using a small subset of $\mathcal{P}_1$ and $\mathcal{P}_2$. Afterwards we partition $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{H}$ into multiple sections and compute lower bounds for each section in $\mathcal{P}_1, \mathcal{P}_2$ and upper bounds for $\mathcal{H}$. We then iterate over all combinations of lower bounds and compare them with the upper bounds. If a combination of lower bounds is entirely dominated by all upper bounds, we can conclude that no solution in these sections will be Pareto-optimal (PO) in $\mathcal{P}$ and safely skip them in the min-heap. To construct $\mathcal{H}$, we apply the same optimization recursively until a base case is reached. Furthermore, we apply multi-threading for each join node t, enabling parallel computation of multiple entries $D[t, S]$ at the same time.

Table 2 Overview over some of the datasets we managed to solve. For all datasets see the full version of the paper.

Dataset	w	Time [h]	Storage [GIB]	#PO Solutions	Percentage nonextreme	Graph #Vertices	Graph #Edges	$\lvert V(\mathbb{T})\rvert$
Osterloh	14	0.15	5.8	83055	99.43	1717	1887	7586
Ahrem	17	3.1	122	219969	98.99	5280	5607	21778
Lottbek	19	10.5	355	316567	99.48	5595	6101	24426
Erlenbach	22	70.7	1754	303565	99.47	5644	6284	25682

The heuristic pruning process introduces a trade-off between its own runtime and the efficiency gains from skipping combinations. By selecting appropriate parameters, we achieved a favorable balance, reducing the runtime by nearly 98% and ultimately solving Ahrem in under seven days.

4.5 Join-forget nodes

To further improve runtime and reduce memory usage, we analyzed recurring patterns in the algorithm process that allowed for optimization. In many instances, after computing a join node, many of these solutions were immediately discarded in subsequent forget nodes. To address this inefficiency, we introduce a new node type, called *join-forget* node, which merges a join node with subsequent forget nodes into a single operation. Formally, for a join node t in the tree decomposition, if its unique parent and all ancestors up to the next non-forget node are forget nodes, we replace t and all these forget nodes by a single join-forget node t'. This new node retains the original bag X_t and additionally stores the set F of forgotten vertices from the removed forget nodes.

We consider which lists would be merged in the upcoming forget nodes of the original join node while combining solutions in the min-heap. More specifically, for a join-forget node with forgotten vertices F, we only create a min-heap for each subset $S' \subseteq X_t \setminus F$. Each such heap allows us to efficiently iterate over all combinations from the union $\bigcup_{S\in\{S'\cup F' \mid F'\subseteq F\}}\{p_1 \cup p_2 \mid p_1 \in \mathcal{P}^S_{t_1}, p_2 \in \mathcal{P}^S_{t_2}\}$.

This setup also enhances our join node heuristic. Instead of computing a heuristic set $\mathcal{H}$ and upper bounds for each pair $\mathcal{P}^S_{t_1}, \mathcal{P}^S_{t_2}$ separately, we reuse the join-forget structure. Specifically, for each subset $S' \subseteq X_t \setminus F$, we compute the same heuristic set $\mathcal{H}$ for all $S \in \{S' \cup F' \mid F' \subseteq F\}$. We do so by using subsets of $\mathcal{P}^S_{t_1}$ and $\mathcal{P}^S_{t_2}$ for all such S. As before, we recursively apply this heuristic until a small base case is reached. A side effect of this strategy is that it significantly increases RAM usage, as each thread now requires not just three lists in memory (namely $\mathcal{P}^S_{t_1}, \mathcal{P}^S_{t_2}$ and $\mathcal{P}^S_{t_1} \oplus \mathcal{P}^S_{t_2}$), but up to $2^{|F|+1} + 1$ lists.

This strategy led to another significant boost in runtime for many of our datasets. For the Ahrem dataset, using join-forget nodes reduced the runtime by 95%, allowing us to solve the instance in almost 8 hours. Moreover, storage usage also decreased by 59%. We emphasize that the improvements are not solely due to the use of join-forget nodes, but by adapting the tree decomposition performance estimator accordingly as well.

