An efficient label-correcting algorithm for the multi-objective shortest path problem

Yannick Kergosien, Antoine Giret, Emmanuel Neron, Gaël Sauvanet

An efficient label-correcting algorithm for the multi-objective shortest path problem

Yannick Kergosien[1], Antoine Giret[1], Emmanuel Néron[1], Gaël Sauvanet[2]

[1] Université de Tours, LIFAT EA 6300, CNRS, ROOT ERL CNRS 7002,
64 avenue Jean Portalis, 37200 Tours

[2] La Compagnie des Mobilités, 30 rue André Theuriet, 37000 Tours

Abstract

This paper proposes an exact algorithm to solve the one-to-one multi-objective shortest path problem. The solution involves determining a set of non-dominated paths between two given nodes in a graph that minimizes several objective functions. This study is motivated by the application of this solution method to determine cycling itineraries. The proposed algorithm improves upon a label-correcting algorithm to rapidly solve the problem on large graphs (i.e., up to millions of nodes and edges). To verify the performance of the proposed algorithm, we use computational experiments to compare it with the best-known methods in the literature. The numerical results confirm the efficiency of the proposed algorithm.

1 Introduction

The shortest path problem is a famous problem in graph theory [16] and was first studied in 1958 [5, 17]. Variants of this problem are still being studied, such as the optimal path problem in stochastic networks [21], the fixed-charge shortest-path problem [22], the time-dependent shortest path problem [54], and the multi-objective shortest path (MOSP) problem. The MOSP problem is to find a set of paths between two given points that minimizes several objective functions. This problem arises in many applications, including the design of transportation networks [13], transportation problems [2], transport risk management [14], tourist trip design [27], satellite scheduling [24], bike tour planning [50], and evacuation planning [44].

We consider a directed graph G composed of a set $\mathcal{V}$ of nodes and a set $\mathcal{A}$ of arcs, a source node s, a destination node t, and a set $\mathcal{K}$ of criteria. Each arc between two nodes i and j is associated with $|\mathcal{K}|$ costs denoted $c_{ij}^k \ \forall \ k \in \mathcal{K}$. We consider the case in which the graph G contains only nonnegative costs ($c_{ij}^k \geq 0$). The MOSP problem consists in determining a set $\mathcal{P}$ of non-dominated paths from node s to node t that minimizes a sum-type objective function for each criterion. In many studies, the graph G corresponds to a road network where an arc represents a road and a node represents a junction. Costs can correspond to distances, travel times, or other criteria, depending on the application. Even if these criteria conflict in theory, they are positively correlated in practice, especially if the graph represents a road network. The exact solution of the MOSP problem gives a set of strictly non-dominated paths, also called the Pareto frontier [61].

The motivation of this study is to determine cycling routes. For ecological reasons, cycling has grown as a suitable alternative transportation mode and, in many countries, public authorities have sought to improve and increase the road network for cyclists. From a cyclist's point of view, the criteria used to design a cycling itinerary include not only distance and travel times, but also other criteria such as safety or height difference. This study was conducted in collaboration with a firm named *La Compagnie des Mobilités*, which develops a product called Geovelo (https://www.geovelo.fr) which is a website and mobile application that proposes appropriate paths for cycling. The proposed paths take into account several criteria such as safety, distance, and physical effort. Finding optimal paths in the shortest possible computational time requires rapidly solving a MOSP problem.

In this paper, we propose an efficient label-correcting algorithm called "Label-Correcting with Dynamic update of Pareto Frontier" (LCDPF), which provides an exact resolution of the MOSP problem. This work is an extension of previous works [50]. The LCDPF algorithm rapidly computes all non-dominated solutions, even for large graphs; its performance is even competitive with that of the best benchmark algorithms. In addition to being simple and flexible, the LCDPF algorithm benefits from state-of-the-art techniques from the related literature and two newly developed techniques, the first of which consists in removing unnecessary nodes from the graph G and the second of which consists in exploiting information collected in the preprocessing phase to quickly update the Pareto frontier during the unidirectional search. This latter technique serves to prune partial paths while searching the space. In contrast, the majority of the benchmark instances in the literature use two criteria: distance and travel time. Given the con-

text of cycling itineraries, we propose herein new instances of the MOSP problem from real road networks with two or three less correlated criteria. The LCDPF algorithm can easily be extended to as many objectives as desired. For example, although the LCDPF algorithm was developed for sum-type objectives, it can be modified for bottleneck objectives such as minimizing the maximum cost in a path, which is equivalent to maximizing the minimum cost in a path [30].

This paper thus makes several contributions: (i) It provides an exact algorithm to solve the MOSP problem based on a classic label-correcting method plus two new techniques (removing unnecessary nodes and quickly updating the Pareto frontier). (ii) It reports the results of a computational experiment that demonstrates the efficiency of the proposed algorithm compared with the best algorithms in the literature and tests new instances inspired by the cycling context with two or three criteria.

This paper is organized as follows: Section 2 introduces classical and recent algorithms for the MOSP problem. Section 3 presents in detail the LCDPF algorithm. Section 4 describes the benchmark instances used for the computational experiments, presents the numerical results of the algorithm for different parameter values, and demonstrates how the number of objectives affects computation time. In addition, computational experiments are used to compare the proposed algorithm with the best-known algorithms from the literature [the bLSET algorithm developed by [47], the pulse algorithm developed by [19], the improved version of the NAMOA* method by [37], and the bi-objective Dijkstra algorithm recently proposed by [51]]. Finally, Section 5 concludes the paper and suggests directions for future work.

2 State of the Art

The MOSP problem is one of the most studied of the multi-objective combinatorial optimization problems [20]. [53] proved that the MOSP problem is $\mathcal{NP}$-complete by reducing the Knapsack problem to a bi-criterion shortest path problem. Later, [30] demonstrated that the MOSP problem is an intractable problem, even for two criteria. In this section, we present the main research works from the MOSP literature, even though the majority of these works address only two objectives. For a more complete review of the topic, we refer the reader to [26] and [57].

The first methods used to solve MOSP problems were the labeling methods proposed by [31] and [39], which were a generalization of existing meth-

ods for solving MOSP problems [6]. Two types of labeling methods exist: label-correcting and label-setting. The main difference between these two methods is the ordering strategy to treat the labels. Label-setting methods are based on an ordering strategy such that, at each iteration, a label is only treated if it is permanent and thus non-dominated until the end of the algorithm. Label-setting methods cannot cannot guarantee polynomial-time convergence on networks with negative edge lengths (but no negative length cycles) [1], whereas label-correcting methods are more general and can use any ordering strategy to explore the labels (e.g., the first-in-first-out rule) [8, 55]. This type of method considers all labels as temporary until the end of the algorithm, at which point they all become permanent. Label-setting algorithms can be considered as special cases of label-correcting algorithms. In labeling methods, two types of selection strategies exist to choose the next label to process: the node-selection strategy, where all labels on a selected node are propagated at the same time, and the label-selection strategy, where labels are separately managed. [29] compare different selection strategies for the label-setting method.

Another type of method to address the MOSP problem includes the ranking methods proposed by [11] for the bi-objective case and by [3] for the multi-objective case. These methods are based on the kth-shortest path problem introduced by [10]. Other studies [55, 12] state that the ranking methods are less competitive than the labeling methods.

A two-phase approach introduced by [42] and [62] is also commonly used to solve MOSP problems. This method determines separately supported solutions (i.e., located on the convex hull boundary) during the first phase, and non-supported solutions during the second phase with, eventually, other supported solutions. The solutions found in the first phase can be determined by using a single objective method based on a weighted sum of objectives. During the next phase, a bi-objective method computes all non-supported solutions. [48] demonstrated the efficiency of this method and compared several strategies for determining all efficient paths. [47] proposed an improved label-setting algorithm called bLSET. According to [15], the bLSET algorithm had the best computational times of the labeling approaches. [15] proposed an accelerated version of Martin's algorithm [39] for MOSP problems. Specifically, they proposed the use of two techniques based on stopping conditions and developed a bidirectional label-setting method. The stopping conditions allow the search to be terminated according to minimum criteria retrieved from the exploration queue and permanent solutions of the Pareto frontier. [25] proposed another bidirectional search to solve multi-objective state space graph problems. The same year, [45] undertook a computational

study of labeling methods for the MOSP problem. They tested 27 variants of the labeling algorithm and considered two labeling techniques (setting or correcting), several data structures for storing the unexplored labels, and several operators for ordering the labels. The performances of the variants were evaluated by applying them to three types of graphs: random, complete, and grid. The results show that, for grid networks, the most efficient label-correcting version uses a first-in-first-out policy to select the labels, whereas the correcting version with a dequeue policy is better for random and complete networks.

[19] designed a new exact method, called the "pulse algorithm," to solve bi-objective shortest path problems on large-size graphs. The pulse algorithm is a recursive method that uses pruning strategies to accelerate path exploration. The experiments demonstrate that the pulse algorithm outperforms the bLSET algorithm on extremely-large-size graphs from the DIMACS dataset. The pulse algorithm may be thought of as a general framework extended to solve other shortest path problems, such as the constrained shortest path problem [35] or the elementary shortest path problem with resource constraints [34]. [52] proposed a generalization of the Dijkstra method to find all extreme supported solutions of a bi-objective shortest path problem in one-to-one or one-to-all versions. They demonstrated that the execution times of the method are $\mathcal{O}(N[m + p\log(p)])$ where p is the number of nodes, m is the number of arcs, and N is the number of extreme supported points in the solution space. [37] proposed a method to compute lower bounds for the bi-objective shortest path problem. These lower bounds are used to improve the NAMOA* method [38], which is a generalization of the A* algorithm to solve multi-objective problems. NAMOA* is a best-first search algorithm for solving the MOSP problem.

Recently, for the bi-objective case, [51] proposed a generalization of Dijkstra's algorithm that uses pruning strategies and a bidirectional search. They first considered the one-to-all bi-objective shortest path problem for which time and space complexities are given, following which they developed a fast algorithm to solve the one-to-one version. Their proposed algorithm outperforms one of the state-of-the-art algorithms to solve the bi-objective shortest path in large road networks (i.e., the pulse algorithm).

