European Journal of Operational Research 276 (2019) 106–118

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

A biobjective Dijkstra algorithm

Antonio Sedeño-noda[a,*], Marcos Colebrook[b]

[a] Departamento de Matemáticas, Estadística e Investigación Operativa, Universidad de La Laguna, C.P. 38271, San Cristóbal de La Laguna, Santa cruz de Tenerife, España
[b] Departamento de Ingeniería Informática y de Sistemas, Universidad de La Laguna, C.P. 38271, San Cristóbal de La Laguna, Santa cruz de Tenerife, España

ARTICLE INFO

Article history:
Received 5 October 2017
Accepted 4 January 2019
Available online 8 January 2019

Keywords:
Network optimization
Biobjective path problems
Efficient paths
Dijkstra's algorithm

ABSTRACT

We generalize the Dijkstra algorithm to the Biobjective Shortest Path (BSP) problem. The proposed method keeps only one candidate label per node in a priority queue of size n. In this way, we introduce a novel algorithm to solve the one-to-all BSP problem determining all non-dominated points in the outcome space and one efficient path associated with each of them. For the case of the one-to-one BSP problem, we incorporate the classical bidirectional search scheme in the proposed algorithm to reduce the number of iterations in practice. The proposed algorithm also includes pruning strategies to avoid the computation of unnecessary labels. The result is a fast algorithm to solve the one-to-one BSP problem in large networks. A computational experiment comparing the performance of the proposed method and the state-of-the-art methods is included.

1. Introduction

In a directed network with arbitrary lengths, the *shortest path* (SP) problem consists of finding directed paths of minimal length from an origin node to all other nodes, or detecting a directed cycle of negative length. The SP problem is one of the most fundamental problems in network optimization. Numerous algorithms that solve SP problems and several real-world applications are reviewed in Ahuja, Magnanti, and Orlin (1993). The SP problem is classified as a one-to-one problem when it is required to determine the shortest path from one origin node to one destination node or as a one-to-all problem when we need to compute the shortest paths from one origin node to all other nodes in the graph. Hereinafter, we refer to SP problems without differentiating between the one-to-all and the one-to-one cases; otherwise, we add the corresponding prefix to distinguish when it is necessary.

Most of the existent algorithms for the standard single objective SP problems fall in the category of labeling methods that iteratively make distance labels *permanent* (guaranteeing that the labels represent the length of an optimal path) for all nodes. Labeling methods are partitioned into *label-setting* and *label-correcting* methods. Label-setting methods are characterized by setting one distance label of a known node permanent in each iteration of the method, while label-correcting methods consider all distance labels as temporary until the end of the algorithm. The most famous and fastest label-setting method is Dijkstra's algorithm (Dijkstra, 1959). Several data structures can be incorporated in the Dijkstra algorithm to improve its performance (see Cormen, Leiserson, Rivest, & Stein, 2009; Fredman & Tarjan, 1987). Moreover, in the case of the one-to-one SP problem, the bidirectional search scheme improves in practice by a factor of 2 the unidirectional search of the Dijkstra algorithm (see Nicholson, 1966; Pohl, 1971). Additionally, A* search in graphs (or path-finder algorithm) is an extension of the unidirectional/bidirectional Dijkstra Algorithm (Goldberg & Harrelson, 2005).

The biobjective and multiobjective shortest path (BSP and MSP) problems have received considerable attention in the literature. There are different classes of solution approaches including labeling methods, the first of which were introduced by Hansen (1980) and Martins (1984) for BSP and MSP problems. They are considered the classical labeling methods for BSP and MSP problems. More recently, computational comparisons to solve the one-to-all MSP problem (see Guerriero & Musmanno, 2001; Paixão & Santos, 2013) and to solve the one-to-one MSP problem (Raith, 2010; Skriver & Andersen, 2000) investigate the performance of different labeling methods. The multiobjective A* (MOA*) algorithm was introduced by Stewart and White (1991). Machuca and Mandow (2012) study the effect of existent heuristic functions for the MOA* algorithm solving large one-to-one BSP problems in road maps. Neither of the proposed algorithms in the previously cited references include bidirectional search. Recently, Demeyer, Goedgebeur, Audenaert, Pickavet, and Demeester (2013) consider the bidirectional search on the classical labeling method for the one-to-one MSP problem, introducing a stopping criterion for

* Corresponding author.
E-mail addresses: asedeno@ull.es (A. Sedeño-noda), mcolesan@ull.edu.es (M. Colebrook).

https://doi.org/10.1016/j.ejor.2019.01.007
0377-2217/© 2019 Elsevier B.V. All rights reserved.

the multiobjective bidirectional search algorithm. Galand, Ismaili, Perny, and Spanjaard (2013) focus on bidirectional preference-based search algorithms for the one-to-one MSP problem.

Other methods include ranking approaches where the solutions are ranked taking into account the values of the first objective function until all efficient solutions are found (Martins & Climaco, 1981). A two phase method has been applied to BSP problems by Mote, Murthy, and Olson (1991). An extensive computational comparison of different types of algorithms to solve the one-to-one BSP is carried out in Raith and Ehrgott (2009). Recently, Duque, Lozano, and Medaglia (2015) introduce the PULSE algorithm based on the *depth-first-search* (DFS) scheme that includes pruning strategies to avoid the computation of dominated or non-promising solutions. The result is a fast algorithm to solve the one-to-one BSP problem in large networks as it is reported in Duque et al. (2015). Currently, the PULSE algorithm appears to be the fastest one in the literature that solves the one-to-one BSP problem in road networks.

The set of efficient solutions of the BSP problem can be divided into two groups: the set of supported solutions and the set of non-supported solutions. Recently, Sedeño-Noda and Raith (2015) develop a Dijkstra-like algorithm that generalizes the Dijkstra algorithm for the BSP problem computing the supported solutions quickly and effectively for large real-world networks.

This work investigates the existence of a generalized Dijkstra algorithm to compute all non-dominated points of the BSP problems. We observe that the implementation of the label selection criteria in the labeling methods solving the MSP problem does not maintain only one candidate label per node as the classical Dijkstra algorithm does for the single objective case. The additional candidate labels generate computationally expensive merge operations between the set of candidate (temporary) labels and the permanent non-dominated labels. Thus, we studied the possibility of keeping only one candidate label per node in the label-selection methods. This implies using a priority queue of maximum size equal to the number of nodes n. In addition, once the algorithm extracts a candidate label from the corresponding priority queue it then uses a function to obtain the next candidate label for this node. In this way, we obtain a novel Dijkstra-like method to find all non-dominated points to the one-to-all BSP problem. We denote by $N=\sum_{i=1}^{n} N_i$ the number of non-dominated points in the outcome space of the one-to-all BSP problem, where N_i is the number of non-dominated points of the one-to-one s-i BSP problem and $N_{\max}=\max_{i=1,\dots,n}\{N_i\}$. The proposed biobjective Dijkstra algorithm needs $O(N+m+n)$ space and runs in $O(N\log n+mN_{\max})$ time, where m is the number of arcs in the network.

Next, the algorithm is then adapted to solve the one-to-one BSP problem. This algorithm incorporates pruning strategies and uses the bidirectional search to reduce the number of iterations that the algorithm performs when solving the one-to-all BSP problem. The result is a fast and practical algorithm to solve the one-to-one BSP problem in large networks.

The paper is organized as follows: Section 2 describes the BSP problem and introduces some known results from the literature. In Section 3, the Dijkstra algorithm is generalized to solve the one-to-all BSP problem. This section describes in detail all the basic operations needed by the proposed algorithm: the label-selection function, the method to obtain a new label for a node and the relaxation process for its successor nodes. This section also includes the proof of the theoretical complexity of the proposed algorithm and other results regarding the correctness of the proposed algorithm. Sections 4 and 5 introduce the pruning strategies and the bidirectional search, respectively, that are included in the proposed algorithm that solves the one-to-one BSP problem. In Section 6, computational experiments comparing performance of the proposed algorithm, the PULSE algorithm and the classical label-setting algorithms are discussed. Finally, Section 7 concludes with some comments and possible lines of future research.

2. The biobjective shortest path problem

Given a directed network $G=(V, A)$. We define $V=\{1,\dots,n\}$ the set of n nodes and A the set of m arcs. Two real-valued costs$c_{ij}=(c_{ij}^1,c_{ij}^2)$ are associated with each arc $(i,j)\in A$. In a road network, these values can represent distance and time, respectively. We denote by $\Gamma_i^-=\{j\in V|(j,i)\in A\}$ and by $\Gamma_i^+=\{j\in V|(i,j)\in A\}$ the sets of predecessor and successor nodes, respectively, for any node $i\in V$. Node s is the *origin* node and t is the *destination* node. Let $i,j\in V$ be two distinct nodes of $G=(V, A)$, we define a directed path p_{ij} as a sequence $\langle i_1,(i_1,i_2),i_2,\cdots,i_{l-1},(i_{l-1},i_l),i_l\rangle$ of nodes and arcs satisfying $i_1=i$, $i_l=j$ and for all $1\le w\le l-1$, $(i_w,i_{w+1})\in A$. The length of a directed path p is the sum of the arc lengths in the path, that is, $c(p)=\sum_{(i,j)\in p}c_{ij}$. Let P be the set of paths from s to t in G.

Assumption 1 ((w.l.o.g.)). The network G contains a directed path from origin node s to any node $i\in V-\{s\}$ (if there is no path in G to some node i, then node i can be removed from G since it cannot lie on any s-t path).

If a flow x_{ij} is associated with each arc (i,j) then the following integer linear programming problem represents the one-to-one *Biobjective Shortest Path* (BSP) problem (see Raith & Ehrgott, 2009):

$$\text{Minimize}\quad c(x)=\left(\sum_{(i,j)\in A}c_{ij}^1x_{ij},\ \sum_{(i,j)\in A}c_{ij}^2x_{ij}\right)\tag{1}$$

subjectto

$$\sum_{j\in\Gamma_i^+}x_{ij}-\sum_{j\in\Gamma_i^-}x_{ji}=\begin{cases}1 & \text{if } i=s\\ 0 & \forall\ i\in V-\{s,t\}\\ -1 & \text{if } i=t\end{cases}\tag{2}$$

$$x_{ij}\in\{0,1\},\qquad \forall(i,j)\in A\tag{3}$$

where x_{ij} takes the value 1 for all arcs (i,j) on a feasible path, and 0 otherwise. Let X be the set defined by constraints (2) and (3) (*decision space*) and let $C=c(X)$ (*outcome space*) be its image under the objective.

A *directed out-spanning tree T* is a spanning tree rooted at node s such that the unique path in the tree from root node s to every other node is a directed path. In the remainder of the paper, we refer to a directed out-spanning tree simply as a tree.

Assumption 2. The network G does not contain a directed cycle with negative length for each single-objective SP problem, i.e. taking into account costs c^1 and c^2, respectively.

