



PEDRO MARISTANY DE LAS CASAS , AND ANTONIO SEDEÑO-NODA  AND RALF
BORNDÖRFER 

An Improved Multiobjective Shortest Path Algorithm

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: <http://www.zib.de>

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

An Improved Multiobjective Shortest Path Algorithm

Pedro Maristany de las Casas¹

Antonio Sedeño-Noda²

Ralf Borndörfer¹

Abstract

We present a new label-setting algorithm for the Multiobjective Shortest Path (MOSP) problem that computes a minimum complete set of efficient paths for a given instance. The size of the priority queue used in the algorithm is bounded by the number of nodes in the input graph and extracted labels are guaranteed to be efficient. These properties allow us to give a tight output-sensitive running time bound for the new algorithm that can almost be expressed in terms of the running time of Dijkstra’s algorithm for the Shortest Path problem. Hence, we suggest to call the algorithm *Multiobjective Dijkstra Algorithm* (MDA). The simplified label management in the MDA allows us to parallelize some subroutines. In our computational experiments, we compare the MDA and the classical label-setting MOSP algorithm by Martins, which we improved using new data structures and pruning techniques. On average, the MDA is 2 to 9 times faster on all used graph types. On some instances the speedup reaches an order of magnitude.

1 Introduction.

The *Shortest Path* (SP) problem is one of the most studied classical *Combinatorial Optimization* problems. Numerous classical algorithms and applications are reviewed in Ahuja et al. [1]. Moreover, the development of new algorithms and speedup techniques to solve SP problems remains an actual research topic as the used graphs and data grow with every new application or required feature. In the SP problem, the quality of paths is measured w.r.t. a single attribute (e.g., cost, length, time, risk, energy consumption, etc.) defined on each arc of the input graph. Given a fixed starting node s , the one-to-all variant of the SP problem seeks to find a shortest path between s and every other node in the graph; in the one-to-one variant, a shortest path between s and a given target node t has to be found.

In many applications of routing problems, a single attribute is not sufficient to define the preference relationship among the paths. This naturally leads to the *Multiobjective Shortest Path* (MOSP) problems, in which several attributes are defined on the arcs, and hence, on the paths. In this scenario, the minimality of paths¹ has to be redefined and leads to solution sets that may be exponentially sized w.r.t. to the problem’s input size. This complicates the running time analysis of MOSP algorithms, as dependence on the input size only allows us to derive bounds that vastly overestimate the running time of many instances. Hence, we study the output sensitive complexity of the presented algorithms. I.e., we seek to bound its running time with a polynomial depending on the size of the input and the output of the problem instances. The running time of the new Multiobjective Dijkstra Algorithm (MDA) turns out to be theoretically and computationally better than the one of existing label-setting MOSP algorithms.

1.1 Literature Review.

The basics in Multiobjective (Combinatorial) Optimization are well explained in Emmrich and Deutz [19], Ehrgott [17], or Ehrgott and Gandibleux [16].

In the 70s, Vincke [39] considered the MOSP for the first time using two objective functions. This Biobjective Shortest Path (BOSP) problem was also considered by Hansen [24], who introduced the first label-setting algorithm for BOSP problems. Serafini [33] showed that the MOSP problem is *NP*-complete. Good surveys on this topic are Ulungu and Teghem [38], Current and Marsh [11], Skriver [34], Tarapata [36], Clímaco and Pascoal [10].

¹Zuse Institute Berlin, Takustraße 7, 14195, Berlin, Germany.

²Departamento de Matemáticas, Estadística e Investigación Operativa, Universidad de La Laguna, 38271 Santa Cruz de Tenerife, España.

¹Minimum paths are called efficient. We give a formal definition in Section 2.

Most methods providing the (minimum or maximum) complete set of efficient paths (cf. [17]) are labeling methods or ranking methods. In this paper, we focus on label-setting methods that follow the ideas from the corresponding algorithms for the SP problem. The first label-setting method for the BOSP problem is due to Hansen [24]. Later, Martins [25] generalized the previous algorithm for MOSP problems. All of the mentioned methods consider min-sum criteria in all the objectives but other objective types have also been considered, e.g., in Gandibleux [22]. Recently, Sedeño-Noda and Colebrook [32] introduced the Biobjective Dijkstra Algorithm (BDA), a new algorithm for the BOSP problem. The BDA’s theoretical running time is the number of efficient solutions times Dijkstra’s [14] running time, hence improving the running time of older label-setting BOSP algorithms. It also outperforms them in the computational experiments. Different approaches to solve MOSP problems could be swarm intelligence graph-based algorithms as presented by Ntakolia and Iakovidis [26] or multi-phase approaches using preference-based optimization as in Di Puglia Pugliese et al. [27].

In general, the cardinality of the sets of efficient paths increases as more objectives are considered simultaneously. Deciding whether a newly found path constitutes an efficient solution given the set of already found efficient paths is hence computationally expensive. This motivates our attempt to parallelize some subroutines in MOSP algorithms. Little work has been published in this direction. In [31] the authors describe a parallel variant of Martins’ algorithm. Their approach targets the heap operations as the main source of parallelism. They focus on the biobjective case and achieve a remarkable asymptotic running time. However, the authors do not report any computational results as they claim that their algorithm *“might be too complicated to be practical”*.

A different approach to overcome the problem of the cardinality of the solution sets of efficient paths is to only output a subset of efficient paths that is *good enough*. This motivates the study of Fully Polynomial Time Approximation Schemes (FPTAS) for MOSP problem in the literature. Tsaggouris and Zaroliagis [37] gave an FPTAS that subdivides the space of possible paths’ costs and stores at most one path per cell in the subdivision. This idea was also used in Breugem et al. [7], where the authors present a new FPTAS based on Martins’ algorithm [25]. Very recently, Maristany et al. [12] introduced a new FPTAS based on the MDA presented in this paper.

1.2 Contribution and Outline

We introduce and analyze the MDA, a new label-setting algorithm that computes a minimum complete set of efficient paths for MOSP problems and generalizes the Biobjective Dijkstra Algorithm (BDA) presented in [32]. The complexity of the BDA benefits from the fact that the efficient paths are stored and ordered in a way that allows dominance checks in constant time. In the multiobjective scenario discussed in this paper, these dominance checks run in linear time w.r.t. the number of elements stored in the considered lists and the number of objectives. This causes different complexity bounds for the BDA and the MDA.

In this paper, we discuss the complexity of the MDA in detail, always considering the number of optimization criteria as part of the input. The result is an output sensitive algorithm for the one-to-all MOSP problem. Contrary to the biobjective case, the derived upper bound exhibits a quadratic dependency on the maximum cardinality of a node’s set of efficient paths. Even though, to the best of our knowledge, the complexity of the MDA is better than the one of existing MOSP algorithms, we focus on the upcoming quadratic term and try to soften its impact parallelizing some parts of the algorithm. The used parallelization techniques are much easier than the ones used in [31] and can also be analyzed in our computational experiments.

In these experiments we compare the new MDA with an improved version of the classical label-setting MOSP algorithm by Martins [13]. The comparison is done using the one-to-one version of the MOSP problem. As proven in [8], this version is not output sensitive, unless $P = NP$. The intuitive reason is that the cardinality of the sets of efficient paths at intermediate nodes can not be bounded a priori by the cardinality of the set of efficient paths at the target node. Still, this version is of theoretical and practical interest: having a target node allows us to use pruning techniques from the literature to reduce the number of computed efficient paths at intermediate nodes. We also endorse our variant of Martins’ algorithm with these techniques, as in the three objective case, they performed better than the early stopping criterion suggested in the original publication [13]. Hence, using the pruning techniques, we are able to experiment on graphs motivated by real world applications such as road or airway networks without incurring huge time and memory consumption. Such experiments for MOSP instances with three objectives are rare in the literature. Our experimental setup differs from the one in [32] in that we do not consider bidirectional versions of the algorithms but instead investigate the impact of parallelization on the MDA running times. The results reveal that the average speedup of the MDA w.r.t. our version of Demeyer’s algorithm [13] is greater than in the biobjective case.

All in all, we study in this paper the impact on the general multiobjective case of a key idea that aims to simplify label setting MOSP algorithms both theoretically and computationally: we only store at most one candidate label per node in the priority queue, and as a result the computationally expensive task of merging

sets of labels can be replaced by a more efficient procedure that selects new candidate labels to include in the queue.

The paper is structured as follows: In Section 2 we define the MOSP problem, formulate it as a general MOCO problem, state assumptions on the cost functions, and discuss their consequences on the structure of efficient paths. In Section 3 we introduce the new MDA and state its correctness and output sensitive complexity. We also explain how the new algorithm can be parallelized. In Section 4 we discuss the classical label-setting algorithm by Martins. The contents of Section 5 focus on the one-to-one version of both algorithms: we introduce pruning techniques that reduce the number of computed labels while preserving the algorithms' exactness at the target node. In Section 6.3 we present the results of our computational experiments. Finally, in Section 7, we summarize the findings of the paper.

2 Multiobjective Shortest Path Problem.

Consider a directed graph $G = (V, A)$ with $n := |V|$ nodes and $m := |A|$ arcs. For a node $v \in V$, we denote its set of outgoing arcs by $\delta^+(v)$ and its set of incoming arcs by $\delta^-(v)$. We assume that G has no parallel arcs and can then incur in a slight abuse of notation by referring to a predecessor node u of v by $u \in \delta^-(v)$ and to a successor node w of v by $w \in \delta^+(v)$. This assumption also allows us to represent a (u, v) -path P in G as a sequence $(u = v_1, \dots, v_k = v)$ of k nodes s.t. $(v_i, v_{i+1}) \in A$ for $i \in \{1, \dots, k-1\}$. Given a cost vector $c_a \in \mathbb{R}^d$, $d \in \mathbb{N}$, for every arc $a \in A$, the cost of the path P is $c(P) := \sum_{i=1}^{k-1} c_{(v_i, v_{i+1})} \in \mathbb{R}^d$. The minimality of paths is defined in terms of the *Pareto order* \prec_D , a strict partial order on \mathbb{R}^d . Given two cost vectors $x, y \in \mathbb{R}^d$, x is said to dominate y and we write $x \prec_D y$ if $x_i \leq y_i$ for all $i \in \{1, \dots, d\}$ and there is at least one $j \in \{1, \dots, d\}$ such that $x_j < y_j$. Then, if consider two (u, v) -paths P and P' , $u, v \in V$, P is said to *dominate* P' if $c(P) \prec_D c(P')$. Moreover, P and P' are called *equivalent* if and only if $c(P) = c(P')$. Finally, in case there is no (u, v) -path in G that dominates P , P is called an *efficient* path and its cost vector $c(P)$ is called a *non-dominated* cost vector.

From the above definitions two interesting solutions sets arise in Multiobjective Optimization and, in particular, in MOSP problems. Clearly, the set of non-dominated cost vectors to a MOSP instance is unique. However, since different paths between the same end-nodes can have equal costs, one can be interested in the *maximum complete set of efficient paths* that contains all efficient paths for the given instance or in a *minimum complete set of efficient paths* that contains exactly one efficient path per non-dominated cost vector. We now state the formal definition of the MOSP problem.

Definition 1 (Multiobjective Shortest Path Problem). Given a directed graph $G = (V, A)$, a root node $s \in V$, and cost vectors $c_a \in \mathbb{R}^d$, $d \in \mathbb{N}$, for every arc $a \in A$, the one-to-all version of the *Multiobjective Shortest Path* (MOSP) problem is to find the maximum or a minimum complete set of efficient (s, v) -paths for every $v \in V$. If additionally a target node $t \in V$ is input, the one-to-one MOSP problem is to find the maximum or a minimum complete set of efficient (s, t) -paths.

