Dynamic preprocessing for the minmax regret robust shortest path problem with finite multi-scenarios

Marta M. B. Pascoal[*], Marisa Resende

August 22, 2015

Department of Mathematics, University of Coimbra
Apartado 3008, EC Santa Cruz, 3001-501 Coimbra, Portugal
Phone: +351 239 791150, Fax: +351 239 832568

Institute for Systems Engineering and Computers – Coimbra (INESCC)
Rua Antero de Quental, 199, 3000-033 Coimbra, Portugal

E-mails: `{marta, mares}@mat.uc.pt`

Abstract

The minmax regret robust shortest path problem aims at finding a path that minimizes the maximum deviation from the shortest paths over all scenarios. It is assumed that different arc costs are associated with different scenarios. This paper introduces a technique to reduce the network, before a minmax regret robust shortest path algorithm is applied. The preprocessing method enhances others explored in previous research. The introduced method acts dynamically and allows to update the conditions to be checked as new network nodes that can be discarded are identified. Computational results on random and Karasan networks are reported, which compare the dynamic preprocessing algorithm and its former static version. Two robust shortest path algorithms as well as the resolution of a mixed integer linear formulation by a solver are tested with and without these preprocessing rules.

Keywords: Robust shortest path, Discrete scenarios, Dynamic preprocessing.

1 Introduction

One approach for dealing with costs uncertainty is to consider several possible scenarios. In the case of the shortest path problem this is done either by associating a discrete set of costs with each arc, or by assuming each arc cost varies within an interval. In this paper, the former case is considered for the minimax regret robust shortest path problem, here simply called robust shortest path problem. This problem consists of finding a path between two nodes of a network, which minimizes the maximum regret cost of each path towards the shortest path, for all scenarios.

Yu and Yang [12] and, more recently, Pascoal and Resende [10], developed algorithms for the robust shortest path problem. Later, inspired by the works of Karasan, Pinar and Yaman [5] and then Catanzaro, Labbé and Salazar-Neumann [3], for the interval data case, Pascoal and Resende

---

[*] Corresponding author

[11] presented theoretical results and algorithms that allow to reduce the network before a robust shortest path algorithm is applied. These preprocessing techniques can identify a priori arcs that are certainly part of any optimal solution, as well as nodes that do not belong to any optimal solution, in order to be deleted later.

The goal of this work is to enhance the preprocessing strategy developed in [11] for nodes. The improvement consists in developing a dynamic rule, in the sense that it is updated as the preprocessing algorithm runs and paths are computed. The idea behind this improvement is to further reduce the network before a robust shortest path algorithm is applied, that is, to increase the number of detected nodes that do not belong to any optimal solution. The latest aspect concerns limiting the number of scenarios to consider in the tests, and thus to save computational time. Empirical experiments compare the new rules with the former. Even though the extension of the rule introduced in [11] for detecting arcs in optimal solutions is expected to enhance the former, the performed tests did not show its usefulness in practice. For this reason that rule is omitted in the following. The interested reader may consult [9] for further details.

The remainder of the paper contains five other sections. Notation and concepts related with the robust shortest path problem are introduced in the next one. In addition, a brief sketch of the labeling and the hybrid robust shortest path algorithms presented in [10] is given. Section 3 is dedicated to the development of the new preprocessing rule and of the algorithm that implements it. An example is provided in Section 4. Results of computational tests on random and Karasan instances, comparing the new rule and its original static version, when used together with the labeling and the hybrid approaches, as well as with using CPLEX for solving a mixed integer formulation of the robust shortest path problem, are reported and discussed in Section 5. Conclusions are drawn in Section 6.

2 Preliminary concepts

A finite multi-scenario model is represented as $G(V, A, S_k)$, where G is a directed graph with a set of nodes $V = \{1, \dots, n\}$, a set of m arcs $A \subseteq \{(i,j) : i,j \in V \text{ and } i \neq j\}$ and a finite set of scenarios $S_k := \{1, \dots, k\}$, $k > 1$. The density or average degree of G is denoted by d, which is given by $d = m/n$. For each arc $(i,j) \in A$, c^s_{ij} represents its cost in scenario s, $s \in S_k$. It is assumed that the graph contains no parallel arcs.

A path from i to j, $i,j \in V$, in graph G, also called an (i,j)-path, is an alternating sequence of nodes and arcs of the form

$$p = \langle v_1, (v_1, v_2), v_2, \dots, (v_{r-1}, v_r), v_r \rangle,$$

with $v_1 = i$, $v_r = j$ and where $v_l \in V$, for $l = 2, \dots, r-1$, and $(v_l, v_{l+1}) \in A$, for $l = 1, \dots, r-1$. Because it is assumed that graphs do not contain parallel arcs, in the following paths will be represented simply by their sequence of nodes.

The set of arcs (nodes) in a path p is denoted by $A(p)$ $(V(p))$. Given two paths p, q, such that the destination node of p is also the initial node of q, the concatenation of p and q is the path formed by p followed by q, and is denoted by $p \diamond q$. The cost of a path p in scenario s, $s \in S_k$, is defined by

$$c^s(p) = \sum_{(i,j)\in A(p)} c^s_{ij}. \tag{1}$$

With no loss of generality, 1 and n denote the origin and the destination nodes of the graph G, respectively. The set of all $(1,n)$-paths in G is represented by $P(G)$.

Let q^s_{ij} represent the shortest (i,j)-path in G, $i,j \in V$, for a given scenario $s \in S_k$. In order to simplify the notation, q^s is used to denote the $(1,n)$-path, q^s_{1n}, and LB^s_{ij} is used to denote the cost of the path q^s_{ij} in scenario s, $c^s(q^s_{ij})$.

The minmax regret robust shortest path problem aims at finding a path in $P(G)$ with the least maximum robust deviation, i.e., satisfying

$$\arg \min_{p\in P(G)} RC(p), \tag{2}$$

where $RC(p)$ is the robustness cost of p, defined by

$$RC(p) := \max_{s\in S_k} RD^s(p), \tag{3}$$

and $RD^s(p)$ represents the robust deviation of path p in scenario s, $s \in S_k$, defined by

$$RD^s(p) := c^s(p) - LB^s_{1n}. \tag{4}$$

An optimal solution of (2) is called a robust shortest path.

A node is called robust 1-persistent if it belongs to some robust shortest $(1,n)$-path. Otherwise, the node is denominated robust 0-persistent. The origin and the destination nodes of the network are trivially robust 1-persistent nodes.

Three methods for finding a robust shortest path were developed in [10]. The two with the best performances in empirical terms were the labeling algorithm (LA) and the hybrid algorithm (HA). The LA is a variant of the labeling approach proposed in [7], adapted to the minmax regret objective function, but using the cost lower and upper-bounds similarly. The HA ranks simple paths for a suitable scenario and limited to an upper-bound that depends on the costs of the computed paths. The ranking is complemented with pruning rules based on the cost bounds imposed for the first method. This allows to discard useless solutions at an early stage.

3 Preprocessing techniques

In [11], a sufficient condition was established to identify robust 0-persistent nodes. This condition allows to test all the nodes that do not belong to a given path in the network. In this section, a new rule is developed to improve the previous preprocessing method, by restricting the number of tested scenarios and also by updating dynamically the tests as new solutions are computed. This rule allows to find a bigger number of robust 0-persistent nodes, than the previous.

For the sake of completeness, first, a result introduced in [11] is recalled to be used later. Proposition 1 concerns the identification of robust 0-persistent nodes.

Proposition 1 ([11]). *Consider a path $p \in P(G)$, and a node $i \notin V(p)$. If*

$$\exists \hat{s} \in S_k : \; RD^{\hat{s}}(q^{\hat{s}}_{1i} \diamond q^{\hat{s}}_{in}) > RC(p), \tag{5}$$

then node i is robust 0-persistent.

Some results are now presented to support an algorithm for identifying robust 0-persistent nodes. As mentioned earlier, the idea behind this version is to make the search dynamic and detect robust 0-persistent nodes, according to the least robustness cost of the $(1,n)$-paths obtained along the process.

Let $RCmin$ be a variable which stores the least robustness cost of a computed $(1,n)$-path at any iteration of the algorithm. Considering only the shortest $(1,n)$-path in scenario 1, q^1, that variable is initialized with

$$RCmin = RC(q^1),$$

for identifying robust 0-persistent nodes. Let Nod denote the set of nodes to be scanned. The condition provided by Proposition 1 can be rewritten, using variable $RCmin$. For any node $i \in Nod$, if

$$\exists \hat{s} \in S_k \; : \; RD^{\hat{s}}(q^{\hat{s}}_{1i} \diamond q^{\hat{s}}_{in}) > RCmin, \tag{6}$$

is satisfied, then the node i is robust 0-persistent. This condition demands the trees of the shortest $(1,j)$-paths and of the shortest (j,n)-paths for each scenario s, denoted by $\mathcal{T}^s_1$ and $\mathcal{T}^s_n$, respectively, $j \in V$, $s \in S_k$, and their costs LB^s_{1j} and LB^s_{jn} to be known. Any shortest path tree algorithm can be used with such purpose [1].