In the case when there are arcs with negative costs in G, from Assumption 2 we can find node potentials $\pi_i=(\pi_i^1,\pi_i^2)$ for all nodes $i\in V$ such that $\bar{c}_{ij}^v=c_{ij}^v+\pi_i^v-\pi_j^v\ge 0$ $\forall(i,j)\in A$ for $v=1,2$ by applying a label-correcting algorithm for each single-objective SP problem starting at node s. This requires additional $O(nm)$ time of computational effort (see Ahuja et al., 1993). Note that it is easy to show that the objective function considering reduced costs of the one-to-one BSP problem becomes:

$$\begin{aligned}\bar{c}(x) &= \left(\sum_{(i,j)\in A}\bar{c}^1_{ij}x_{ij}, \sum_{(i,j)\in A}\bar{c}^2_{ij}x_{ij}\right)\\ &= \left(\sum_{(i,j)\in A}c^1_{ij}x_{ij}+\pi^1_s-\pi^1_t, \sum_{(i,j)\in A}c^2_{ij}x_{ij}+\pi^2_s-\pi^2_t\right)\\ &= c(x)+\pi_s-\pi_t,\end{aligned}$$

and the objective function with the reduced cost of the one-to-all BSP problem is:

$$\begin{aligned}\bar{c}(x) &= \left(\sum_{(i,j)\in A}\bar{c}^1_{ij}x_{ij}, \sum_{(i,j)\in A}\bar{c}^2_{ij}x_{ij}\right)\\ &= \left(\sum_{(i,j)\in A}c^1_{ij}x_{ij}+(n-1)\pi^1_s\right.\\ &\qquad \left.-\sum_{i\in V-\{s\}}\pi^1_i, \sum_{(i,j)\in A}c^2_{ij}x_{ij}+(n-1)\pi^2_s-\sum_{i\in V-\{s\}}\pi^2_i\right)\\ &= c(x)+(n-1)\pi_s-\sum_{i\in V-\{s\}}\pi_i\end{aligned}$$

Therefore, the BSP problem with objective $c(x)$ is equivalent to the BSP problem with objective $\bar{c}(x)$ since $c(x)-\bar{c}(x)$ is only a constant term. In the rest of the paper, we assume without loss of generality that the costs $c_{ij}=(c^1_{ij}, c^2_{ij})$ take non-negative values.

Definition 1. A path (feasible solution) $p\in P$ ($x\in X$) is called *efficient* if there does not exist any $p'\in P$ ($x'\in X$) with $c^1(p')\le c^1(p)$ ($c^1(x')\le c^1(x)$) and $c^2(p')\le c^2(p)$ ($c^2(x')\le c^2(x)$) with at least one inequality being strict. The image $c(p)$ ($c(x)$) of an efficient path p (or solution x) is called a *non-dominated point*.

Definition 2. *Supported efficient paths* (*supported efficient solutions*) are those efficient paths (efficient solutions) that can be obtained as optimal paths (solutions) of a weighted sum problem $\min_{p\in P}(\lambda_1 c^1(p)+\lambda_2 c^2(p))$ ($\min_{x\in X}(\lambda_1 c^1(x)+\lambda_2 c^2(x))$) for some $\lambda_1>0$ and $\lambda_2>0$. All other efficient paths (solutions) are called *non-supported*.

The *supported non-dominated points* lie on the lower-left boundary of the convex hull ($conv(C)$) of the feasible set C in the *outcome space*, whereas non-supported points lie in the interior of $conv(C)$.

The focus of this paper is to design a fast algorithm to determine one efficient path for each non-dominated point in the BSP problem. From Assumption 2, it follows that for each non-dominated point, an efficient path exists that is a simple path (i.e. a path without repeating nodes). Moreover, the BSP problem satisfies the principle of optimality stated below (Martins, 1984).

Proposition 1. *Principle of Optimality. Every efficient path p_{st} from node s to node t contains only efficient sub-paths p_{si} from s to any intermediate node $i\in p_{st}$.*

According to the principle of optimality, an optimal solution for the BSP problem can be determined by finding successive optimal sub-paths; thus, a labeling method can be used to solve the problem. Furthermore, the non-dominance test can be applied to each intermediate node of some candidate path to determine whether it is efficient.

The lexicographic order ($\prec_L$) in $\mathbb{R}^2$ is useful in defining efficient procedures when verifying if a path is efficient and to order the non-dominated points.

Definition 3. Let (y,z) and (y',z') be two points in $\mathbb{R}^2$. We say that (y,z) is lexicographically smaller than (y',z'), denoted by $(y,z)\prec_L(y',z')$, if $y<y'$ or ($y=y'$ and $z<z'$) holds.

We denote by$(y,z)\le(y',z')\Leftrightarrow y\le y'$and $z\le z'$. Similarly, $(y,z)\ge(y',z')\Leftrightarrow y\ge y'$ and $z\ge z'$.

3. Expanding the Dijkstra algorithm for the BSP

The main purpose of labeling methods is to compute for each node i in V a set of labels $L[i]$ that store the images of all efficient paths from node s to node i. Individual labels at node i are $L[i][1], L[i][2], \dots, L[i][N_i]$ where N_i is the number of non-dominated points of the one-to-one BSP problem from node s to node i. Labels $L[i][k]$ with k varying in $\{1,\dots, N_i\}$ for all $i\in V$ must be dynamically created (as the value of N_i is unknown before the end of the algorithm). From this point of view, we suppose that $L[i]$ is a single linked list or a vector and, in a particular stage of the algorithm, $L[i]$ contains only non-dominated labels. At the end of the algorithm $L[i]$ is the set of non-dominated points associated with the s-i BSP problem. In particular, we assume that label $L[i][k]$ stores the information of the kth non-dominated point when the labels in $L[i]$ are stored in lexicographic order. For label $L[i][k]=(d^1,d^2, j, r)$, j is the predecessor node of node i in the kth efficient path, r denotes the position in $L[j]$ of the non-dominated label of node j that allows the kth efficient path for node i ($L[j][r]$) to be obtained, and d^1,d^2 are the distance labels of node i for each objective. The values of j and r associated with the label allow the kth efficient path of the s-i BSP problem to be identified. We use the next (recursive) *PrintPath* procedure to print the kth efficient path from node s to node i:

```
Procedure PrintPath(s, i, k, L);
(1)  If ($s \neq i$) then
(2)      PrintPath(s, L[i][k].j, L[i][k].r, L);
(3)  Print "i ";
```

The classical labeling methods can be viewed as derivations from two prototype procedures, referred to as the label-selection method and the node-selection method. In the first case, at each iteration, a label is selected (for example, the kth label associated with node i) and a new label is obtained for each node j successor to i. In the second case, at each iteration, a node i is selected and all the labels belonging to $L[i]$ are used in order to obtain a set of new labels for each successor of node i.

The first main difference between the proposed method and the classical label-selection methods is now presented. In order to introduce a Dijkstra-like method for the one-to-all BSP problem, our algorithm maintains a priority queue (i.e. a heap) that stores at most one candidate label for each node i in V. The candidate label of node i is not in $L[i]$. We denote this heap by H, with maximum size n. Any label in H is (i, d^1,d^2, j, r) where i identifies the final node of the path from node s with distances d^1 and d^2. As before, j is the predecessor node of node i in the candidate path and r denotes the position in $L[j]$ of the non-dominated label of node j that allows this label to be obtained. Any label in H has a *key* given by the pair of distances (d^1,d^2). The algorithm iteratively selects the label l in H with a lexicographic minimum key. Therefore, the label selection operation in our method is then just as simple as the selection of a label of a node in a Dijkstra algorithm. As in Dijkstra's algorithm, no label in H is definitive until it becomes extracted from H. Initially, H only contains the label (s, 0, 0, -, -). Additionally, we use a boolean vector *inH* where *inH*[i] is true if heap H contains a label associated with node i, and false otherwise.

Invariant 1. Our algorithm keeps the invariant that the label l in H associated with node i is not dominated by any label in $L[i]$, for all i in V.

3.1. Selection key and determination of a new label

There are many possible ways to select a label in H, for example FIFO order and others (Paixão & Santos, 2013; Skriver & Andersen, 2000). As previously mentioned, we consider that the selection criterion of the label l^* is given by

$$l^* = \arg \operatorname{lex}\min_{l\in H}\{(l.d^1, l.d^2)\}$$

In other words, l^* is the lexicographic smallest label in H. This operation can be easily implemented in any priority queue. We denote the above operation as *Find-min*(H). The following heap operations are used in our algorithm (see Cormen et al., 2009): *Create-Heap*(H), *Insert*(*label*, H), *Decrease-key*(*label*, H), and *Delete-min*(H).

Let $l^* = (i, d^1, d^2, j, r)$ be the label extracted from H in an iteration of the algorithm. In this case, the label l^* associated with a node i will be added to $L[i]$ since $(l^*.d^1, l^*.d^2)$ is currently a non-dominated label by Invariant 1. The question here is if the label l^* will belong to $L[i]$ at the end of the algorithm. For that, we must assure that there are not any non-explored paths from s to i whose pair of distances dominate $(l^*.d^1, l^*.d^2)$. Theorem 1 (in Section 3.3) confirms it. In this way, the non-dominated (permanent) labels of any node are determined in lexicographic min order as the Dijkstra algorithm determines the labels in non-decreasing costs. This choice simplifies the adding operation of label l^* to $L[i]$. Thus, once l^*associated with a node i is extracted from H, it is added at the end of $L[i]$ and no label in $L[i]$ is dominated by l^*. This operation requires $O(1)$ time. For simplicity, we code these operations as:

```
$l^*$ = Find-min($H$); Delete-min($H$);
Add $l^*$at the end of $L[i]$; $N_i = N_i + 1$;
$InH[i]$ = False; // $i$ is the node with label $l^*$
```

The classical Dijkstra method extracts from the heap one label per node. The proposed algorithm must extract exactly N_i labels for each node i in V. Therefore, once label l^* associated with a node i is extracted from H, the algorithm computes a new candidate label l^{new} for node i (whenever it exists). Following the Invariant 1, the label l^{new} must not be dominated by any label in $L[i]$. Moreover, l^{new} must not be in $L[i]$ to discard alternative efficient paths (we only want to compute one efficient path per each non-dominated point). The next label for node i must be the lexicographically smallest non-dominated label among the labels that can be obtained from the labels in $L[j]$, for any node j predecessor of node i. Therefore, we must search for a non-dominated label l with distances (a, b) for node i such that $(l^*.d^1, l^*.d^2) \prec_L (a, b)$. As the label (a, b) must be also a non-dominated point (that is not included in $L[i]$), we obtain that $a > l^*.d^1$ and $b < l^*.d^2$. Thus, the expression of the operation obtaining a new candidate label l^{new} for node i is:

$$l^{new} = \arg \operatorname{lex}\min_{\forall j\in\Gamma_i^-, \forall l\in L[j]} \left\{(l.d^1 + c^1_{ji}, l.d^2 + c^2_{ji}) | l.d^1 + c^1_{ji} > l^*.d^1 \text{ and } l.d^2 + c^2_{ji} < l^*.d^2\right\}.$$

The pseudo code of the function *NewCandidateLabel* (i, l^*) is given next. The computational effort of the function *NewCandidateLabel* is $O(\sum_{j\in\Gamma_i^-}|L[j]|)$. The function *NewCandidateLabel* outcomes as the second main difference between the proposed method and the classical label-selection methods. The proposed algorithm needs only one new candidate label for node i (whenever it exists) to be included in heap H. Instead, the classical labeling methods could maintain many candidate labels for node i. Moreover, in the dominance test it is only necessary to check if the last label in $L[i]$ dominates the candidate label. This already was observed by Captivo, Clímaco, Figueira, Martins, and Santos (2003), but we introduce later a formal proof of this result (see Theorem 2). Therefore, the computational effort to check dominance is $O(1)$ time. Meanwhile, in the classical label-setting algorithm checking dominance requires an effort $O(\log|L[i]|)$, since the temporal labels are not obtained in lexicographic min order (supposing that the temporal labels are stored in lexicographic min order). Note that in the classical label-setting method the merge operation adds a non-dominated label to the temporal labels set and deletes from it all dominated labels (for example, it needs $O(|L[i]|)$ time to delete dominated labels). This could impose a major computational cost in practice compared with the time employed by the proposed algorithm

```
Function NewCandidateLabel $(i, l^*)$
 $d^1 = +\infty$; $d^2 = +\infty$; $l^{new} = Null$;
For $j \in \Gamma_i^-$ do
    For $l \in L[j]$ do
      If $((l.d^1 + c^1_{ji} < d^1)$ or $(l.d^1 + c^1_{ji} == d^1$ and $l.d^2 + c^2_{ji} < d^2))$
      //lexmin candidate label
        If $(l.d^1 + c^1_{ji} > l^*.d^1$ and $l.d^2 + c^2_{ji} < l^*.d^2)$
{
        //non-dominated candidate label
          $d^1 = l.d^1 + c^1_{ji}$; $d^2 = l.d^2 + c^2_{ji}$;
          $l^{new} = (i, d^1, d^2, j, r)$; // r is the position of l in $L[j]$
      }
Return $l^{new}$;
```