Numerous problems derive from the MOSP problem, such as the multi-objective uncertain shortest path problems or the resource-constrained shortest path problems. [49] studied a MOSP problem with uncertain edge lengths and, to solve the problem, proposed two main approaches based on the label-correcting algorithm to find robust solutions that exploit several concepts of robust efficiency, namely, multi-scenario efficiency, flimsily

and highly robust efficiency, and point-based and set-based min-max robust efficiency. Experiments on two types of networks (grid and NetMaker) demonstrate the performance of the proposed algorithms. The resource-constrained shortest path (RCSP) problem is similar to the MOSP problem in which the only objective is to minimize (or maximize) subject to additional constraints that bound the value of other objectives. Thus, the optimal solution of a RCSP problem belongs to the set of all non-dominated solutions of the related MOSP problem. This difference implies that some optimization techniques for solving RCSP problems may become useless for solving MOSP problems (or vice versa). [18] studied the effectiveness of several techniques integrated within a classic label-setting algorithm. Among the techniques proposed, one is a preprocessing technique that aims to reduce the graph based on the resolution of several one-to-all mono-criterion shortest path problem. From randomly generated and real network data, they provided a systematic comparison of the performance of the different algorithms. To have an overview of RCSP solution methods, we refer the reader to [33]. A classification and a generic formulation for the RCSP is proposed as well as commonly used RCSP solution methods, such as the label correcting algorithm. [58] proposed a exact bidirectional A* algorithm, called "RC-BDA", to solve RCSP problems. They demonstrate that, although bidirectional A* approaches perform poorly in most studies in terms of computation speed, they exploit the resource constraints to produce an efficient bidirectional A* algorithm. [58] also analyzed computationally and theoretically the sensitivity of the algorithm's performance. In the same year, [46] studied a generalization of the resource constrained shortest path problem where the resources are taken in a monoid. For example, this includes stochastic and non-linear resource constrained shortest path problems. He proposed a generic solution method with several bounding algorithms allowing to discard partial paths during the exploration phase of the solution method. Computational experiments shown that the proposed method outperform existing ones on non-linear pricing subproblem of a column generation approach to an airline operations problem.

Recently, [9] proposed a bidirectional pulse algorithm to exactly solve a RCSP problem. This algorithm is based on the pulse algorithm and a bidirectional search executed in parallel. The algorithm has two parameters: the maximum number of labels to store at each node and the maximum depth for which the search type may still be adjusted (i.e., to achieve a balance between breadth and depth searches). They also undertook a sensitivity analysis to understand the algorithm components and performance. Computational experiments involving large-scale road networks are done in the

United States and the results are compared to those of [58]. The proposed bidirectional pulse algorithm is four times faster on average, although the RC-BDA algorithm solved a few more instances.

3 Label Correction with Dynamic update of Pareto Frontier Algorithm

The LCDPF algorithm is composed of two phases: The first phase, called the "preprocessing phase," serves to determine supported solutions by solving several mono-objective shortest path problems. This phase determines upper and lower bounds of the final Pareto frontier at each node of the graph, making this information available for the next phase. The preprocessing phase also determines whether a node is unnecessary to find the Pareto frontier. Every unnecessary node is removed from the graph, which may accelerate the next phase of the algorithm. The second phase consists in determining the Pareto frontier (i.e., finding all the non-dominated solutions) by using the information obtained in the first phase. To accomplish this, the LCDPF algorithm is based on a label-correcting algorithm into which we integrated additional improvements to increase efficiency. Section 3.1 presents the second phase of the LCDPF algorithm, called the "label-correcting phase," following which the details of the preprocessing phase are given in Section 3.2. Section 3.3 uses an example to illustrate both phases of the algorithm. Finally, Section 3.4 discusses the complexity of the LCDPF algorithm.

3.1 Label-Correcting Phase

The LCDPF algorithm is based on a label-correcting algorithm introduced by [39]. The goal of the algorithm is to determine all non-dominated paths from source node s to destination node t (i.e., the final Pareto frontier). The LCDPF algorithm uses the following notation:

- l_u is a label representing a feasible partial path from node s to a node of the graph. A label is defined by a tuple $(i, \mathcal{U}, l_p)$, where
  - i is the end node of the partial path.
  - $\mathcal{U}$ is the vector containing the $|\mathcal{K}|$ criteria of the feasible path from s to i. u_k is the value of criterion k in $\mathcal{U}$.

  - l_p is the predecessor node label from which l_u is determined (only a pointer to l_p is needed). This information allows us to determine the corresponding path from s to i.

- $l_v \prec l_u$ means that the label l_u is strictly dominated by l_v in the Pareto sense.

- LN_i is the set of labels at node i.

- Q is the set of labels that represents the label-correcting exploration queue, which contains unexplored labels.

- Y^{max} is a parameter that defines when the dynamic update of the current Pareto frontier is to be applied.

The LCDPF algorithm also uses results from preprocessing phase, which are listed below:

- A vector of $|\mathcal{K}|$ lower bounds denoted $LB^i = (lb^i_1, \ldots, lb^i_{|K|})$ that gives a lower bound on each criterion of any path from i to t. It is therefore impossible to determine a path from i to t with a value less than lb^i_k for every criterion k.

- A set of feasible paths exists from node i to t. We denote $\mathcal{V}^n_i = (v^n_{i1}, \ldots, v^n_{i|\mathcal{K}|})$ as the vector of $|\mathcal{K}|$ criteria associated with the nth feasible path from node i to t.

The LCDPF algorithm is detailed in Algorithm 1. At the end of the LCDPF algorithm, the final Pareto frontier is given by LN_t. Algorithm 1 lists the five main steps of the LCDPF algorithm.

- **Initialization step.** This step (lines 2 to 4) consists in running the preprocessing phase, initializing as an empty set the list of labels associated with each node, and initializing the exploration queue and LN_s to the same first label. The goal of the preprocessing phase, described in Section 3.2.1, is to remove unnecessary nodes from the graph, initialize the Pareto frontier (LN_t) with feasible initial paths, compute the lower bounds (LB_i), and determine a set of feasible paths from node i to node t ($\mathcal{V}^n_i$).

- **Select the next label to propagate.** In each main iteration of the algorithm, labels at a given node i are propagated through the edges from i until no additional labels remain to consider in Q (line

Algorithm 1 LCDPF algorithm

function LCDPF($G = (\mathcal{V}, \mathcal{A})$, s, t, Y^{max})
  Run the preprocessing phase, which computes the values of LB^i and $\mathcal{V}_i^n$, initializes LN_t, and removes unnecessary nodes from G.
  Initialize LN_i to $\emptyset$, $\forall\ i \in \mathcal{V},\ i \neq t$.
  Initialize LN_s and Q to $(s, \{0, 0, \ldots, 0\}, -)$.
  while $Q \neq \emptyset$ **do**
    $l_u = (i, \mathcal{U}, l_p) \leftarrow$ GET THE NEXT LABEL TO TREAT(Q)
    Remove l_u from Q .
    for all nodes j such that $(i, j) \in \mathcal{A}$ **do**
      Create a new label $l_{u'} = (j, \{u_1 + c_{ij}^1, u_2 + c_{ij}^2, \ldots, u_{|\mathcal{K}|} + c_{ij}^{|\mathcal{K}|}\}, l_u)$ at node j from l_u.
      if $l_{u'}$ is not dominated by a label in LN_j (i.e., $\nexists\ l_v \in LN_j / l_v \prec l_{u'}$) **then**
        Create a new label $l_{lb} = (t, \{u'_1 + lb_1^j, u'_2 + lb_2^j, \ldots, u'_{|\mathcal{K}|} + lb_{|\mathcal{K}|}^j\}, -)$.
        if l_{lb} is not dominated by a label in LN_t (i.e., $\nexists\ l_v \in LN_t / l_v \prec l_{lb}$) **then**
          Add $l_{u'}$ to LN_j.
          Remove all labels from LN_j that are dominated by $l_{u'}$; let S be the set of removed labels.
          Remove all labels in S from Q.
          if $j \neq t$ **then**
            Add $l_{u'}$ to Q.
            every Y^{max} iterations of propagation of a label **do**
              $LN_t \leftarrow$ UPDATE THE CURRENT PARETO FRONTIER(LN_t,$l_{u'}$)
            end every
          end if
        end if
      end if
    end for
  end while
  return LN_t
end function

5). The exploration order is given by the GET THE NEXT LABEL TO TREAT function, which selects the next label l_u from Q. Several strategies to select the next most pertinent label are available in the literature; for example, [7] use a best-promise exploration order as an acceleration strategy for the weight-constrained shortest path problem with replenishment. The majority of these strategies can be generalized as follows: the next label $l_u = (i, \mathcal{U}, -)$ to explore minimizes the objective function

$$\min \sum_{k \in \mathcal{K}} [u_k + h(i,k)].\alpha_k$$

where

- $h(i,k)$ is a function that computes or estimates the cost of the shortest path from i to t considering criterion k;
- α_k is the weight associated with criterion k.

Both parameters define the strategy to select the next label to consider. For example, if $\alpha_1 = 1$, $\alpha_k = 0 \ \forall \ k \in \mathcal{K}\backslash\{1\}$, and $h(i,k) = 0 \ \forall \ i \in G \ \forall \ k \in \mathcal{K}$, then the labels in Q are sorted in increasing order of the first criterion u_1. As in the A* algorithm introduced by [32], the search is accelerated by implementing an effective function $h(i,k)$. The number of labels explored during the LCDPF algorithm (Algorithm 1) is related to the order in which they are selected. In our case, a good function to use is $h(i,k) = lb_k^i$, which corresponds to the lower bounds computed in the preprocessing phase. The earlier that promising labels are explored, the more accurate is the approximation of the Pareto frontier. Several strategies were tested, and the results are presented in Section 4, Computational Experiments.

- **Create and evaluate new labels.** From node i and label l_u, a new label $l_{u'}$ is created for each successor of i (lines 8 and 9 of Algorithm 1). First, the new label $l_{u'}$ is considered for the next steps only if it is not dominated by a label in LN_j (line 10). Otherwise, this label $l_{u'}$ would lead to a dominated path in LN_t. Next, lines 11 and 12 verify if label $l_{u'}$ is promising by using the lower bounds at node j. $l_{u'}$ represents a partial path P_j from node s to j. An ideal and dummy path from node s to node t, represented by label l_{lb}, is deduced such that the value of criterion k is given by $u'_k + lb_k^j$, where u'_k is the value of the partial path P_j on criterion k. If l_{lb} is dominated by at least one label in LN_t, then the ideal path is dominated by a solution of the current Pareto frontier,

and the partial path P_j is of no interest, so its label $l_{u'}$ can be pruned. This pruning method is commonly used in the literature; it was first introduced by [60] and later extended by [19]. The closer the current Pareto frontier is to the final Pareto frontier, the greater is the number of pruned labels, which will accelerate the search. For this reason, the last step, "update the current Pareto frontier," is important. Figure 1 illustrates a case where a label associated with a path P_j should be pruned ($|\mathcal{K}| = 2$).