4.6 Introduce-join-forget nodes

Together with the newly defined join-forget node, we searched for new patterns in the algorithm computations to reduce the storage consumption further. To this end, we introduce a second node type called *introduce-join-forget* node, which further generalizes the concept

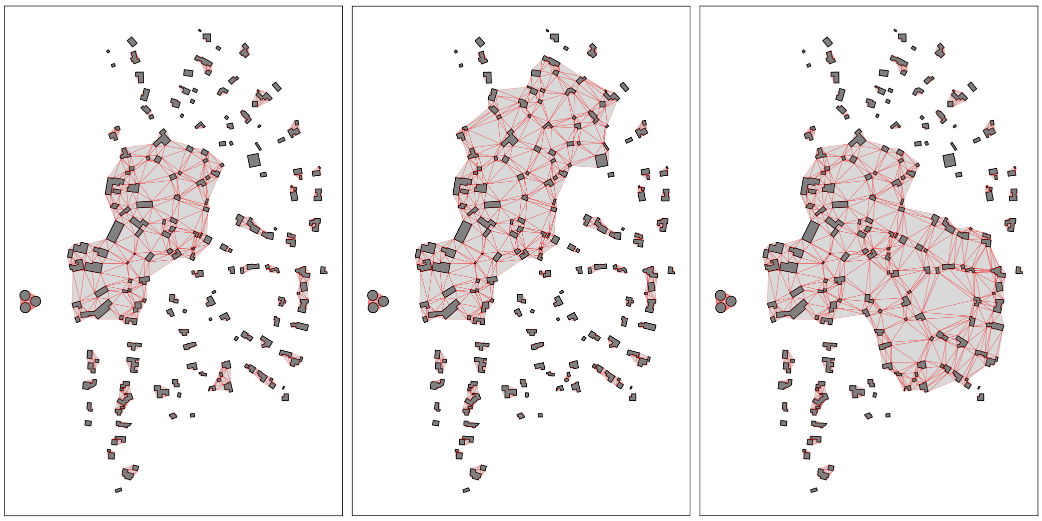

Figure 1 Three PO aggregations of the dataset Osterloh. Input polygons are filled dark gray, chosen triangles are filled light gray. Middle: nonextreme solution. Left/Right: Closest extreme solutions to the nonextreme solution.

of join-forget nodes. After replacing applicable structures with join-forget nodes, we consider each join-forget node t with child nodes t_1, t_2, where at least one child (t_1, t_2, or both) is an introduce node. We remove all consecutive introduce nodes among the children and store their introduced vertices as additional information in t.

A major inefficiency of standard introduce nodes arises from the exponential growth in possible subsets $S \subseteq X_t$. When a new vertex v is introduced, each existing solution $p \in \mathcal{P}_t^S$ for $S \subseteq X_t \setminus \{v\}$ is, with a constant weight adjustment, duplicated for both S and $S \cup \{v\}$. This significantly increases both memory usage and I/O overhead in the outsourcing step, making it a significant bottleneck. To resolve this, we avoid explicit duplication of solutions in introduce-join-forget nodes. Let $I \subseteq X_t$ be the set of vertices whose introduction was skipped on one side. If a set $\mathcal{P}^S$ is required for the min-heap computations and $S \cap I \neq \emptyset$, we instead load the solutions of the surface-pointer file of $\mathcal{P}^{S \setminus I}$ and add the constant weight adjustments according to $S \cap I$. Additionally, we create an origin-pointer entry for such a solution only if they are identified as Pareto-optimal in the min-heap. This strategy also reduces the growth of the origin-pointer file.

Together with adapting the performance estimator, this final improvement reduced the memory consumption by an additional 92%, resulting in a total of only 122GB of storage required for the Ahrem dataset, while also achieving a further 15% decrease in runtime, bringing the total runtime down to 7 hours.

5 Results

Using an efficient data structure and multiple algorithmic improvements described in Section 4, we were able to compute the full Pareto set for many datasets, including instances with treewidths of up to 22, within a reasonable amount of time. Table 2 shows a subset of datasets we successfully solved. All of these computations were performed on a high performance computing system. We used 96 threads of two Intel Xeon “Sapphire Rapids” 2.10GHz and about two terabytes of storage.