3.2. Updating the labels in H. Relaxation

In Dijkstra's algorithm, the *Relaxation* process updates the distance label of any node j successor of node i when it is possible to improve the current distance label of the path to j by adding (i, j) to the path to i. Similarly, once the label l^*associated with a node i is extracted from H, it is possible that labels l in H corresponding to successor nodes j of node i can be (lexicographically) improved. In order to facilitate the description of the algorithm, we keep with each node j in H the values d^1_j, d^2_j being the pair of distance values of the label for node j in H. Thus, if $(l^*.d^1 + c^1_{ij}, l^*.d^2 + c^2_{ij}) \prec_L (d^1_j, d^2_j)$ and $(l^*.d^1 + c^1_{ij}, l^*.d^2 + c^2_{ij})$ is not dominated by the last label in $L[j]$ then we must set $(d^1_j, d^2_j) = (l^*.d^1 + c^1_{ij}, l^*.d^2 + c^2_{ij})$ (relaxation operation). The body of the procedure *RelaxationProcess* is:

```
Procedure RelaxationProcess$(i, H, l^*)$
For all $j \in \Gamma_i^+$ do
    If $((l^*.d^1 + c^1_{ij} < d^1_j)$ or $(l^*.d^1 + c^1_{ij} == d^1_j$ and $l^*.d^2 + c^2_{ij} < d^2_j))$
    //relaxation $(i, j)$
      If$(N_j == 0)$ or $((l^*.d^1 + c^1_{ij} > L[j][N_j].d^1$ and $l^*.d^2 + c^2_{ij} < L[j][N_j].d^2))$
{
    //non-dominated label
        $d^1_j = l^*.d^1 + c^1_{ij}$; $d^2_j = l^*.d^2 + c^2_{ij}$;
        $l = (j, d^1_j, d^2_j, i, N_i)$;// $N_i$ is the position of $l^*$ in $L[i]$
        If $(InH[j] == False)$ {Insert($l$, $H$); $InH[j] = True$;}
        Else decrease-key($l$, $H$);
    }
```

Note that when checking if $(l^*.d^1 + c^1_{ij}, l^*.d^2 + c^2_{ij})$ is not dominated by the last label in $L[j]$ ($L[j][N_j]$, where N_j is the current size of $L[j]$), then it will be non-dominated by any label in $L[j]$. Thus, the effort of the procedure *RelaxationProcess* is $O(|\Gamma_i^+|)$ time since all operations require $O(1)$ time (included the *Insert* and *Decrease-key* heap operations). This computational cost is equal to the effort made in the single-objective Dijkstra algorithm.

Note that the relaxation process does not require a merge operation on the labels $L[j]$ as in the classical label methods. As Dijkstra algorithm, it is only required the relaxation operation on the candidate label of node j in H, for all j successors of node i. This implies less computational cost in this part of the proposed method.

3.3. Basic scheme of the biobjective Dijkstra algorithm, correctness and theoretical complexity

The full scheme of the biobjective Dijkstra (BDijkstra) algorithm can now be considered. The pseudo-code of the BDijkstra algorithm is:

```
BDijkstra Algorithm; //Solving the one-to-all BSP problem
(1)   CreateHeap(H);
(2)   SetN_i = 0; $d_i^1 = +\infty$; $d_i^2 = +\infty$; InH[i] = False; forall $i \in V - \{s\}$;
(3)   $N_s = 0$; $d_s^1 = 0$; $d_s^2 = 0$; $l = (s, 0, 0, -, -)$; Insert(l, H); InH[s] = True;
(4)   While $(H \neq \emptyset)$ do
(5)       $l^*$ = Find-min(H); Delete-min(H);
(6)       $N_i = N_i + 1$; $L[i][N_i] = l^*$; InH[i] = False; // i is the node with label $l^*$
(7)       $l^{new}$ = NewCandidateLabel $(i, l^*)$;
(8)       If $(l^{new} \neq NULL)$ {Insert $(l^{new}, H)$;
              InH[i] = True; $d_i^1 = l^{new}.d^1$; $d_i^2 = l^{new}.d^2$;}
(9)       RelaxationProcess $(i, H, l^*)$;
```

We need to prove Theorems 1 and 2 before showing the correctness of the BDijkstra algorithm.

Theorem 1. *Let $G = (V, A)$ be a directed network with non-negative cost vector $c_{ij} = (c_{ij}^1, c_{ij}^2)$ for all arcs $(i, j) \in A$. The label $l^* = \arg\operatorname{lex}\min_{l \in H}\{(l.d^1, l.d^2)\}$ corresponds to an efficient path from node s to node i, that is, there is no path p from s to i in G whose distance $(c^1(p), c^2(p))$dominates $(l^*.d^1, l^*.d^2)$.*

Proof. The first label extracted from the priority queue is (s, 0, 0, -, -) and is a non-dominated label for node s since the costs are non-negative. At the beginning of an iteration, let $l^* = \arg\operatorname{lex}\min_{l \in H}\{(l.d^1, l.d^2)\}$ be the label extracted from H. For the purpose of contradiction, suppose that path p' from s to i with cost $(l^*.d^1, l^*.d^2)$ is not an efficient path. Note that the label l^* was calculated from the non-dominated labels in $L[j]$, for all predecessor nodes j to node i, by the function *NewCandidateLabel* (i, l^*) or by the function *RelaxationProcess*. Therefore, $(l^*.d^1, l^*.d^2)$ is the lexicographically smallest non-dominated label among the labels that can be obtained from the labels in $L[j]$, for any node j predecessor of node i. Sincep' is a dominated path, a non-examined efficient path p from s to i with label$(c^1(p), c^2(p))$ exists such that $(c^1(p), c^2(p)) \prec_L (l^*.d^1, l^*.d^2)$. Note that the label $(c^1(p), c^2(p))$ does not belong to $L[i]$ by Invariant 1. Since $(c^1(p), c^2(p))$ is not in H, there are labels associated to intermediate nodes in the path p that still have not been scanned. Let k be the intermediate node in the path p such that its distance label $(d^1(k),d^2(k))$ is not in $L[k]$, but the label associated to the predecessor node k' to node k in p is in $L[k']$. Such node must exist and the extreme case is when k' is node s. Therefore, the path p can be viewed as the path from s to k with cost $(d^1(k),d^2(k))$ plus a sub-path p_{ki} from k to i with cost $(c^1(p_{ki}), c^2(p_{ki}))$. From $c^1(p_{ki}) \geq 0$ and $c^2(p_{ki}) \geq 0$, we obtain that $(c^1(p), c^2(p)) = (d^1(k),d^2(k)) + (c^1(p_{ki}), c^2(p_{ki})) = (d^1(k) + c^1(p_{ki}), d^2(k) + c^2(p_{ki})) \geq (d^1(k), d^2(k))$ and, therefore, $(d^1(k),d^2(k)) \prec_L (c^1(p), c^2(p))$ or $(d^1(k),d^2(k)) = (c^1(p), c^2(p))$.

Now, we examine two cases when $(d^1(k),d^2(k)) \prec_L (c^1(p), c^2(p))$ (the proof when $(d^1(k),d^2(k)) = (c^1(p), c^2(p))$ is similar):

Case 1) If $(d^1(k),d^2(k))$ is in H, then we know that $(l^*.d^1, l^*.d^2) \prec_L (d^1(k), d^2(k))$ or $(l^*.d^1, l^*.d^2) = (d^1(k), d^2(k))$since both labels are in H and l^* is the lexicographically smallest label. Therefore $(l^*.d^1, l^*.d^2) \prec_L (d^1(k), d^2(k)) \prec_L (c^1(p), c^2(p))$ andp' is not dominated by p, and, thus p' is an efficient path from s to i.

Case 2) If $(d^1(k),d^2(k))$ is not in H, then a label (a, b) associated with node k exists in H such that $(a, b) \prec_L (d^1(k), d^2(k))$ (here $(a, b) \neq (d^1(k), d^2(k))$ since the proposed algorithm only computes a non-dominated label with the same values per node). The reason is that $(d^1(k),d^2(k))$ has been examined from node k' using a label in $L[k']$ and this label does not belong to H. Therefore we have that $(l^*.d^1, l^*.d^2) \prec_L (a, b)$ or $(l^*.d^1, l^*.d^2) = (a, b)$ since both labels are in H. Thus,$(l^*.d^1, l^*.d^2) \prec_L (a, b) \prec_L (d^1(k), d^2(k)) \prec_L (c^1(p), c^2(p))$. Again, p' is not dominated by p, and, thus p' is an efficient path from s to i. □

Note that Theorem 1 holds for the case of more than two objectives since we only used the lexicographic order to prove it. Therefore, Theorem 1 can be used to design a Dijkstra algorithm for the MSP problem. However, the next result is only fulfilled in the biobjective case.

Theorem 2. *Let $G = (V, A)$ be a directed network with non-negative cost vector $c_{ij} = (c_{ij}^1, c_{ij}^2)$ for all arcs $(i, j) \in A$. In the BDijkstra algorithm, the dominance test on a label distance (a, b) for a node i only needs to use the last label in $L[i]$.*

Proof. The BDijkstra algorithm computes the set of non-dominated (permanent) labels $L[i]$ in lexicographic min order. The labels in $L[i]$ appear in lexicographic min order since they are added at the end in $L[i]$. Therefore, in an iteration of the BDijkstra algorithm, let (a, b)be a candidate label of node i to be stored in H. Label (a, b) must satisfy $(c, d) \prec_L (a, b)$, where (c, d) is the last label in $L[i]$. Suppose that (a, b) is not dominated by (c, d), but it is dominated by a label (e, f) prior to label (c, d) in $L[i]$. This implies that $(e, f) \prec_L (c, d)$. If $(e = c)$ and $(f \leq d)$ then we have (e, f) dominates (c, d) or $(e, f) = (c, d)$, following to a contradiction (BDijkstra algorithm only computes one non-dominated point for all efficient paths with the same image in the outcome space). Therefore, we have $e < c$ and, therefore $c < a$, $b < d$ and $(e, f) \leq (a, b)$ with $(e, f) \neq (a, b)$ by dominance. Thus, $e < c$ and $f \leq b < d$. This observation means that the label (e, f) dominates label (c, d). This is a contradiction since (c, d) is a non-dominated label by Theorem 1. □

Theorem 2 helps to simplify the dominance test. The proposed algorithm computes all non-dominated points in the outcome space of the one-to-all BSP with origin s and destination i, for all nodes $i \in V - \{s\}$ from Proposition 1. That is, the correctness of the proposed algorithm can be proved by induction using Proposition 1 and Theorems 1 and 2. If we denote by $N = \sum_{i=1}^{n} N_i$ the number of non-dominated points in the outcome space of BSP, where N_i is the number of non-dominated points of the s-i BSP and $N_{\max} = \max_{i \in V}\{N_i\}$, we arrive at the following result

Lemma 1. *The BDijkstra algorithm runs in $O(N \log n + mN_{\max}^2)$ time and uses $O(N + m + n)$ space.*

Proof. Note that the size of the heap H in the *BDijkstra* algorithm is at most n. In addition, any heap operation takes constant time with the exception of the *Delete-min* operation which requires $O(\log n)$ time when a Fibonacci heap is used (see Fredman, & Tarjan., 1987). *BDijkstra* performs exactly N_i iterations for each node i in V. Thus, the total number of iterations in *BDijkstra* is N. In each iteration, a label l associated with node i is extracted from the heap H in $O(\log n)$ time and added to $L[i]$ in constant time. As previously mentioned, the function *NewCandidateLabel* (i, l^*) needs $O(\sum_{j \in \Gamma_i^-} |L[j]|)$. Finally, the procedure *RelaxationProcess* requires $O(|\Gamma_i^+|)$ time. Considering all iterations and all nodes, we obtain:

$$O\left(\sum_{i \in V} N_i \left(\log n + \sum_{j \in \Gamma_i^-} |L[j]| + |\Gamma_i^+|\right)\right).$$

Since, $|L[j]| \leq N_j \leq N_{\max}$, the above sum is

$$O\left(\sum_{i \in V} N_i \left(\log n + |\Gamma_i^-| N_{\max} + |\Gamma_i^+|\right)\right)$$
$$= O(N \log n + mN_{\max}^2 + mN_{\max}).$$

If we only consider dominant terms this sum becomes $O(N \log n + mN_{\max}^2)$. Note that the space used by the algorithm is

$O(N+m+n)$ since any label requires constant space and we must store the graph and at most n labels in H. □

The computational effort of the recursive procedure *PrintPath* is $O(n)$ time since any non-dominated path calculated by the BDijkstra algorithm is a simple path. Therefore, the process of printing all non-dominated paths requires $O(nN)$ additional time.