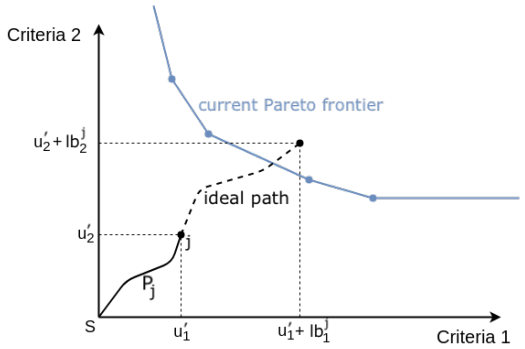

Figure 1: Pruning label.

- **Add new labels to the list of labels associated with a node and update this list.** Each promising label $l_{u'}$ is added to the list of labels LN_j associated with a node j (line 13). This label $l_{u'}$ can dominate one or several labels in LN_j, which should therefore be removed from LN_t (line 14). If S is the set of removed labels, every label in $\mathcal{S}$ that is no longer interesting can remain in the exploration queue, so the labels are also removed from the exploration queue (line 15). Finally, if j is not the destination node, the label $l_{u'}$ is added to the exploration queue Q (line 17).

- **Update the current Pareto frontier.** This step (line 5) refers to the dynamic update of the Pareto frontier during the search, which consists in trying to improve the current Pareto frontier by using the current label $l_{u'}$ and upper bounds at node j ($\mathcal{V}_j^n$). This step aims at accelerating the convergence of the current Pareto frontier to the final Pareto frontier throughout the search, which is a time-consuming procedure and does not necessarily improve the current Pareto frontier; for this reason, the procedure is applied only every Y^{max} iterations

of the propagation of a label. The procedure called UPDATE THE CURRENT PARETO FRONTIER is described by Algorithm 2. For each feasible path n from j to t computed during the preprocessing phase and given by $\mathcal{V}_j^n = (v_{j1}^n, \ldots, v_{j|\mathcal{K}|}^n)$, a feasible path from s to t represented by label l_{ub} is deduced by using $l_{u'}$ so that criterion k is given by $u'_k + v_{jk}^n$. If no label in LN_t dominates l_{ub}, each new path given by l_{ub} is added to the current Pareto frontier. If a label l_{ub} is added to LN_t, then every path in LN_t dominated by l_{ub} is removed. Figure 2 presents the case where a path P_j given by $l_{u'}$ allows us to improve the current Pareto frontier by adding a new solution and removing two others ($|\mathcal{K}| = 2$).

Algorithm 2

function UPDATE THE CURRENT PARETO FRONTIER (LN_t, $l_{u'} = (j, \mathcal{U}', l_u)$)
  for all feasible paths from j to t given by $\mathcal{V}_j^n$ **do**
    Create a new label $l_{ub} = (t, \{u'_1 + v_{j1}^n, u'_2 + v_{j2}^n, \ldots, u'_{|\mathcal{K}|} + v_{j|\mathcal{K}|}^n\}, -)$.
    if l_{ub} is not dominated by any label in LN_t (i.e., $\nexists l_v \in LN_t / l_v \prec l_{ub}$) **then**
      Add l_{ub} to LN_t.
      Remove all labels from LN_t that are dominated by l_{ub}.
    end if
  end for
end function

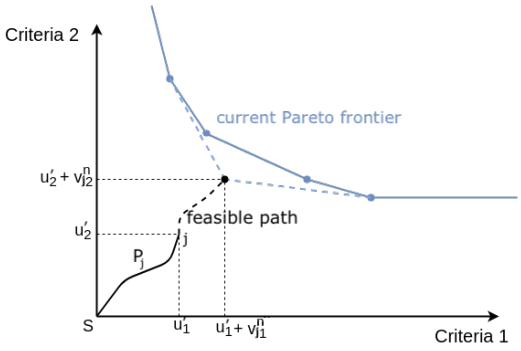

Figure 2: Update of Pareto frontier.

This step uses a concept similar to that used for a bidirectional search [15, 51] but in a different way. The common concept involves combining two labels associated with a node to find a feasible path to update the Pareto frontier. However, in the bidirectional algorithm, this combination is executed during the exploration step (i.e., while propagating the labels in both directions, backward and forward) using the propagated labels. In contrast, we propose to combine the propagated labels with labels that have been computed in the preprocessing phase.

3.2 Preprocessing phase

The goal of the preprocessing phase is to remove unnecessary nodes from the graph, initialize the Pareto frontier LN_t with initial feasible paths, compute the lower bounds LB_i, and determine a set of feasible paths from node i to t ($\mathcal{V}_i^n$). This phase is divided into two steps: The first step initializes LN_t and starts to compute the lower bounds LB_i and some feasible paths from node i to t. The second step finishes determining the lower bounds LB_i and the feasible paths from node i to t, and then deduces the unnecessary nodes to remove from the graph. Both steps use a set of the solution to the mono-objective shortest path problem solved by Dijkstra's algorithm [17].

3.2.1 First step

Let the parameter $\mathcal{W}$ be a set of tuples that contains weights associated with each criterion, and let w_k^n be the weight of the nth tuple of $\mathcal{W}$ associated with criterion k such that

$$\sum_{k \in \mathcal{K}} w_k^n = 1 \ \forall \ n \in \{1, \dots, |\mathcal{W}|\}.$$

We suppose that the first $|\mathcal{K}|$ tuples in $\mathcal{W}$ are defined such that, $\forall\ n \in \{1, \dots, |\mathcal{K}|\}$, we have $w_n^n = 1$ and $\forall\ k \in \mathcal{K} \backslash n,\ w_k^n = 0$ [e.g., $|\mathcal{K}| = 3$ and $\mathcal{W} = \{(\infty, \prime, \prime), (\prime, \infty, \prime), (\prime, \prime, \infty), (\prime.\ni\ni, \prime.\ni\ni, \prime.\ni\triangle), \dots\}$]. The first step of the preprocessing phase consists in solving $|\mathcal{W}|$ mono-objective shortest path problems from the destination node t to the source node s, where every arc in the graph is reversed. We refer to this type of problem as a reversed mono-objective shortest path problem. The nth reversed shortest path problem minimizes a linear combination of criteria by using the weights of the nth tuple in $\mathcal{W}$. For all $n \in \{1, \dots, |\mathcal{K}|\}$, the nth problem is equivalent to a

mono-criterion shortest path problem considering only the criterion $k = n$. In this case, the mono-criterion shortest path problem given criterion k is solved by using a lexicographic optimization strategy [20] in which criterion k is first optimized, and then the remaining criteria are optimized in the same arbitrary order. The goal of this approach is to consider only non-dominated paths at each node from among the paths with the same optimal objective value for criterion k.

After solving the $|\mathcal{W}|$ mono-objective shortest path problems from node t to node s and using Dijkstra's algorithm, we retrieve all feasible paths from node i to node t that were explored in the resolution of each $n \in \mathcal{W}$ mono-objective shortest path problem. These feasible paths correspond to $\mathcal{V}_i^n = (v_{i1}^n, \ldots, v_{i|\mathcal{K}|}^n)$ and are determined from a linear combination of criteria using the nth tuple of weights. We obtain the following information from these feasible paths:

- For the MOSP problem, we obtain $|\mathcal{W}|$ supported solutions given by $\mathcal{V}_s^n \ \forall \ n \in \mathcal{W}$ based on Geoffrion's theorem [28] [see Section 3.8.2 in [59]]. $\mathcal{V}_s^n = (v_{s1}^n, \ldots, v_{s|\mathcal{K}|}^n)$ corresponds to the optimal path between node s and node t that minimizes a linear combination of criteria using the nth tuple of weights. All these supported solutions belong to the final Pareto frontier and are used to initialize the Pareto frontier (LN_t). Initializing the Pareto frontier with non-dominated solutions increases the amount of pruning and thus accelerates the LCDPF algorithm.

- Lower bounds $LB_i = (lb_1^i, \ldots, lb_{|K|}^i)$ such that $lb_k^i = v_{ik}^k \ \forall \ k \in \mathcal{K}$ because every v_{ik}^k value is determined by Dijkstra's algorithm to solve the mono-criterion shortest path problem for criterion k.

- Upper bounds $\mathcal{V}_i^n$ correspond to non-dominated paths from node i to node t and serve to update the current Pareto frontier during the search.

Note that, given the stopping criterion of Dijkstra's algorithm, not all $\mathcal{V}_i^n$ are determined for every node i in the graph (and, therefore, not all LB_i are determined). Thus, enumerating all non-dominated paths $\mathcal{V}_i^n \ \forall \ n \in \{1, \ldots, |\mathcal{W}|\}$ for all nodes i in the graph is not necessary and can be time-consuming; however, enumerating them for a subset of pertinent nodes may be worthwhile, where "pertinent nodes" refers to nodes that will be strongly involved during the exploration phase (propagation of partial paths). The second step of the preprocessing phase aims at computing other partial paths from pertinent nodes to node t.