Let V_0 be used to collect the robust 0-persistent nodes. According to Proposition 1, and to the initialization of $RCmin$, Nod is initialized by

$$Nod = V \backslash V(q^1).$$

The value of variable $RCmin$ may change along the algorithm. The $(1,n)$-paths computed by the algorithm are stored in a list X_P, without repetitions. The set of nodes to scan may also change, every time a new $(1,n)$-path p such that $p \notin X_P$ has a robustness cost not greater than $RCmin$. If $RC(p) < RCmin$, $RCmin$ is updated with $RC(p)$. In what follows, it is shown how to update Nod, depending on the obtained path p satisfying $RC(p) \leq RCmin$.

When searching for robust 0-persistent nodes, Proposition 1 establishes that the analysis of the nodes of path p, $V(p)$, can be skipped. Thus, if $RC(p) = RCmin$, the nodes of $V(p)$ can be removed from Nod, and, if $RC(p) < RCmin$, the search focuses all the nodes outside $V(p)$ that were not already identified as robust 0-persistent. For a selected node $i \in Nod$, path p has the particular form $q^s_{1i} \diamond q^s_{in}$, $s \in S_k$. Then, one can write

$$Nod = \begin{cases} Nod \backslash V(q^s_{1i} \diamond q^s_{in}) & \text{if} \quad RC(q^s_{1i} \diamond q^s_{in}) = RCmin \\ V \backslash (V(q^s_{1i} \diamond q^s_{in}) \cup V_0) & \text{if} \quad RC(q^s_{1i} \diamond q^s_{in}) < RCmin \end{cases} \tag{7}$$

Nodes may be scanned more than once, because the analyzed $(1,n)$-paths may have nodes in common. This makes that some tests may be repeated after $RCmin$ is updated. Besides, in order to avoid repeating the path robust deviations, it is useful to store them, as

$$RD^s_i = RD^s(q^s_{1i} \diamond q^s_{in}) \, , \; s \in S_k \, , \; i \in V \backslash \{1, n\}. \tag{8}$$

A list X_N is used to store the nodes that have already been analyzed along the process.

The number of scenarios used to test condition (5) may make the robust 0-persistent nodes test computationally demanding. In [11] this test uses k scenarios. The same holds for condition (6), so in order to make this task lighter, in the following only a small number of scenarios to test, M, $M \leq k$, will be considered. Moreover, for each node $i \in Nod$, when the first scenario $s_i \in S_k$ for which (6) holds is known, then i is a robust 0-persistent node and its analysis can halt. Hence, the tests for scenarios $s_i + 1, \dots, M$, can be skipped. Generally, if $\max\{s_i : i \in Nod\} \neq M$, the computation of the trees $\mathcal{T}_1^s$ can be skipped for $s \in \{\max\{s_i : i \in Nod\} + 1, \dots, M\}$. The pseudo-code is given in Algorithm 1.

Algorithm 1: Dynamic version for finding robust 0-persistent nodes

```
for $s = 1, \dots, k$ do
    Compute the tree $\mathcal{T}_n^s$;
    for $j = 1, \dots, n$ do  $LB_{jn}^s \leftarrow c^s(q_{jn}^s)$ ;
$RCmin \leftarrow RC(q^1)$;
$X_P \leftarrow \{q^1\}$ ; $X_N \leftarrow \emptyset$;
$Nod \leftarrow V \backslash V(q^1)$ ; $V_0 \leftarrow \emptyset$;
while $Nod \neq \emptyset$ do
    Choose a node $i \in Nod$;
    $Nod \leftarrow Nod - \{i\}$;
    if $i \notin X_N$ then
        $X_N \leftarrow X_N \cup \{i\}$;
        for $s = 1, \dots, M$ do
            if tree $\mathcal{T}_1^s$ was not yet determined then  Compute the tree $\mathcal{T}_1^s$ ;
            $RD_i^s \leftarrow LB_{1i}^s + LB_{in}^s - LB_{1n}^s$;
            if $RD_i^s > RCmin$ then
                $V_0 \leftarrow V_0 \cup \{i\}$;
                break;
            if $q_{1i}^s \diamond q_{in}^s \notin X_P$ then
                $X_P \leftarrow X_P \cup \{q_{1i}^s \diamond q_{in}^s\}$;
                $RC(q_{1i}^s \diamond q_{in}^s) \leftarrow \max\Big\{RD_i^s, \max\big\{RD^r(q_{1i}^s \diamond q_{in}^s) : r \in S_k \backslash \{s\}\big\}\Big\}$;
                if $RC(q_{1i}^s \diamond q_{in}^s) = RCmin$ then  $Nod \leftarrow Nod \backslash V(q_{1i}^s \diamond q_{in}^s)$ ;
                if $RC(q_{1i}^s \diamond q_{in}^s) < RCmin$ then
                    $RCmin \leftarrow RC(q_{1i}^s \diamond q_{in}^s)$;
                    $Nod \leftarrow V \backslash (V(q_{1i}^s \diamond q_{in}^s) \cup V_0)$;
    else
        for $s = 1, \dots, M$ do
            if $RD_i^s > RCmin$ then
                $V_0 \leftarrow V_0 \cup \{i\}$;
                break;
return $V_0$
```

In terms of the worst case computational time complexity, the first phase of Algorithm 1 is similar to the first phase of the static version [11]. The former initializes $RCmin$ with $RC(q^1)$, which means it is performed in $O_1^a = \mathcal{O}(km + kn) = \mathcal{O}(km)$ time for acyclic networks and in $O_1^c = \mathcal{O}(k(m + n \log n))$ for general networks.

The second phase concerns searching for robust 0-persistent nodes, which compared to the static version has the additional work of calculating RD_i^s, $i \in Nod$, $s \in S_k$, updating set Nod, and repeating the tests (6) due to the updates of $RCmin$. For the first task, assuming that the trees $\mathcal{T}_1^s$ and $\mathcal{T}_n^s$, and the associate costs for all scenarios were previously computed, RD_i^s, $i \in Nod$, $s \in S_k$, is obtained in $\mathcal{O}(k)$ time. The second task concerns the update of Nod and involves differences and unions of sets with n nodes at most. These operations require an $\mathcal{O}(n)$ complexity, when using indexation by hash sets [2]. The third procedure demands $\mathcal{O}(1)$ operations for each scenario in S_k, and each node $i \in Nod$, since RD_i^s was already determined.

In a worst case, the three tasks above are performed $k(n-2)$ times at most, one per each scenario s, $s \in S_k$, and each node selected in Nod, with up to $n-2$ nodes. Thus, an additional work of $\mathcal{O}(kn^2 + k^2n)$ is added to the second phase of the static version. In conclusion, Algorithm 1 has a time complexity of $\mathcal{O}(kn^2 + k^2n)$ for all types of networks, since $\log n \ll n$ and $m < n^2$.

4 Example

In the following, the dynamic algorithm for finding robust 0-persistent nodes introduced in Section 3 is exemplified. In order to better understand the differences introduced in the previous algorithm with respect to the static preprocessing method presented in [11], the application of the two approaches is described.

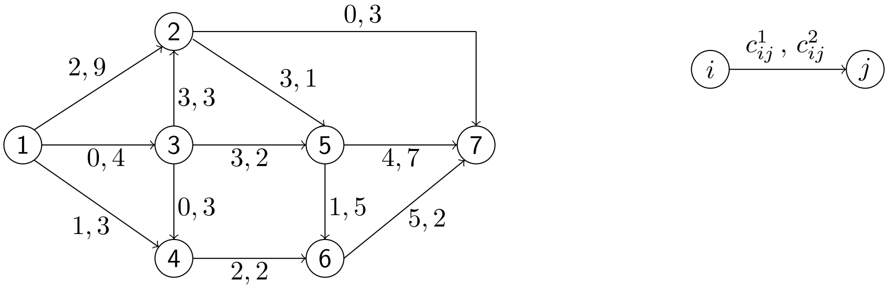


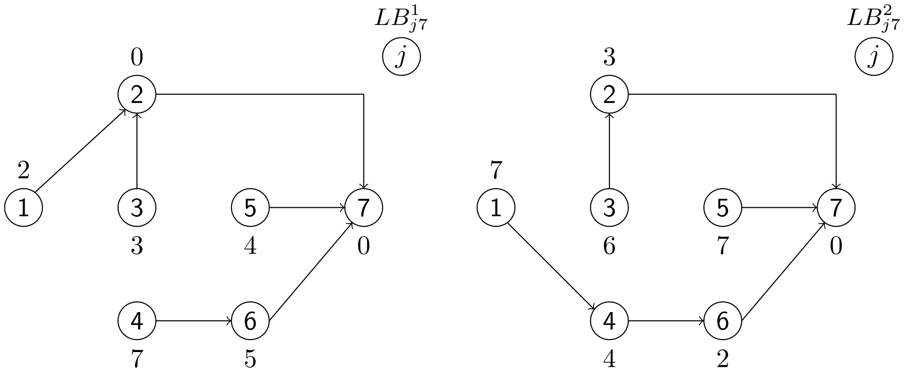
Figure 1: Network $G(V, A, S_2)$

Let $G(V, A, S_2)$ be the network depicted in Figure 1. Figure 2 shows the shortest path trees from every node to node 7 in G, in scenario 1 – Figure 2.(a) – and in scenario 2 – Figure 2.(b).

Figure 3 shows the shortest path trees from node 1 to any node in $G(V, A, S_2)$, in scenario 1 – Figure 3.(a) – and in scenario 2 – Figure 3.(b).