The determination of a new label by the function *NewCandidateLabel* (i, l^*) clearly induces a dependence on the quadratic term in $N_{\max}$ in the time complexity of the algorithm. In the next sub-section, the *NewCandidateLabel* (i, l^*) is improved using additional $O(m)$ space to obtain a BDijkstra algorithm running in $O(N \log n + mN_{\max})$ time.

3.4. Enhancement of the function NewCandidateLabel

We use a data structure *LastLabelArc*[i][j] for all arcs$(j, i) \in A$. Therefore, the space equals the number of arcs, m. For that, *LastLabelArc*[i][j] can be implemented as an additional field for each node j in the adjacency predecessor list of node i. *LastLabelArc*[i][j] stores the position r of the first label l in $L[j]$ ($L[j][r]$) satisfying that $(L[i].[N_i].d^1, L[i].[N_i].d^2) \prec_L (l.d^1 + c^1_{ji}, l.d^2 + c^2_{ji})$ and $(l.d^1 + c^1_{ji}, l.d^2 + c^2_{ji})$ is not dominated by $(L[i].[N_i].d^1, L[i].[N_i].d^2)$. Note that the labels in $L[j]$ appear in lexicographic min order and therefore:

$$(L[j].[r].d^1 + c^1_{ji}, L[j].[r].d^2 + c^2_{ji}) \prec_L (L[j].[r+1].d^1 + c^1_{ji}, L[j].[r+1].d^2 + c^2_{ji}).$$

Thus, *LastLabelArc*[i][j] stores the position of the last label in $L[j]$ tested to define the last label of node i using the arc (j, i) in the function *NewCandidateLabel* (i, l^*). Initially, *LastLabelArc*[i][j] is equal to 0 for all $(j, i) \in A$ and it is updated in function *NewCandidateLabel*$+ (i, l^*)$ as follows:

```
Function NewCandidateLabel+ $(i, l^*)$
 $d^1 = +\infty$; $d^2 = +\infty$; $l^{new} = Null$;
For $j \in \Gamma^-_i$ do {
    $r = LastLabelArc[i][j]$; $found = False$;
    While $(r \le N_j)$ and $(found == False)$ do {
      $l = L[j][r]$;
      If $(l.d^1 + c^1_{ji} > l^*.d^1$ and $l.d^2 + c^2_{ji} < l^*.d^2)$ {
      //non-dominated candidate label
        found $= True$;
        If $((l.d^1 + c^1_{ji} < d^1)$ or $(l.d^1 + c^1_{ji} == d^1$ and $l.d^2 + c^2_{ji} < d^2))$
        //lexmin candidate label
              {$d^1 = l.d^1 + c^1_{ji}$; $d^2 = l.d^2 + c^2_{ji}$; $l^{new} = (i, d^1, d^2, j, r)$;}
      }
      If $(found == False)$ $r = r + 1$;
    }
    $LastLabelArc[i][j] = r$;
}
Return $l^{new}$;
```

Lemma 2. *The overall effort of the function NewCandidateLabel*$+ (i, l^*)$ *in the BDijkstra algorithm is* $O(mN_{\max})$ *time.*

Proof. Observe that *LastLabelArc*[i][j] never decreases in all iterations when a label associated to node i is extracted from H. Thus, the set of permanent labels $L[j]$ for each arc $(j, i) \in A$ is completely examined exactly once in all the iterations when node i is extracted from H. Since $|L[j]| \le N_j \le N_{\max}$, this imposes an $O(\sum_{i \in V} \sum_{j \in \Gamma^-_i} N_j = mN_{\max})$ time in all iterations. Additionally, *LastLabelArc*[i][j] does not increase at most $N_i$times for each arc $(j, i) \in A$. This implies a total time of $O(\sum_{i \in V} \sum_{j \in \Gamma^-_i} N_i = mN_{\max})$ in all iterations. Thus, the run time of the *NewCandidateLabel*$+ (i, l^*)$ in all iterations is $O(mN_{\max})$ *time.*

Theorem 3. *The BDijkstra algorithm runs in* $O(N \log n + mN_{\max})$ *time and uses* $O(N+m+n)$ *space.*

Proof. From Lemmas 1 and 2 theorem holds.

The theoretical complexity of the BDijkstra algorithm matches with the theoretical complexity of the single-objective Dijkstra algorithm multiplied by $N_{\max}$. That is, $O(N_{\max}(n \log n + m))$ using the relation $N \le nN_{\max}$.

In the next two sections, we consider the one-to-one BSP problem taking into account the existent pruning strategies and the bidirectional search to reduce the number of iterations that the proposed algorithm performs when solving the one-to-all BSP problem.

4. Computing only the necessary non-dominated labels. Pruning strategies

The computation of the complete non-dominated point set N_t for the *s*-*t* BSP problem does not necessarily require the computation of the complete set N_i for each node i in V-$\{t\}$. Given a node i in V-$\{t\}$, an ideal situation would only require computation of those non-dominated labels associated with node i that correspond to efficient sub-paths contained in some efficient path from node s to node t. Several pruning strategies can be used which avoid the computation of labels associated with node i that are not promising to obtain non-dominated labels for node t exist in the literature (see Skriver & Andersen, 1991; Machuca & Mandow, 2012). Duque et al. (2015) used these pruning strategies. An integration of these into the proposed algorithms is now discussed.

We begin with some additional notation. We compute the lexicographically shortest path tree rooted at s considering the $\operatorname{lex\,min}_{p \in P}(c^1(p), c^2(p))$ problem. For this tree, distance labels $d^s_i = (d^{s1}_i, d^{s2}_i)$ for any node i in V are stored. This problem can be solved by adapting a Dijkstra algorithm in such a way that the label of any node i is modified when the distance label d^{s1}_i is improved or when there is a tie in the d^{s1}_ivalues but d^{s2}_i can be improved. Similarly, we compute the lexicographically shortest path tree rooted at t in the inverse graph G considering the $\operatorname{lex\,min}_{p \in P}(c^1(p), c^2(p))$ to obtain $d^t_i = (d^{t1}_i, d^{t2}_i)$ for any node i in V as the pair of lex-min distances from node i to node t. Note that the inverse graph of G is obtained by swapping the role of the set of successors with the set of predecessors. In order to avoid confusion on the notation introduced, we denote by $w^s_i = (w^{s1}_i, w^{s2}_i)$ for any node i in V the distance labels associated to the lexicographically shortest path tree rooted at s considering the $\operatorname{lex\,min}_{p \in P}(c^2(p), c^1(p))$ problem. Similarly, we compute the lexicographically shortest path tree rooted at t in the inverse graph G considering the $\operatorname{lex\,min}_{p \in P}(c^2(p), c^1(p))$ problem to obtain $w^t_i = (w^{t1}_i, w^{t2}_i)$ for any node i in V. In other words, the proposed bidirectional algorithm in Section 5 will need to perform four initial lexicographic shortest path tree computations to obtain the four bicriteria-labels d^s_i,d^t_i,w^s_i and w^t_i for any node i in V. Instead, the unidirectional version introduced in this section, only requires two lexicographic shortest path tree computations to obtain the two bicriteria-labels d^t_i and w^t_i for any node i in V.

4.1. Pruning by nadir points

Consider the *s*-*i* BSP problem for any node i in V and let $d^s_i = (d^{s1}_i, d^{s2}_i)$ and $w^s_i = (w^{s1}_i, w^{s2}_i)$ be the labels introduced previously. The ideal point for the *s*-*i* BSP problem is given by the pair (d^{s1}_i, w^{s2}_i). The *nadir point* for the *s*-*i* BSP problem is given by the pair (w^{s1}_i, d^{s2}_i). This pair of values is the vector in the objective space that establishes an upper bound for each objective. Therefore, in the algorithm it must hold that any distance label (a, b) in $L[i]$ satisfies $a \le w^{s1}_i$ and $b \le d^{s2}_i$; otherwise, we are sure that (a, b) is a dominated label.

A second aim is to prune any candidate label in H associated to node i that follows a label for node t exceeding w^{t1}_s or d^{t2}_s. Thus, any candidate distance label (d_i^1, d_i^2) for a node i plus the optimal pair of distances (d_i^{t1}, w_i^{t2})(ideal point) to reach node t from node i is not dominated by the nadir point (w_s^{t1}, d_s^{t2}) for the s-t BSP problem. In other words, the algorithm only considers candidate labels (d_i^1, d_i^2) for node i satisfying $d_i^1 + d_i^{t1} \le w_s^{t1}$ and $d_i^2 + w_i^{t2} \le d_s^{t2}$. This condition can easily be incorporated in the function *NewCandidateLabel*+ and in the *RelaxationProcess* and its evaluation takes constant time.

4.2. Pruning by efficient set

The proposed algorithm maintains Invariant 1, that is, any candidate label (d_i^1, d_i^2) for node i in H is not dominated by any label in $L[i]$. In this case, we want to prune any candidate label (d_i^1, d_i^2) for node i that leads to labels dominated by some label in $L[t]$. Thus, any candidate distance label (d_i^1, d_i^2) for a node i plus the optimal pair of distances (d_i^{t1}, w_i^{t2})(ideal point) to reach node t from node i must not be dominated by any label in $L[t]$. In other words, the algorithm only considers candidate labels (d_i^1, d_i^2) for node i satisfying that the pair $(d_i^1 + d_i^{t1}, d_i^2 + w_i^{t2})$ is not dominated by the last label in $L[t]$ (see Theorem 2) and this pair is not in $L[t]$ (to avoid duplicate labels in $L[t]$). This pruning strategy can easily be incorporated in the function *NewCandidateLabel*+ and in the *RelaxationProcess*. The incorporation of these strategies does not increase the theoretical complexity of the BDijkstra algorithm.