3.2.2 Second step

To find additional non-dominated paths from pertinent node i to node t (and thereby determine additional $\mathcal{V}_i^n$), we solve a reversed one-to-all mono-objective shortest path problem (from t to all nodes) for each tuple in $\mathcal{W}$. To solve this problem, we use the same Dijkstra algorithm as that used at the previous step but change the stopping criterion from "node s is reached" to "all pertinent nodes are reached." To estimate this set of pertinent nodes, the stopping criterion of the reversed one-to-all mono-objective shortest path problem is obtained as follows: We define the worst dummy path such that

$$\mathcal{V}_s^{\text{worst}} = (\ v_{s1}^{\text{worst}} = \max_{\forall\ n \in \mathcal{W}}(v_{s1}^n),\ \ldots,\ v_{s|\mathcal{K}|}^{\text{worst}} = \max_{\forall\ n \in \mathcal{W}}(v_{s|\mathcal{K}|}^n)).$$

The nth iteration of Dijkstra's algorithm is stopped when the next node i to visit is associated with $\mathcal{V}_i^n = (v_{i1}^n, \ldots, v_{i|\mathcal{K}|}^n)$ such that $v_{ik}^n \geq v_{sk}^{\text{worst}}\ \forall\ k \in \mathcal{K}$. Note that $\mathcal{V}_s^{\text{worst}}$ is determined after the first step of the preprocessing phase. The use of this stopping criterion implies that not all nodes are explored during each resolution of the one-to-all mono-objective shortest path problem (only one node is explored for each tuple in $\mathcal{W}$). The result is the same when determining LB^i by solving the one-to-all mono-objective shortest path problem only for the first $|\mathcal{K}|$ tuples in $\mathcal{W}$. This information allows us to determine the nodes that are "useless" for the resolution of the multi-objective shortest path problem.

Lemma 1 *When determining LB^i, if node i is explored at most once during the resolution of the $|\mathcal{K}|$ one-to-all mono-criterion shortest path problem, then any path through node i computed during the exploration phase is dominated by a solution of the initial Pareto frontier computed after the preprocessing phase. Thus, node i can be removed from graph G after the preprocessing phase.*

The proof of the Lemma 1 is available in Section 1 of online supplement. In addition to finding other partial paths from pertinent nodes to node t, the goal of the second step is to delete other nodes from the graph by using Lemma 1. The size of $\mathcal{W}$ and its values are parameters of the LCDPF algorithm and determine the number of initial non-dominated solutions in the Pareto frontier and the number of solutions for a mono-objective shortest path problem. An appropriate size for $\mathcal{W}$ is a trade-off between the time required for the preprocessing phase and the contribution of the initial solutions to reducing the exploration of the graph during the exploration phase.

3.3 Example

Figures 3 and 4 show an example of an instance with two criteria and $\mathcal{W} = \{(1,0),(0,1),(0.5,0.5)\}$. The graph of Figure 3 is composed of 10 nodes and 15 arcs. Figure 4 overviews the results after the first step of the preprocessing phase. The Pareto frontier LN_t is initialized with the supported solution $(2,8)$, $(8,2)$, and $(4,5)$. Each value of $\mathcal{V}_i^n$ is noted on the graph. For some nodes (e.g., nodes 1, 3, 6, and 8), not all $\mathcal{V}_i^n$ are determined because of the stopping criterion of Dijkstra's algorithm used to solve mono-criterion shortest path problems.

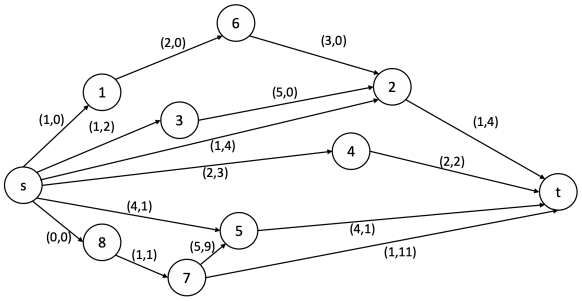

Figure 3: Graph example.

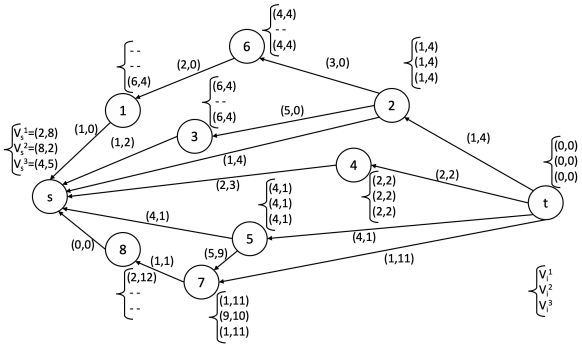

Figure 4: Results of first step of preprocessing phase.

Figure 5 (in gray) shows the results of the second step of the preprocessing phase. $\mathcal{V}_i^n$ are determined for all nodes except node 8, where $\mathcal{V}_8^2$ is not determined. Based on Lemma 1, this node can be removed from the graph because it has been explored only once during the resolution of the first two one-to-all mono-criterion shortest path problems.

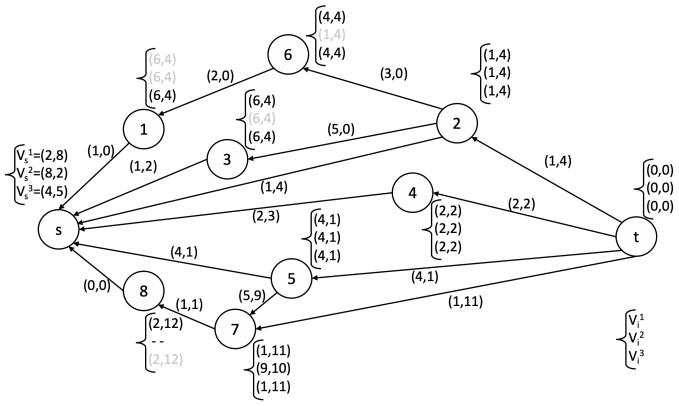

Figure 5: Results of second step of preprocessing phase.

Figure 6 shows the results of the first iteration of the LCDPF algorithm. Label l_0 is propagated and creates five new labels on nodes 1–5. After creating the label $l_1 = (1, 0)$ on node 1, the current Pareto frontier is updated by adding a new non-dominated solution $(7, 4)$ owing to label l_1 and the values of $\mathcal{V}_1^1$. Label $l_2 = (1, 2)$ created at node 3 is not added in LN_3 because l_2 is not a promising label: the dummy path computed from l_2 and lower bounds $v_{3,1}^1 = lb_1^3$ and $v_{3,2}^2 = lb_2^3$ is dominated by at least one solution in the current Pareto frontier [e.g., $(4, 5)$].

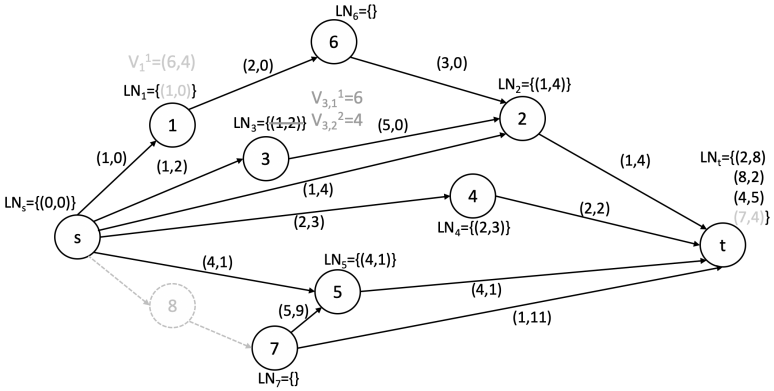

Figure 6: Results of first iteration of LCDPF algorithm.

3.4 Time Complexity of LCDPF Algorithm

The complexity of the preprocessing phase is polynomial because it consists in solving $|\mathcal{K}|$ mono-objective shortest path problems, adding non-dominated solutions to the initial Pareto frontier, and removing some nodes from the graph. Each mono-objective shortest path problem is solved by applying Dijkstra's algorithm with a specific stopping criterion to avoid exploring the whole graph. The worst case corresponds to exploring the entire graph for each iteration of Dijkstra's algorithm (note that, in this case, no node is removed from the graph). When using a classic implementation of Dijkstra's algorithm, the complexity of the preprocessing phase has the computational complexity $\mathcal{O}(|\mathcal{K}|(|\mathcal{A}| + |\mathcal{V}|) \log |\mathcal{V}|)$.

The complexity of the exploration phase depends on the number of labels to explore, which, theoretically, may be an exponential number. If no technique improves the pruning of labels during the search, the algorithm will enumerate all labels (i.e., feasible paths) that are not dominated at a given node. Only labels dominated by another label are pruned, so the algorithm finds the optimal Pareto frontier when the exploration queue is empty.

A key element of the LCDPF algorithm is the choice of data structure and, more specifically, the structure of the list of labels LN_j and the exploration queue Q. To decrease the complexity of the algorithm, a sagacious choice is to use the same data structure, such as a multi-index container as implemented in the Boost Multi-Index Containers Library [43]. This structure saves the data with different sorting and access semantics; in our case, one for each criterion, one for the node index, and one for the exploration strategy $h(i, k)$. The complexity to insert, delete, locate, and modify an element in this structure of p elements is $\mathcal{O}(\log p)$, $\mathcal{O}(1)$, $\mathcal{O}(\log p)$, and $\mathcal{O}(\log p)$, respectively. The parameter p corresponds to $|LN_j|$ for the structure of the label list at each node j and to $|Q|$ for the structure of the exploration queue Q. Both depend on the number of explored paths.

This type of structure allows the GET NEXT LABEL TO TREAT() procedure to have complexity $\mathcal{O}(\log |Q|)$. The complexity of lines 10 and 12 is related to the operation that checks a label l_v in both LN_j and LN_t that dominates the labels $l_{u'}$ and l_{lb}, respectively. This operation can be done with complexities $\mathcal{O}(|K||LN_j|)$ and $\mathcal{O}(|K||LN_t|)$ for LN_j and LN_t, respectively. Line 14, which removes the dominated labels, has complexity $\mathcal{O}(|K||LN_j|)$. In addition, the procedure UPDATE THE CURRENT PARETO FRONTIER has complexity $\mathcal{O}(|\mathcal{W}||K||LN_t|)$ because, for each tuple n in $\mathcal{W}$, it checks if a label l_v exists in LN_t that dominates the label l_{ub}.

To conclude, for each iteration of the main loop of the algorithm (one iteration for each label to explore in Q), the worst-case scenario corresponds to the creation of new labels (one label per successor node of the current label), where each new label is promising and is used to update the Pareto frontier. Therefore, the worst-case complexity of each iteration of the main loop of the LCDPF algorithm is $\mathcal{O}(\Delta_G(|K||LN_j| + |\mathcal{W}||K||LN_t|))$, where Δ_G is the maximum degree of graph G.