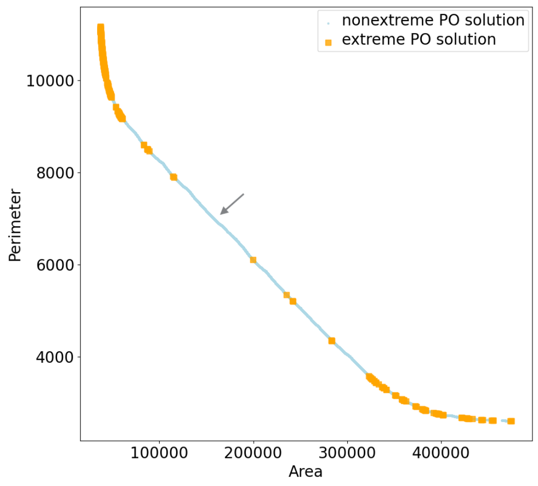

Figure 2 The weights of all PO solutions of Osterloh. The arrow indicates which nonextreme solution is shown in Figure 1.

For the polygon aggregation problem, it is known that all extreme solutions are hierarchically compatible [28]. In Figure 1 we illustrate this by showing two consecutive extreme solutions p_1, p_2 and one nonextreme solution p' that lies between them (i.e. $A(p_1) < A(p') < A(p_2)$ and $P(p_1) > P(p') > P(p_2)$). While the nonextreme solution p' is also a viable aggregation, it exhibits a significantly different structure compared to the extreme solutions. Focusing solely on extreme solutions may lead to large gaps in the solution space, as seen in Figure 2.

Across the datasets we solved, extreme solutions accounted for only less than one percent of the total PO solutions on average. As a result, computing the set of nonextreme solutions can therefore lead to a significant richer range of solutions for the user to choose from, while the extreme solutions are forced being hierarchical depending on each other.

We published a tool to interactively investigate all PO solutions for the dataset Osterloh. The tool is available at `https://github.com/Tarek-pub/Bicriteria_Aggregation_plotting`.

6 Conclusions

We presented the first algorithms for computing Pareto sets using dynamic programming over tree decompositions and showed that this framework can naturally be applied to various multiobjective optimization problems. The main motivation for our work was the article of Rottmann et al. [28], who raised the question of whether it is possible to compute the Pareto set in output-polynomial time. We also conducted an experimental analysis of the polygon aggregation problem on real-world instances and developed several techniques to improve both runtime and memory usage.

We want to highlight that the theoretical running times of our algorithms are only worst-case bounds. In practice, because not every bag in a tree decomposition has the maximum possible size and the number of join nodes is often relatively small, the actual running time will usually be smaller. Additionally, it is a common phenomenon for multiobjective optimization that Pareto sets are not too large for real-world inputs. Hence, the dependency on the size of the largest Pareto set is not prohibitive in practice. For problems like the multiobjective minimum spanning tree problem, the multiobjective TSP, and many other problems, this is further supported by theoretical results in smoothed analysis, which shows that Pareto sets are expected to be polynomially bounded when instances are randomly perturbed [10].

References

1 Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, IJCAI'15, pages 275–282. AAAI Press, 2015. doi:10.1613/jair.5312.

2 Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 596–605. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743510.

3 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. doi:10.1137/0608024.

4 Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A modular library for computing tree decompositions. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 28:1–28:21, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2017.28.

5 René Beier, Heiko Röglin, Clemens Rösner, and Berthold Vöcking. The smoothed number of pareto-optimal solutions in bicriteria integer optimization. *Math. Program.*, 200(1):319–355, September 2023. doi:10.1007/s10107-022-01885-6.

6 Nadja Betzler, Rolf Niedermeier, and Johannes Uhlmann. Tree decompositions of graphs: Saving memory in dynamic programming. *Discret. Optim.*, 3(3):220–229, 2006. Graphs and Combinatorial Optimization. doi:10.1016/j.disopt.2006.05.008.

7 Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, STOC '93, pages 226–234, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167161.

8 Hans L. Bodlaender and Fedor V. Fomin. Tree decompositions with small cost. *Discret. Appl. Math.*, 145(2):143–154, 2005. Structural Decompositions, Width Parameters, and Graph Labelings. doi:10.1016/j.dam.2004.01.008.