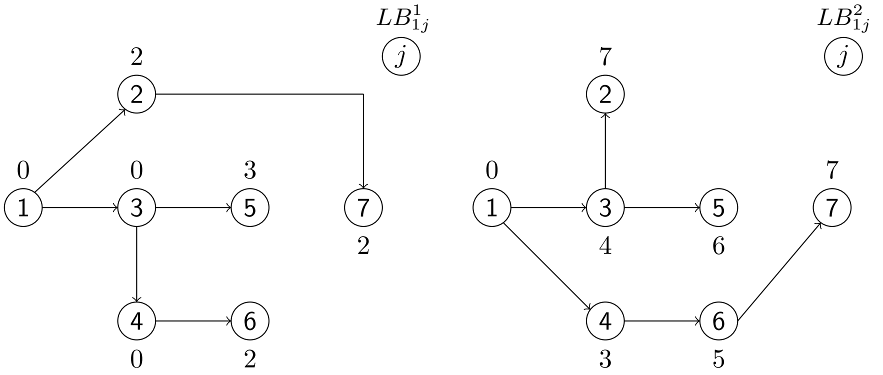
In what follows the number of scenarios tested is limited to $M \in \{1, 2\}$. As mentioned in the previous section, this constraint was not used in [11], it will be applied to both approaches for the sake of comparing them.

Static approach The variable $RCmin$ is initialized by the minimum robustness cost of the shortest $(1,7)$-paths of G. According to Figure 2, $q^1 = \langle 1, 2, 7 \rangle$, with $LB_{17}^1 = 2$, and $q^2 = \langle 1, 4, 6, 7 \rangle$, with $LB_{17}^2 = 7$. Given that $c^2(q^1) = 12$ and $c^1(q^2) = 8$, one has $RC(q^1) = 5$ and $RC(q^2) = 6$. Hence, q^1 is the shortest $(1,7)$-path with the least robustness cost and, therefore, $RCmin = 5$.



(a) in scenario 1 (b) in scenario 2

Figure 2: Shortest path trees rooted at node 7 in $G(V, A, S_2)$



(a) in scenario 1 (b) in scenario 2

Figure 3: Shortest path trees rooted at node 1 in $G(V, A, S_2)$

This value does not change along the static method and the set of arcs Nod to scan is initialized by $V \backslash V(q^1) = \{3, 4, 5, 6\}$.

- $M = 1$

  Starting with node 3, the inequality (6) is not satisfied in scenario 1, given that

  $$RD^1_3 = LB^1_{13} + LB^1_{37} - LB^1_{17} = 1 \leq RCmin.$$

  The same thing happens for nodes 4, 5 and 6, because

  $$RD^1_i = LB^1_{1i} + LB^1_{i7} - LB^1_{17} = 5 \leq RCmin\,,\ i = 4, 5, 6.$$

  Therefore, no robust 0-persistent nodes are detected when considering only scenario 1, $V_0 = \emptyset$.

- $M = 2$

  For scenario 2, the nodes 3, 4 and 6 still do not satisfy (6), given that

  $$RD^2_3 = LB^2_{13} + LB^2_{37} - LB^2_{17} = 3 \leq RCmin,$$

and

$$RD_i^2 = LB_{1i}^2 + LB_{i7}^2 - LB_{17}^2 = 0 \leq RCmin\,,\ i = 4,6.$$

Nevertheless, (6) holds for node 5 and scenario 2,

$$RD_5^2 = LB_{15}^2 + LB_{57}^2 - LB_{17}^2 = 6 > RCmin,$$

therefore, node 5 is the only one identified as robust 0-persistent, $V_0 = \{5\}$.

Dynamic approach According to Algorithm 1, $RCmin$ is initialized with $RC(q^1) = 5$ and Nod with $V \backslash V(q^1) = \{3,4,5,6\}$.

- $M = 1$

  Starting by scanning node 3, condition (6) is not satisfied for scenario 1. Then, the robustness cost of the path $q_{13}^1 \diamond q_{37}^1 = \langle 1,3,2,7\rangle$ is determined, $RC(\langle 1,3,2,7\rangle) = 3$, and this value improves $RCmin$. Additionally, by (7), Nod is updated to $V \backslash V(\langle 1,3,2,7\rangle) = \{4,5,6\}$, since at this point $V_0 = \emptyset$.

  For the updated $RCmin = 3$, when choosing nodes 4, 5 and 6 to scan, inequality (6) is always satisfied for scenario 1, given that

  $$RD_i^1 = 5 > RCmin\,,\ i = 4,5,6.$$

  Consequently, all the nodes in Nod are identified as robust 0-persistent, i.e., $V_0 = \{4,5,6\}$.

- $M = 2$

  Condition (6) holds for node 3 and scenarios 1 and 2, with the initial $RCmin = 5$. Then, the path associated with node 3 for scenarios 1 and 2, $q_{13}^s \diamond q_{37}^s$, $s \in S_2$, is given by $\langle 1,3,2,7\rangle$, which has a robustness cost of 3. The remaining steps are those presented for $M = 1$, thus $V_0 = \{4,5,6\}$.

For this example, Algorithm 1 is more effective than its static version, given that it detects more robust 0-persistent nodes than the former version.

Computing a robust shortest path after preprocessing The reduced network obtained from preprocessing is depicted in Figure 4. The arcs in G are represented with a dashed line in Figure 4. The robust 0-persistent nodes, 4, 5 and 6, are removed from G, as well as all the arcs that start or end in these nodes.

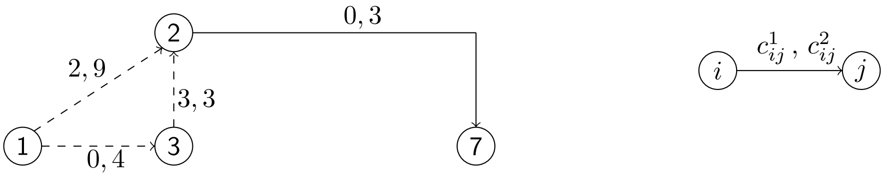


Figure 4: Reduced network after preprocessing

There are only two $(1,7)$-paths containing arc $(2,7)$ in the reduced network, $q^1 = \langle 1,2,7\rangle$, with $RC(q^1) = 5$, and $q = \langle 1,3,2,7\rangle$, with $RC(q) = 3$. Therefore, q is the robust shortest $(1,7)$-path in G.

5 Computational experiments

This section is dedicated to the computational comparison of the static (presented in [11]) and the dynamic (in Algorithm 1) methods for preprocessing robust 0-persistent nodes, and to their impact on solving the robust shortest path problem when combined with the LA and the HA introduced in [10]. Additionally, the integer formulation of the robust shortest path problem with discrete scenarios

$$\begin{array}{lll} \min & \max\limits_{s\in S_k} \left(\sum\limits_{(i,j)\in A} c_{ij}^s x_{ij} - LB_1^s \right) & \\ \text{s. t.} & \sum\limits_{(1,j)\in A} x_{1j} - \sum\limits_{(i,1)\in A} x_{i1} = -1 & \\ & \sum\limits_{(u,j)\in A} x_{uj} - \sum\limits_{(i,u)\in A} x_{iu} = 0, & u \in V - \{1,n\} \\ & \sum\limits_{(n,j)\in A} x_{nj} - \sum\limits_{(i,n)\in A} x_{in} = 1 & \\ & x_{ij} \in \{0,1\}, & (i,j)\in A \end{array} \tag{9}$$

was solved using CPLEX [4], after having been rewritten as a mixed integer linear problem.

Algorithm 1 and its static version were implemented in Matlab 7.12 and the IBM ILOG CPLEX Optimization Studio, 12.6.2. version, was used to solve the linear formulation obtained from (9). The tests ran on a computer equipped with an Intel Pentium Dual CPU T2310 1.46GHz processor and 2GB of RAM. As a preliminary task for all the codes, Dijkstra's algorithm [1] is used to solve the single destination shortest path problem for a given scenario. As mentioned earlier, the preprocessing techniques were combined with the LA and the HA in [10], and with CPLEX. The robust shortest path problem was solved with and without preprocessing.

5.1 Test problems

The benchmarks used in the experiments are divided into two main classes: randomly generated directed graphs and Karasan graphs.

The tests performed on random graphs are divided in three groups. The first two include those with arc costs assigned with integer numbers in $U(0,c)$, $c > 0$. In this case, networks with n nodes, density d and k scenarios are denoted by $R_{n,d}^{k,c}$. For the considered instances, $n \in \{500, 7000\}$, $d \in \{5, 10, 20\}$, $k \in \{2, 3\}$ and $c \in \{100, 10\,000\}$. In addition, $n \in \{2000, 5000\}$ is used, when LA and HA are applied for $c = 100$, given that these instances have already been considered in [9]. The third group considers networks with two scenarios and negatively correlated costs. Following the procedure in [8], half of the arc costs in scenario 1 are integers randomly chosen in $U(0, c/2)$ and the costs in scenario 2 are random integers in $U(c/2, c)$, $c > 0$. The remaining arc costs are assigned similarly, but to scenarios 2 and 1. Such networks are denoted by $R_{n,d}^{2,c,NC}$. The tests comprised instances with $n \in \{500, 7000\}$, $d \in \{5, 10, 20\}$ and $c = 100$.