4 Computational Experiments

This section presents the computational experiments used to test the LCDPF algorithm. First, Section 4.1 introduces two types of instances used to test the efficiency of the proposed algorithm. Section 4.2 shows how the label-selection strategy affects the results of the LCDPF algorithm and the choice of the set $\mathcal{W}$ of weights in mono-criterion searches that serve to initialize and update the Pareto frontier. Section 4.3 shows how the number of criteria affects the LCDPF algorithm. Finally, Section 4.4 compares the LCDPF algorithm with the best methods available in the literature. The LCDPF algorithm was implemented in C++ using the Boost Graph Library. The computational experiments were done on a Linux system equipped with an Intel Xenon W3520 2.67 GHz CPU, eight cores, and 24 GB of RAM. Each experiment used only one core.

4.1 Benchmark instances

Two types of instances were used. The first type is a classic set of instances from the literature on the MOSP problem. The second type is a set of instances generated by a real application for cycling itineraries.

4.1.1 DIMACS instances

The first type of instances are the DIMACS instances from the 9^{th} DIMACS challenge. Table 1 lists the characteristics of the eight graphs used for the computational experiments. The criteria of these graphs are the physical distance and travel time.

Name	Description	Number of nodes	Number of arcs
NY	New York City	264 346	733 846
BAY	San Francisco Bay Area	321 270	800 172
COL	Colorado	435 666	1 057 066
FLA	Florida	1 070 376	2 712 798
NW	Northwest USA	1 207 945	2 840 208
NE	Northeast USA	1 524 453	3 897 636
CAL	California and Nevada	1 890 815	4 657 742
LKS	Great Lakes	2 758 119	6 885 658

Table 1: Characteristics of DIMACS instances.

4.1.2 Geovelo instances

To emulate more realistically the real application (i.e., cycling itineraries), we used actual road networks based on data from OpenStreetMap (https://www.openstreetmap.org) to build a new set of instances. In this network, a node corresponds to a junction and an arc between two junctions is associated with a set of values (i.e., the travel distance, road security, and other information regarding the type of road). To determine the security cost associated with an arc, each road was graded between zero (maximal safety) and five (minimal safety). The security cost is given by the product of the security grade and the arc length. The three graphs extracted are for Paris (France), Berlin (Germany), and the San Francisco Bay Area (USA). The graph of Paris contains 29 086 nodes and 64 538 arcs. The graph of Berlin is interesting because Germany has an active OpenStreetMap community, so safety information is accurately reported; the Berlin graph contains 59 673 nodes and 145 840 arcs. The San Francisco graph (SF hereinafter), which is also interesting because of the road network structure (in fact, urban road networks in the USA are rather larger than those in Europe), contains 174 975 nodes and 435 959 arcs. This set of instances is noted as OSM graphs. For each graph, we generated 60 random instances (i.e., pairs of origin-destination nodes), which were ordered by increasing number of non-dominated solutions in the final Pareto frontier and were grouped into three sets of identical size: P1, P2, P3 for Paris, B1, B2, B3 for Berlin, and SF1, SF2, SF3 for San Francisco. The data are available on the VRP-REP open-data platform [40] at http://www.vrp-rep.org/references/item/kergosien-et-al-2021.html.

4.2 Comparison of label-selection strategies and parameter setting $\mathcal{W}$

The choice of the label-selection strategy, described in Section 3.1, is important and affects the performances of the LCDPF algorithm. Specifically, it determines in what order the labels are considered and therefore influences the number of explored labels and the computation time. To test several strategies, we used the OSM graphs just described with two criteria (distance and security).

Let α_1 be the weight associated with the distance criterion and α_2 be the weight associated with the security criterion. Three strategies were tested:

- The first label-selection strategy (STR1) selects the label with the shorter distance, as in the classic Dijkstra method. For each node i and criterion k, the cost of the shortest path from i to t is not considered, so $h(i,k) = 0 \ \forall \ i \in \mathcal{V}, k \in \{1,2\}$. Furthermore, to only consider distance, we set $\alpha_1 = 1$ and $\alpha_2 = 0$.

- For the distance criterion, the second label-selection strategy (STR2) selects the label with the smallest sum of distance from the source node associated with label u_1 and the shortest path distance from i to t. This method is similar to an A* method. In this strategy, the cost $h(i,1)$ represents the shortest distance between nodes i and t (v_{i1}^1) computed in the second step of the preprocessing phase (see Section 3.2.2). The security cost of the shortest path from i to t is not considered, so $h(i,2) = 0 \ \forall \ i \in \mathcal{V}$. Furthermore, to consider only distance, we set $\alpha_1 = 1$ and $\alpha_2 = 0$.

- The third label-selection strategy (STR3) selects the label with the smaller linear combination of distance and insecurity for the labels u_1 and u_2, with equal weights $\alpha_1 = 0.5$ and $\alpha_2 = 0.5$. For each node i and criterion k, the cost of the shortest path from i to t is not considered, so $h(i,k) = 0 \ \forall \ i \in \mathcal{V}, \ k \in \{1,2\}$.

Other strategies based on the insecurity criterion instead of the distance criterion were tested; however, the results were either similar or worse than the results of these three strategies.

In the first phase of the LCDPF algorithm, $|\mathcal{W}|$ mono-objective shortest path problems are solved (cf. Section 3.2). The objective function of each problem is a weighted linear combination of all criteria for which the weights are contained in $\mathcal{W}$. The results allow us to initialize the Pareto frontier, to compute lower bounds and upper bounds, and to deduce the nodes to be

removed from the graph. The upper bounds are then used during the exploration phase to update the current Pareto frontier (Algorithm 2). However, to update the current Pareto frontier, we consider only a subset $\mathcal{W}' \subseteq \mathcal{W}$ to evaluate the performance of this procedure (i.e., only $\mathcal{V}_j^n \ \forall \ n \in \mathcal{W}'$ are used in line 2 of Algorithm 2). For each label-selection strategy, we test different values of $\mathcal{W}$ and $\mathcal{W}'$. Let us define the following:

- $\mathcal{W}_1 = \mathcal{W}'_1 \equiv \{(1,0),(0,1)\}$;
- $\mathcal{W}_2 = \mathcal{W}'_2 \equiv \{(1,0),(0,1),(0.5,0.5)\}$;
- $\mathcal{W}_3 = \mathcal{W}'_3 \equiv \{(1,0),(0,1),(0.75,0.25),(0.5,0.5),(0.25,0.75)\}$.

Finally, after preliminary experimental analysis, the parameter Y^{max} was set to 15 for the remainder of the numerical experiments, which means that, for each 15 iterations of the propagation of label l, the update of the Pareto frontier was applied by using l.

Tables 2–4 show how $\mathcal{W}$ and $\mathcal{W}'$ affect the execution time of the LCDPF algorithm when using the strategies STR1–STR3, respectively. The first column shows the graph name, and columns two to ten give the average running time in seconds for each pair $\mathcal{W}$ and $\mathcal{W}'$. Note that $\mathcal{W} = \emptyset$ means that the Pareto frontier is not dynamically updated.

	$\mathcal{W}_1$		$\mathcal{W}_2$			$\mathcal{W}_3$			
	$\emptyset$	$\mathcal{W}'_1$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\mathcal{W}'_3$
P1	0.04	**0.03**	0.04	0.04	0.04	0.08	0.07	0.07	0.07
P2	0.14	**0.08**	0.13	0.10	0.09	0.16	0.15	0.15	0.15
P3	1.30	**0.29**	0.94	0.32	**0.29**	0.71	0.36	0.36	0.34
B1	0.24	**0.11**	1.30	0.13	0.12	0.26	0.23	0.22	0.23
B2	1.61	**0.34**	4.36	0.38	0.35	0.93	0.54	0.54	0.53
B3	9.79	1.86	2.41	1.88	1.66	3.92	1.89	1.78	**1.56**
SF1	0.37	**0.17**	0.31	0.22	0.22	0.48	0.42	0.41	0.41
SF2	30.46	6.08	19.57	6.09	5.23	11.13	5.67	5.17	**4.42**
SF3	158.22	45.83	122.80	45.49	41.61	98.24	44.50	41.57	**38.59**

Table 2: Influence of $\mathcal{W}$ and $\mathcal{W}'$ on LCDPF algorithm execution times (s) when using strategy STR1.

As can be observed in Tables 2 and 4, the best results for execution time are obtained with the combination $\{\mathcal{W}_3; \mathcal{W}'_3\}$, especially for large graphs. This means that dynamically updating the Pareto frontier improves the

	$\mathcal{W}_1$		$\mathcal{W}_2$			$\mathcal{W}_3$			
	$\emptyset$	$\mathcal{W}'_1$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\mathcal{W}'_3$
P1	**0.03**	**0.03**	0.04	0.05	0.04	0.08	0.07	0.07	0.07
P2	0.07	**0.06**	0.10	0.09	0.09	0.15	0.15	0.15	0.15
P3	0.30	**0.27**	0.34	0.31	0.29	0.40	0.38	0.37	0.37
B1	0.11	**0.10**	0.15	0.14	0.12	0.23	0.23	0.23	0.23
B2	0.34	**0.32**	0.43	0.40	0.35	0.58	0.56	0.56	0.58
B3	1.82	**1.75**	1.95	1.84	**1.75**	2.05	1.99	1.98	1.97
SF1	0.18	**0.17**	0.26	0.26	0.22	0.42	0.41	0.41	0.41
SF2	6.38	6.17	6.63	6.40	**6.13**	6.90	6.75	6.62	6.60
SF3	62.02	**60.45**	62.49	60.77	60.52	62.49	61.33	61.07	61.34

Table 3: Influence of $\mathcal{W}$ and $\mathcal{W}'$ on LCDPF algorithm execution times (s) when using strategy STR2.