In this paper we focus on the MOSP variant that seeks to find a minimum complete set of efficient paths, i.e. solution sets of efficient and non-equivalent paths. Label-setting MOSP algorithms iteratively take an already found path and extend it along the outgoing arcs of the path's end node. For this dynamic programming approach to work, the *principle of optimality* has to hold. This requires some assumptions on the structure of the arc costs. We only consider MOSP instances (G, s, c) without negative cost cycles C , i.e., $c(C) := \sum_{a \in C} c_a \geq 0$ for every cycle C in G . As a consequence, we can search for simple paths only since such a path exists for every non-dominated point in the outcome space. Arcs with negative cost components are allowed. For such instances, non-negative reduced costs \tilde{c} that preserve efficient paths can be found in time polynomial in the graph's size. The new instance (G, s, \tilde{c}) can then be solved with the presented label-setting methods. Hence, we consider w.l.o.g. MOSP instances with non-negative arc cost. In this setting, subpath optimality, also known as *principle of optimality*, holds as proven in [25].

Hardness As proven in [24] and [33], the MOSP problem is intractable and *NP* complete, even for $d = 2$. The one-to-all MOSP problem is known to be output-sensitive, while the one-to-one version is not, unless $P = NP$ (cf. [8]).

3 Multiobjective Dijkstra Algorithm.

We now describe the new *Multiobjective Dijkstra Algorithm* (MDA), which is shown in Algorithm 1. Given a one-to-all MOSP instance, the MDA computes a minimum complete set of efficient paths between s and every other node $v \in V$. We choose to represent paths with an implicit, less memory-consuming representation:

labels. Given an (s, v) -path P in G ending with the arc $(u, v) \in A$, the unique label representing P is a tuple $l_v := (v, c_{l_v} := c(P), l_{pred})$ consisting of the node $v \in V$, the costs $c(P)$ of P which we rewrite as c_{l_v} and a pointer to the label l_{pred} that corresponds to the (s, u) -subpath of P . This one to one correspondence of paths and labels allows us to inherit the notation introduced in Section 2: if a path is efficient, we call the corresponding label a *non-dominated label*.

Definition 2 (Comparison of Labels). Consider a d -dimensional MOSP instance (G, s, c) and let l and l' be two labels corresponding to two (s, v) -paths P and P' , respectively.

1. *Dominance and Non-Equivalence.* Then, l dominates l' and both are *non-equivalent* iff $c_l \prec_D c_{l'}$ and $c_l \neq c_{l'}$. In this case, we write $l \preceq_D l'$. If there is no other label \tilde{l} representing an (s, v) -path such that $\tilde{l} \preceq_D l$, l is called a *non-dominated label*. Additionally, if L_v is a set of labels representing (s, v) -paths, we write $L_v \preceq_D l$ iff there is a label $\tilde{l} \in L_v$ s.t. $\tilde{l} \preceq_D l$. Otherwise, we write $L_v \not\preceq_D l$.
2. *Lexicographic ordering.* l is said to be lexicographically smaller than l' (we write $l \prec_{lex} l'$) iff c_l is lexicographically smaller than $c_{l'}$, i.e., iff $c_{l,k} < c_{l',k}$ for the first index $k \in \{1, \dots, d\}$ s.t. $c_{l,k} \neq c_{l',k}$.

Algorithm 1: Multiobjective Dijkstra Algorithm

Blue lines only for one-to-one version described in Section 5.

```

Input : Graph  $G = (V, A)$ , Arc Costs  $c_a \in \mathbb{R}_{\geq}^d$ , Node  $s \in V$ .
Input one-to-one: Target node  $t \in V$ , Lower bound  $\underline{c}_v$ ,  $v \in V$ , Upper bound label at target  $\bar{l}_t$ .
Output : Set  $L_v$  of non-dominated labels  $\forall v \in V$  OR Non-dominated target labels  $L_t$ .

1 Priority Queue  $H \leftarrow \emptyset$ ;
2 for  $v \in V$  do Efficient labels  $L_v \leftarrow \emptyset$ ;
   /* We assume  $L$  to store pointers to the sets  $L_v$  and hence, contains the updated sets  $L_v$  during the whole algorithm. */
3  $L \leftarrow \bigcup_{v \in V} L_v$ ;
4 for  $a \in A$  do lastProcessedLabel[ $a$ ]  $\leftarrow 0$ ;
5 Label  $l_s \leftarrow (s, (0, \dots, 0), \text{NULL})$ ;
6  $H \leftarrow H.insert(l_s)$ ;
7 while  $H \neq \emptyset$  do
8    $l_v^* \leftarrow H.extract\_lexmin()$ ;
9    $v \leftarrow l_v^*.node$ ;
10   $L_v.push\_back(l_v^*)$ ;
11   $l_v^{new} \leftarrow nextCandidateLabel(v, lastProcessedLabel, \delta^-(v), L, \underline{c}_v, \bar{l}_t)$ ;
12  if  $l_v^{new} \neq \text{NULL}$  then  $H.insert(l_v^{new})$ ;
13  for  $w \in \delta^+(v)$  do  $H \leftarrow propagate(l_v^*, w, H, L, \underline{c}_w, \bar{l}_t)$ ;
14 return  $L_v$  for all  $v \in V$ ; OR return  $L_t$ ;

```

The MDA is a label-setting algorithm managing a set L_v of non-dominated labels at each node $v \in V$. In the algorithm, a \prec_{lex} -sorted priority queue H stores *tentative* labels that correspond to paths that have been explored during the algorithm but are not yet known/decided to be non-dominated. At any point during the algorithm, H stores at most one label per node. Hence, H 's size is bounded by n . Throughout the algorithm it is guaranteed that the extraction of a lexicographically smallest label from H yields a *non-dominated label*.

The algorithm first creates a start label $l_s = (s, (0, \dots, 0), \text{NULL})$, associated with the origin node s , and inserts it into H . The main part of the algorithm is a loop that ends once H becomes empty. An iteration of the loop starts with the extraction of a lexicographically smallest label l_v^* from H , which is added to the end of L_v since it is guaranteed to be non-dominated. This is the only way labels are added to the lists L_v , i.e., made permanent. As a consequence, the sets L_v , $v \in V$, are also sorted according to \prec_{lex} . Each iteration pursues two main tasks. The first is to find the *next tentative/candidate label* for node v that can be added to H and the second is the *propagation* of l_v^* along the outgoing arcs of v . In the pseudocodes we will use $L := \bigcup_{v \in V} L_v$ to simplify notation when passing the sets of permanent labels to the subroutines.

Next Candidate Label

Once a label l_v^* for node v is extracted from H , a new tentative label l_v^{new} for node v must be found (if it exists) and added to H . This is the price for keeping only one tentative label per node in H instead of keeping a set of tentative labels for the same node. This is a crucial difference between Algorithm 1 and the classical label-setting MOSP algorithms. The label l_v^{new} must not be dominated by any existing label in L_v and is computed by extending existing non-dominated labels at predecessor nodes $u \in \delta^-(v)$ along the arc (u, v) . Among the

resulting labels, l_v^{new} is set to be a lexicographically smallest, non-dominated one. More precisely, we define

$$l_v^{new} := \arg \operatorname{lexmin}_{\substack{l \in L_u, \\ u \in \delta^-(v)}} \{l_v := (v, c_l + c_{uv}, l) \mid L_v \not\prec_D l_v\}. \quad (1)$$

If no label l_v^{new} can be found in (1), nothing is added to the priority queue H . Procedure *nextCandidateLabel* shows how l_v^{new} is found algorithmically.

Remark 1 (Exploiting the lexicographic order in *nextCandidateLabel*). Let $N_v \in \mathbb{N}$ be the number of permanent labels at node $v \in V$ at the end of the MDA. Then, the procedure *nextCandidateLabel*(v, \dots) is called $N_v + 1$ times during the algorithm. If we fix a predecessor node $u \in \delta^-(v)$ of v , does the algorithm need to traverse the whole list L_u of permanent labels every time a new candidate label for v is searched? The answer is *no* and has in fact a big impact on the overall complexity of the MDA. For all nodes $w \in V$, the sets L_w are only modified by the insertion of labels at the end of the lists in Line 10 of Algorithm 1. Since these are extracted from H as lexicographically smallest ones in every iteration, the lists L_w inherit the ordering of the priority queue: they are sorted in lexicographically increasing order.

Assume *nextCandidateLabel*(v, \dots) is called for the k^{th} time, $k \in \{1, \dots, N_v\}$. We claim that if during the $(k-1)^{\text{th}}$ search the first $i \in \mathbb{N}$ labels in L_u were considered and only the extension along (u, v) of the label at position $L_u[i]$ was non-dominated at L_v , the current call to *nextCandidateLabel* can start from the i^{th} label in L_u ². The reason is that between the $(k-1)^{\text{th}}$ and k^{th} search for a next candidate label for v , L_v is not modified. Moreover, if L_u is modified, then just by appending labels behind its i^{th} position. Hence, if the labels in L_u prior to position i did not qualify as candidates labels to become the result of (1) in the $(k-1)^{\text{th}}$ search, they will also not qualify as such during the k^{th} search. This uses the fact that both, L_u and L_v , are sorted in lexicographically increasing order. As a result, for every arc $(u, v) \in A$, *nextCandidateLabel* is called $N_v + 1$ times and in total, during all these calls, $\mathcal{O}(N_u + N_v)$ dominance checks against L_v are performed. The second summand accounts for those calls to *nextCandidateLabel*(v, \dots) in which no new predecessor label in L_u is processed, i.e., those calls in which the index `lastProcessedLabel`[(u, v)] is not increased.

Procedure *nextCandidateLabel*

Blue lines only for one-to-one version described in Section 5.

Input : Node v , Indices `lastProcessedLabel`, Node set $Q \subseteq \delta^-(v)$, Permanent labels L .

Input one-to-one: Lower bound of v to $t \in v$, Upper bound label at $t \in \bar{l}_t$.

Output : New lexicographically smallest, non-dominated label for v , if one exists.

```

1 Label  $l_v \leftarrow (v, (\infty, \dots, \infty), \text{NULL})$ ;
2 for  $u \in Q$  do
3   for  $k \in [\text{lastProcessedLabel}[(u, v)], |L_u|]$  do
4     Label  $l_u \leftarrow L_u[k]$ ;
5     Label  $l_{new} \leftarrow (v, c_{l_u} + c_{uv}, l_u)$ ; /* Extension of  $l_u$  along  $(u, v)$ . */
6     lastProcessedLabel[( $u, v$ )]  $\leftarrow k$ ;
7     /* For the artificial expansion of  $l_{new}$  to  $t$ , there is no predecessor label but it is also not needed. Thus, we set
       it to NULL. */
8     if  $\bar{l}_t \not\prec_D (t, c_{l_{new}} + \underline{c}_v, \text{NULL})$  and  $L_t \not\prec_D (t, c_{l_{new}} + \underline{c}_v, \text{NULL})$  then
9       if  $L_v \not\prec_D l_{new}$  then
10        if  $l_{new} \prec_{lex} l_v$  then  $l_v \leftarrow l_{new}$ ;
11        break;
12 if  $c_{l_v} \neq (\infty, \dots, \infty)$  then return NULL;
13 return  $l_v$ ;
```

In Section 3.3 we will discuss a parallel version of *nextCandidateLabel* in which we will need to pass a subset of predecessors of v to this function. This explains the third input argument Q . In this section we will always set $Q = \delta^-(v)$.

Label Propagation

The second main step in any iteration is to propagate the extracted label l_v^* to the successor nodes w in $\delta^+(v)$ of v . Let $l_w = (w, c_{l_v^*} + c_{vw}, l_v^*)$ be such a propagated tentative label. If l_w is dominated by any label in L_w ,

²Note that the $(k-1)^{\text{th}}$ search stops at position i because of Line 11 of *nextCandidateLabel*.