We include the pseudo code of the function *NewCandidateLabelWithPruning* and the *RelaxationProcessWithPruning* used by the BDijkstra algorithm containing the two previous pruning strategies.

```
Function NewCandidateLabelWithPruning $(i, l^*)$
$d^1 = +\infty$; $d^2 = +\infty$; $l^{new} = Null$;
For $j \in \Gamma_i^-$ do {
    $r = LastLabelArc[i][j]$; $found = False$;
    While $(r \le N_j)$ and $(found == False)$ do {
      $l = L[j][r]$;
      If $(l.d^1 + c_{ji}^1 > l^*.d^1$ and $l.d^2 + c_{ji}^2 < l^*.d^2)$
      //non-dominated candidate label
          If $(l.d^1 + c_{ji}^1 + d_i^{t1} \le w_s^{t1}$ and $l.d^2 + c_{ji}^2 + w_i^{t2} \le d_s^{t2}$ and $(N_t == 0$ or
          $(l.d^1 + c_{ji}^1 + d_i^{t1} > L[t][N_t].d^1$ and $l.d^2 + c_{ji}^2 + w_i^{t2} < L[t][N_t].d^2)))$
          {//pruning strategies
                $found = True$;
                If $((l.d^1 + c_{ji}^1 < d^1)$ or $(l.d^1 + c_{ji}^1 == d^1$ and $l.d^2 + c_{ji}^2 < d^2))$
                //lexmin candidate label
                      {$d^1 = l.d^1 + c_{ji}^1$; $d^2 = l.d^2 + c_{ji}^2$; $l^{new} = (i, d^1, d^2, j, r)$;}
          }
      If $(found == False)$ $r = r + 1$;
    }
    $LastLabelArc[i][j] = r$;
}
Return $l^{new}$;
```

```
Procedure RelaxationProcessWithPruning $(i, H, l^*)$
For all $j \in \Gamma_i^+$ do
    If $((l^*.d^1 + c_{ij}^1 < d_j^1)$ or $(l^*.d^1 + c_{ij}^1 == d_j^1$ and $l^*.d^2 + c_{ij}^2 < d_j^2))$ //relaxation
$(i, j)$
      If $(N_j == 0)$ or $(l^*.d^1 + c_{ij}^1 > L[j][N_j].d^1$ and $l^*.d^2 + c_{ij}^2 < L[j][N_j].d^2)$
      //non-dominated label
          If $(l^*.d^1 + c_{ij}^1 + d_j^{t1} \le w_s^{t1}$ and $l^*.d^2 + c_{ij}^2 + w_j^{t2} \le d_s^{t2}$ and $(N_t == 0$ or
          $(l^*.d^1 + c_{ij}^1 + d_j^{t1} > L[t][N_t].d^1$ and $l^*.d^2 + c_{ij}^2 + w_j^{t2} < L[t][N_t].d^2)))$
          {//pruning strategies
            $d_j^1 = l^*.d^1 + c_{ij}^1$; $d_j^2 = l^*.d^2 + c_{ij}^2$;
            $l = (j, d_j^1, d_j^2, i, N_i)$; // $N_i$ is the position of $l^*$ in $L[i]$
            If $(InH[j] == False)$ {$Insert(l, H)$; $InH[j] = True$;}
            Else $decrease\text{-}key(l, H)$;
        }
```

The BDijkstra algorithm ends when H becomes empty. Demeyer et al. (2013) introduced a different stopping criterion in the unidirectional multiobjective label-setting algorithm given by Martins (1984). Let T be the set of temporary non-dominated labels in this algorithm. Define $(\min^1, \min^2)$ as the point determined by the minimum values among all the labels in T taking into account both objectives separately. The stopping criterion in Demeyer et al. (2013) consists of stopping when $(\min^1, \min^2)$ is dominated by some label in $L[t]$. We note here that the pruning strategies guarantee stopping the BDijkstra algorithm before this criterion is satisfied. The reason is that at the end of the algorithm, $L[t]$ must contain the label $w_s^t = (w_s^{t1}, w_s^{t2})$. However, the value of the first objective for any label in $L[t]$ is less than or equal to w_s^{t1}. In particular, the algorithm in Demeyer et al. (2013) must wait to compute this last label. Therefore, this algorithm might have to wait until $\min^1$becomes greater than or equal to w_s^{t1} for all labels in T. The use of pruning strategies forces the label $w_s^t = (w_s^{t1}, w_s^{t2})$ to be known initially. A label (d_i^1, d_i^2) for node i satisfying that the last label in $L[t]$ dominates $(d_i^1 + d_i^{t1}, d_i^2 + w_i^{t2})$ is never added. In particular, $d_i^1 + d_i^{t1} \le w_s^{t1}$ for any label in H by the pruning by nadir point strategy. In other words, the pruning by nadir point strategy is stronger than the stopping criterion introduced in Demeyer et al. (2013), because it never adds a label in H with a first objective value greater than w_s^{t1}. This stopping criterion is evaluated in each iteration of the algorithm implying an additional computational cost. In Section 6, we will comment on the practical behavior of this stopping criterion.

Another way to reduce the number of iterations in the proposed algorithms when solving the s-t BSP problem is considered in the next section.

5. Bidirectional scheme of the biobjective Dijkstraś algorithm

The basic idea of a bidirectional shortest path algorithm (Pohl, 1971) is to make two simultaneous searches: a forward search starting from the origin node s and backward search from the destination node t. The algorithm keeps two priority queues HF and HB that store the temporary labels from s in the forward search and from t in the backward search in the inverse graph of G. Denote by dF_i the distance label of node i in the forward search and dB_i the distance label of node i in the backward search. For the single objective case, Nicholson (1966) proved that the shortest path is found when the cost of the shortest path found so far (i.e. the minimum sum of the forward and the backward label of the same node) is smaller than or equal to the sum of the minimum in HF and HB. For example, we maintain the length μ of the best path seen so far (initially, $\mu = \infty$). Whenever the algorithm scans an arc (i, j) in the forward search and node j was scanned in the backward search, it updates μ if $dF_i + c_{ij} + dB_j < \mu$. A similar operation is made when scanning an arc in the backward search. The stopping criterion holds when the sum of the minimum labels in HF and HB is greater than or equal to μ (Goldberg & Harrelson, 2005).

Our bidirectional BDijkstra algorithm uses two priority queues HF and HB for each one of the search directions (forward and backward, respectively). In the forward search, the algorithm keeps the set of labels $LF[i]$ for all i in V. In the backward search starting from node t, the algorithm uses the set of labels $LB[i]$ for all i in V. In an iteration, one label from HF (whenever it exists) and one label from HB (whenever it exists) are extracted. Thus the algorithm performs one step in the forward search and one step in the backward search in any iteration. Additionally, the algorithm uses a set of labels $L_{solution}$ where all non-dominated points of the efficient s-t paths are stored. In a forward step, if a new label l is added to $LF[i]$ and $LB[i]$ is not empty, this new label l is combined with all labels of $LB[i]$ in order to form complete paths between node s and node t. The non-dominated labels obtained are then added to $L_{solution}$. Moreover, any label in $L_{solution}$ dominated by any added label must be deleted from $L_{solution}$. In this case, we need a function *Admissible* $((a, b), L_{solution})$ that returns *True* when the pair of tentative distances (a, b) is not in $L_{solution}$ and it is not dominated by any label in $L_{solution}$. Otherwise, the *Admissible* $((a, b), L_{solution})$ returns *False*. The function *Admissible* $((a, b), L_{solution})$

requires $O(\log|L_{solution}|)$ time making a binary search on $L_{solution}$ (this list keeps a lexicographic order on the labels).

```
Bidirectional BDijkstra with Pruning (BBDijkstra) Algorithm;
(1)  Compute distances $d_i^s = (d_i^{s1}, d_i^{s2})$ for all nodes $i$ in $V$ // see section 4
(2)  Compute distances $d_i^t = (d_i^{t1}, d_i^{t2})$ for all nodes $i$ in $V$ // see section 4
(3)  Compute distances $w_i^s = (w_i^{s1}, w_i^{s2})$ for all nodes $i$ in $V$ // see section 4
(4)  Compute distances $w_i^t = (w_i^{t1}, w_i^{t2})$ for all nodes $i$ in $V$ // see section 4
(5)  $CreateHeap(HF)$; $CreateHeap(HB)$; $L_{solution} = \emptyset$;
(6)  Set$NF_i = 0$; $dF_i^1 = +\infty$; $dF_i^2 = +\infty$; $InHF[i] = False$; for all $i \in V - \{s\}$;
(7)  Set$NB_i = 0$; $dB_i^1 = +\infty$; $dB_i^2 = +\infty$; $InHB[i] = False$; for all $i \in V - \{t\}$;
(8)  $NF_s = 0$; $dF_s^1 = 0$; $dF_s^2 = 0$; $l = (s, 0, 0, -, -)$; $Insert(l, HF)$; $InHF[s] = True$;
(9)  $NB_t = 0$; $dB_t^1 = 0$; $dB_t^2 = 0$; $l = (t, 0, 0, -, -)$; $Insert(l, HB)$; $InHB[t] = True$;
(10) While $(HF \neq \emptyset)$ and $(HB \neq \emptyset)$ do
(11)     // forward direction
(12)     $l^* =$ Find-min($HF$); Delete-min($HF$);
(13)     $NF_i = NF_i + 1$; $LF[i][NF_i] = l^*$; $InHF[i] = False$; // $i$ is the node with label $l^*$
(14)     For $l \in LB[i]$ do
(15)         If $Admissible((l^*.d^1 + l.d^1, l^*.d^2 + l.d^2), L_{solution})$)
(16)             Add $l^* + l$ to $L_{solution}$; Remove from $L_{solution}$ any label dominated by $l^* + l$;
(17)     $l^{new} = NewCandidateLabelWithPruning\ (i, l^*)$;
(18)     If $(l^{new} \neq NULL)$ {Insert $(l^{new}$,HF);
             $InHF[i] = True$;$dF_i^1 = l^{new}.d^1$; $dF_i^2 = l^{new}.d^2$;}
(19)     $RelaxationProcessWithPruning\ (i, HF, l^*)$;
(20)     // backward direction
(21)     $l^* =$ Find-min($HB$); Delete-min($HB$);
(22)     $NB_i = NB_i + 1$; $LB[i][NB_i] = l^*$; $InHB[i] = False$; // $i$ is the node with label $l^*$
(23)     For $l \in LF[i]$ do
(24)         If $Admissible((l^*.d^1 + l.d^1, l^*.d^2 + l.d^2), L_{solution})$)
(25)         Add $l^* + l$ to $L_{solution}$; Remove from $L_{solution}$ any label dominated by $l^* + l$;
(26)     $l^{new} = NewCandidateLabelWithPruningBackward\ (i, l^*)$;
(27)     If $(l^{new} \neq NULL)$ {Insert $(l^{new}$,HB);
             $InHB[i] = True$;$dB_i^1 = l^{new}.d^1$; $dB_i^2 = l^{new}.d^2$;}
(28)     $RelaxationProcessWithPruningBackward\ (i, HB, l^*)$;
```

The stopping criterion used in our proposed algorithms is to stop when HF or HB becomes empty. Finally, we use $LastLabelArcF[i][j]$ for all arcs $(j, i) \in A$ in the forward search and $LastLabelArcB[i][j]$ for all arcs $(i, j) \in A$ in the backward search. A description of the pseudo-code of the bidirectional BDijkstra algorithm is included with the aim of giving a complete description of the proposed algorithm.

```
Function NewCandidateLabelWithPruningBackward $(i, l^*)$
 $d^1 = +\infty$; $d^2 = +\infty$; $l^{new} = Null$;
For $j \in \Gamma_i^+$ do {
    $r = LastLabelArcB[i][j]$; $found = False$;
    While $(r \leq NB_j)$ and $(found == False)$ do {
      $l = LB[j][r]$;
      If $(l.d^1 + c_{ij}^1 > l^*.d^1$ and $l.d^2 + c_{ij}^2 < l^*.d^2)$
      //non-dominated candidate label
        If $(l.d^1 + c_{ij}^1 + d_i^{s1} \leq w_t^{s1}$ and $l.d^2 + c_{ij}^2 + w_i^{s2} \leq d_t^{s2}$ and
        $Admissible((l.d^1 + c_{ij}^1 + d_i^{s1}, l.d^2 + c_{ij}^2 + w_i^{s2}), L_{solution})$) {
        //pruning strategies
            $found = True$;
            If $((l.d^1 + c_{ij}^1 < d^1)$ or $(l.d^1 + c_{ij}^1 == d^1$ and $l.d^2 + c_{ij}^2 < d^2))$
               //lexmin candidate label
                 $d^1 = l.d^1 + c_{ij}^1$; $d^2 = l.d^2 + c_{ij}^2$; $l^{new} = (i, d^1, d^2, j, r)$;
         }
      If $(found == False)$ $r = r + 1$;
    }
    $LastLabelArcB[i][j] = r$;
}
Return $l^{new}$;
```

```
Procedure RelaxationProcessWithPruningBackward $(i, HB, l^*)$
For all $j \in \Gamma_i^-$ do
   If $((l^*.d^1 + c_{ji}^1 < dB_j^1)$ or $(l^*.d^1 + c_{ji}^1 == dB_j^1$ and $l^*.d^2 + c_{ji}^2 < dB_j^2))$
   //relaxation $(j, i)$
    If$(NB_j == 0$ or $(l^*.d^1 + c_{ji}^1 > LB[j][NB_j].d^1$ and $l^*.d^2 + c_{ji}^2 < LB[j][NB_j].d^2))$
    //non-dominated label
     If $(l^*.d^1 + c_{ji}^1 + d_j^{s1} \leq w_t^{s1}$ and $l^*.d^2 + c_{ji}^2 + w_j^{s2} \leq d_t^{s2}$ and
     $Admissible((l^*.d^1 + c_{ji}^1 + d_j^{s1}, l^*.d^2 + c_{ji}^2 + w_j^{s2}), L_{solution})$)
     //pruning strategies {
      $dB_j^1 = l^*.d^1 + c_{ji}^1$; $dB_j^2 = l^*.d^2 + c_{ji}^2$;
      $l = (j, dB_j^1, dB_j^2, i, NB_i)$; // $NB_i$ is the position of $l^*$ in $LB[i]$
      If $(InHB[j] == False)$ {$Insert(l, HB)$; $InHB[j] = True$;}
      Else $decrease\text{-}key(l, HB)$;
     }
```

In the pseudo-code of BBDijkstra, the function *NewCandidateLabelWithPruning* and the *RelaxationProcessWithPruning* used in the forward direction are basically the same given in Section 4.2 where L is replaced by LF, H by HF, $LastLabelArc\ [i][j]$ by $LastLabelArcF[i][j]$, and $L[t]$ is replaced by $L_{solution}$. Now, the function *Admissible* must be called to prune using the efficient set strategy. For a complete description of the proposed algorithm, the pseudo-code of the function *NewCandidateLabelWithPruningBackward* and the *RelaxationProcessWithPruningBackward* are presented.