	$\mathcal{W}_1$		$\mathcal{W}_2$			$\mathcal{W}_3$			
	$\emptyset$	$\mathcal{W}'_1$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\emptyset$	$\mathcal{W}'_1$	$\mathcal{W}'_2$	$\mathcal{W}'_3$
P1	0.04	**0.03**	0.04	0.04	0.04	0.08	0.07	0.08	0.07
P2	0.17	**0.08**	0.15	0.10	0.10	0.16	0.15	0.15	0.15
P3	3.25	**0.52**	2.44	0.55	**0.52**	1.78	0.60	0.59	0.55
B1	0.34	**0.12**	0.21	0.14	0.13	0.27	0.24	0.23	0.23
B2	3.58	**0.45**	2.46	0.49	**0.45**	1.51	0.64	0.62	0.61
B3	33.41	4.50	22.87	4.52	3.98	12.33	4.20	3.89	**3.27**
SF1	0.59	**0.18**	0.36	0.24	0.23	0.52	0.43	0.43	0.42
SF2	132.30	21.15	84.89	21.07	17.72	44.33	18.10	16.31	**13.05**
SF3	1,126.77	336.01	893.35	335.97	310.70	722.35	326.38	307.68	**284.61**

Table 4: Influence of $\mathcal{W}$ and $\mathcal{W}'$ on LCDPF algorithm execution times (s) when using strategy STR3.

algorithm efficiency when using the strategy STR1 or STR3. However, the data of Table 3 show that the best results for execution time are obtained with two types of combinations, $\{\mathcal{W}_1; \mathcal{W}'_1\}$ and $\{\mathcal{W}_2; \mathcal{W}'_2\}$, which lead to approximately the same results. The use of the combination $\{\mathcal{W}_3; \mathcal{W}'_3\}$ with the strategy STR2 does not minimize the corresponding execution times. In fact, dynamically updating the Pareto frontier requires comparing labels to determine which are non-dominated (this step is time consuming), and the efficiency of this step is related to the label-selection strategy. Whichever combination of $\mathcal{W}$ and $\mathcal{W}'$ is used, the strategy STR3 based on a linear

combination is the less effective. In general, the strategy STR1 based on the Dijkstra method and using the combination $\{\mathcal{W}_3; \mathcal{W}'_3\}$ produces superior results compared with the strategy STR2 based on the A* method, even with the combination $\{\mathcal{W}_1; \mathcal{W}'_1\}$. However, the algorithm that uses the strategy STR2 requires less memory than the strategy STR1. Therefore, the strategies STR1 and STR2 were used for the remainder of the computational experiments.

4.3 Influence of the number of criteria

Increasing the number of criteria increases significantly the difficulty of the problem because the number of solutions in the final Pareto frontier increases considerably. To the best of our knowledge, the majority of the studies to have addressed the MOSP problem used either instances with only two criteria or instances with a small graph and up to ten criteria. To determine how the number of criteria affects the results, we conducted two types of experiments on two versions of the LCDPF algorithm:

(i) $LCDPF_1$: This experiment used the LCDPF algorithm with the following parameters: $Y^{max} = 15$, strategy STR1 is used, and $\mathcal{W}_3 = \mathcal{W}'_3$ if $|\mathcal{K}| = 2$; otherwise $\mathcal{W} = \mathcal{W}'$ contains a set of mono-objective searches and a set of identical weights for each criterion [e.g., if $|\mathcal{K}| = 5$ then $\mathcal{W} = \mathcal{W}' = \{(1,0,0,0,0), (0,1,0,0,0), (0,0,1,0,0), (0,0,0,1,0), (0,0,0,0,1), (0.2,0.2,0.2,0.2,0.2)\}$].

(ii) $LCDPF_2$: This experiment used the LCDPF algorithm with the following parameters: $Y^{max} = 15$, strategy STR2 is used, and $\mathcal{W} = \mathcal{W}'$ set to a corresponding set of mono-objective searches regardless of the number of criteria [e.g., if $|\mathcal{K}| = 5$ then $\mathcal{W} = \mathcal{W}' = \{(1,0,0,0,0), (0,1,0,0,0), (0,0,1,0,0), (0,0,0,1,0), (0,0,0,0,1)\}$].

The first type of experiment is based on the benchmark instances of the resource-constrained shortest path problem proposed by [4]. As in [19], the instances were adapted, and each type of resource was considered as a criterion in addition to the distance criterion. From these instances, we created six instances with three, five, and eleven criteria. Table 5 gives the size of the graphs and the results for each instance (i.e., execution times for versions of the LCDPF algorithm and the number of solutions in the final Pareto frontier).

Instance	Number of nodes	Number of arcs	Number of criteria								
			$\|\mathcal{K}\| = 3$			$\|\mathcal{K}\| = 5$			$\|\mathcal{K}\| = 11$		
			$LCDPF_1$ Time (s)	$LCDPF_2$ Time (s)	$\|S\|$	$LCDPF_1$ Time (s)	$LCDPF_2$ Time (s)	$\|S\|$	$LCDPF_1$ Time (s)	$LCDPF_2$ Time (s)	$\|S\|$
RCSP 5	100	990	**0.00**	**0.00**	4	**0.00**	**0.00**	4	**0.01**	**0.01**	16
RCSP 7	100	999	**0.01**	**0.01**	42	**0.08**	**0.08**	394	9.17	**8.68**	6346
RCSP 13	200	2080	**0.00**	**0.00**	23	**0.01**	**0.01**	65	**0.04**	**0.04**	253
RCSP 15	200	1960	**0.01**	**0.01**	24	**0.05**	**0.05**	219	12.86	**12.75**	6331
RCSP 21	500	4847	0.01	**0.00**	5	**0.01**	**0.01**	50	0.04	**0.03**	166
RCSP 23	500	4868	**0.02**	**0.02**	55	**0.41**	0.43	617	**74.94**	78.52	12 045

Table 5: Influence of the number of criteria when using RCSP instances.

Both versions of the LCDPF algorithm can solve instances of up to eleven criteria within a similar reasonable time. As expected, the execution time increases with the number of criteria. For example, the last instance is solved in 20 ms and has 55 non-dominated solutions in the Pareto frontier for the case of three criteria whereas, for eleven criteria, 12 045 non-dominated solutions are found in 74.94 s for $LCDPF_1$ and in 78.52 s for $LCDPF_2$.

The second type of experiment was based on the Geovelo instances for which we added a third criterion equal to one for each arc. This criterion serves to determine how many arcs compose a path from the source node to the destination node. This kind of criterion is in conflict with the two other criteria and implies the existence of a large number of non-dominated solutions in the Pareto frontier. Table 6 summarizes the execution times (in s) of both versions of the LCDPF algorithm and gives the number of solutions in the Pareto frontier for each group of Geovelo instances, for two and three criteria.

Instances	Number of criteria					
	$\|\mathcal{K}\| = 2$			$\|\mathcal{K}\| = 3$		
	$LCDPF_1$ Time (s)	$LCDPF_2$ Time (s)	$\|S\|$	$LCDPF_1$ Time (s)	$LCDPF_2$ Time (s)	$\|S\|$
P1	0.07	**0.03**	11.20	0.07	**0.05**	33.90
P2	0.15	**0.06**	41.65	0.30	**0.22**	200.60
P3	0.34	**0.27**	145.50	95.68	**95.18**	1623.00
B1	0.23	**0.10**	39.95	0.77	**0.73**	214.10
B2	0.53	**0.32**	144.35	93.33	**82.71**	1583.50
B3	**1.56**	1.75	367.05	10 178.70	**9590.16**	8861.00
SF1	0.41	**0.17**	60.40	111.93	**108.69**	873.45
SF2	**4.42**	6.17	429.85	51 512.63	**43 054.82**	12 273.50
SF3	**38.59**	60.45	1327.65	-	-	-

Table 6: Influence of the number of criteria when using Geovelo instances.

The results given in Table 6 clearly demonstrate that the number of solutions in the Pareto frontier increases with the number of criteria. For example, the instance group B3 has an average of 367.05 solutions when two criteria are considered and an average of 8861 solutions when three criteria are considered. This results in an increase in computing time. For the SF instances of group SF3, neither version of the LCDF algorithm can enumerate all the solutions in the Pareto frontier because of insufficient memory, whereas, for two criteria, all the instances are solved in less than 2

minutes. Regardless of the number of criteria, the computation times of both versions are similar for groups of small instances (i.e., P1, P2, P3, B1, and B2). However, for groups of large instances (i.e., B3, SF1, SF2, and SF3), the $LCDPF_1$ algorithm is faster than the $LCDPF_2$ algorithm for two criteria, whereas the $LCDPF_2$ algorithm is faster than the $LCDPF_1$ algorithm for three criteria. This may be explained by the required computation time of the dynamic update of Pareto frontier, which is much greater for three criteria than for two criteria, and knowing that this technique is extensively used in the $LCDPF_1$ algorithm compared to $LCDPF_2$ algorithm.

4.4 Comparison with the literature

The computational experiments presented in this section compare the proposed algorithm with five of the best-known methods in the literature, and we report the results of each method, including the computation times. Given the differences between computational systems used in these studies and that used herein, we normalize the running times by using the PassMark benchmark [56]. Section 2 of online supplement lists the computational systems used in each study, and we determine the time-scaling factor from the Thread mark of PassMark (higher values indicate that the corresponding CPU is faster).

First, we compare the results of both versions of our algorithm to the results of the bLSET algorithm presented by [47] and the pulse algorithm presented by [19] when applied to the NY, BAY, COL, FLA, and NW graphs of the 9^{th} DIMACS challenge. The parameter settings used for all experiments discussed in this section are the same as those in Section 4.3. We use the same pairs of source and target nodes, and the same classification of instances used by [19]. Based on the number of solutions in the Pareto frontier, the 30 instances of each graph are grouped into three equal-sized clusters, denoted S (small), M (medium), and L (large). Table 7 presents the computational results of bLSET, the pulse algorithm, and the LCDPF algorithms. The first column indicates the graph name and the group. The columns entitled bLSET, pulse2, $LCDPF_1$, and $LCDPF_2$ give the average execution times (in seconds) and the number of solved instances within a timeout of 3600 s for each algorithm. The last column gives the average number of non-dominated solutions in the Pareto frontier.