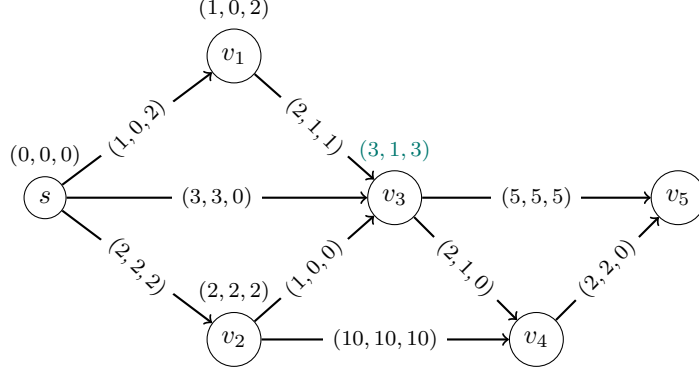


Figure 1: MOSP instances (G, s, c) with $d = 3$. The tuples on the nodes represent the permanent cost labels in L_v .

it is discarded. If that is not the case and if there is no label for w in the priority queue H , l_w is inserted into the priority queue. If there is a label for w in H , we check if l_w is lexicographically smaller. If so, w 's label in H is updated to be l_w . Otherwise, l_w is discarded. To have constant-time access to a node's label in H , a copy of these labels is stored in a vector, indexed by nodes. Whenever we extract or update a node's label in H , we update the corresponding entry in the vector. This allows us to use constant time functions $H.\text{getLabel}(v)$ and $H.\text{contains}(v)$ to get a node's heap label or check if one exists. The pseudocode is shown in *propagate*.

Procedure propagate

Blue lines only for one-to-one version described in Section 5.

Input : Label l_v , Node $w \in \delta^+(v)$, Priority Queue H , Permanent labels L .

Input one-to-one: Lower bound \underline{c}_w , Upper bound label at target \bar{l}_t .

Output : Updated Priority Queue H .

```

1 Label  $l_{new} \leftarrow (w, c_{l_v} + c_{vw}, l_v)$ ;
2 /* For the artificial expansion of  $l_{new}$  to  $t$ , there is no predecessor label but it is also not needed. Thus, we set it to
   NULL. */
3 if  $\bar{l}_t \not\prec_D (t, c_{l_{new}} + \underline{c}_w, \text{NULL})$  and  $L_t \not\prec_D (t, c_{l_{new}} + \underline{c}_w, \text{NULL})$  then
4   if  $L_w \not\prec_D l_{new}$  then
5     if ! $H.\text{contains}(w)$  then
6        $H.\text{insert}(l_{new})$ ;
7     else if  $l_{new} \prec_{lex} H.\text{getLabel}(w)$  then
8        $H.\text{update}(l_{new})$ ;
9 return  $H$ ;

```

The MDA terminates once H becomes empty. It returns the sets L_v of non-dominated labels at every node v .

Example 1. In Figure 1, we sketch an intermediate step in Algorithm 1. The label extracted from the priority queue has cost $(3, 1, 3)$ at node v_3 . The label is directly made permanent at L_{v_3} . The algorithm now searches for a new candidate label for v_3 among the permanent labels of the predecessors of v_3 (Line 11). A label with costs $(3, 3, 0)$ coming from s and a label with costs $(3, 2, 2)$ coming from v_2 fulfill the requirements since they are non-dominated at L_{v_3} . The algorithm picks the lexicographically smallest among those, i.e., $(3, 2, 2)$, as v_3 's next label in the priority queue. Now, the label with cost $(3, 1, 3)$ is propagated to the successor nodes of v_3 (Line 13). A new label for v_5 with cost $(8, 6, 8)$ is added to H (Line 6 of propagate). The new label with cost $(5, 2, 3)$ at node v_4 dominates the label with cost $(12, 12, 12)$ at v_4 that was already in H . Hence, v_4 's label in the priority queue is improved (Line 8 of propagate).

3.1 Correctness.

We sketch the proof of correctness of Algorithm 1. A detailed proof for the biobjective case can be found in [32] and can be extended to hold in the general multiobjective case. As in Dijkstra's [14] algorithm, the key of

the correctness of the MDA is that labels that are extracted from the priority queue can be made permanent. Formally:

Lemma 1 (Permanent labels). *Consider an instance $(G = (V, A), s, c)$ of the MOSP problem with non-negative costs. Let l_v be a label at node $v \in V$ extracted from H in Line 8 of Algorithm 1. Then, l_v is a non-dominated label and can be made permanent in L_v .*

A proof of Lemma 1 can be found in [25], [32]. As discussed in Section 2, having non-negative costs guarantees the efficiency of subpaths. All in all we can prove the following theorem by induction:

Theorem 1 (Correctness of Algorithm 1). *Given a MOSP instance (G, s, c) with non-negative costs, Algorithm 1 computes a minimum complete efficient set L_v for every $v \in V$ that is reachable from s . Labels in L_v correspond to efficient simple (s, v) -paths.*

3.2 Complexity in the One-to-All Case.

The complexity analysis of the MDA differs from the one for the Biobjective Dijkstra Algorithm presented in [32]. In the biobjective case, the lexicographically increasing ordering of the sets L_v (recall that these sets only contain non-dominated and non-equivalent labels) causes the contained labels to be sorted increasingly in the first cost component and decreasingly in the second component (cf. [9], [32]). Since a new candidate label l_v^{new} is lexicographically greater than all labels in L_v , checking if $L_v \preceq_D l_v$ can be done in constant-time by just checking if the second cost component of the last label in L_v is greater than the second cost component of l_v . In the general multiobjective case, this interplay between the lexicographically increasing order and the Pareto order not happen in general: the check $L_v \preceq_D l_v$ is done in $\mathcal{O}(d|L_v|)$ since all labels in L_v have to be checked in the worst case (see Example 2).

Example 2. *For the case $d = 3$, consider the following list containing labels representing different (s, v) -paths for a node $v \in V$:*

$$L = [(v, (1, 5, 1), l_{pred_1}), (v, (2, 3, 8), l_{pred_2}), (v, (3, 2, 6), l_{pred_3})].$$

These labels are non-dominated and are sorted in lexicographically increasing order. The check $L \preceq_D (v, (4, 6, 2), l_{pred_4})$ could be answered with just one comparison if the check starts with the first element of L . However, $L \preceq_D (v, (4, 4, 7), l_{pred_5})$ would require to check the whole set L .

To analyze the running time of the MDA, we set $N := \sum_{v \in V} |L_v|$ and $N_{\max} := \max_{v \in V} |L_v|$. The dimension d of the cost vectors is considered as an input parameter. First, we determine the time needed by each operation in the algorithm.

Dominance and lexicographic checks Checking $l \preceq_D l'$ and $l \prec_{lex} l'$ for two labels l, l' with d -dimensional cost vectors takes $\mathcal{O}(d)$ comparisons. For any node $v \in V$, checking $L_v \preceq_D l$ for a candidate label l representing an (s, v) -path, takes $\mathcal{O}(d|L_v|)$ comparisons. Hence, any check $L_v \preceq_D l$ is done in $\mathcal{O}(dN_{\max})$.

Operations on priority queue and lists of permanent labels We assume that in Algorithm 1, H is a Fibonacci Heap [21]. Recall that the size of the heap is bounded by n . Hence, the *extract_min* operation has a running time of $\mathcal{O}(d \log(n))$. The *insert* and *update* operations run in $\mathcal{O}(d)$. The lists L_v of permanent labels are just modified by the *push_back* function in Line 10 of Algorithm 1. This operation adds a label at the end in constant time.

Initialization The initialization phase entails the creation of the empty heap H and of an empty list L_v for every node v . Additionally, the initial label at node s has to be built and inserted into H . The individual operations are done in constant time but the iteration over all nodes yields a $\mathcal{O}(n)$ complexity for this phase.

Number of iterations Every iteration starts with the extraction of a lexicographically smallest label l_v^* from H . Every extracted label corresponds to an efficient path, so the total number of iterations is N .

Next Candidate Label For any node v , Algorithm 1 searches $|L_v| + 1$ times for a new tentative label using *nextCandidateLabel*(v, \dots) (Line 11 of Algorithm 1). As explained in Remark 1, it is guaranteed that for every arc $(u, v) \in A$, $\mathcal{O}(|L_u| + |L_v|)$ dominance checks are performed (Line 9) during calls to *nextCandidateLabel*(v, \dots). Taking also the lexicographic checks (Line 10) into account, we get a running time of

$$\mathcal{O}((|L_u| + |L_v|)(dN_{\max} + d)) = \mathcal{O}(dN_{\max}^2) \quad (2)$$

Table 1: Running time of Dijkstra algorithms depending on the number of objectives.

	w.r.t N and N_{\max}	w.r.t. N_{\max} using $nN_{\max} \geq N$
$d = 1$	$\mathcal{O}(n \log(n) + m)$	$\mathcal{O}(n \log(n) + m)$
$d = 2$	$\mathcal{O}(N \log(n) + N_{\max}m)$	$\mathcal{O}(N_{\max}(n \log(n) + m))$
$d \geq 3$	$\mathcal{O}(d(N \log(n) + N_{\max}^2m))$	$\mathcal{O}(dN_{\max}(n \log(n) + N_{\max}m))$

per arc $(u, v) \in A$ for all calls to *nextCandidateLabel*(v, \dots). Hence, summing over all arcs, we get a total running time of

$$\mathcal{O}(dmN_{\max}^2) \quad (3)$$

for all calls to *nextCandidateLabel*.

Label Propagation A single call to *propagate* performs dominance and lexicographic checks (Lines 4 and 7) and possibly inserts or updates an element to the heap. Hence, the call runs in $\mathcal{O}(dN_{\max})$. In every iteration of Algorithm 1, the extracted label l_v^* is propagated to all successor nodes of v (Line 9). Thus, per iteration, we get a running time of $\mathcal{O}(|\delta^+(v)|dN_{\max})$. Summing over all iterations, the running time of *propagate* is:

$$\mathcal{O}\left(\sum_{v \in V} |L_v| |\delta^+(v)| dN_{\max}\right) = \mathcal{O}(dN_{\max}^2m). \quad (4)$$

Knowing that the MDA performs N iterations in the main loop, we use (3) and (4) to conclude that its the running time is

$$\mathcal{O}(d(N \log(n) + N_{\max}^2m)) \underbrace{\quad}_{nN_{\max} \geq N} \mathcal{O}(dN_{\max}(n \log(n) + N_{\max}m)).$$

Table 1 gives an overview on the complexity of the one-to-all (MO)SP problems depending on the number of objective functions.

3.3 Parallelization

Some computations in the presented algorithm can easily be parallelized. The most obvious one is the dominance check between a set L_v of permanent labels at a node v and a tentative label l . The operation $L_v \preceq_D l$ can be implemented as shown in *parallelDominates*. It is a recursive approach that splits the array L_v in two halves as long as the resulting splits contain more than $B \in \mathbb{N}$ elements. Each such split is called a *task*. The algorithm returns **TRUE** if at least one task finds an element that dominates l . The worst case regarding the number of comparisons is when $L_v \not\preceq_D l$. If tasks are processed in parallel, one task per thread, each thread will do $\mathcal{O}(B)$ dominance checks. Parallelization using tasks offers some advantages that we will discuss in Section 6.1.3.

Procedure parallelDominates

Input : Label l , List of non-dominated labels L , Index *start*, Index *end*, Bound B .

Output: **TRUE** iff $L \preceq_D l$.

*/** $L_{[start, end]} := \{L[i] \mid i \in [start, end]\}$ is a subarray of L . */

```

1 if  $end - start < B$  then return  $L_{[start, end]} \preceq_D l$ ;
2 Boolean  $dom_1 = \text{FALSE}$ ;  $dom_2 = \text{FALSE}$ ;
3 parallel tasks
4    $dom_1 \leftarrow \text{parallelDominates}(l, L, start, \lfloor \frac{start+end}{2} \rfloor, B)$ ;
5    $dom_2 \leftarrow \text{parallelDominates}(l, L, \lfloor \frac{start+end}{2} \rfloor + 1, end, B)$ ;
6 return  $dom_1$  OR  $dom_2$ ;
```

As shown in Table 1 the price for considering more than two objectives is a factor N_{\max} in the second summand of the algorithm's time complexity. If we neglect the overhead that arises due to communication of threads and assume that k threads are available, we can set $B = \frac{N_{\max}}{k}$ and reduce the overall running time of the algorithm depending on the number of threads. Note that the threads do not write or read to/from same locations during *parallelDominates*, i.e., communication is minimal. However, N_{\max} is not known a priori and thus some parameter tuning might be needed when implementing *parallelDominates*.