The Karasan graphs have the structure presented in [5], i.e. they are acyclic and layered graphs. Each arc cost is assigned with a random integer in $U(0,c)$, $c > 0$. In the following, Karasan networks with n nodes, width w and k scenarios are denoted by $K^{k,c}_{n,w}$. The source and the destination are dummy nodes that link to the first and from the last layers, respectively. The tests include instances with $n \in \{30, 60, 90\}$, $w \in \{10, 20\}$, $k \in \{2, 3\}$ and $c = 100$. The considered widths are bigger than usual (see, for instance, [5, 6]) because the preprocessing techniques were not effective for networks with small width compared to the number of nodes.

For each network dimension, ten instances were generated. For each instance, the static and the dynamic preprocessing algorithms were applied, and (6) was tested for the scenarios $1, \dots, M$, with $M \in \{1 \dots k\}$. The robust shortest path problems were solved by LA, HA and CPLEX, after preprocessing. Alternatively, these methods solved the same instances from scratch, with no preprocessing.

5.2 Results

In order to analyze the performance of the static and the dynamic algorithms, the average total running times (in seconds) are calculated for each network dimension. Let P, NP and AP represent the average CPU times to preprocess robust 0-persistent nodes, to solve the robust shortest path problem with no preprocessing, and to do the same after preprocessing, respectively. Let also TP denote the average overall CPU time for finding a robust shortest path combined with preprocessing, i.e., $TP = P + AP$. Additionally, let N represent the average number of detected robust 0-persistent nodes. The application of the static and the dynamic methods is distinguished by the indices s and d, respectively.

The least total CPU time to find the robust shortest path with LA, HA and CPLEX is bold typed, for each type of instances that have been considered.

Random networks First, the results obtained for random networks when costs range in $[0, c]$, $c = 100$, are considered. The averages are presented in Tables 1 – 4. The number of detected robust 0-persistent nodes is high for the networks with the lowest densities ($d \in \{5, 10\}$), particularly for Algorithm 1 rather than for its static version – Table 1. Moreover, for fixed n, d and M, less nodes tend to be detected when k increases, since N_s and N_d also decrease. For all the instances, in spite of the preprocessing work demanded by Algorithm 1 being heavier than the required by the static version, $P_s < P_d$, the additional effort of the dynamic version leads to the detection of more robust 0-persistent nodes, $N_s < N_d$.

Tables 2 – 4 show that preprocessing robust 0-persistent nodes can be more effective to solve the robust shortest path problem by LA, HA or CPLEX, rather than without using preprocessing. Combining dynamic preprocessing with finding a robust shortest path was the most efficient method when HA was applied for $M = 1$ on the biggest networks ($n = 2000$, $d = 5$ and $k = 3$; $n = 5000$, except for $d = 20$ and $k = 3$, and $n = 7000$, for $d = 10$ or for $d = 20$ and $k = 2$). The same happened when LA was applied on most of the networks (except for $n = 500$ and $d = 20$). In case of CPLEX, the dynamic procedure stood out for all the smallest networks, except for $d = 10$ and $k = 2$, and for the biggest networks, for the single cases $d = 10$ and $k = 2$.

	M	P_s	P_d	N_s	N_d		M	P_s	P_d	N_s	N_d
$R^{2,100}_{500,5}$	1	0.713	0.772	267	491	$R^{2,100}_{2000,5}$	1	4.744	4.872	1518	1992
	2	0.948	0.986	361	495		2	6.094	6.384	1788	1995
$R^{3,100}_{500,5}$	1	1.142	1.083	130	410	$R^{3,100}_{2000,5}$	1	5.745	5.977	764	1730
	2	1.384	1.308	222	479		2	7.150	7.353	1126	1963
	3	1.557	1.527	279	493		3	8.559	8.748	1336	1992
$R^{2,100}_{500,10}$	1	0.877	1.060	149	430	$R^{2,100}_{2000,10}$	1	4.315	4.632	911	1943
	2	0.199	1.238	196	483		2	5.637	5.883	1144	1990
$R^{3,100}_{500,10}$	1	1.201	1.318	65	170	$R^{3,100}_{2000,10}$	1	6.313	6.846	106	1250
	2	1.520	1.584	120	324		2	8.047	8.431	188	1806
	3	1.777	1.915	151	389		3	9.474	10.094	264	1925
$R^{2,100}_{500,20}$	1	0.856	1.572	19	103	$R^{2,100}_{2000,20}$	1	4.823	5.845	8	1662
	2	1.127	1.630	34	201		2	6.218	7.203	14	1862
$R^{3,100}_{500,20}$	1	1.145	1.328	2	16	$R^{3,100}_{2000,20}$	1	7.140	8.809	57	266
	2	1.481	1.723	4	44		2	9.802	9.774	138	710
	3	1.800	1.939	5	97		3	11.392	11.421	179	963
$R^{2,100}_{5000,5}$	1	20.486	20.905	3247	4990	$R^{2,100}_{7000,5}$	1	129.504	134.173	4838	6908
	2	26.391	26.770	4110	4994		2	142.658	143.929	5667	6995
$R^{3,100}_{5000,5}$	1	25.895	25.979	1477	4646	$R^{3,100}_{7000,5}$	1	186.513	196.549	4006	6421
	2	31.760	32.382	2193	4966		2	205.702	212.144	5037	6925
	3	37.897	38.531	2633	4994		3	222.072	223.820	5561	6990
$R^{2,100}_{5000,10}$	1	21.449	21.797	1260	4939	$R^{2,100}_{7000,10}$	1	132.743	141.608	1606	6598
	2	27.264	27.888	1788	4993		2	145.890	152.684	1892	6991
$R^{3,100}_{5000,10}$	1	27.601	26.594	353	3782	$R^{3,100}_{7000,10}$	1	199.218	212.207	324	5634
	2	34.233	33.236	661	4724		2	216.220	228.328	736	6798
	3	40.223	39.827	900	4915		3	232.169	246.511	1055	6953
$R^{2,100}_{5000,20}$	1	22.396	27.453	119	4404	$R^{2,100}_{7000,20}$	1	149.545	160.489	1014	5217
	2	29.453	33.095	208	4806		2	164.483	180.172	1412	6681
$R^{3,100}_{5000,20}$	1	28.653	42.394	60	1383	$R^{3,100}_{7000,20}$	1	213.284	237.520	58	2841
	2	34.135	42.061	108	2713		2	232.389	248.409	122	4177
	3	41.741	45.958	155	3248		3	244.730	270.563	160	4838

Table 1: Average preprocessing CPU times (in seconds) and number of detected robust 0-persistent nodes

	M	NP	AP_s	AP_d	TP_s	TP_d		M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100}_{500,5}$	1	**0.596**	0.042	0.001	0.755	0.773	$R^{2,100}_{2000,5}$	1	**4.837**	0.267	0.007	5.011	4.879
	2		0.024	0.000	0.972	0.986		2		0.083	0.003	6.177	6.387
$R^{3,100}_{500,5}$	1	**0.857**	0.089	0.014	1.231	1.097	$R^{3,100}_{2000,5}$	1	6.297	0.748	0.054	6.493	**6.031**
	2		0.059	0.003	1.443	1.311		2		0.455	0.009	7.605	7.362
	3		0.042	0.002	1.599	1.529		3		0.297	0.002	8.856	8.750
$R^{2,100}_{500,10}$	1	**0.696**	0.089	0.010	0.966	1.070	$R^{2,100}_{2000,10}$	1	**4.634**	0.763	0.009	5.078	4.641
	2		0.079	0.003	1.278	1.241		2		0.599	0.005	6.236	5.888
$R^{3,100}_{500,10}$	1	**1.108**	0.155	0.080	1.356	1.398	$R^{3,100}_{2000,10}$	1	**6.757**	1.595	0.330	7.908	7.176
	2		0.110	0.032	1.630	1.616		2		1.509	0.047	9.556	8.478
	3		0.101	0.020	1.878	1.935		3		1.431	0.015	10.905	10.109
$R^{2,100}_{500,20}$	1	**0.772**	0.194	0.127	1.050	1.699	$R^{2,100}_{2000,20}$	1	**5.086**	1.950	0.127	6.773	5.972
	2		0.183	0.089	1.310	1.719		2		1.994	0.039	8.212	7.242
$R^{3,100}_{500,20}$	1	**1.053**	0.203	0.175	1.348	1.503	$R^{3,100}_{2000,20}$	1	**7.309**	2.007	1.629	9.147	10.438
	2		0.198	0.157	1.679	1.880		2		1.813	0.915	11.615	10.689
	3		0.215	0.133	2.015	2.072		3		1.767	0.715	13.159	12.136
$R^{2,100}_{5000,5}$	1	26.757	2.259	0.003	22.745	**20.908**	$R^{2,100}_{7000,5}$	1	144.962	4.689	0.044	**134.193**	134.217
	2		0.845	0.006	27.236	26.776		2		2.436	0.003	145.094	143.932
$R^{3,100}_{5000,5}$	1	32.438	6.072	0.081	31.967	**26.060**	$R^{3,100}_{7000,5}$	1	207.573	8.097	0.248	**194.610**	196.797
	2		4.414	0.056	36.174	32.438		2		4.811	0.011	210.513	212.155
	3		3.517	0.016	41.414	38.547		3		3.430	0.002	225.502	223.822
$R^{2,100}_{5000,10}$	1	26.601	9.967	0.014	31.416	**21.811**	$R^{2,100}_{7000,10}$	1	197.869	24.793	0.378	157.536	**141.986**
	2		7.530	0.005	34.794	27.893		2		21.766	0.006	167.656	152.690
$R^{3,100}_{5000,10}$	1	31.671	10.070	0.843	37.671	**27.437**	$R^{3,100}_{7000,10}$	1	225.077	30.871	7.508	230.089	**219.715**
	2		8.812	0.077	43.045	33.313		2		29.951	0.113	246.171	228.441
	3		7.930	0.018	48.153	39.845		3		29.269	0.006	261.438	246.517
$R^{2,100}_{5000,20}$	1	33.868	14.781	0.430	37.177	**27.883**	$R^{2,100}_{7000,20}$	1	183.919	28.231	3.213	177.776	**163.702**
	2		14.009	0.069	43.462	33.164		2		24.308	0.832	188.791	181.004
$R^{3,100}_{5000,20}$	1	**34.468**	12.513	6.661	41.166	49.055	$R^{3,100}_{7000,20}$	1	**228.817**	37.535	15.694	250.819	253.214
	2		11.397	3.018	45.532	45.079		2		42.442	6.812	274.831	255.221
	3		14.929	1.968	56.670	47.926		3		38.458	4.130	283.188	274.693