The results given in Table 7 indicate that the running times of the bLSET algorithm are on average longer than those of the other algorithms. For the majority of instance groups, the $LCDPF_1$ and $LCDPF_2$ algorithms are clearly faster on average and solve all instances of the groups within

Cluster Name	bLSET		Pulse		$LCDPF_1$		$LCDPF_2$		$\|S\|$
	Time (s)	Solved	Time (s)	Solved	Time (s)	Solved	Time (s)	Solved	
NY S	62.39	10	0.32	10	0.73	10	**0.29**	10	34.10
NY M	301.16	10	52.32	10	2.48	10	**1.39**	10	147.40
NY L	881.26	10	1367.66	7	**9.56**	10	12.57	10	422.70
BAY S	6.78	10	0.16	10	0.40	10	**0.13**	10	8.80
BAY M	55.24	10	5.70	10	1.18	10	**0.45**	10	49.90
BAY L	317.43	10	105.55	10	3.46	10	**2.50**	10	171.80
COL S	7.30	10	**0.20**	10	0.86	10	0.28	10	18.20
COL M	233.03	10	381.94	9	2.91	10	**1.84**	10	87.10
COL L	865.76	10	508.76	10	4.98	10	**3.47**	10	328.40
FLA S	330.12	10	**0.35**	10	2.61	10	0.74	10	14.70
FLA M	566.15	9	347.91	10	5.06	10	**1.86**	10	94.10
FLA L	2627.43	4	888.59	9	13.03	10	**10.47**	10	552.30
NW S	260.73	10	1.99	10	2.51	10	**0.77**	10	39.00
NW M	1109.98	8	81.96	10	4.90	10	**2.33**	10	124.20
NW L	1443.66	8	538.54	9	8.70	10	**5.96**	10	281.60

Table 7: Comparison of execution times (s) for bLSET, pulse, $LCDPF_1$, and $LCDPF_2$ algorithms.

15 s [note that, according to the PassMark benchmark, the setup of [19] is 1.31 times faster than the proposed setup]. The execution time of the $LCDPF_2$ algorithm is, on average, slightly less than the execution time of the $LCDPF_1$ algorithm. Finally, all algorithms behave the same; that is, the execution time increases exponentially with the number of non-dominated paths.

We also compared the results of the proposed algorithm with those of the adaptation of the NAMOA* algorithm presented by [37]. Their paper presents a pre-calculation method called KDLS for lower-bound sets. The computational experiments of [37] are based on the NY graph of the 9^{th} DIMACS challenge using two objectives: travel time and economic cost. The latter was introduced by [36] and is computed based on the toll and fuel-consumption costs according to road category. In this graph, only a single arc is considered for each pair of nodes. In the NY graph, for each set of arcs between the same pair of nodes, the first arc in the lexicographic order of the file is retained.

Table 8 gives the execution times of the NAMOA* algorithm proposed

by [37] (with several parameter settings of the KDLS method) and of both versions of the LCDPF algorithm applied to the same 20 pairs of source and target nodes. The first column gives the instance number, columns two to eight give the execution times (in seconds) of each parameter setting of the KDLS method (including the NAMOA* algorithm), and column nine gives the execution times of the $LCDPF_1$ and $LCDPF_2$ algorithms. The last column shows the number of non-dominated solutions for each instance.

#	KDLS ∞	KDLS 5	KDLS 4	KDLS 3	KDLS 2	KDLS 1	KDLS 0	$LCDPF_1$	$LCDPF_2$	$\|S\|$
1	973.3	922.6	920.4	799.7	539.1	572.6	499.1	**8.8**	9.5	1089
2	-	26 331.7	29 985.8	24 325.0	25 229.0	21 363.7	25 781.5	**344.8**	410.5	1469
3	6.9	7.3	7.1	6.6	5.7	4.2	5.1	0.5	**0.2**	16
4	-	-	-	-	-	-	-	-	-	-
5	29 607.9	26 395.4	25 897.6	21 532.1	21 872.4	20 011.3	15 710.4	**183.0**	230.6	2451
6	18 751.4	19 247.8	18 438.3	11 775.9	8491.7	6285.8	5697.4	**88.3**	118.7	1502
7	485.5	454.8	428.4	370.3	309.5	231.1	129.7	4.2	**3.0**	272
8	-	-	-	-	-	-	-	**2183.2**	2967.9	7391
9	4914.3	5088.5	4527.8	4646.8	3827.4	3418.6	3379.2	**57.4**	94.9	919
10	1188.7	1149.6	1144.1	1163.3	1067.3	895.2	722.7	**17.8**	23.4	774
11	613.6	575.4	514.8	404.4	363.2	346.5	362.4	**7.7**	10.6	631
12	24 559.0	21 720.4	19 037.9	15 565.9	20 373.9	14 272.4	18 605.4	**289.9**	397.5	1573
13	-	-	-	33 794.3	31 365.8	23 976.0	21 782.2	**223.3**	286.8	3046
14	24 086.2	22 925.4	22 569.8	27 995.3	20 002.3	18 693.8	24 683.8	**278.8**	352.6	2957
15	0.7	0.7	1.1	0.7	0.7	0.8	0.9	**0.1**	**0.1**	1
16	7303.9	5201.4	6555.1	7497.1	6355.2	5691.6	7399.7	**118.4**	134.8	2034
17	12 308.8	10 951.1	9034.0	8574.6	7668.2	10 714.7	12 849.8	**138.2**	172.1	1724
18	1644.6	1572.2	1426.9	1312.1	1476.3	1135.3	1168.0	**17.6**	21.4	1276
19	-	-	-	-	-	-	-	**698.1**	886.2	4224
20	-	-	-	-	-	-	-	**430.4**	597.3	3262

Table 8: Execution times (s) of the NAMOA* algorithm with several parameter settings of KDLS method compared with execution times of both versions of the LCDPF algorithm.

The results listed in Table 8 indicate that the execution times of the NAMOA* with several parameter settings of KDLS proposed by [37] exceed those of both versions of the LCDPF algorithm (even if we take into account a factor of 0.76, computed from the PassMark benchmark, which would reduce their execution times). For resolved instances, the $LCDPF_1$ and $LCDPF_2$ algorithms are 50 times faster on average than the best KDLS method. Neither NAMOA* algorithm solves instances 4, 8, 19, and 20 within 12 h (mainly because of insufficient memory space in our case), and none of the algorithms resolves instance 4. Comparing the results for the original instances of the NY graph for the 9^{th} DIMACS challenge with the results for the new instances based on the same graph proposed by [37] reveals an increasing number of non-dominated solutions in the Pareto frontier and a longer execution time on average for both versions of the LCDPF algorithm. This demonstrates that replacing an objective by another more conflicting objective (e.g., economic cost instead of distance) appears to increase the difficulty of the problem. For all 19 instances solved, the $LCDPF_1$ algorithm is faster than the $LCDPF_2$ algorithm for 16 instances with an average reduction in computation time of 22%.

Finally, we compare the $LCDPF_1$ and $LCDPF_2$ algorithms with the BBDijkstra and BDijkstra algorithms proposed by [51]. These two recent algorithms outperform one of the state-of-the-art algorithms to solve the bi-objective shortest path problems in large road networks (i.e., the pulse algorithm). We used the same instances from the 9^{th} DIMACS challenge (NY, BAY, COL, FLA, NE, CAL, and LKS graphs) and the same 100 pairs of source and target nodes for each graph. Each run was limited to 3600 seconds for all algorithms.

Table 9 lists the average, minimum, and maximum numbers of solutions in the Pareto frontier for solved instances, the number of solved instances for each algorithm, and the average, minimum, and maximum running times for each algorithm. Note that the average running times are determined with a computational time of 3600 seconds for unsolved instances.

The results of both versions of the LCDPF algorithm are similar to those of the BBDijkstra and BDijkstra algorithms and have shorter average running times for the FLA, NE, CAL, and LKS graphs. However, the CPU used in [51] is 65% faster than the CPU used to execute the LCDPF algorithm (according to the PassMark benchmark). After standardizing the running times with the help of the time-scaling factor, the $LCDPF_1$ and $LCDPF_2$ algorithms have shorter running times for all graphs except for the smallest graph (NY). The proposed algorithms are at least twice as fast as the BBDijkstra and BDijkstra algorithms for the FLA and NE graphs, at least 50%

		$\|S\|$	BBDijkstra	BDijkstra	$LCDPF_1$	$LCDPF_2$
NY	Solved/100	100/100	**100/100**	**100/100**	**100/100**	**100/100**
	Avg.	119.79	**0.74**	1.32	2.10	1.62
	Min	1	0.22	0.15	0.13	0.03
	Max	646	8.46	21.75	16.31	26.80
BAY	Solved/100	100/100	**100/100**	**100/100**	**100/100**	**100/100**
	Avg.	143.77	**1.28**	2.16	3.21	3.36
	Min	1	0.27	0.18	0.16	0.04
	Max	825	16.39	33.42	29.57	51.07
COL	Solved/100	100/100	**100/100**	**100/100**	**100/100**	**100/100**
	Avg.	346.51	**10.89**	12.20	12.83	16.18
	Min	1	0.36	0.25	0.18	0.04
	Max	2612	255.25	355.57	336.38	412.90
FLA	Solved/100	100/100	**100/100**	99/100	**100/100**	**100/100**
	Avg.	673.72	83.86	129.21	**50.55**	71.04
	Min	2	0.93	0.62	0.40	0.08
	Max	6292	1596.66	1626.24	1076.84	1637.23
NE	Solved/100	99/100	**99/100**	**99/100**	**99/100**	**99/100**
	Avg.	808.19	246.95	199.83	**135.46**	172.32
	Min	7	1.56	1.05	0.65	0.15
	Max	3145	3414.14	1308.06	1143.06	1440.57
CAL	Solved/100	99/100	**98/100**	**98/100**	97/100	**98/100**
	Avg.	862.54	216.99	267.29	**204.62**	228.56
	Min	1	1.90	1.24	0.74	0.15
	Max	6962	2543.59	2786.61	1013.86	2074.06
LKS	Solved/100	74/100	68/100	**74/100**	68/100	69/100
	Avg.	1917.80	1577.87	1421.14	**1,325.55**	1380.35
	Min	1	2.93	1.92	1.21	0.26
	Max	7547	3560.20	3286.70	1357.64	2210.31

Table 9: Comparison of execution times (s) of BBDijkstra, BDijkstra, $LCDPF_1$, and $LCDPF_2$ algorithms.

faster for the CAL graph, and at least 70% faster for the LKS graph. These results demonstrate that both versions of the LCDPF algorithm are particularly efficient for the largest graphs. However, the BDijkstra algorithm solves 74 instances for the LKS graphs, whereas the $LCDPF_1$ algorithm solves 68 instances and the $LCDPF_2$ algorithm solves 69 instances.

The results of the $LCDPF_1$ and $LCDPF_2$ algorithms thus seem quite

similar. Although the $LCDPF_1$ algorithm is slightly faster than the $LCDPF_2$ algorithm, the $LCDPF_2$ algorithm solves slightly more instances than the $LCDPF_1$ algorithm. These results all show that, even if the $LCDPF_1$ algorithm uses a poorer label-selection strategy (STR1) than that (STR2) used by the $LCDPF_2$ algorithm, the intense reliance on dynamically updating the Pareto frontier makes the $LCDPF_1$ algorithm as efficient as the $LCDPF_2$ algorithm.