After extracting a label at node v from the heap, the next heap label for v is determined in function *nextCandidateLabel*. This search can also be split into tasks by separating the predecessor nodes of v into groups. Let

$$p_v := \sum_{u \in \delta^-(v)} (|L_u| - \text{lastProcessedLabel}[(u, v)]) \quad (5)$$

be the number of predecessor labels of v that have to be considered in *nextCandidateLabel*³. Given an upper bound $B' \in \mathbb{N}$, $p_v > B'$, on the number of labels that each task will consider, we can split the workload. Ideally each task will find a candidate label for v among $\frac{p_v}{B'}$ predecessor labels. After doing so, the next heap label l_v^{new} for v is found by choosing the lexicographically smallest candidate label. We assume that there is a function *splitPredecessors*($v, \text{lastProcessedLabel}, B'$) that returns groups of predecessors $P_i \subseteq \delta^-(v)$ such that $\cup P_i = \delta^-(v)$. Then, the function *nextCandidateLabelParallel* can be used to compute (1) in parallel if the tasks generated in Line 4 are processed by different threads.

Procedure *nextCandidateLabelParallel*

Input : Node v , Indices *lastProcessedLabel*, Permanent Labels L , Bound B' .

Output: New lexicographically smallest, non-dominated label for v , if one exists.

- 1 Predecessor Subsets $[P_i]_{i \in I} \leftarrow \text{splitPredecessors}(v, \text{lastProcessedLabel}, B')$;
 - 2 Labels $[l_{v,i}]_{i \in I} \leftarrow [\text{NULL}]_{i \in I}$;
 - 3 **parallel tasks** $i \in I$
 - 4 $l_{v,i} \leftarrow \text{nextCandidateLabel}(v, \text{lastProcessedLabel}, P_i, L)$;
 - 5 **return** $\text{lex min}\{l_{v,i} \mid i \in I, l_{v,i} \neq \text{NULL}\}$;
-

4 Martins' Algorithm

As noted in the introduction, label setting MOSP algorithms are usually variants and improvements of Martins' algorithm [25]. In Algorithm 2, we choose to present the variant introduced in [13] since the authors claim (and report computational evidence) that it is a sped up version of Martins' algorithm.

The set of temporary labels in Algorithm 2 is also a lexicographically ordered priority queue H but its size is not bounded by the number of nodes in the graph: more than one label per node can be stored in the H simultaneously. At the end of the algorithm, the sets of node labels L_v for every node $v \in V$ store a minimum complete set of efficient (s, v) -paths. However, during the search dominated labels can appear therein. Hence, every time a new non-dominated label l_v is added to a set L_v , we have to check whether it dominates existing labels therein. In this case, these labels have to be removed. This operation, often called *merge* between l_v and L_v , is one of the main differences between Martins' algorithm and the new MDA.

Initially, in Algorithm 2, a label with zero costs at the starting node s is generated and inserted into H and into L_s . The main loop of the algorithm goes on until H becomes empty. In every iteration the lexicographically smallest label l_v^* is extracted from H . For all successor nodes w of v , the tentative label $l_w := (w, cl_v^* + c_{vw}, l_v^*)$ is build. Now the merge operation between L_w and l_w takes places: l_w is added into H and L_w if it is not dominated by any other label in L_w . If added, l_w might dominate some of the existing labels in L_w and these labels are deleted from L_w .

Remark 2 (Martins' Algorithm – Correctness and Running Time). The proof of correctness of Algorithm 2 can be found for example in [25] or [18]. The Algorithm runs in $\mathcal{O}(ndN^2)$ time, where, as in Section 3.2, N is the total number of non-dominated labels at the end of the algorithm. A complete proof of this bound is given in [7].

From Table 1 and Remark 2, we can see that the running time of our algorithm is superior to that of the algorithm by Martins. Firstly, bounding the number of labels in the priority queue allows us to explicitly mirror the priority queue extraction in the running time of Algorithm 1. Secondly, the MDA avoids the tedious merge operation (Line 15 in Algorithm 2). Hence, a label extracted later will never dominate an existing one. The price for avoiding the merge operation are the calls to *nextCandidateLabel*.

Removing labels from the sets L_v as in (Line 15 of Algorithm 2) hides a major drawback: removed, hence dominated labels, still remain in the priority queue H . Let l_v be a label just extracted from H in Line 7 of Algorithm 2 and assume that it was deleted from L_v in a prior iteration. Due to subpath optimality every propagated label gotten from l_v will also be dominated. Hence, the iteration should be aborted after extracting

³Note that in (5) L_u is not the final set of permanent labels at u but the set containing the ones found at the moment the function is called.

Algorithm 2: Demeyer’s et al. [13] variant of Martins’ algorithm.

Blue lines only for one-to-one version described in Section 5.

```

Input :  $G = (V, A)$ ,  $c_a \in \mathbb{R}_{\geq}^d$  for  $a \in A$ ,  $s \in V$ .
Input one-to-one: Target node  $t \in V$ , Lower bound  $\underline{c}_v$ ,  $v \in V$ , Upper bound label at target  $\bar{l}_t$ .
Output : Sets of non dominated labels  $L_v$ ,  $v \in V$  OR Non-dominated target labels  $L_t$ .

1 Priority queue  $H \leftarrow \emptyset$ ;
2 for  $v \in V$  do  $L_v \leftarrow \emptyset$ ;
3 Initial label  $l_{init} \leftarrow (s, (0, \dots, 0), \text{NULL})$ ;
  /* The priority queue  $H$  is sorted in lexicographically increasing order. */
4  $H.\text{insert}(l_{init})$ ;
5  $L_s.\text{insert}(l_{init})$ ;
6 while  $H \neq \emptyset$  do
7   Label  $l_v^* \leftarrow H.\text{extract\_min}()$ ;
8   Node  $v \leftarrow l_v^*.node$ ;
9   for  $w \in \delta^+(v)$  do
10    Label  $l \leftarrow (v, c_{l_v^*} + c_{(v,w)}, l_v^*)$ ;
11    if  $\bar{l}_t \not\leq_D (t, c_l + \underline{c}_w, \text{NULL})$  and  $L_t \not\leq_D (t, c_l + \underline{c}_w, \text{NULL})$  then
12      if  $L_w \not\leq_D l$  then
13         $H.\text{insert}(l)$ ;
14        Remove elements  $L_{rem} \leftarrow \{l_w \in L_w \mid l \preceq_D l_w\}$ ;
15         $L_w \leftarrow (L_w \cup \{l\}) \setminus L_{rem}$ ;
16 return  $L_v$  for  $v \in V$ ; OR return  $L_t$ ;

```

l_v . A constant-time check of whether l_v is still in L_v can be done choosing an appropriate data structure (e.g., a hash map). In Section 6.1 we will discuss the implications of the choice of different data structures in more detail. Removing the label l_v from L_v and from H would imply finding a not necessarily minimum element in a priority queue, which is also a costly operation.

5 One-to-One MOSP Problem.

In the one-to-one MOSP problem we are interested in the efficient paths between two input nodes $s, t \in V$. The output of the MDA in this scenario is a minimum complete set of non-dominated labels at t . We adapt Algorithm 1 to solve one-to-one MOSP instances introducing *pruning techniques* to discard tentative labels at nodes $v \neq t$ that provably will not be expanded to non-dominated labels at t . The pruning techniques discussed in what follows are widely used and their impact on the computational performance of the algorithms presented here is remarkable. A detailed presentation can be found e.g., in [15].

5.1 Pruning by dominance.

For a node $v \in V$, let $\underline{c}_v \in (\mathbb{R}_{\geq} \cup \{\infty\})^d$ be a vector of costs that in each dimension $j \in \{1, \dots, d\}$ lower bounds the cost of a (v, t) -path w.r.t. the j^{th} cost component. Given such a lower bound for every node in the graph, tentative labels at an extracted node can early be recognized to be provably irrelevant for the final set of non-dominated labels.

Let l_v be a tentative label for node v . If the extended cost $c_{l_v} + \underline{c}_v$ is dominated by the cost of any label in L_t , then l_v can be pruned from the search space. We find feasible lower bounds for the nodes’ cost in a preprocessing step that performs d (single criteria) t -to-all *lexicographic SP queries* on the reverse digraph $\bar{G} := (V, \{(v, u) \mid (u, v) \in A\})$. The cost of a reversed arc (v, u) in the j^{th} query is the j^{th} cost component of the original arc (u, v) . The queries being lexicographic means while optimizing according to one cost component only, the tie break criterion in case two labels have the same (single) costs is their lexicographic ordering. Given a d -dimensional MOSP instance, there are $d!$ possible ways of ordering the objectives to perform such queries. This would result in a preprocessing phase without polynomial running time. To overcome this issue, we restrict ourselves to the computation of d such queries, each of them having one of the d cost components as its first/main optimization criteria. To build d adequate permutations of the objectives, we keep a closed list (*round-robin* list) from 1 to d and make shift to the left for every new query. We then set $\underline{c}_{v,j}$ to the cost of the shortest (t, v) -path

w.r.t. the j^{th} SP query, i.e., the one with the j^{th} cost component as first optimization criteria.

In Algorithm 1 the described pruning strategy can be used to reduce the number of labels built in *nextCandidateLabel* and *propagate*. The corresponding checks appear in Line 8 of *nextCandidateLabel* and Line 3 of *propagate*.

5.2 Upper Target Bound Pruning.

The dominance check at the target node's set of non-dominated labels we introduced in the last subsection might be costly as the size of L_t gets bigger during the MDA. A less aggressive, yet faster pruning technique can be applied if an upper bound on the costs of the efficient (s, t) -paths is known. To find such an upper bound we can reuse the d lexicographic all-to-one queries that we used to determine \mathcal{C}_v . For the query w.r.t. the j^{th} cost component, let P_{st}^j be the found shortest (s, t) -path and $\pi^j := c(P_{st}^j)$ the corresponding multiobjective cost vector. Then

$$\bar{c}_t := \left(\max_{j \in \{1, \dots, d\}} \pi_1^j, \dots, \max_{j \in \{1, \dots, d\}} \pi_d^j \right) \quad (6)$$

defines an upper bound on the costs of all efficient (s, t) -paths. This is guaranteed because the ran queries were performed using the lexicographic ordering of the labels as tie breakers. The results of the single queries correspond to efficient paths that dominate a path with costs \bar{c}_t . Thus, the artificial cost $c_{l_v} + \mathcal{C}_v$ of a candidate label l_v at a node v can be compared with \bar{c}_t (see [15] for a complete proof). In case $\bar{c}_t \prec_D c_{l_v} + \mathcal{C}_v$, l_v can be discarded. Upper target bound pruning is weaker than pruning by dominance because for any non-dominated label $l \in L_t$ we have $\bar{c}_t \not\prec_D c_l$. However, pruning by \bar{c}_t is faster since it involves only the comparison of two cost vectors. We therefore use it before pruning by dominance, in Line 8 of *nextCandidateLabel* and Line 3 of *propagate*. To be consistent with our notation, we introduce an artificial label $\bar{l}_t := (t, \bar{c}_t, \text{NULL})$ and use it in our pseudocode when we apply upper target bound pruning.

5.3 Pruning and early stop condition in Martins' algorithm.

The described pruning techniques can also be included in Algorithm 2 when it is used to solve one-to-one MOSP instances (Line 11). Each time a tentative label l_{tent} at a node w is determined (Line 10), we can extend its costs towards node t and get $\mathcal{C}_{w,t} := c_{l_{tent}} + \mathcal{C}_w$. We can discard l_{tent} if either $\bar{c}_t \prec_D \mathcal{C}_{w,t}$ or if there exists a label in L_t that dominates the artificial label with costs $\mathcal{C}_{w,t}$.