These functions incorporate the pruning strategies taking into account the ideal point for the paths starting at s to node i given by (d_i^{s1}, w_i^{s2}) and the nadir point to reach node t from node s given by the pair (w_t^{s1}, d_t^{s2}). As a result two pointers/positions are needed to derive a path for each label in $L_{solution}$ associated with node i. One pointer is to the label in $LF[j]$ where j is the predecessor node of node i in the path from s to i. The second pointer is to the label in $LB[k]$, being k the predecessor node of i in the path from t to i in the inverse graph G (alternatively k is the successor node of i in the path from i to t in G). Therefore the labels here have one additional field. The exact scheme is omitted here for the sake of brevity. In this case, Procedure *PrintPath* is called twice to determine the sub-paths s-i and i-t per label in $L_{solution}$.

Demeyer et al. (2013) enhanced the algorithm of Martins (1984) by considering a bidirectional search in this classical multiobjective label-setting method. Furthermore, Demeyer et al. (2013) introduced a new stopping criterion on the two simultaneous searches. Let TF and TB be the set of temporary labels in the forward and backward direction, respectively. Define $(\text{minF}^1, \text{minF}^2)$ and $(\text{minB}^1, \text{minB}^2)$as the points determined by the minimum values among all the labels in TF and TB, respectively, taking into account both objectives separately. The stopping criterion in Demeyer et al. (2013) consists of stopping when $(\text{minF}^1+\text{minB}^1, \text{minF}^2+\text{minB}^2)$ is dominated by some label in $L_{solution}$. We do not consider this stopping criterion in our algorithm because a label (d_i^1, d_i^2) for node i satisfying that the pair $(d_i^1 + d_i^{t1}, d_i^2 + w_i^{t2})$ is dominated by a label in $L_{solution}$ is never added in the forward direction. Similarly, (d_i^1, d_i^2) is never added in the backward search if $({d_i}^1 + d_i^{s1}, {d_i}^2 + w_i^{s2})$ is dominated by a label in $L_{solution}$. Following similar arguments as in Section 4.2, it is possible to show that the pruning by the nadir point strategy is stronger than the stopping criterion introduced in Demeyer et al. (2013).

In the next section, we examine the performance of the unidirectional and bidirectional BDijkstra algorithm.

6. Computational results

We investigate the computational performance of the proposed unidirectional BDijkstra (BDijkstra) and bidirectional BDijkstra (BBDijkstra) algorithms with pruning strategies, introduced in Section 4, when solving the one-to-one BSP problem. We compare its running times with the state-of-the-art methods. The first method considered is the PULSE algorithm. In Duque et al. (2015),

Table 1
Characteristics of road network instances.

Name	Description	# nodes	# 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
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

the PULSE algorithm was compared against the bounded label-setting method (bLSET) by Raith (2010) which is among the best performers for the BSP on real road networks. The PULSE algorithm was faster than bLSET on 123 out of 150 very-large scale instances.

The PULSE algorithm is based on a depth first search (DFS) starting at node s on the graph. This recursive algorithm travels through the entire network storing the partial path (the path from s to the current node reached by a pulse). Each time the node t is recursively reached, it obtains a feasible solution that might be efficient. The PULSE algorithm enumerates all possible paths from node s to the rest of nodes in the graph when the pruning strategies are not considered. In order to avoid the computation of excessive dominated s-t paths, the PULSE algorithm applies the pruning strategies before expanding the partial path with a new non-visited node. The reader is referred to Duque et al. (2015) for more details on the algorithm.

We have also implemented a unidirectional (BLset) and a bidirectional label-setting (BBLset) algorithm including the stopping criterion introduced by Demeyer et al. (2013). A preliminary experiment reveals that these algorithms needed a significant amount of time to solve the road network instances. Pruning strategies given in Section 4 are then incorporated into these algorithms. The result was that the classical label-setting algorithms with pruning strategies and without the stopping criterion in Demeyer et al. (2013) are faster than label-setting algorithms including the stopping criterion. This fact is the first relevant conclusion of our study, and is the reason why we compare our proposed algorithm with a standard label-setting algorithm incorporating the same pruning strategies. Thus, our experiment considers the unidirectional algorithms PULSE, BLset and BDijkstra and the bidirectional algorithms BBLset and BBDijkstra. All of them use the same pruning strategies. Computational experiments are conducted on an Intel Xeon 3.70 Gigahertz x 8 with 64 gigabyte of RAM running Ubuntu 16.04. All the algorithms were written in C and compiled with option O3. The algorithms were limited to 3600 seconds for the solution of each instance.

6.1. Instance sets

We use large road networks from the DIMACS Shortest Path Implementation Challenge (2013) for our computational experiments. Characteristics of the road network instances are shown in Table 1. It should be noted that these road network instances from NE, CAL and LKS are larger than those considered in Duque et al. (2015). The arc costs represent distance and travel time. We randomly chose 100 different origin-destination pairs since the performance of the algorithms depends on how far origin and destination nodes are apart. All algorithms are tested on the same set of origin-destination pairs.

Grid networks represent rectangular grids with arcs between each node and its immediate neighbors in the grid, arc costs are selected randomly between 1 and 10. The source node and the target node are beyond the grid structure. The source node $s=1$ is connected to all the nodes in the left margin and the node target $t=n$ is connected to all the nodes in the right margin. We use problem instances with grid height and width as shown in Table 2 tested in Sedeño-Noda and Raith (2015). We obtain 49 observations per algorithm using this set of instances.

We also have considered the NetMaker generator proposed by Skriver and Andersen (2000). We used 67 instances with n varying from 5000 to 30,000. For each one of them, we chose $s=1$ and $t=n$. All the algorithms solving these instances used at most 0.6 seconds. The results of this experiment are not included here for the sake of brevity. However, the full data collected from these three experiments appear in the ExperimentData file as supplementary material. In each table of this file, we show the CPU times in seconds used for each algorithm in the resolution of the corresponding s-t BSP problem. The number of non-dominated points N_t are also reported.

6.2. Results

This section summarizes the data collected from the experimentation. We start considering the experiment with road network instances. Results are shown in Table 3 where average, minimum and maximum observed running times are given (best average running times appear in bold). In addition, Table 3 includes the number of instances solved within the time limit by the corresponding algorithm from the 100 instances considered in each road network.

We note that the average performance of BDijkstra and BBDijkstra algorithms are the best. For example, the average times on all road instances of BDijkstra and BBDijkstra algorithm are 290.45 and 305.51 seconds, respectively. Furthermore, the BDijkstra algorithm solves the largest number of instances in the experiment. For example, the BDikjstra successfully solved 670 instances out of 700 in the experiment (96%) and the BBDijkstra solved 665 (95%). Next, the BLSet algorithm is the third best on average (480.56 seconds) and is the third algorithm in solved problems (644/700 = 92%). The BBLset algorithm appears next in this ranking with an average time of 509.42 seconds. The percentage of instances solved by this algorithm is 91%. However the PULSE algorithm is always the slowest algorithm (average time of 1243.57 seconds) and solves the least number of problems (483/700 = 69%). Note that this is an advantage to the computed average times that appear in Table 3. Each non-solved instance contributes 3600 seconds instead of larger times to the average times. However, the minimal running times of PULSE are always the shortest. Also note that the differences among the min CPU times are not significant when we compare these differences with the differences among the average CPU times. The PULSE algorithm performs well when the number of non-dominated points is small. This fact is better understood when we further analyze the problems by N_t as shown in Figs. 1–3, for the NY, COL and CAL instances, respectively. Clearly, the running times of all the algorithms increase as N_t increases.

In Fig. 1, we observe that the PULSE algorithm needs considerably more time than the others when the values of N_t are greater than 100. The PULSE algorithm beats the other algorithms for instances with N_t smaller than 50 in the NY road network. Alternatively, the BBDijkstra algorithm is the best for any instance with N_t bigger than 110 in the NY road network. The second best algorithm is the BDijkstra algorithm.

In the case of the COL road network, Fig. 2, the PULSE algorithm beats the other algorithms for any instance with N_t smaller than 90. However, note that N_t ranges from 1 to approximately 2600 in the COL road network. Observe that the start points associated with the PULSE algorithm disappear from Fig. 2 when N_t is greater than 750. For the CAL road network (Fig. 3), the PULSE algorithm beats the other algorithms for any instance with N_t smaller than 150. However, note that N_t ranges from 1 to approximately

Table 2
Characteristics of grid instances.

Name	Height	Width	Number of nodes	Number of arcs
G1-G7	300	300,350,...,600	90,002 to 180,002	359,400 to 718,800
G8-G14	350	300,350,...,600	105,002 to 210,002	419,400 to 838,800
G15-G21	400	300,350,...,600	120,002 to 240,002	479,400 to 958,800
G22-G28	450	300,350,...,600	135,002 to 270,002	539,400 to 1,078,800
G29-G35	500	300,350,...,600	150,002 to 300,002	599,400 to 1,198,800
G36-G42	550	300,350,...,600	165,002 to 330,002	659,400 to 1,318,800
G43-G49	600	300,350,...,600	180,002 to 360,002	719,400 to 1,438,800

Table 3
Computational results on road networks: no. of solved problems in 3600 seconds/100, average, minimum and maximum CPU time and number of non-dominated points (N_t).