Table 2: Average CPU times (in seconds) for algorithm HA with and without preprocessing

The fact that more robust 0-persistent nodes were detected by the dynamic method than by the static method contributes for a more significant reduction of the network and, consequently, of the average CPU times when finding a robust shortest path after preprocessing, $AP_s > AP_d$. In conclusion, the dynamic version outperformed the static version. Besides, preprocessing with the dynamic search was also a better alternative than solving the problem without any preprocessing, $TP_d < NP$.

In spite of the previous considerations, HA was the fastest method for the smallest networks without preprocessing, being replaced by CPLEX for the biggest and densest networks. Still, for the majority of the tested networks, the results of HA combined with preprocessing were more effective than the homologous results for LA and CPLEX, except when $n = 7000$, $d = 10$ and $k = 3$, or $n = 7000$ and $d = 20$, for which CPLEX outperformed the remaining algorithms.

LA was never the fastest method, however it was the most sensitive to preprocessing, and showed the most drastic reductions with respect to NP. This can be explained by the fact that removing nodes from the network allows to discard a considerable number of labels in LA, making easier the computation of an optimal solution. For HA, despite the fact that eliminating nodes reduces the effort on calculating reduced costs, preprocessing does not have so much impact, given that the search for a robust shortest path is more focused on selecting suitable deviation arcs and that can be done in few iterations without preprocessing [10]. Another aspect to take into account is that with the increase of n, the available number of deviation arcs may increase substantially, making the problem harder to solve with HA, rather than with CPLEX. In fact, CPLEX depended mainly on the number of arcs of the network, which means that its stability was not greatly affected by the network structure.

For each fixed n, d and k, the smaller the number of scenarios for testing (6), the less effort was required for computing the shortest path trees rooted at node 1. Hence, small values of M implied small preprocessing times. This is valid for both the static and the dynamic approaches. The latter is always better than the first in detecting robust 0-persistent nodes, $N_s < N_d$, when M is fixed – Table 1. In general, the best value of M to consider in order to ensure that finding a robust shortest path with preprocessing is faster than solving the problem without preprocessing, must assure that $P < NP$ and that the number of detected robust 0-persistent nodes is sufficient to reduce the CPU time which may not exceed $NP - P$. Tables 2 – 4 show that Algorithm 1 was more effective than its static version with this respect when $M = 1$, except if $n = 7000$, with $d = 5$ for HA and CPLEX and with $d = 10$, $k = 3$ for CPLEX, and if $n = 500$, with $d = 10$, $k = 2$ for CPLEX and with $d = 20$ for LA. When $M = 2, 3$, the dynamic preprocessing combined with LA or CPLEX was the most efficient method in very few cases.

	M	NP	AP_s	AP_d	TP_s	TP_d		M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100}_{500,5}$	1	0.859	0.221	0.005	0.934	**0.777**	$R^{2,100}_{2000,5}$	1	7.522	1.807	0.045	6.551	**4.917**
	2		0.114	0.003	1.062	0.989		2		0.541	0.016	6.635	6.400
$R^{3,100}_{500,5}$	1	1.268	0.465	0.040	1.607	**1.123**	$R^{3,100}_{2000,5}$	1	10.922	5.475	0.323	11.220	**6.300**
	2		0.304	0.010	1.688	1.318		2		3.206	0.049	10.356	7.402
	3		0.213	0.007	1.770	1.534		3		2.194	0.034	10.753	8.782
$R^{2,100}_{500,10}$	1	1.763	0.528	0.033	1.405	**1.093**	$R^{2,100}_{2000,10}$	1	9.164	5.160	0.102	9.475	**4.734**
	2		0.447	0.009	1.646	1.247		2		3.948	0.031	9.585	5.914
$R^{3,100}_{500,10}$	1	1.948	0.675	0.396	1.876	**1.714**	$R^{3,100}_{2000,10}$	1	19.705	11.980	2.196	18.293	9.042
	2		0.558	0.139	2.078	1.723		2		10.839	0.234	18.886	**8.665**
	3		0.517	0.065	2.294	1.980		3		10.032	0.083	19.506	10.177
$R^{2,100}_{500,20}$	1	3.389	0.824	0.603	**1.680**	2.175	$R^{2,100}_{2000,20}$	1	33.345	12.860	0.833	17.683	**6.678**
	2		0.789	0.371	1.916	2.001		2		13.329	0.210	19.547	7.413
$R^{3,100}_{500,20}$	1	3.910	0.914	0.839	**2.059**	2.167	$R^{3,100}_{2000,20}$	1	42.829	12.605	10.543	19.745	19.352
	2		0.883	0.761	2.364	2.484		2		12.132	6.474	21.934	16.248
	3		0.878	0.629	2.678	2.568		3		11.531	4.808	22.923	**16.229**
$R^{2,100}_{5000,5}$	1	59.962	13.748	0.160	34.234	**21.065**	$R^{2,100}_{7000,5}$	1	199.571	56.169	0.832	185.673	**135.005**
	2		4.952	0.032	31.343	26.802		2		30.301	0.477	172.959	144.406
$R^{3,100}_{5000,5}$	1	103.437	43.294	0.615	69.189	**26.594**	$R^{3,100}_{7000,5}$	1	264.490	107.745	3.821	294.258	**200.370**
	2		31.870	0.198	63.630	32.580		2		65.654	0.809	271.356	212.953
	3		25.157	0.152	63.054	38.683		3		47.681	0.615	269.753	224.435
$R^{2,100}_{5000,10}$	1	134.070	53.750	0.187	75.199	**21.984**	$R^{2,100}_{7000,10}$	1	363.601	286.662	1.666	419.405	**143.274**
	2		46.056	0.132	73.320	28.020		2		263.508	0.603	409.398	153.287
$R^{3,100}_{5000,10}$	1	149.398	70.250	6.142	97.851	**32.736**	$R^{3,100}_{7000,10}$	1	464.506	363.082	23.083	562.300	235.290
	2		62.080	0.616	96.313	33.852		2		326.167	1.349	542.387	**229.677**
	3		55.514	0.224	95.737	40.051		3		306.959	1.201	539.128	247.712
$R^{2,100}_{5000,20}$	1	311.511	81.121	2.391	103.517	**29.844**	$R^{2,100}_{7000,20}$	1	409.501	240.741	45.757	390.286	206.246
	2		82.884	0.527	112.337	33.622		2		236.265	3.168	400.748	**183.340**
$R^{3,100}_{5000,20}$	1	301.563	79.006	46.820	107.659	89.214	$R^{3,100}_{7000,20}$	1	512.466	394.824	175.524	608.108	413.044
	2		78.269	22.580	112.404	64.641		2		384.554	86.136	616.943	334.545
	3		78.830	14.193	120.571	**60.151**		3		380.670	52.660	625.400	**323.223**

Table 3: Average CPU times (in seconds) for algorithm LA with and without preprocessing

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100}_{500,5}$	1	3.387	2.087	1.960	2.800	**2.732**
	2		1.943	1.800	2.891	2.786
$R^{3,100}_{500,5}$	1	4.620	2.365	2.045	3.507	**3.128**
	2		1.950	1.840	3.334	3.148
	3		1.945	1.820	3.502	3.347
$R^{2,100}_{500,10}$	1	4.685	3.233	2.993	4.110	4.053
	2		2.683	2.147	**2.882**	3.385
$R^{3,100}_{500,10}$	1	5.159	3.445	2.600	4.646	**3.918**
	2		2.695	2.475	4.215	4.059
	3		2.520	2.305	4.297	4.220
$R^{2,100}_{500,20}$	1	5.116	3.540	2.813	4.396	4.385
	2		3.463	2.210	4.590	**3.840**
$R^{3,100}_{500,20}$	1	5.472	3.265	2.770	4.410	**4.098**
	2		2.905	2.585	4.386	4.308
	3		2.750	2.415	4.550	4.354
$R^{2,100}_{7000,5}$	1	140.900	8.307	4.263	**137.811**	138.436
	2		5.227	1.950	147.885	145.879
$R^{3,100}_{7000,5}$	1	195.734	8.550	3.435	**195.063**	199.984
	2		5.685	2.420	211.387	214.564
	3		4.590	2.315	226.662	226.135
$R^{2,100}_{7000,10}$	1	163.533	25.300	6.003	158.043	**147.611**
	2		17.563	2.947	163.453	155.631
$R^{3,100}_{7000,10}$	1	215.976	14.335	7.495	**213.553**	219.702
	2		11.060	4.320	227.280	232.648
	3		10.195	2.150	242.364	248.661
$R^{2,100}_{7000,20}$	1	**153.454**	20.167	6.647	169.712	167.136
	2		19.077	3.683	183.560	183.855
$R^{3,100}_{7000,20}$	1	**222.188**	30.863	18.517	244.147	256.037
	2		29.733	12.693	262.122	261.102
	3		29.027	7.497	273.757	278.060