In [13] the authors introduce an *early stop condition* for Martins' algorithm that reduces the number of needed iterations. Let $\min_i(H) := \min_{l \in H} (c_l)_i$ be the minimum value among the i^{th} cost component of all labels in the priority queue H . Recalling that arc cost are non-negative, it is easy to see that if $\mathcal{C}_H := (\min_1(H), \dots, \min_d(H))$ is dominated by a label to the target node that is in the list L_t , no label in H will be expanded to a non-dominated label at the target.

Remark 3 (Search Space Labels and Pruning by Dominance). There is no guarantee that Martins' algorithm and the MD algorithm will extract the same label in the i^{th} iteration of their corresponding main loop. Hence, the set of labels at the target node t can look differently for both algorithms during the execution. This has an impact on the possibility to prune labels by dominance. In general, the following holds: if pruning by dominance is used, at the end of both algorithms the sets of non-dominated labels at t will coincide but the sets of non-dominated labels at intermediate nodes do not.

The running time of the preprocessing needed to define the lower and upper bounds used for the presented pruning techniques is polynomial in the input size since we run d SP queries only. In the presented pseudocodes we assume that the bounds are part of the input. In what follows we will refer to Martins' algorithm with pruning by dominance and upper target bound pruning as *Improved Martins' Algorithm* (IMA).

6 Experiments

We compare the IMA and the MDA on different one-to-one MOSP instances with $d = 3$. The reasons to do so are twofold: on one hand, we had data from applications involving three cost components; on the other hand, contrary to what happens when switching from two to three cost components, there is no algorithmic modification when using more than three objectives. Moreover, it is likely that due to memory or time requirements, we could have not considered the large instances used in this work for $d > 3$. Our intention however was to observe the behavior of the algorithms on large and practically relevant MOSP instances. This is also the reason why our experimental results consider only one-to-one MOSP instances. In Section 6.1 we detail how both algorithms

Table 2: Datastructures and pruning techniques used in our implementations.

	IMA	MDA
Pruning	By Dominance Upper Target	By Dominance Upper Target
Heap	Bin. Heap ($<_{lex}$)	Bin. Heap ($<_{lex}$)
L_v	Hash Tables	Array

were implemented. Section 6.2 specifies the instances used in our experiments and how they were generated. Finally, in Section 6.3 we report and analyze the results.

6.1 Implementation Details

Both algorithms are implemented in C and we use the GCC compiler version 7.5 to build the binaries⁴. The relevant implementation aspects are the data structures, the used pruning techniques, and how the MDA was parallelized.

6.1.1 Data Structures

In both algorithms we use a binary heap as the priority queue H . In the MDA, the sets L_v of permanent labels at the nodes are implemented as dynamically allocated arrays. If a label l is to be added to an array L_v and the allocated size is completely in use, we double the array’s size. If the number of permanent labels at a node $i \in V$ is $N_i \in [2^{k_i}, 2^{k_i+1}]$ for some $k_i \in \mathbb{N}_0$, allocating the size of the arrays as described here causes an overhead in the storage space needed by the algorithm of at most $\sum_{i \in V} 2^{k_i+1} \leq 2N$ times the size of a label⁵. In *propagate* the heap’s label of a node v is possibly updated. To check if an update has to be done, we avoid searching for v ’s label in the heap by storing the node’s heap labels in a vector indexed by nodes. This means that we store at most n labels twice during the algorithm but it guarantees constant-time access to v ’s label in H .

Concerning our implementation of the IMA, we tried to benefit from the improvements introduced in [13]. Note that dominated labels are only deleted from the sets of node labels L_v . Hence, in every iteration, we have to check if the label extracted from the heap of temporary labels has already been deleted from the corresponding L_v set. To perform this check in constant time, we implement the L_v sets as HashTables. We used the HashTable implementation from the open source GNOME Library GLib 2.64.5 [23]. Since each set L_v does only contain labels at node v , it is enough to hash the labels using their cost vector only. We hash a label’s cost components using our own implementation of the `hash_combine` function included in the Boost libraries [6].

6.1.2 Pruning

In both algorithms we use pruning by dominance and upper target pruning. In the IMA we first tried to use the early stop condition from Section 5.3. Doing so helped to reduce the running time of the algorithm but not as much as using pruning by dominance and upper target pruning, i.e., our final implementation does not include the early stop condition. The reason is that it needs to traverse the heap many times looking for the minimum value of the labels’ cost components in all but the first dimension. Table 2 shows an overview of the used data structures and pruning techniques.

6.1.3 Parallelization

We use version 4.5 of the *OpenMP API for Parallel Programming* [5] to parallelize the dominance checks and the *nextCandidateLabel* function as described in Section 3.3. We initialize k threads at the beginning of the algorithm and let all but one (the *master* thread) threads wait until tasks are generated. Tasks are put into a queue from which all active threads can take an element and perform the required computations. The assignment from tasks to threads is controlled by the Open MP task scheduler. Once all tasks are completed their results are compared and aggregated to one final return value.

In the recursive parallel functions *parallelDominates* and *nextCandidateLabelParallel* we introduced bounds B and B' respectively. Workload that is below these bounds is performed by the serial versions of the functions.

⁴Compiler optimization level was set to `O3`.

⁵Note that despite the remarked downside, this is a widely used procedure for efficient memory allocation.

In our experiments we set $B = 3,000$ and $B' = 50^6$. In the description of the *nextCandidateLabelParallel* we assumed the existence of a function *splitPredecessors* that builds groups of predecessor labels. To keep this task computationally cheap, we iterate through the predecessor nodes of the input node v and group predecessor nodes together until the sum of the corresponding permanent labels is greater or equal than B' .

6.2 Instance description

In our experiments we consider four different types of instances. The properties of the underlying graphs are summarized in Table 3.

Grid Graphs We considered a directed 100×100 grid graph. Arcs between neighbouring nodes exist in both directions. This results in a graph with 10000 nodes and 39600 arcs. Every arc (u, v) has 3-dimensional costs and $c_{uv} = c_{vu}$. The costs were generated uniformly at random with values between 1 and 10 for each component separately. We use 50 different 3-dimensional cost functions on the 100×100 grid described here. These instances were already used and described in [28]. Each node gets an id. The ids start at 0 at the lower left node of the grid and increase first vertically (the node on the upper right corner has id 99) and then horizontally (the node to the right of node 0 has id 100). For every pair consisting of the grid graph and a cost function, we create 60 (s, t) pairs. We do so by taking 20 random node pairs with ids differing by less than 3333 (Grid-small), 20 random node pairs with ids differing between 3333 and 6666 (Grid-medium), and 20 random node pairs with ids differing by more than 6666 (Grid-big). Since we consider 50 cost functions, we get 1000 Grid-small instances (G-S), 1000 Grid-medium instances (G-M), and 1000 Grid-big instances (G-B).

NetMaker NetMaker graphs are synthetic graphs. They were introduced in [35] and have been used for benchmarks in multiple publications (see for example [29]). The considered graphs have 5000 to 30000 nodes and 29591 to 688398 arcs. The density of the graphs ranges from 5.92 to 23.05. As explained in [30], NetMaker graphs are build around a Hamiltonian cycle that ensures the connectivity of the graph. Additionally, the nodes of the graph are assumed to be numbered and arcs are only allowed to connect nodes in a certain range w.r.t. their numbering. This helps avoiding direct paths or shortcuts between nodes. Arcs have 3-dimensional costs between 1 and 1000. For an arc a there is always a cost component with costs between 1 and 333, a cost component with costs between 334 and 666, and a cost component between 667 and 1000. Cost generation for an arc is a two stage random process: first the interval in which the cost component will lie is chosen randomly (among the not yet used intervals for this arc costs) and then the actual costs are generated randomly.

Road Networks Road networks are directed graphs often used to benchmark shortest path queries [2]. We use the road networks available from the *9th DIMACS Implementation Challenge on Shortest Paths* [20]. The available arc cost functions are the arcs' distance and the arcs' travel time. Additionally, we added a third cost component to every arc that is always 1, i.e., we aim to minimize the number of arcs along a path (or also sometimes called number of hubs). For every considered graph, we selected 50 random (s, t) pairs to generate our instances.

Airway Networks Airway Networks are directed graphs used in flight planning for commercial airlines [4]. Nodes can either represent airports or so called waypoints. A waypoint is an intersection of two airways/segments which in turn represent the arcs of the graph. An example of the current worldwide airway network can be seen at www.skyvector.com⁷. Using data provided by our partner Lufthansa Systems GmbH & Co. KG. we built the airway network over Europe and got a directed graph with 70,425 nodes and 137,628 edges. The first cost component of every arc is the underlying segment's length. The second cost component is the travel time along the segment given how the wind was on November 26th, 2016, at 06am. The third cost component are the segment's overflight costs [3]. Overflight costs can be thought of as the costs that airlines have to pay to every country that their aircraft overfly along a route. We uniformly at random built 100 pairs of airports to define the used instances. A 101st instance from Tenerife to Berlin was solved.

⁶We set these values after some parameter tuning since they work well for all instance types. Instance dependent considerations and choices could yield better results.

⁷Use the option "World Hi".

Table 3: Overview of the used graphs.

	Nodes n		Arcs m		Density m/n	
	min.	max.	min.	max.	min.	max.
Grid Graphs	10,000		39,600		3.96	
NetMaker	5,000	30,000	29,591	688,398	5.92	23.05
Road Graphs	264,346	14,081,816	730,100	33,866,826	2.39	2.76
EU Airway Network	70,425		137,628		1.95	

6.3 Results

All experiments were run on a machine with an Intel Xeon CPU E5-2680 0 @ 2.70GHz processor. It has 2 CPUs per node and 8 cores per CPU. The available RAM was 64GB. In Sections 6.3.3-6.3.6 we analyze the results for every graph type separately. We first compare the results of the serial versions of the IMA and the MDA. Afterwards we focus on the parallel version of the MDA considering different number of threads.

6.3.1 Understanding the Presented Results

For both algorithms we set a time limit to $T = 5400s$. The running time of the instances that are not solved when the time limit is reached is set to T . Recall that by N we denote the total number of non-dominated labels. Additionally, we refer to the number of non-dominated labels at the target node t by N_t . Let \mathcal{I} be an instance. We denote by $T_{\mathcal{I},C}$ and $T_{\mathcal{I},M}$ the running time of the IMA and the MDA respectively. Then, the speedup for \mathcal{I} is $s_{\mathcal{I}} := T_{\mathcal{I},C}/T_{\mathcal{I},M}$. The average speedups reported here are the geometric means of the speedups in the corresponding instance set. Instances that could not be solved because the memory limit was reached are neglected when building the averages. An interesting figure when comparing the performance of the algorithms is the number of extracted labels, which we denote by N_{ext} . Note that in the IMA N_{ext} and N can differ since labels in L_v are not permanent until the end of the algorithm (see Remark 3).

6.3.2 Parallel Performance

Even if the dominance checks account for 85% of the running time in every instance, the benefits through parallelization were not as expected. The reason is that the sets L_v of permanent labels are arrays and checking dominance between two labels entails the comparison of $d = 3$ integers. Hence, it is only for very large sets of permanent labels that the net benefit through parallelization is greater than the overhead caused through thread communication. On the other hand, the parallelization of the *nextCandidateLabel* offered better results. Looking for a node's next heap label among the unexplored permanent predecessor labels of this node can be, in particular for big instances, an expensive computation. Splitting the predecessors into groups and assigning each group to a task becomes beneficial as the density of the underlying graph grows. When reporting speedups, we always take the serial running time divided by the parallel running time.

6.3.3 Grid Instances

Even if the 100×100 grid is the smallest considered graph, the random costs on the arcs make the resulting instances have many efficient solutions at the target nodes. Table 4 compares the serial results of both algorithms on grid instances. When reporting about labels (N_{ext} , N , or N_t), only instances that were solved by both algorithms within the time limit are considered. Solution times and speedup are treated as explained in Section 6.3.1.