9 Fritz Bökler. *Output-sensitive complexity of multiobjective combinatorial optimization with an application to the multiobjective shortest path problem.* PhD thesis, Dortmund University, Germany, 2018. doi:10.17877/DE290R-19130.

10 Tobias Brunsch and Heiko Röglin. Improved smoothed analysis of multiobjective optimization. *J. ACM*, 62(1):4:1–4:58, 2015. doi:10.1145/2699445.

11 Günther Charwat and Stefan Woltran. Efficient problem solving on tree decompositions using binary decision diagrams. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, pages 213–227, Cham, 2015. Springer. doi:10.1007/978-3-319-23264-5_19.

12 H. William Corley and I. Douglas Moon. Shortest paths in networks with vector weights. *Journal of Optimization Theory and Application*, 46(1):79–86, 1985. doi:10.1007/BF00938761.

13 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 1st edition, 2015. doi:10.1007/978-3-319-21275-3.

14 Tamal K. Dey. Improved bounds on planar k-sets and k-levels. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 156–161. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646104.

15 Matthias Ehrgott. Integer solutions of multicriteria network flow problems. *Investigacao Operacional*, 19:229–243, 1999. doi:10.1007/978-3-642-59179-2_2.

16 Matthias Ehrgott. *Multicriteria Optimization (2. ed.)*. Springer, 2005. doi:10.1007/3-540-27659-9.

17 Pierre Hansen. Bicriterion path problems. In *Multiple Criteria Decision Making: Theory and Applications*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127, 1980. doi:10.1007/978-3-642-48782-8_9.

18 Kustaa Kangas, Mikko Koivisto, and Sami Salonen. A faster tree-decomposition based algorithm for counting linear extensions. *Algorithmica*, 82(8):2156–2173, 2020. doi:10.1007/s00453-019-00633-1.

19 Joshua Marc Könen, Heiko Röglin, and Tarek Stuck. Bicriteria Aggregation. Software, swhId: swh:1:dir:b6eacf189792239e9115cf288be27c6753b8df17 (visited on 2025-09-09). URL: https://github.com/Tarek-pub/Bicriteria_Aggregation, doi:10.4230/artifacts.24714.

20 Joshua Marc Könen, Heiko Röglin, and Tarek Stuck. Bicriteria Aggregation Plotting. InteractiveResource (visited on 2025-09-09). URL: https://github.com/Tarek-pub/Bicriteria_Aggregation_plotting, doi:10.4230/artifacts.24713.

21 Tuukka Korhonen. Single-exponential time 2-approximation algorithm for treewidth. *CoRR*, abs/2104.07463, 2021. doi:10.48550/arXiv.2104.07463.

22 Ankur Moitra and Ryan O'Donnell. Pareto optimal solutions for smoothed analysts. *SIAM J. Comput.*, 41(5):1266–1284, 2012. doi:10.1137/110851833.

23 Matthias Müller-Hannemann and Karsten Weihe. On the cardinality of the pareto set in bicriteria shortest path problems. *Ann. Oper. Res.*, 147(1):269–286, 2006. doi:10.1007/s10479-006-0072-1.

24 Adli Mustafa and Mark Goh. Finding integer efficient solutions for bicriteria and tricriteria network flow problems using DINAS. *Comput. Oper. Res.*, 25(2):139–157, 1998. doi:10.1016/S0305-0548(97)00027-0.

25 George L. Nemhauser and Zev Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969. doi:10.1287/mnsc.15.9.494.

26 Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.

27 Heiko Röglin and Shang-Hua Teng. Smoothed analysis of multiobjective optimization. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 681–690. IEEE Computer Society, 2009. doi:10.1109/FOCS.2009.21.

28 Peter Rottmann, Anne Driemel, Herman J. Haverkort, Heiko Röglin, and Jan-Henrik Haunert. Bicriteria aggregation of polygons via graph cuts. In Krzysztof Janowicz and Judith Anne Verstegen, editors, *11th International Conference on Geographic Information Science, GIScience 2021, September 27-30, 2021, Poznań, Poland (Virtual Conference) - Part II*, volume 208 of *LIPIcs*, pages 6:1–6:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.GIScience.2021.II.6.

29 Anders J. V. Skriver and Kim Allan Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Comput. Oper. Res.*, 27(6):507–524, 2000. doi:10.1016/S0305-0548(99)00037-4.