Table 4: Average CPU times (in seconds) for CPLEX with and without preprocessing

Tables 5 – 8 summarize the average results obtained when arc costs range between 0 and $c = 10\,000$. A bigger effort is required to preprocess nodes when the arc costs are larger than for the previous set of benchmarks. This is reflected by the increase of the number of detected robust 0-persistent nodes by both the static and the dynamic approaches – Table 5. With the dynamic procedure, in particular, this is more expressive for the biggest networks and less significant for the smallest networks. Nevertheless, this procedure is still better than the static in terms of detected robust 0-persistent nodes.

In general, the best CPU times to solve the robust shortest path problem (with and without preprocessing) for $c = 10\,000$ are bigger than the homologous values for $c = 100$, for all the methods and the smallest networks. The same happened for the biggest networks, with LA and HA, except when $d = 20$, whereas CPLEX was faster when larger costs were involved.

For $c = 10\,000$, the improvement of combining the dynamic procedure for solving the robust shortest path problem was observed in more cases than for $c = 100$, rather than combining the static procedure or determining a solution from scratch. The only exception was the application of CPLEX for $n = 7000$, which was faster to solve from scratch. Still for $c = 10\,000$, HA combined with the dynamic procedure is faster on instances with $n = 500$ ($d \in \{5, 10\}$ and $k = 3$), for which no kind of preprocessing is effective in terms of CPU times for $c = 100$, and on instances with

$n = 7000$, $d = 5$ and $k = 2$, for which the static procedure is more efficient when $c = 100$ – Tables 2 and 6. The latter situation happened for LA when $n = 500$, $d = 20$ and $k = 2$ – Tables 3 and 7 – and for CPLEX when $n = 500$, $d = 10$ and $k = 2$ – Tables 4 and 8.

	M	P_s	P_d	N_s	N_d		M	P_s	P_d	N_s	N_d
$R^{2,10000}_{500,5}$	1	1.168	1.249	351	460	$R^{2,10000}_{7000,5}$	1	149.159	151.669	3700	6923
	2	1.412	1.504	426	495		2	156.563	160.888	4725	6996
$R^{3,10000}_{500,5}$	1	2.083	2.060	163	304	$R^{3,10000}_{7000,5}$	1	195.342	208.062	4107	6457
	2	2.242	2.652	266	441		2	208.873	219.120	5105	6931
	3	2.390	2.566	319	479		3	221.730	245.684	5604	6991
$R^{2,10000}_{500,10}$	1	1.278	1.449	211	371	$R^{2,10000}_{7000,10}$	1	150.612	163.895	361	6800
	2	1.453	1.636	284	487		2	167.918	174.420	781	6993
$R^{3,10000}_{500,10}$	1	1.964	2.055	72	226	$R^{3,10000}_{7000,10}$	1	205.741	220.140	118	5858
	2	2.209	2.421	128	350		2	223.808	231.268	284	6845
	3	2.481	2.696	152	413		3	228.126	234.946	435	6967
$R^{2,10000}_{500,20}$	1	1.382	1.532	108	276	$R^{2,10000}_{7000,20}$	1	156.746	177.824	2569	5862
	2	1.612	1.847	124	369		2	162.099	180.452	2773	6827
$R^{3,10000}_{500,20}$	1	2.281	2.230	22	5	$R^{3,10000}_{7000,20}$	1	202.199	214.773	106	2750
	2	2.206	2.544	33	38		2	223.909	234.823	220	4699
	3	2.484	2.860	50	89		3	240.718	253.720	286	5556

Table 5: Average preprocessing CPU times (in seconds) and number of detected robust 0-persistent nodes

Average results when arc costs vary between 0 and $c = 100$ and are negatively in the two scenarios are shown in Tables 9 – 12. It can be observed that preprocessing is more time demanding on these instances than with costs in $U(0, 100)$. Besides, the same happened for some instances with costs in $U(0, 10\,000)$, in particular for the smallest networks with the highest densities, namely, $d = 10$, $M = 2$ and $d = 20$. In terms of the number of identified robust 0-persistent nodes, the dynamic procedure is still more effective than the static, but globally, fewer robust 0-persistent nodes are found than when $c = 100$. In particular, the preprocessing is not effective for the tested denser networks, where they were rarely detected.

In general, all the smallest CPU times for finding the robust shortest path are bigger than those obtained by all the methods when costs range in $U(0, 100)$ and even in $U(0, 10\,000)$ for most of the cases. In terms of running time, the preprocessing was not effective for HA, given that the problem was much easier to solve from scratch – Table 10. For LA and CPLEX, fewer situations tend to be improved with the combination of the dynamic procedure to solve the problem, when compared with the experiments for costs in $U(0, 100)$. Namely, for LA, when $n = 7000$ and $d = 20$ – Tables 3 and 11, which was outperformed by the combination with the static procedure. The same situation happened for CPLEX, when $n = 500$ and $d = 20$, and, additionally, when $n = 7000$ and $d = 10$, which was outperformed by solving the problem with no preprocessing – Tables 4 and 12.

Karasan networks Now the results obtained for Karasan networks are presented. The averages are shown in Tables 13 – 16.

Like for random networks, for this new class of graphs the dynamic algorithm outperformed the static in terms of the number of preprocessed nodes ($N_d > N_s$). Besides, it can be observed that the wider the Karasan graph, the easier it is to identify robust 0-persistent nodes – Table 13. When the graph width increases, there exist more arcs between layers, which can lead to bigger

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,10000}_{500,5}$	1	**1.153**	0.073	0.007	1.241	1.256
	2		0.046	0.002	1.458	1.506
$R^{3,10000}_{500,5}$	1	2.205	0.219	0.047	2.302	**2.107**
	2		0.119	0.005	2.361	2.657
	3		0.087	0.003	2.477	2.569
$R^{2,10000}_{500,10}$	1	**1.283**	0.157	0.032	1.435	1.481
	2		0.103	0.002	1.556	1.638
$R^{3,10000}_{500,10}$	1	2.198	0.319	0.095	2.283	**2.150**
	2		0.283	0.018	2.492	2.439
	3		0.250	0.007	2.731	2.703
$R^{2,10000}_{500,20}$	1	**1.357**	0.256	0.067	1.638	1.599
	2		0.246	0.018	1.858	1.865
$R^{3,10000}_{500,20}$	1	**2.311**	0.468	0.221	2.749	2.451
	2		0.429	0.094	2.635	2.638
	3		0.417	0.073	2.901	2.933
$R^{2,10000}_{7000,5}$	1	174.960	15.111	0.055	164.270	**151.724**
	2		7.520	0.001	164.083	160.889
$R^{3,10000}_{7000,5}$	1	216.540	8.870	0.270	**204.212**	208.332
	2		8.659	0.010	217.532	219.130
	3		21.035	0.006	242.765	245.690
$R^{2,10000}_{7000,10}$	1	184.114	42.287	0.184	192.899	**164.079**
	2		39.346	0.063	207.264	174.483
$R^{3,10000}_{7000,10}$	1	252.671	66.776	1.140	272.517	**221.280**
	2		31.455	0.045	255.263	231.313
	3		29.198	0.012	257.324	234.958
$R^{2,10000}_{7000,20}$	1	182.971	23.473	1.230	180.219	**179.054**
	2		22.954	0.052	185.053	180.504
$R^{3,10000}_{7000,20}$	1	**224.993**	36.775	16.417	238.974	231.190
	2		35.700	3.877	259.609	238.700
	3		36.281	1.348	276.999	255.068

Table 6: Average CPU time (in seconds) for HA with and without preprocessing

path deviation costs, thus increasing the possibilities of satisfying condition (6). For the considered Karasan instances, no type of preprocessing was effective for the biggest networks ($n \in \{60, 90\}$) with the smallest width, given that no robust 0-persistent nodes were found with the static or the dynamic procedures. For the biggest width, more robust 0-persistent nodes were detected by the dynamic strategy, except for $n = 90$ and $k = 3$.

In terms of the CPU times to solve the robust shortest path problem, of the three methods used, the combination with the dynamic procedure was the most effective for CPLEX. That is shown on Table 16, for almost all the instances, expect the biggest with $k = 3$. Other exceptions were observed for the LA and HA cases – Tables 14 and 15. However, for the latter instances, the combination with the dynamic preprocessing resulted better, but with worse CPU times, than CPLEX without preprocessing.