From the G-S group both algorithms managed to solved all instances. On average the MDA was five times faster than IMA, the speedup ranging from 1.0 to 13.5 among all instances. In the G-M group source and destination nodes were allowed to lie farther apart from each other. This resulted in instances having 4,755 target nodes on average and around 4,8M (IMA) resp. 4,3M (MDA) labels in the search space. The average speedup of $\times 7.02$ was greater than in the G-S instance set. We note that in the G-M set the IMA failed to solve 31 instances because it reached the time limit. The trend continued in the G-B group: the IMA did not manage to solve 200 instances, while the MDA handled all but four. All failed instances failed because the time limit was reached. Among the set of instances solved by both algorithms, the average number of labels at the target nodes was 6,501 and the search space contained around 10,8M labels on average. It is remarkable that, as seen in Figure 2 or Table 5, the MDA could handle instances with more than 74,5M labels without reaching the time limit. In particular the bottom plot in Figure 2 shows how the running time increase depending on N is way lower for the MDA. The IMA instances do not appear for large values of N since the algorithm runs out of time.

Table 4: IMA vs. serial MDA on the Grid instance sets. The comparison only considers instances that were solved by both algorithms.

		IMA				MDA		Speedup
		N_t	time [s]	N_{ext}	N	time [s]	$(N=)N_{ext}$	
		Solved 1,000/1,000				Solved 1,000/1,000		
G-S	Avg.	1,478	1.643	1,682,714	1,499,954	0.323	1,500,523	$\times 5.09$
	Min.	1	0.001	3	3	0.001	4	$\times 1.00$
	Max.	11,330	3439.764	21,003,709	18,675,973	373.429	18,422,930	$\times 13.50$
		Solved 969/1,000				Solved 1,000/1,000		
G-M	Avg.	4,755	83.648	4,849,986	4,333,563	11.907	4,343,790	$\times 7.02$
	Min.	127	0.355	95,468	87,721	0.075	85,081	$\times 4.18$
	Max.	18,814	5395.960	26,934,232	23,963,251	634.437	23,559,011	$\times 10.36$
		Solved 800/1,000				Solved 996/1,000		
G-B	Avg.	6,501	687.311	10,776,651	9,596,356	75.830	9,591,892	$\times 9.06$
	Min.	693	23.129	1,427,727	1,290,449	3.437	1,304,614	$\times 6.67$
	Max.	19,441	5384.353	28,589,673	25,390,745	633.355	25,155,008	$\times 11.13$

Table 5: Parallelization of Grid instances. For 2 to 5 threads the speedup w.r.t. the serial version is shown.

		N_t	N	Threads			
				2	3	4	5
G-S	Avg.	533	169,491	0.915	0.871	0.828	0.810
	Min.	1	4	0.500	0.500	0.500	0.500
	Max.	11,330	18,422,930	2.000	2.000	2.000	2.000
G-M	Avg.	3,271	2,552,942	0.990	0.929	0.885	0.863
	Min.	127	85,081	0.731	0.707	0.654	0.617
	Max.	25,931	44,589,845	1.225	1.182	1.175	1.167
G-B	Avg.	6,934	10,787,338	1.009	1.030	0.999	0.984
	Min.	693	1,304,614	0.884	0.808	0.76	0.729
	Max.	39,710	74,520,469	1.366	1.328	1.335	1.335

The parallelization for Grid instances follows the general remarks stated in Section 6.3.2: in every instance set the average and maximum speedup is greatest when two threads are active. The average speedup is only greater than one for the G-B instance group using at most three threads. Table 5 shows the comparison of the parallel results. We stopped after experimenting with 5 threads because the average speedup for G-B consolidated below 1.0. We believe that the reason is that the Grid instances are sparse.

6.3.4 NetMaker Instances

The serial results of the Netmaker instances are shown in Table 6. On average, these instances are easier to solve than the grid instances since the search space contains less non-dominated labels than the Grid instances. The intuitive reason could be the existence of the Hamiltonian cycle in the graph: all other arcs constitute possible shortcuts w.r.t. the path connecting s and t along the cycle. Then, whenever a shortcut ends *close enough* to the target node, the cycle offers a way to reach it without using many arcs. Thus, the average speedup increases with the graph size but goes up to $\times 3.09$ only. Note that the average number of labels at the target node remains similar among all groups of NetMaker instances. Also the maximum number of labels remains similar in all but the last group, where it reaches 11,566. Solvability is not a big issue among the Netmaker instances: the MDA solves always at least as many instances as the IMA but the latter solves always more than 90%. There are Netmaker instances on which the IMA is faster. However, the corresponding search space is always very small: if the IMA does not have to delete labels from the lists L_v , it can be faster than the MDA. Figure 3 shows the running time needed by both algorithms to solve all NetMaker instances depending on N_t and N . The picture is not as clear as for the Grid instances but as N increases the better performance of the MDA becomes more notable.

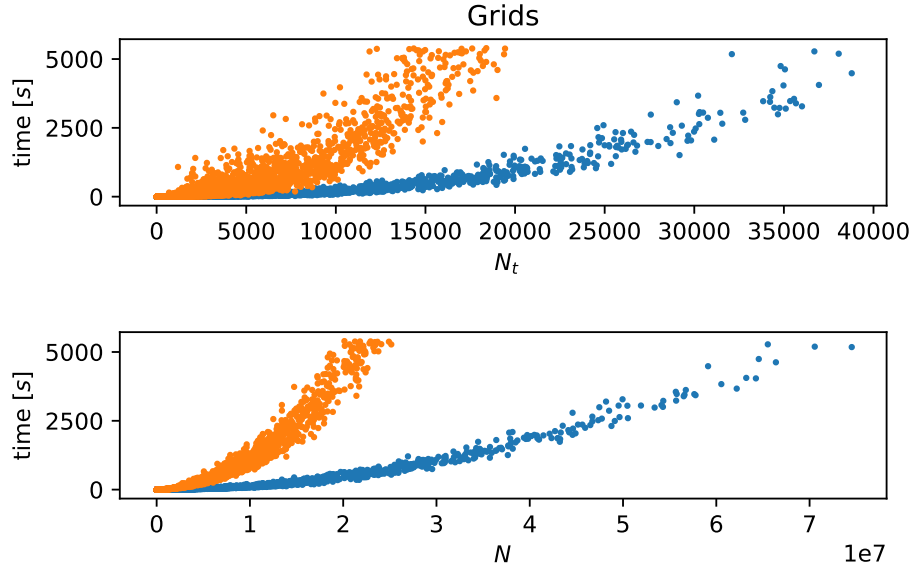


Figure 2: Solution times depending on N_t and N . Orange dots – IMA. Blue dots – MDA

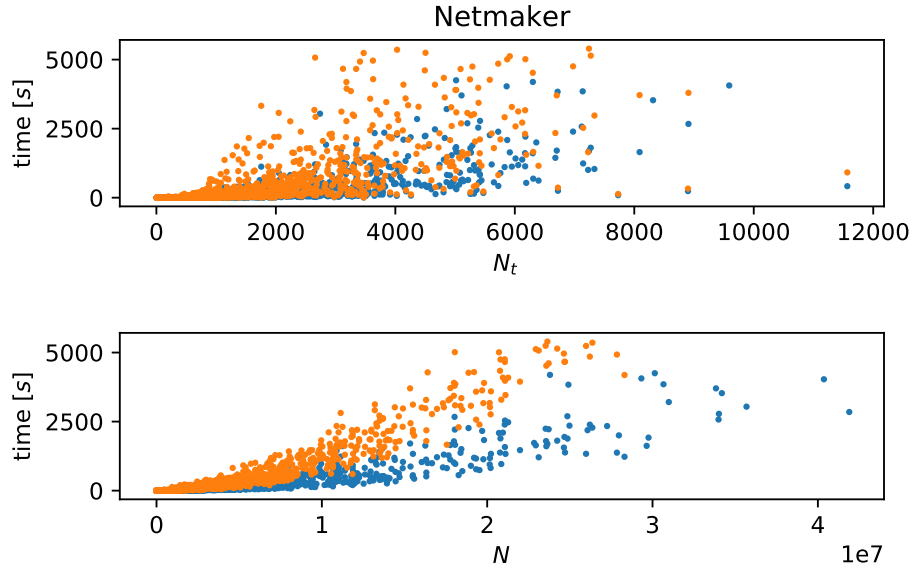


Figure 3: Solution times depending on N_t and N . Orange dots – IMA. Blue dots – MDA

Table 6: IMA vs. serial MDA on the Netmaker instance sets. The comparison only considers instances that were solved by both algorithms.

		IMA				MDA			
		N_t	time [s]	N_{ext}		N	time [s]	$(N=)N_{ext}$	Speedup
Net-5000			Solved 240/240				Solved 240/240		
	Avg.	1,142	8.096	1,220,551		902,825	3.505	909,528	$\times 2.31$
	Min.	2	0.001	9		9	0.001	44	$\times 0.53$
	Max.	7,730	973.745	8,603,674		5,857,185	424.093	5,708,330	$\times 5.52$
Net-10000			Solved 220/220				Solved 220/220		
	Avg.	1,477	24.28	3,021,651		2,239,276	9.391	2,252,072	$\times 2.58$
	Min.	11	0.002	131		129	0.001	182	$\times 0.68$
	Max.	6,679	2913.098	17,422,129		12,353,185	1443.320	13,201,644	$\times 6.63$
Net-15000			Solved 238/240				Solved 240/240		
	Avg.	1,701	57.313	5,169,096		3,793,429	20.243	3,805,065	$\times 2.83$
	Min.	86	0.013	4,236		3,667	0.005	2,879	$\times 0.63$
	Max.	7,227	3703.363	23,973,287		15,878,181	1705.126	16,303,998	$\times 7.34$
Net-20000			Solved 236/240				Solved 239/240		
	Avg.	1,808	88.243	6,720,274		4,910,920	29.589	4,843,874	$\times 2.98$
	Min.	39	0.011	2,748		2,258	0.006	2,191	$\times 1.33$
	Max.	8,907	5397.193	34,581,606		24,069,039	2671.336	23,640,287	$\times 7.67$
Net-25000			Solved 234/240				Solved 239/240		
	Avg.	1,715	95.105	9,157,740		6,628,449	31.505	6,542,399	$\times 3.02$
	Min.	13	0.004	271		241	0.003	313	$\times 1.17$
	Max.	6,397	5356.872	38,396,986		26,193,328	2537.255	27,844,522	$\times 6.80$
Net-30000			Solved 225/240				Solved 239/240		
	Avg.	1,857	122.074	8,994,510		6,695,299	39.524	6,561,003	$\times 3.09$
	Min.	36	0.008	1,032		938	0.005	1,107	$\times 1.11$
	Max.	11,566	5246.558	38,405,012		27,139,791	2392.773	28,305,417	$\times 7.46$

Table 7: Parallel Netmaker

		Threads							
		N_t	N	2	3	4	5	6	7
Net-5000	Avg.	696	406,038	1.279	1.195	1.169	1.160	1.141	1.085
	Min.	2	44	1.000	0.974	0.910	0.864	0.797	0.716
	Max.	7,730	5,708,330	1.667	1.667	1.560	2.000	1.667	1.667
Net-10000	Avg.	941	1,032,373	1.270	1.190	1.166	1.151	1.136	1.085
	Min.	11	182	1.000	0.984	0.963	0.932	0.892	0.806
	Max.	6,679	13,201,644	2.000	1.588	1.540	1.531	1.597	1.539
Net-15000	Avg.	1,126	1,937,120	1.269	1.189	1.173	1.158	1.146	1.090
	Min.	86	2,879	0.870	0.712	0.680	0.638	0.589	0.523
	Max.	7,227	23,797,481	1.616	1.584	1.586	1.578	1.659	1.600
Net-20000	Avg.	1,252	2,723,965	1.266	1.188	1.171	1.159	1.147	1.095
	Min.	39	2,191	1.016	0.909	0.870	0.827	0.775	0.696
	Max.	8,907	30,669,161	1.542	1.503	1.594	1.590	1.668	1.615
Net-25000	Avg.	1,114	3,126,577	1.278	1.204	1.191	1.176	1.167	1.114
	Min.	0	313	1.000	0.975	0.960	0.940	0.899	0.750
	Max.	6,397	27,963,730	1.667	1.523	1.551	1.547	1.613	1.554
Net-30000	Avg.	1,344	4,074,560	1.263	1.187	1.173	1.163	1.152	1.101
	Min.	36	1,107	0.956	0.833	0.799	0.772	0.712	0.648
	Max.	11,566	41,898,880	1.570	1.484	1.552	1.548	1.610	1.552

Among all used graphs the ones in the NetMaker instances reach the greatest density (up to 23.05). Hence, the search for a new label in *nextCandidateLabel* iterates over labels in many predecessor nodes. As we noted, there are not many non-dominated labels per node in these instances, so the workload caused by the dominance checks is rarely split into tasks. We parallelized these instances using up to seven threads. Contrary to what we saw in the Grid instances, the parallel version always outperforms the serial one (see Table 7). However, the best average speedup is still reached using two threads only and ranges between 1.263 for the Net-3000 instances and 1.279 for the Net-5000 instances.