5 Conclusion

We propose herein a new exact algorithm called the "Label-Correcting with Dynamic update of Pareto Frontier" (LCDPF) algorithm to solve the one-to-one MOSP problem. The LCDPF algorithm is an improved label-correcting algorithm that rapidly solves the MOSP problem on large graphs with up to millions of nodes and edges. We undertook computational experiments to compare two versions of the proposed algorithm with the best-known methods from the literature, and the results demonstrate that both versions outperform most existing methods. We also generated new instances based on three graphs corresponding to the urban areas of Paris, Berlin, and the San Francisco Bay Area, including at least two conflicting objective functions (travel distance and security cost).

The LCDPF algorithm is based on the assumption that the graph in question contains only nonnegative costs. However, if no negative cycle exists, the algorithm can be modified to consider negative costs in the graph. The first phase of the algorithm uses the Bellman-Ford-Moore [5, 23, 41] algorithm instead of Dijkstra's algorithm, which increases the computation time of this phase and invalidates Lemma 1. Nevertheless, the second phase remains unchanged.

Finally, the LCDPF algorithm can be extended to consider, in addition to sum-type objectives, bottleneck objectives such as minimizing the maximum cost of a path or maximizing the minimum cost of a path. To consider this type of objective, Dijkstra's algorithm is easily adapted and some minor steps are added to the LCDPF algorithm, such as the dominance condition, label comparison, and label propagation.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network flows: theory, algorithms, and applications*. Cambridge, Mass.: Alfred P. Sloan School of Management, Massachusetts Institute of Technology, 1993.

[2] Y. Aneja and K. Nair. Bicriteria transportation problem. *Management Science*, 25(1):73–78, 1979.

[3] J. Azevedo, M. E. O. S. Costa, J. J. E. S. Madeira, and E. Q. V. Martins. An algorithm for the ranking of shortest paths. *European Journal of Operational Research*, 69(1):97–106, 1993.

[4] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.

[5] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[6] D. Bertsekas. *Network Optimization: Continuous and Discrete Models*. Belmont: Athena Scientific, 1998.

[7] M. A. Bolívar, L. Lozano, and A. L. Medaglia. Acceleration strategies for the weight constrained shortest path problem with replenishment. *Optimization Letters*, 8(8):2155–2172, 2014.

[8] J. Brumbaugh-Smith and D. Shier. An empirical investigation of some bicriterion shortest path algorithms. *European Journal of Operational Research*, 43:216–224, 1989.

[9] N. Cabrera, A. L. Medaglia, L. Lozano, and D. Duque. An exact bidirectional pulse algorithm for the constrained shortest path. *Networks*, 76(2):128–146, 2020.

[10] W. Carlyle and R. Wood. Near-shortest and k-shortest simple paths. *Networks*, 46:98–109, 2005.

[11] J. Clímaco and E. Martins. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11:399–404, 1982.

[12] J. C. Clímaco, J. M. Craveirinha, and M. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Networks*, 41(4):206–220, 2003.

[13] J. Current, C. Revelle, and J. Cohon. The median shortest path problem: a multiobjective approach to analyze cost vs. accessibility in the design of transportation networks. *Transportation Science*, 21(3):188–197, 1987.

[14] J. Current, C. ReVelle, and J. Cohon. The minimum-covering/shortest-path problem. *Decision Sciences*, 19(3):490–503, 1988.

[15] S. Demeyer, J. Goedgebeur, P. Audenaert, M. Pickavet, and P. Demeester. Speeding up Martins algorithm for multiple objective shortest path problems. *4OR*, 11(4):323–348, 2013.

[16] N. Deo and C. Pang. Shortest-path algorithms: Taxonomy and annotation. *Networks*, 14(2):275–323, 1984.

[17] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, (1):269–271, 1959.

[18] I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks: An International Journal*, 42(3):135–153, 2003.

[19] D. Duque, L. Lozano, and A. Medaglia. An exact method for the biobjective shorstest path problem for large-scale road networks. *European Journal of Operational Research*, (242):788–797, 2015.

[20] M. Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

[21] A. Eiger, P. B. Mirchandani, and H. Soroush. Path preferences and optimal paths in probabilistic networks. *Transportation Science*, 19(1):75–84, 1985.

[22] F. G. Engineer, G. L. Nemhauser, M. W. Savelsbergh, and J.-H. Song. The fixed-charge shortest-path problem. *INFORMS Journal on Computing*, 24(4):578–596, 2012.

[23] L. R. Ford Jr and D. R. Fulkerson. *Flows in networks*. Princeton university press, 2015.

[24] V. Gabrel and D. Vanderpooten. Enumeration and interactive selection of efficient paths in a multiple criteria graph for scheduling an earth observing satellite. *European Journal of Operational Research*, 139(3):533–542, 2002.

[25] L. Galand, A. Ismaili, P. Perny, and O. Spanjaard. Bidirectional preference-based search for state space graph problems. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[26] R. G. Garroppo, S. Giordano, and L. Tavanti. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Computer Networks*, 54(17):3081–3107, 2010.

[27] D. Gavalas, C. Konstantopoulos, K. Mastakas, and G. Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, 20(3):291, 2014.

[28] A. M. Geoffrion. Proper efficiency and the theory of vector maximization. *Journal of mathematical analysis and applications*, 22(3):618–630, 1968.

[29] F. Guerriero and R. Musmanno. Label correcting methods to solve multicriteria shortest path problems. *Journal of optimization theory and applications*, 111:589–613, 2001.

[30] P. Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. Springer, 1980.

[31] P. Hansen. Multiple criteria decision making theory and application. *Lecture Notes in Economics and Mathematical Systems*, 177:109–127, 1980.

[32] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[33] S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pages 33–65. Springer, 2005.

[34] L. Lozano, D. Duque, and A. L. Medaglia. An exact algorithm for the elementary shortest path problem with resource constraints. *Transportation Science*, 50(1):348–357, 2015.

[35] L. Lozano and A. L. Medaglia. On an exact method for the constrained shortest path problem. *Computers & Operations Research*, 40(1):378–384, 2013.

[36] E. Machuca and L. Mandow. Multiobjective route planning with pre-calculated heuristics. In *Proc. of the 15th Portuguese Conference on Artificial Intelligence (EPIA 2011)*, pages 98–107, 2011.

[37] E. Machuca and L. Mandow. Lower bound sets for biobjective shortest path problems. *Journal of Global Optimization*, 64(1):63–77, 2016.

[38] L. Mandow and J. D. L. Cruz. Multiobjective A* search with consistent heuristics. *Journal of the ACM (JACM)*, 57(5):27, 2010.

[39] E. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, (16):236–245, 1984.

[40] J. E. Mendoza, C. Guéret, M. Hoskins, H. Lobit, V. Pillac, T. Vidal, and D. Vigo. Vrp-rep: the vehicle routing community repository. In *Third Meeting of the EURO Working Group on Vehicle Routing and Logistics Optimization (VeRoLog). Oslo, Norway*, 2014.

[41] E. F. Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

[42] J. Mote, I. Murthy, and D. L. Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53(1):81–92, 1991.

[43] J. M. L. Munoz. Boost multi-index containers library. *C/C++ Users Journal*, 22(9):6, 2004.

[44] I. A. Ndiaye, E. Neron, and A. Jouglet. Macroscopic evacuation plans for natural disasters. *OR spectrum*, 39(1):231–272, 2017.

[45] J. M. Paixão and J. L. Santos. Labeling methods for the general case of the multi-objective shortest path problem–a computational study. In *Computational Intelligence and Decision Making*, pages 489–502. Springer, 2013.

[46] A. Parmentier. Algorithms for non-linear and stochastic resource constrained shortest path. *Mathematical Methods of Operations Research*, 89(2):281–317, 2019.

[47] A. Raith. Speed-up of labelling algorithms for biobjective shortest path problems. In *Proceedings of the 45th annual conference of the ORSNZ. Auckland, New Zealand*, pages 313–322, 2010.

[48] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & OR*, 36(4):1299–1331, 2009.

[49] A. Raith, M. Schmidt, A. Schöbel, and L. Thom. Extensions of labeling algorithms for multi-objective uncertain shortest path problems. *Networks*, 72(1):84–127, 2018.

[50] G. Sauvanet and E. Neron. Search for the best compromise solution on multiobjective shortest path problem. *Electronic Notes in Discrete Mathematics*, (36):615–622, 2010.

[51] A. Sedeño-Noda and M. Colebrook. A biobjective Dijkstra algorithm. *European Journal of Operational Research*, 276(1):106–118, 2019.

[52] A. Sedeño-Noda and A. Raith. A Dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem. *Computers & Operations Research*, 57:83–94, 2015.

[53] P. Serafini. Some considerations about computational complexity for multiobjective combinatorial problems. In *Lecture Notes in Economics and Mathematical Systems*, pages 222–232. Springer-Verlag, 1987.

[54] H. D. Sherali, A. G. Hobeika, and S. Kangwalklai. Time-dependent, label-constrained shortest path problems with applications. *Transportation Science*, 37(3):278–293, 2003.

[55] A. Skriver and K. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27:507–524, 2000.

[56] P. Software. Cpu benchmarks. `https://www.cpubenchmark.net/`, July 2019.

[57] 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*, 17:269–287, 2007.

[58] B. W. Thomas, T. Calogiuri, and M. Hewitt. An exact bidirectional A* approach for solving resource-constrained shortest path problems. *Networks*, 73(2):187–205, 2019.

[59] V. T'kindt and J.-C. Billaut. *Multicriteria scheduling: theory, models and algorithms*. Springer Science & Business Media, 2006.

[60] C. Tung and K. Chew. A multicriteria pareto-optimal path algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

[61] E. Ulungu and J. Teghem. Multi-objective combinatorial optimization problems: A survey. *Journal of Multi-Criteria Decision Analysis*, 3:83–104, 1994.

[62] E. Ulungu and J. Teghem. The two phases method : An efficient procedure to solve biobjective combinatorial optimization problems. *Foundations of Computing and Decision Sciences*, 20(2):149–165, 1995.