In general, for the widest Karasan networks, except $n = 90$, $k = 3$, HA was the fastest method to find a robust shortest path. More concretely, without preprocessing for the smallest networks and with preprocessing for the remaining. Instead, for the Karasan networks with the smallest width, LA combined with the static procedure was the fastest method for the smallest networks, and CPLEX combined with the dynamic preprocessing was the fastest for the biggest networks.

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,10000}_{500,5}$	1	1.859	0.310	0.033	1.478	**1.282**
	2		0.118	0.009	1.530	1.513
$R^{3,10000}_{500,5}$	1	2.981	1.074	0.361	3.157	**2.421**
	2		0.629	0.045	2.871	2.697
	3		0.444	0.012	2.834	2.578
$R^{2,10000}_{500,10}$	1	2.238	0.946	0.358	2.224	1.807
	2		0.613	0.008	2.066	**1.644**
$R^{3,10000}_{500,10}$	1	4.125	1.636	0.798	3.600	2.853
	2		1.331	0.292	3.540	**2.713**
	3		1.227	0.126	3.708	2.822
$R^{2,10000}_{500,20}$	1	3.028	1.628	0.793	3.010	2.325
	2		1.514	0.347	3.126	**2.194**
$R^{3,10000}_{500,20}$	1	4.787	2.005	2.082	4.286	4.312
	2		1.889	1.848	**4.095**	4.392
	3		1.818	1.508	4.302	4.368
$R^{2,10000}_{7000,5}$	1	251.859	122.866	0.937	272.025	**152.606**
	2		75.886	0.334	232.449	161.222
$R^{3,10000}_{7000,5}$	1	338.993	112.566	3.616	307.908	**211.678**
	2		62.828	0.897	271.701	220.017
	3		42.721	0.662	264.451	246.346
$R^{2,10000}_{7000,10}$	1	399.071	359.487	2.888	510.099	**166.783**
	2		323.788	0.632	491.706	175.052
$R^{3,10000}_{7000,10}$	1	486.093	384.061	18.511	589.802	238.651
	2		366.564	5.469	590.372	236.737
	3		348.150	0.662	576.276	**235.608**
$R^{2,10000}_{7000,20}$	1	378.675	235.617	22.316	392.363	200.140
	2		234.804	1.662	396.903	**182.114**
$R^{3,10000}_{7000,20}$	1	488.825	384.548	180.010	586.747	394.783
	2		371.658	58.189	595.567	293.012
	3		364.307	24.839	605.025	**278.559**

Table 7: Average CPU time (in seconds) for LA with and without preprocessing

6 Conclusions

In this work, a new technique was developed to identify robust 0-persistent nodes of a network. This technique is a dynamic version of the preprocessing strategy presented in [11], because the involved tests are updated as new paths are computed. The dynamic technique was exemplified and its improvement towards the former version was empirically tested. The experiments comprised random instances with different methods for generating costs, namely: (1) costs in $U(0, 100)$, (2) costs in $U(0, 10\,000)$ and (3) negatively correlated costs between scenarios, as well as (4) Karasan instances with costs in $U(0, 100)$,

The experiments showed that both methods were more effective for the first two sets of instances than for the last two. In particular, for the random networks it was easier to detect robust 0-persistent nodes for the smallest densities. On the contrary, for the Karasan networks that task was easier for the widest instances. Nevertheless, the dynamic preprocessing was almost always able to detect more robust 0-persistent nodes than the former static version. For these cases, the increase of the number of these nodes found by the dynamic method ranged between: (1) 11% and 20 675%, (2) 25% and 4864%, (3) 71% and 62 033%, and (4) 14% and 1100%. For some of the instances, the dynamic preprocessing was faster than the static, however, for most of them it was more time

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,10000}_{500,5}$	1	3.180	1.915	1.665	3.083	**2.914**
	2		1.820	1.585	3.232	3.089
$R^{3,10000}_{500,5}$	1	3.705	2.175	1.910	4.258	**3.970**
	2		1.935	1.835	4.177	4.487
	3		1.905	1.785	4.295	4.351
$R^{2,10000}_{500,10}$	1	4.087	2.705	2.205	3.983	3.654
	2		2.360	1.840	3.813	**3.476**
$R^{3,10000}_{500,10}$	1	4.161	2.490	1.920	4.454	**3.975**
	2		2.450	1.845	4.659	4.266
	3		2.390	1.825	4.871	4.521
$R^{2,10000}_{500,20}$	1	4.342	3.200	2.135	4.582	**3.667**
	2		3.140	1.830	4.752	3.677
$R^{3,10000}_{500,20}$	1	5.156	2.780	2.680	5.061	4.910
	2		2.710	2.597	4.916	5.141
	3		2.675	1.985	5.159	**4.845**
$R^{2,10000}_{7000,5}$	1	**136.069**	8.135	5.775	157.294	157.444
	2		7.345	2.150	163.908	163.038
$R^{3,10000}_{7000,5}$	1	**201.303**	8.480	3.505	203.822	211.567
	2		6.460	2.070	215.333	221.190
	3		5.290	1.885	227.020	247.569
$R^{2,10000}_{7000,10}$	1	**142.025**	13.055	4.200	163.667	168.095
	2		10.375	2.460	178.293	176.880
$R^{3,10000}_{7000,10}$	1	**207.203**	20.900	6.340	226.641	226.480
	2		17.665	5.115	241.473	236.383
	3		11.675	2.730	239.801	237.676
$R^{2,10000}_{7000,20}$	1	**149.038**	16.940	4.515	173.686	182.339
	2		15.285	2.675	177.384	183.127
$R^{3,10000}_{7000,20}$	1	**213.566**	25.740	16.360	227.939	231.133
	2		24.720	6.575	248.629	241.398
	3		22.250	4.470	262.968	258.190

Table 8: Average CPU time (in seconds) for CPLEX with and without preprocessing

demanding. For the latter cases, the increase in the running times was at most: (1) 522%, (2) 18%, (3) 20%, and (4) 35%.

The computational experiments also evaluated the performance of methods to find the robust shortest path before and after the application of the dynamic or the static preprocessing techniques. These tests involved the HA and the LA introduced in [10], as well as CPLEX for solving a linear version of the robust shortest path problem formulation given in (9).

The results obtained by each algorithm were similar for (1) and (2). In general, the LA and the HA after dynamic preprocessing outperformed their combination with the static version for almost all the cases. Besides, the LA was always more efficient with preprocessing than with no preprocessing at all. The same only happened with the HA for networks with a large number of nodes using the dynamic preprocessing when considering just $M = 1$ testing scenario, even for the cases for which the static approach was not efficient.

For (3), the number of robust 0-persistent nodes found statically and dynamically was small. Therefore, applying preprocessing in these cases, be it static or dynamic, was not advantageous in terms of the total running time. Preprocessing might, however, be useful in these cases, if the robust shortest path computation can be independent from the initial reduction of the network, given that $AP_d < AP_s$. The results were similar for (4) when the graph width is small in comparison with the

	M	P_s	P_d	N_s	N_d		M	P_s	P_d	N_s	N_d
$R^{2,100,NC}_{500,5}$	1	1.148	1.225	217	372	$R^{2,100,NC}_{7000,5}$	1	127.238	142.450	951	5406
	2	1.336	1.433	260	477		2	146.496	149.616	2132	6777
$R^{2,100,NC}_{500,10}$	1	1.225	1.384	10	58	$R^{2,100,NC}_{7000,10}$	1	148.747	159.104	3	1864
	2	1.463	1.756	17	140		2	164.714	174.971	16	3977
$R^{2,100,NC}_{500,20}$	1	1.461	1.572	0	1	$R^{2,100,NC}_{7000,20}$	1	142.003	146.625	0	0
	2	1.627	1.905	1	1		2	159.410	162.777	0	0

Table 9: Average preprocessing CPU times (in seconds) and number of detected robust 0-persistent nodes

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100,NC}_{500,5}$	1	**1.103**	0.118	0.016	1.266	1.241
	2		0.098	0.003	1.434	1.436
$R^{2,100,NC}_{500,10}$	1	**1.292**	0.263	0.126	1.488	1.510
	2		0.263	0.053	1.726	1.809
$R^{2,100,NC}_{500,20}$	1	**1.952**	0.616	0.526	2.077	2.098
	2		0.492	0.456	2.119	2.361
$R^{2,100,NC}_{7000,5}$	1	**140.581**	22.621	1.870	149.859	144.320
	2		20.574	0.056	167.070	149.672
$R^{2,100,NC}_{7000,10}$	1	**162.279**	59.872	29.073	208.619	188.177
	2		49.033	5.264	213.747	180.235
$R^{2,100,NC}_{7000,20}$	1	**179.833**	39.471	38.982	181.744	185.607
	2		39.059	37.243	198.469	200.020

Table 10: Average CPU time (in seconds) for HA with and without preprocessing

number of nodes. Yet, the number of detected robust 0-persistent nodes was much higher for graphs that are wider than longer. Again, in these cases, the dynamic method behaved better than the static with respect to the number of nodes found, as well as in terms of running times in general. Like before, when applying CPLEX, the total times for solving the problems did not always decrease when preprocessing was used.