We analyze the parallel results of the NetMaker instances depending on the density of the graphs. As the density increases, the parallelization of the *nextCandidateLabel* search becomes more meaningful. Figure 4 shows the average speedup w.r.t. the serial version of the algorithm for every density among the NetMaker graphs depending on the number of threads. Grouping instances by density doesn't change the fact that using only two threads yields best results. For all threads the trend is the same: sorting instances by increasing densities yields an increasingly sorted view on the running time improvement through parallelization. The only outlier is the group of instances with the lowest density of 6 since the parallelization of these instances performs better than the one for instance groups up to a density of 16. The best group of instances is always the one containing the graphs with the highest density (23). The parallelization of these instances using two threads yields a speedup of 1.39. Using seven threads the average speedup is 1.31. We stopped after seven threads because even though switching from five to six threads yields an improvement for some instance sets (see Table 7), the running times went up again using seven threads.

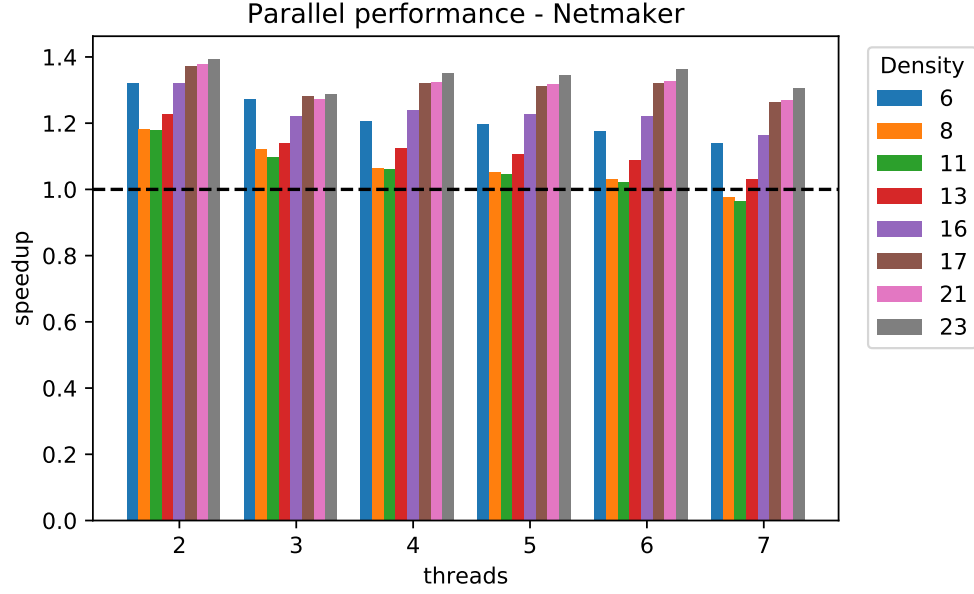


Figure 4: Average parallel performance of the Netmaker instances analyzed by graph densities. Speedups above 1.0 imply that the parallel version outperforms the serial one.

Table 8: Solvability of Road Networks using serial versions of algorithms.

Graph	IMA			MDA		
	solved	out of time	out of mem.	solved	out of time	out of mem.
BAY	20	25	5	31	19	0
COL	16	32	2	24	26	0
NE	4	45	1	11	39	0
NY	28	16	6	36	14	0

Table 9: IMA vs. serial MDA on the Roads instance set. The comparison only considers instances that were solved by both algorithms.

		IMA			MDA			Speedup
		N_t	time [s]	N_{ext}	N	time [s]	$(N=)N_{ext}$	
		Solved 28/50			Solved 36/50			
NY	Avg.	1,431	57.418	19,696,620	17,851,922	8.530	17,875,409	$\times 6.73$
	Min.	16	0.065	15,655	15,023	0.023	15,018	$\times 2.83$
	Max.	6,502	4719.710	60,288,358	53,266,643	516.811	53,422,870	$\times 12.93$
		Solved 20/50			Solved 31/50			
BAY	Avg.	822	15.693	14,438,830	13,656,241	2.869	13,623,620	$\times 5.47$
	Min.	5	0.057	719	689	0.021	710	$\times 2.67$
	Max.	3,136	3959.391	53,032,972	50,683,607	399.339	50,484,009	$\times 9.91$
		Solved 16/50			Solved 24/50			
COL	Avg.	1,614	65.130	19,221,293	18,163,472	9.550	18,231,712	$\times 6.82$
	Min.	2	0.076	651	640	0.028	638	$\times 2.64$
	Max.	8,250	4916.035	96,335,451	91,431,937	507.162	91,260,276	$\times 10.82$
		Solved 4/50			Solved 11/50			
NE	Avg.	1,290	17.208	16,745,972	15,932,695	3.212	15,903,145	$\times 5.36$
	Min.	82	0.640	182,599	171,620	0.206	176,968	$\times 3.11$
	Max.	4,392	2492.843	63,279,074	60,330,338	282.710	60,165,456	$\times 8.82$

6.3.5 Road Instances

As shown in Table 3, road networks are the biggest graph considered in our experiments. We have chosen to report results related to the instances defined on the BAY, COL, NE, and NY graphs since these are the ones where a meaningful number of instances could be solved within the time limit and given the available memory. Table 8 shows how the IMA and the MDA performed in terms of solvability. Following the trend, the MDA always managed to solve more instances than the IMA. The ones that could not be solved by the MDA reached the time limit. The IMA had a memory consumption issue: the hash tables used to store the nodes' labels cause some instances to exhaust the available memory.

Table 9 shows that the average speedup ranges from 5.36 for the NE instances to 6.82 for the COL instances. The maximum speedup is close to an order of magnitude for every road instance set but for the one using the NE graph (only 8.82). Since in the table we only compare instances solved by both algorithms, the data does not show how many labels the MDA is able to analyze within the given time limit. This can be seen in Table 10: in its serial version, the MDA manages to process more than 295M labels in a BAY instance. This is also the maximum number of processed labels among all solved instances analyzed in our experiments. The maximum number of labels that the IMA managed to process was 63M in an instance using the COL network. The running time depending on the number of permanent labels N is shown in Figure 5.

In Table 10 we see that parallelizing the MDA has not a big impact on the number of instances that could be solved. In BAY and COL instance sets, using three threads helps to solve one more instance than in the serial or 2-threaded versions. In the NE and NY instance sets, already two threads manage to solve one more instance than in the serial version. Table 11 summarizes the speedup gained through parallelization. It is remarkable that the average speedups are always very close to one. The reason is that as the N_t values in Table 9 and Table 10 indicate, single nodes do not have many permanent labels in the solved instances. Hence, the parallel versions of the dominance checks and of *nextCandidateLabel* are rarely called.

6.3.6 Airway Instances

The Airway instances turned out to be easy to solve: the first two cost components, distance and travel time, are correlated [4]. The third cost component, the overflight charges, causes some detours from the time/distance

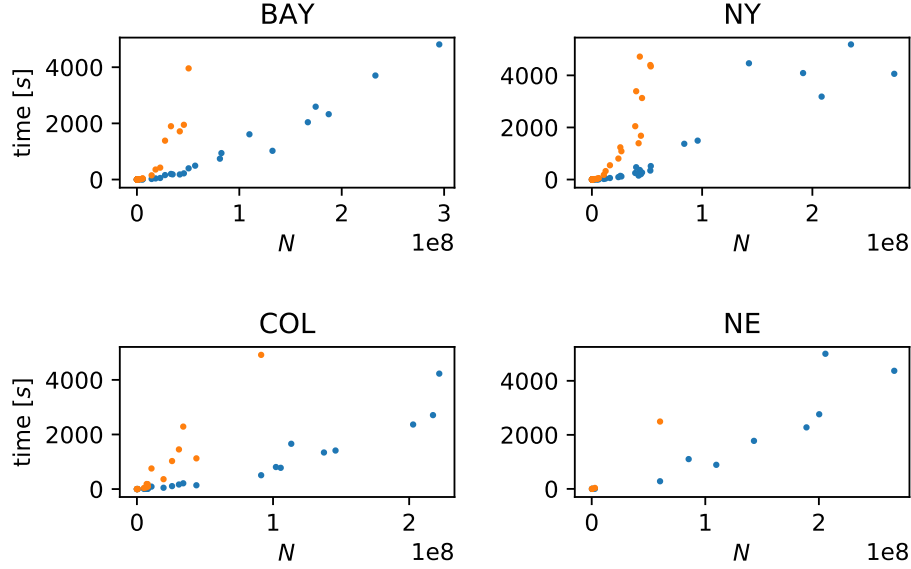


Figure 5: Road solutions depending on N . Orange dots – IMA. Blue dots – MDA.

Table 10: Road Networks Parallel Solvability. 50 instances were considered for every graph. The *thread* columns show how many of them were solved.

	max. N_t	max. N	Threads				
			1	2	3	4	5
BAY	18,465	295,246,523	31	32	33	32	32
COL	13,507	227,698,068	24	24	25	25	25
NE	7,851	280,697,693	11	12	12	11	12
NY	15,665	274,267,247	36	37	37	37	37

Table 11: Parallel speedup on Road Instances. The speedup is built comparing with the running time of the serial version.

		2	Threads		
			3	4	5
BAY	Avg.	1.002	0.996	0.978	0.976
	Min.	0.802	0.78	0.732	0.7
	Max.	1.258	1.26	1.226	1.204
COL	Avg.	1.0	0.987	0.966	0.953
	Min.	0.851	0.801	0.741	0.708
	Max.	1.221	1.288	1.265	1.263
NE	Avg.	1.005	1.002	0.998	0.99
	Min.	0.914	0.826	0.862	0.812
	Max.	1.329	1.242	1.291	1.191
NY	Avg.	0.991	0.97	0.932	0.905
	Min.	0.778	0.764	0.708	0.677
	Max.	1.267	1.235	1.205	1.139

Table 12: IMA vs. serial MDA on the Airway instance set.