		N_t	BBDijkstra	BBLset	BDijkstra	BLset	PULSE
NY	Solved/100	100/100	100/100	100/100	100/100	100/100	98/100
	Avg.	119.79	**0.74**	4.17	1.32	3.78	119.71
	Min	1	0.22	0.20	0.15	0.14	0.10
	Max	646	8.46	84.84	21.75	69.48	1137.92
BAY	Solved/100	100/100	100/100	100/100	100/100	100/100	98/100
	Avg.	143.77	**1.28**	5.08	2.16	6.29	234.87
	Min	1	0.27	0.26	0.18	0.19	0.11
	Max	825	16.39	99.92	33.42	102.38	2714.16
COL	Solved/100	100/100	100/100	100/100	100/100	100/100	86/100
	Avg.	346.51	**10.89**	47.01	12.20	39.08	657.88
	Min	1	0.36	0.35	0.25	0.24	0.16
	Max	2612	255.25	1087.77	355.57	1126.05	2442.59
FLA	Solved/100	100/100	100/100	98/100	99/100	97/100	69/100
	Avg.	673.72	**83.86**	314.32	129.21	310.28	1219.87
	Min	2	0.93	0.85	0.62	0.62	0.40
	Max	6292	1596.66	3559.18	1626.24	3477.06	2349.48
NE	Solved/100	99/100	99/100	93/100	99/100	97/100	49/100
	Avg.	808.19	246.95	581.36	**199.83**	516.28	1960.60
	Min	7	1.56	1.52	1.05	1.00	0.62
	Max	3145	3414.14	2873.28	1308.06	3496.64	2063.56
CAL	Solved/100	99/100	98/100	93/100	98/100	94/100	58/100
	Avg.	862.54	**216.99**	654.46	267.29	587.30	1706.22
	Min	1	1.90	1.79	1.24	1.32	0.73
	Max	6962	2543.59	3466.02	2786.61	3438.27	2247.70
LKS	Solved/100	74/100	68/100	55/100	74/100	56/100	25/100
	Avg.	1917.80	1577.87	1959.56	**1421.14**	1900.95	2805.81
	Min	1	2.93	2.90	1.92	2.00	1.09
	Max	7547	3560.20	3284.93	3286.70	3047.47	3186.06

Average time is calculated with a computational time of 3600 seconds for unsolved instances.
Maximum computational time of the solved instances.

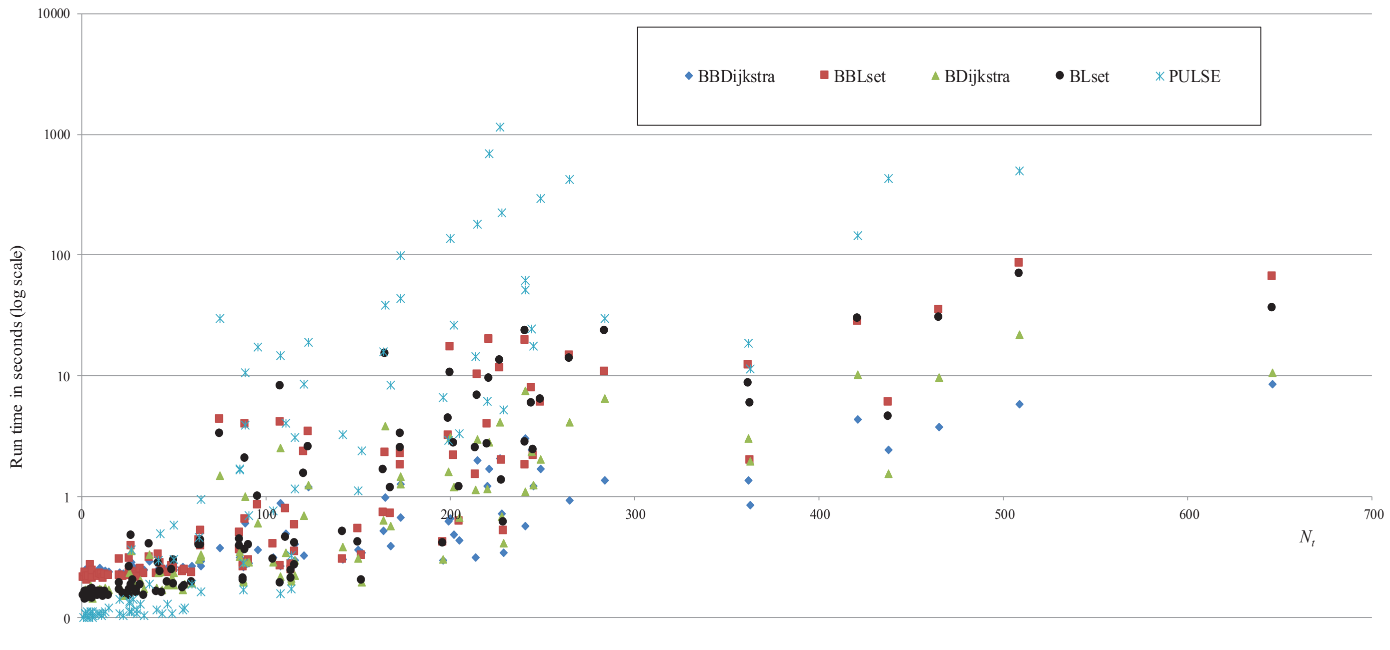

Fig. 1. Running times vs. number of non-dominated solutions for the NY road instances.

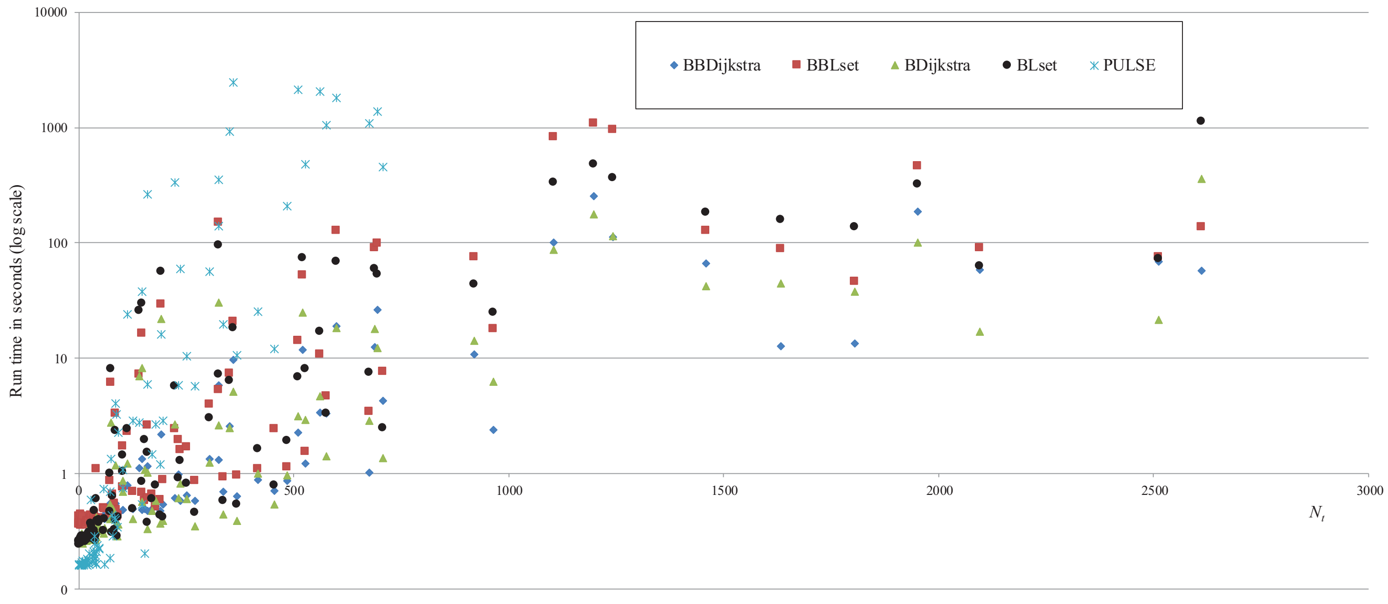

Fig. 2. Running times vs. number of non-dominated solutions for the COL road instances.

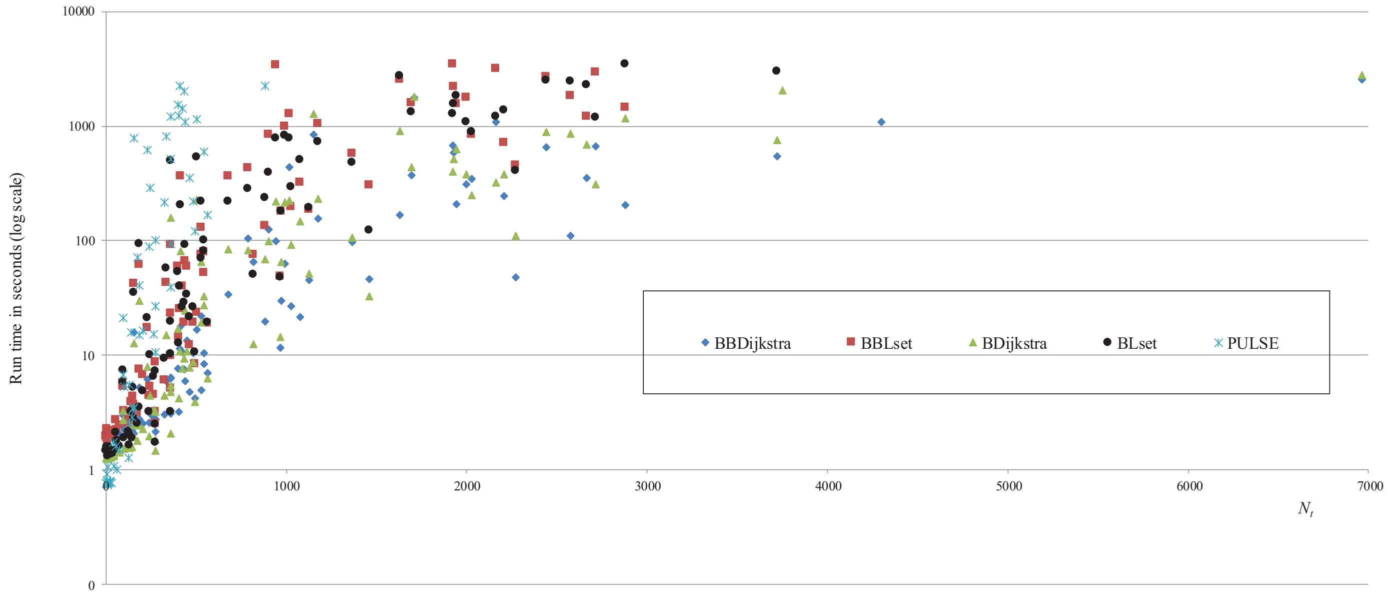

Fig. 3. Running times vs. number of non-dominated solutions for the CAL road instances.

4300 in this case. Again, the marker associated with the PULSE algorithm disappears from Fig. 3 when N_t is greater than 900. Figs. 1–3 allow the differences between the CPU times employed by the algorithms to be observed (note that the CPU times appear in log scale). Clearly, the best algorithm is the BBDijkstra and the worst is the PULSE algorithm. The second best is the BDijkstra algorithm. Identifying the order for the remaining three algorithms depends on the values of N_t. For this reason, as in Duque et al. (2015), we categorized the 100 instances generated from each network into three groups according to N_t. To do so, we sorted the instances in increasing order of N_t and allocated the first 34 instances into the low group (L), the next 33 into the medium group (M), and the remaining 33 into the high group (H).

Analyzing the results by low, medium and high number of non-dominated points (Table 4), we see that the performance of the BB-Dijkstra is superior for the more difficult problems with a medium and high number of non-dominated points. The PULSE algorithm is the best for low groups in the NY, BAY and COL road networks. However, the differences among the average CPU times are less than 0.2 seconds in this case. Note that the PULSE algorithm is only capable to determine the true set of non-dominated points for low groups. In summary, the BBDijkstra and BDijkstra algorithms are the two best algorithms in the remaining cases and the BBDijkstra algorithm outperforms the BDijkstra algorithm in the majority of the cases.

From this experiment, the BBDijkstra algorithm is the best choice to solve the BSP problem in all road networks with the exceptions of NE and LKS roads networks. The BDijkstra algorithm is the best choice for NE and LKS road networks.

Observed results for the grid network instances are now considered. Surprisingly, results are very different for grid networks. We illustrate runtimes for the different instances in Fig. 4 and summarize the observed results in Table 5. The problems have between 403 and 991 non-dominated points. The grid network

Table 4
Road network average runtimes: by number of efficient solutions (N_t).