		IMA				MDA		Speedup
		N_t	time [s]	N_{ext}	N	time [s]	$(N =)N_{ext}$	
		Solved 101/101				Solved 101/101		
Aviation	Avg.	38	0.026	20,441	17,298	0.0080	17,332	$\times 3.22$
	Min.	1	0.012	3	3	0.0043	3	$\times 2.61$
	Max.	328	1.302	335,464	288,434	0.2213	279,129	$\times 5.88$

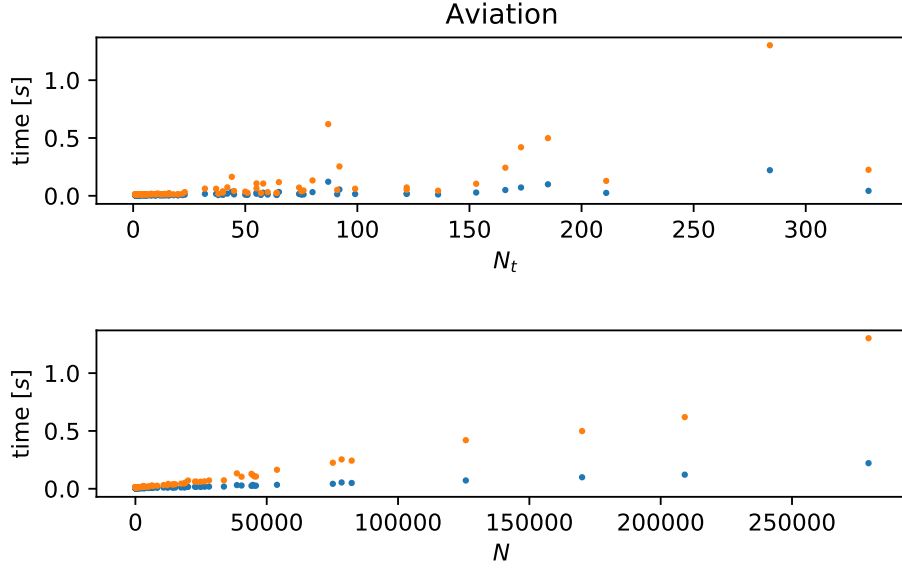


Figure 6: Solution times depending on N_t and N . Orange dots – IMA. Blue dots – MDA

optimal routes but is still correlated with the distance flown over single countries [3]. As a consequence, the N_t ranged between 1 and 328 in our experiments, while N lay between 3 and 279,129 in the MDA solutions. The average solution times were 0.0026s for the IMA and 0.0080 for the MDA causing an average speedup of $\times 3.22$. The details are shown in Table 12. Figure 6 shows solutions times depending on N_t and on N . Figure 7 shows some efficient paths between Tenerife and Berlin.

As a consequence of the low number of labels in the search space and our choice of B and B' , the parallel versions of the dominance checks and of *nextCandidateLabel* are not called while solving Airway instances.

7 Conclusion

We introduced a new algorithm for the Multiobjective Shortest Path Problem which we suggest to call Multiobjective Dijkstra Algorithm (MDA) because throughout the algorithm only the best known unprocessed label for every node is stored in the heap, hence bounding its size by the number of nodes in the graph. Additionally, the new algorithm extracts only non-dominated labels from the heap. These properties allow us to get a running time of $\mathcal{O}(d(N \log(n) + N_{\max}^2 m))$ that to the best of our knowledge is superior to the running time of any MOSP algorithm known so far. We implemented our algorithm and an improved version of the classical label setting MOSP algorithm by Martins [25]. Extensive computational experiments showed a remarkable speedup of the new algorithm both on synthetic (Grids and NetMaker) and real world (Roads and Airway Networks) instances. Additionally, we tried to parallelize the new MDA and analyzed the parallel performance from a

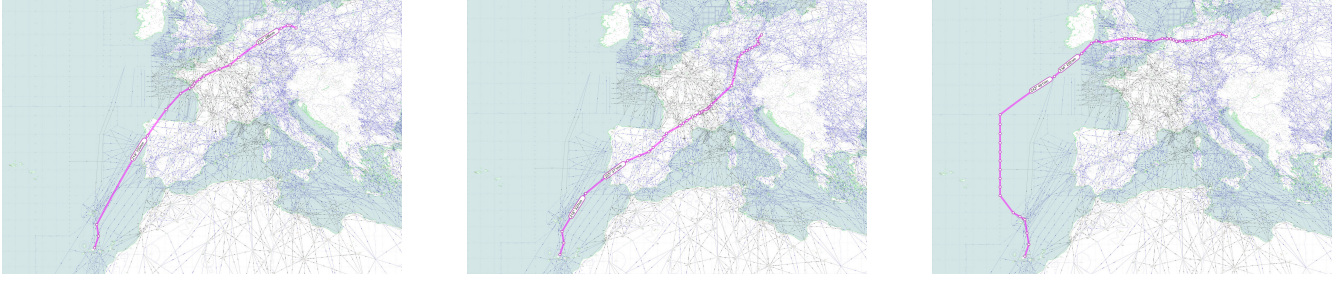


Figure 7: Routes from Tenerife to Berlin. The set of efficient paths at the Berlin node contains 284 solutions. The left figure shows the distance shortest route, the figure in the middle the time optimal route, and the one to the right the cheapest w.r.t to overflight costs.

theoretical and computational point of view. In [31] the authors introduced new parallel data structures that yield a great theoretical running time improvement but did not implement them. We tried to parallelize the dominance checks and label searches but both operations being performed on very efficient C arrays made it difficult to see big improvements. Parallelization of MOSP algorithms remains a challenge. The dominance check between a lexicographically increasing label list L of non-dominated and non-equivalent labels and a label l caused the increased complexity of the multiobjective MOSP compared to the BOSP algorithm presented in [32]. New ideas regarding an efficient characterization of the unsupported non-dominated solutions are needed to reduce the complexity of the $L \preceq_D l$ dominance check we used in our algorithm. All in all, for more than two cost components, the search space size that the new MDA is able to handle and its memory efficiency are beyond the state of the art if we consider exact MOSP algorithms that do not use heavy preprocessing.

Acknowledgements

We thank Lufthansa Systems GmbH & Co. KG for providing us with the data used to define the Airway instances, as well as for the many fruitful discussions. Dr. Andrea Raith provided us the NetMaker and Grid instances, which were also very helpful.

References

- [1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., USA, 1993.
- [2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks, 2015.
- [3] Marco Blanco, Ralf Borndörfer, Nam Dung Hoang, Anton Kaier, Pedro Maristany de las Casas, Thomas Schlechte, and Swen Schlobach. Cost projection methods for the shortest path problem with crossing costs. In Gianlorenzo D’Angelo and Twan Dollevoet, editors, *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59, 2017.
- [4] Marco Blanco, Ralf Borndörfer, Nam-Dung Hoang, Anton Kaier, Adam Schienle, Thomas Schlechte, and Swen Schlobach. Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind. In Marc Goerigk and Renato Werneck, editors, *16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, volume 54 of *OpenAccess Series in Informatics (OASICS)*, pages 12:1–12:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [5] OpenMP Architecture Review Board. Openmp application programming interface 4.5. <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>, 2015. Last accessed August 31, 2020.
- [6] Boost. Boost C++ libraries. https://www.boost.org/doc/libs/1_73_0/doc/html/hash/reference.html#boost.hash_combine, 2020. Last accessed August 31, 2020.

- [7] Thomas Breugem, Twan Dollevoet, and Wilco van den Heuvel. Analysis of FPTASes for the multi-objective shortest path problem. *Computers & Operations Research*, 78:44–58, feb 2017.
- [8] Friedrich Konstantin Bökler. *Output-sensitive complexity of multiobjective combinatorial optimization with an application to the multiobjective shortest path problem*. PhD thesis, Technische Universität Dortmund, 2018.
- [9] M.E. Captivo, J. Clímaco, J. Figueira, E. Martins, and J.L. Santos. Solving bicriteria 0-1 knapsack problems using a labeling algorithm. *Computers and Operations Research*, 30(12):1865–1886, 2003.
- [10] J.C.N. Clímaco and M.M.B. Pascoal. Multicriteria path and tree problems: Discussion on exact algorithms and applications. *International Transactions in Operational Research*, 19(1-2):63–98, 2012.
- [11] J. Current and M. Marsh. Multiobjective transportation network design and routing problems: Taxonomy and annotation. *European Journal of Operational Research*, 65(1):4–19, 1993.
- [12] Pedro Maristany de las Casas, Ralf Borndörfer, Luitgard Kraus, and Antonio Sedeño-Noda. An FPTAS for dynamic multiobjective shortest path problems. *Algorithms*, 14(2):43, jan 2021.
- [13] Sofie Demeyer, Jan Goedgebeur, Pieter Audenaert, Mario Pickavet, and Piet Demeester. Speeding up martins’ algorithm for multiple objective shortest path problems. *4OR*, 11:323–348, 2013.
- [14] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] Daniel Duque, Leonardo Lozano, and Andrés L. Medaglia. An exact method for the biobjective shortest path problem for large-scale road networks. *European Journal of Operational Research*, 242(3):788–797, 2015.
- [16] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 22(4):425–460, 2000.
- [17] Mathias Ehrgott and Xavier Gandibleux. *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*. International Series in Operations Research & Management Science. Springer US, 2006.
- [18] Matthias Ehrgott. *Multicriteria Optimization*. Springer-Verlag, 2005.
- [19] Michael Emmerich and André Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing*, 17, 05 2018.
- [20] Center for Discrete Mathematics and Theoretical Computer Science. 9th dimacs implementation challenge - shortest paths. <http://users.diag.uniroma1.it/challenge9/download.shtml#benchmark>, 2010. Last accessed August 31, 2020.
- [21] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, jul 1987.
- [22] Xavier Gandibleux, Frédéric Beugnie, and Sabine Randriamasy. Martins’ algorithm revisited for multi-objective shortest path problems with a MaxMin cost function. *4OR*, 4(1):47–59, mar 2006.
- [23] GLib. Glib reference manual. <https://developer.gnome.org/glib/stable/glib-Hash-Tables.html>, 2014. Last accessed August 31, 2020.
- [24] Pierre Hansen. Bicriterion path problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application*, pages 109–127, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.
- [25] Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, May 1984.
- [26] Charis Ntakolia and Dimitris K. Iakovidis. A swarm intelligence graph-based pathfinding algorithm (SIGPA) for multi-objective route planning. *Computers & Operations Research*, 133:105358, sep 2021.

- [27] Luigi Di Puglia Pugliese, Janusz Granat, and Francesca Guerriero. Two-phase algorithm for solving the preference-based multicriteria optimal path problem with reference points. *Computers & Operations Research*, 121:104977, sep 2020.
- [28] Francisco Javier Pulido, Lawrence Mandow, and José Luis Pérez de la Cruz. Multiobjective shortest path problems with lexicographic goal-based preferences. *European Journal of Operational Research*, 239(1):89–101, nov 2014.
- [29] Andrea Raith and Matthias Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, apr 2009.
- [30] Andrea Raith, Marie Schmidt, Anita Schöbel, and Lisa Thom. Extensions of labeling algorithms for multi-objective uncertain shortest path problems. *Networks*, 72(1):84–127, mar 2018.
- [31] Peter Sanders and Lawrence Mandow. Parallel label-setting multi-objective shortest path search. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, may 2013.
- [32] Antonio Sedeño-noda and Marcos Colebrook. A biobjective dijkstra algorithm. *European Journal of Operational Research*, 276(1):106–118, jul 2019.
- [33] P. Serafini. Some considerations about computational complexity for multiobjective combinatorial problems. *Recent advances and historical development of vector optimization*, 294:222–232, 1986.
- [34] A.J.V. Skriver. A classification of bicriterion shortest path (bsp) algorithms. *Asia-Pacific Journal of Operational Research*, 17(2):199–212, 2000.
- [35] A.J.V. Skriver and K.A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers and Operations Research*, 27(6):507–524, 2000.
- [36] Z. Tarapata. Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science*, 17(2):269–287, 2007.
- [37] George Tsaggouris and Christos Zaroliagis. Multiobjective optimization: Improved fptas for shortest paths and non-linear objectives with applications. In Tetsuo Asano, editor, *Algorithms and Computation*, pages 389–398, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [38] E.L. Ulungu and J Teghem. Multi-objective shortest path problem: A survey. In D. Gluckaufova, D. Loula, and M. Cerny, editors, *Proceedings of the International Workshop on Multicriteria Decision Making: Methods - Algorithms - Applications at Liblice, Czechoslovakia*. Institute of Economics, Czechoslovak Academy of Sciences, Prague, pages 176–188, 1991.
- [39] P. Vincke. Problemes multicriteres. *Cahiers du Centre d’ Etudes de Recherche Operationelle*, b16:425–439, 1974.