	N_t		BBDijkstra		BBLset		BDijkstra		BLset		PULSE	
	G	Range	CPU	S	CPU	S	CPU	S	CPU	S	CPU	S
NY	L	1–41	0,24	34	0,24	34	0,18	34	0,18	34	**0,12**	34
	M	42–142	**0,35**	33	0,86	33	0,43	33	0,81	33	3,80	33
	H	150–646	**1,63**	33	11,52	33	3,39	33	10,46	33	358,83	31
BAY	L	1–47	0,28	34	0,28	34	0,21	34	0,21	34	**0,14**	34
	M	47–124	**0,46**	33	0,97	33	0,52	33	0,89	33	10,42	33
	H	126–825	**3,14**	33	14,12	33	5,82	33	17,96	33	701,17	31
COL	L	1–47	0,39	34	0,41	34	0,29	34	0,30	34	**0,20**	34
	M	48–251	**0,63**	33	2,69	33	1,76	33	4,49	33	351,11	30
	H	271–2612	**31,96**	33	139,35	33	34,90	33	113,62	33	1969,54	19
F LA	L	2–139	**1,15**	34	2,08	34	1,43	34	2,58	34	2,58	34
	M	156–633	**3,02**	33	12,56	33	4,85	33	14,34	33	460,72	30
	H	636–6292	**249,93**	33	937,78	31	385,21	32	923,24	30	3233,19	5
NE	L	7–276	2,03	34	3,81	34	**1,97**	34	4,01	34	25,99	34
	M	278–886	**43,88**	33	240,42	33	54,47	33	184,44	33	2314,44	15
	H	930–3145	702,37	32	1517,34	26	**549,04**	32	1703,18	27	3600,00	0
CAL	L	1–239	**2,80**	34	6,35	34	3,00	34	6,57	34	50,84	34
	M	247–903	**16,86**	33	94,34	33	33,83	33	102,43	33	1517,99	24
	H	941–6962	**637,79**	31	1882,32	26	773,05	31	1670,49	27	3600,00	0
LKS	L	1–1314	**39,50**	34	98,09	34	133,47	34	123,91	34	1370,03	24
	M	1315–4603	1316,38	30	2237,01	21	**834,74**	33	2032,80	22	3600,00	0
	H	4833–7547	3549,68	4	3600,00	0	**3431,71**	7	3600,00	0	3600,00	0

Average time is calculated with a computational time of 3600 seconds for unsolved instances.

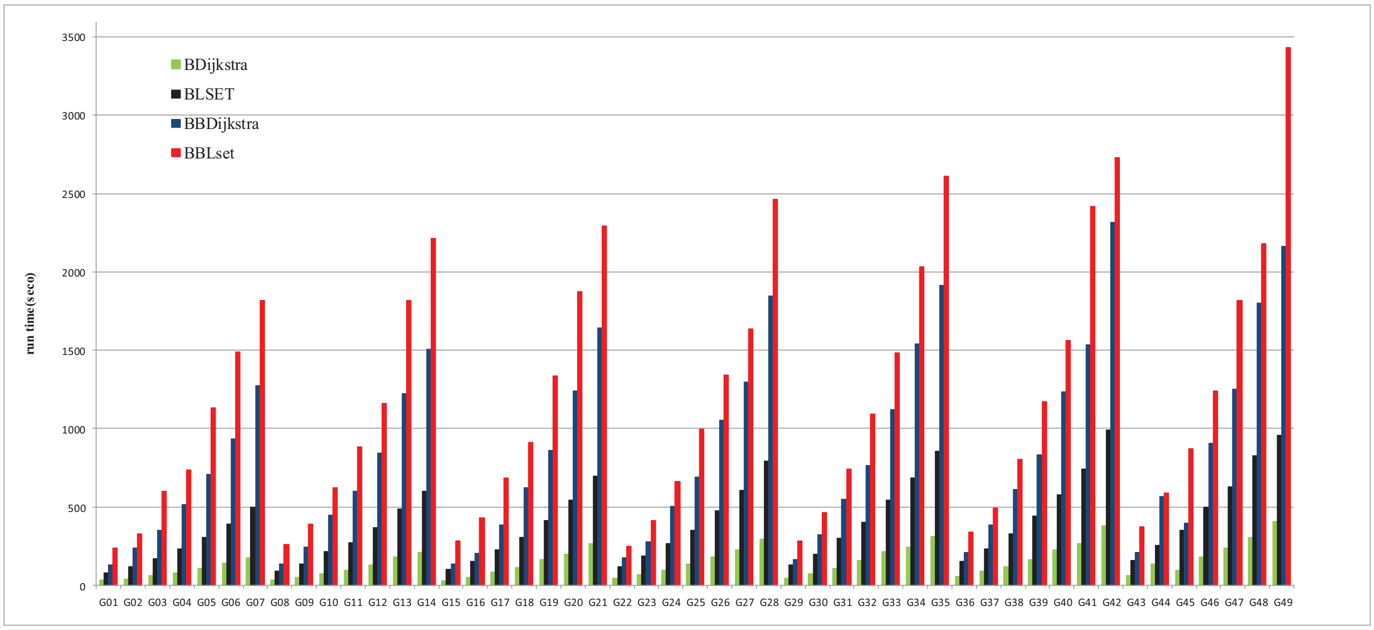

Fig. 4. Run time of the algorithms for grid network instances.

Table 5
Summary of the computational results on grid instances: average, min and max of the CPU time, no. of solved problems in 3600 seconds/100 and average, min and max of N_t.

Algorithm	Average CPU time	Max CPU time	Min CPU time	#Solved	Average N_t	Min N_t	Max N_t
BBDijkstra	833.74	2315.50	126.62	49	706.29	403	991
BBLset	1183.42	3431.30	237.38	49	706.29	403	991
BDijkstra	**147.12**	408.25	27.48	49	706.29	403	991
BLset	395.88	990.46	78.00	49	706.29	403	991
PULSE	3600.00			0			

Average time is calculated with a computational time of 3600 seconds for unsolved instances.

instances were harder to solve than the road network instances. In particular, the PULSE algorithm was unable to solve any of these instances. The best algorithm is BDijkstra, followed by BLSet. The bidirectional versions of these algorithms are worse than their unidirectional counterparts. The observed behavior may be explained by the fact that the grid networks have a very peculiar network structure, and arc cost functions that do not reflect real-world costs (Table 5).

7. Conclusions

An initial study of the classical label-selection methods for the biobjective SP problem revealed that neither of them keep one candidate label per node only as the classical Dijkstra algorithm for the single objective case. We saw an opportunity to reduce the computational effort employed in the merge operations among the candidate labels and the examined non-dominated labels in the existent label-selection methods. To do so, once the algorithm extracts from the corresponding priority queue its candidate label, the proposed algorithm uses the function *NewCandidateLabel*() to obtain the next candidate label for this node. In this way, we obtain a novel Dijkstra-like method to find all non-dominated points of BSP problems. The proposed biobjective Dijkstra algorithm determines the non-dominated points of the one-to-all BSP problems. Moreover, the space needed by the algorithm is $O(N+m+n)$, which is minimal. We prove that its running time is $O(N\log n + mN_{\max})$ with an additional $O(m)$ space.

The one-to-one BSP problem is then considered. We take into account the existent pruning strategies and the bidirectional search to reduce the number of iterations that the proposed algorithm performs when solving the one-to-all BSP problem. The pruning strategies also ideally lead to compute those non-dominated labels associated with any node i that correspond to efficient sub-paths in some efficient path from node s to node t. Moreover, these pruning strategies avoid the computation of labels associated with node i that are not promising to obtain non-dominated labels for node t. The result is a fast and practical algorithm to solve the one-to-one BSP problem in large networks. For example, the proposed BBDijkstra (bidirectional version) and BDijkstra (unidirectional version) algorithms clearly outperform the PULSE algorithm given in Duque et al. (2015), being this last algorithm one of the state-of the-art algorithms to solve the BSP problem in large road networks. This claim is supported by the experimental study included in this paper. Additionally, the proposed algorithms are faster than the classical unidirectional and bidirectional label-setting algorithms including pruning strategies. This last fact confirms that keeping one candidate label per node only is a good choice when solve one-to-one BSP problem.

Future lines of research extend to experimentation with multiobjective SP problem and to parallelize the proposed algorithm with similar techniques and tools as used for the Dijkstra algorithm for the single objective problem. We also want to modify the function *NewCandidateLabel*() to obtain a new Fully Polynomial Time Approximation Scheme (FPTAS) for the MSP problem (see Breugem, Dollevoet, & Van den Heuvel, 2017).

Acknowledgments

We would like to thank Senior Lecturer Andrea Raith for her generosity in sharing with us her grid and NetMaker instances. We would also like to thank the referees for their insightful comments that contributed to improve this paper. This work has been partially supported by Spanish Government Research Project MTM2016-74877-P.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2019.01.007.

References

Ahuja, R., Magnanti, T., & Orlin, J. B. (1993). *Network flows: theory, algorithms and applications*. Englewood Cliffs, NJ: Prentice-Hall.

Breugem, T., Dollevoet, T., & Van den Heuvel, W. (2017). Analysis of FPTASes for the multi-objective shortest path problem. *Computers and Operations Research, 78*, 44–58.

Captivo, M. E., Clímaco, J. N., Figueira, J. R., Martins, E. Q., & Santos, J. L. (2003). Solving bicriteria 0-1 knapsack problems using a labeling algorithm. *Computers and Operations Research, 30*, 1865–1886.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. (3rd ed.). Cambridge, MA: The MIT Press.

Demeyer, S., Goedgebeur, J., Audenaert, P., Pickavet, M., & Demeester, P. (2013). Speeding up Martins' algorithm for multiple objective shortest path problems. *4OR - A Quarterly Journal of Operations Research, 11*(4), 323–348.

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

DIMACS (2013). *9th DIMACS implementation challenge - shortest path* Accessed May http://www.dis.uniroma1.it/challenge9/download.shtml.

Duque, D., Lozano, L., & Medaglia, A. L. (2015). An exact method for the biobjective shortest path problem for large-scale road networks. *European Journal of Operational Research, 242*(3), 788–797.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM, 34*(3), 596–615.

Galand, L., Ismaili, A., Perny, P., & Spanjaard, O. (2013). Bidirectional preference-based search for state graph problems. In Proceedings of the 6th international symposium on combinatorial search *(SOCS 2013)* (pp. 80–88).

Goldberg, A. V., & Harrelson, C. (2005). Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms: 1* (pp. 156–165).

Guerriero, F., & Musmanno, R. (2001). Label correcting methods to solve multicriteria shortest path problems. *Journal of Optimization Theory and Applications, 111*, 589–613.

Hansen, P. (1980). Bicriterion path problems. In *Proceedings of the 3rd international conference on multiple criteria decision making, theory and application, Hagen/Königswinter 1979: 177* (pp. 109–127). Berlin: Springer Verlag.

Machuca, E., & Mandow, L. (2012). Multiobjective heuristic search in road maps. *Expert Systems with Applications, 39*, 6435–6445.

Martins, E., & Clímaco, J. (1981). On the determination of the nondominated paths in multiobjective network problem. *Methods in Operations Research, 40*, 255–258.

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

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

Nicholson, J. A. T. (1966). Finding the shortest route between two points in a network. *The Computer Journal, 9*(3), 275–280.

Paixão, J., & Santos, J. (2013). Labeling methods for the general case of the multi--objective shortest path problem – A computational study. *Computational intelligence and decision making* (pp. 489–502). Netherlands: Springer.

Pohl, I. (1971). Bi-directional Search, *Machine intelligence* (6, pp. 124–140). Edinburgh: Edinburgh Univ. Press.

Raith, A., & Ehrgott, M. (2009). A comparison of solution strategies for biobjective shortest path problems. *Computers and Operations Research, 36*, 1299–1331.

Raith, A. (2010). Speed-up of labeling algorithms for biobjective shortest path problems. In *Proceedings of the 45th annual conference of the ORSNZ* (pp. 313–322). Operations Research Society of New Zealand.

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

Skriver, A. J. V., & Andersen, K. A. (2000). A label correcting approach for solving bicriterion shortest-path problems. *Computers and Operations Research, 27*, 507–524.

Stewart, B. S., & White, C. C. (1991). Multiobjective A*. *Journal of the ACM, 38*, 775–814.