When solving the problem with CPLEX, the reduction of the network was not enough to pay off the running times increase, for the biggest instances. In fact, in general, CPLEX was the least sensitive method to the application of preprocessing techniques, and also the most stable with

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100,NC}_{500,5}$	1	2.093	1.009	0.287	2.157	1.512
	2		0.924	0.014	2.260	**1.447**
$R^{2,100,NC}_{500,10}$	1	2.954	1.968	1.633	3.193	3.017
	2		1.914	1.056	3.377	**2.812**
$R^{2,100,NC}_{500,20}$	1	3.599	2.193	2.124	3.654	3.696
	2		1.084	1.080	**2.711**	2.985
$R^{2,100,NC}_{7000,5}$	1	294.481	287.377	24.298	414.615	166.748
	2		212.268	1.469	358.764	**151.085**
$R^{2,100,NC}_{7000,10}$	1	591.997	415.733	222.316	564.480	381.420
	2		411.210	77.898	575.924	**252.869**
$R^{2,100,NC}_{7000,20}$	1	609.060	405.509	403.522	**547.512**	550.147
	2		404.279	401.172	563.689	563.949

Table 11: Average CPU time (in seconds) for LA with and without preprocessing

	M	NP	AP_s	AP_d	TP_s	TP_d
$R^{2,100,NC}_{500,5}$	1	3.348	1.985	1.860	3.133	**3.085**
	2		1.860	1.690	3.196	3.123
$R^{2,100,NC}_{500,10}$	1	**3.544**	2.540	2.265	3.765	3.649
	2		2.470	2.170	3.933	3.926
$R^{2,100,NC}_{500,20}$	1	5.137	3.710	3.560	5.171	5.132
	2		2.610	2.600	**4.237**	4.505
$R^{2,100,NC}_{7000,5}$	1	**130.742**	7.335	3.865	134.573	146.315
	2		4.970	2.280	151.466	151.896
$R^{2,100,NC}_{7000,10}$	1	**150.769**	15.540	11.355	164.287	170.459
	2		14.002	5.255	178.716	180.226
$R^{2,100,NC}_{7000,20}$	1	**161.366**	32.995	31.025	174.998	177.650
	2		32.310	29.970	191.800	192.747

Table 12: Average CPU time (in seconds) for CPLEX with and without preprocessing

$w = 10$	M	P_s	P_d	N_s	N_d	$w = 20$	M	P_s	P_d	N_s	N_d
$K^{2,100}_{30,10}$	1	0.057	0.066	8	14	$K^{2,100}_{30,20}$	1	0.077	0.055	14	16
	2	0.067	0.078	9	18		2	0.082	0.071	20	23
$K^{3,100}_{30,10}$	1	0.074	0.090	1	1	$K^{3,100}_{30,20}$	1	0.068	0.092	4	4
	2	0.093	0.110	1	12		2	0.093	0.096	6	21
	3	0.109	0.130	2	14		3	0.103	0.106	8	24
$K^{2,100}_{60,10}$	1	0.108	0.134	0	0	$K^{2,100}_{60,20}$	1	0.103	0.127	3	9
	2	0.133	0.171	0	1		2	0.134	0.158	6	30
$K^{3,100}_{60,10}$	1	0.153	0.168	0	0	$K^{3,100}_{60,20}$	1	0.127	0.158	0	1
	2	0.171	0.212	0	0		2	0.159	0.182	0	14
	3	0.203	0.234	0	0		3	0.182	0.212	0	18
$K^{2,100}_{90,10}$	1	0.185	0.203	0	0	$K^{2,100}_{90,20}$	1	0.164	0.212	0	3
	2	0.192	0.248	0	0		2	0.219	0.251	0	10
$K^{3,100}_{90,10}$	1	0.216	0.277	0	0	$K^{3,100}_{90,20}$	1	0.217	0.258	0	0
	2	0.287	0.346	0	0		2	0.238	0.317	0	0
	3	0.292	0.410	0	0		3	0.301	0.367	0	0

Table 13: Average preprocessing CPU times (in seconds) and number of detected robust 0-persistent nodes

respect to the structure of the network. This was specially clear for Karasan instances, which were difficult to solve by the LA and HA, even though these are small size problems, but solved very quickly by CPLEX.

The biggest problems, in networks with 7000 nodes, 140 000 arcs and three scenarios, were solved in less than 16 seconds by the HA, 180 seconds by the LA, and 19 seconds by CPLEX, after preprocessing. In average, finding the robust shortest path in Karasan instances after dynamic preprocessing took up to 280 seconds with the HA, 86 seconds with the LA, and 3 seconds with CPLEX.

Acknowledgments This work has been partially supported by the Portuguese Foundation for Science and Technology under project grants PEst-OE/ EEI/UI308/2014 and SFRH/BD/51169/2010.

$w = 10$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,10}$	1	0.082	0.017	0.011	**0.074**	0.077
	2		0.015	0.007	0.082	0.085
$K^{3,100}_{30,10}$	1	0.153	0.031	0.027	**0.105**	0.117
	2		0.028	0.010	0.121	0.120
	3		0.025	0.008	0.134	0.138
$K^{2,100}_{60,10}$	1	0.295	0.257	0.113	0.365	**0.247**
	2		0.235	0.109	0.368	0.280
$K^{3,100}_{60,10}$	1	4.322	4.217	4.212	4.370	4.380
	2		4.202	4.127	4.373	4.339
	3		4.178	4.084	4.381	**4.318**
$K^{2,100}_{90,10}$	1	10.358	10.209	10.164	10.394	10.367
	2		10.188	9.924	10.380	**10.172**
$K^{3,100}_{90,10}$	1	281.696	280.593	280.053	280.809	280.330
	2		280.026	279.156	280.313	279.502
	3		279.532	278.433	279.824	**278.843**

$w = 20$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,20}$	1	**0.053**	0.009	0.004	0.086	0.059
	2		0.007	0.004	0.089	0.075
$K^{3,100}_{30,20}$	1	**0.072**	0.014	0.008	0.082	0.100
	2		0.014	0.005	0.107	0.101
	3		0.013	0.003	0.116	0.109
$K^{2,100}_{60,20}$	1	**0.132**	0.039	0.026	0.142	0.153
	2		0.030	0.013	0.164	0.171
$K^{3,100}_{60,20}$	1	0.217	0.076	0.063	**0.203**	0.221
	2		0.072	0.047	0.231	0.229
	3		0.069	0.023	0.251	0.235
$K^{2,100}_{90,20}$	1	0.336	0.242	0.210	0.406	0.422
	2		0.226	0.066	0.445	**0.317**
$K^{3,100}_{90,20}$	1	3.743	3.722	3.560	3.939	3.818
	2		3.608	3.114	3.846	3.431
	3		3.593	2.939	3.894	**3.306**

Table 14: Average CPU time (in seconds) for algorithm HA with and without preprocessing

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In T. Tokuyama, editor, *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer Berlin Heidelberg, 2007.

[3] D. Catanzaro, M. Labbé, and M. Salazar-Neumann. Reduction approaches for robust shortest path problems. *Computers & Operations Research*, 38:1610–1619, 2011.

[4] IBM. IBM ILOG CPLEX Optimization Studio. Retrieved from `http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/`.

[5] O. E. Karasan, M. C. Pinar, and H. Yaman. The robust shortest path problem with interval data. Technical report, Bilkent University, Ankara, Turkey, 2001.

$w = 10$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,10}$	1	0.161	0.009	0.004	**0.066**	0.070
	2		0.007	0.003	0.074	0.081
$K^{3,100}_{30,10}$	1	0.136	0.012	0.012	**0.086**	0.117
	2		0.010	0.005	0.103	0.115
	3		0.008	0.004	0.198	0.134
$K^{2,100}_{60,10}$	1	1.239	1.037	1.033	**1.145**	1.167
	2		1.037	1.026	1.170	1.197
$K^{3,100}_{60,10}$	1	13.702	13.038	13.037	13.191	13.205
	2		13.038	12.996	13.209	13.208
	3		13.040	12.838	13.243	**13.072**
$K^{2,100}_{90,10}$	1	8.886	8.095	8.096	8.280	8.299
	2		8.077	8.073	**8.269**	8.321
$K^{3,100}_{90,10}$	1	87.015	86.340	86.102	86.556	86.379
	2		86.112	85.998	86.399	86.344
	3		86.095	85.746	86.387	**86.156**
$w = 20$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,20}$	1	0.064	0.003	0.002	0.080	**0.057**
	2		0.002	0.001	0.084	0.072
$K^{3,100}_{30,20}$	1	0.115	0.009	0.008	**0.077**	0.100
	2		0.008	0.002	0.101	0.098
	3		0.007	0.001	0.110	0.107
$K^{2,100}_{60,20}$	1	0.435	0.084	0.040	0.187	**0.167**
	2		0.058	0.013	0.192	0.171
$K^{3,100}_{60,20}$	1	1.173	1.055	1.042	1.182	1.200
	2		1.050	0.126	1.209	**0.308**
	3		1.049	0.121	1.231	0.333
$K^{2,100}_{90,20}$	1	2.015	1.988	1.079	2.152	1.291
	2		1.980	0.360	2.199	**0.611**
$K^{3,100}_{90,20}$	1	22.374	22.034	22.010	22.251	22.268
	2		22.003	21.667	22.241	21.984
	3		21.982	21.036	22.283	**21.403**

Table 15: Average CPU time (in seconds) for algorithm LA with and without preprocessing

[6] R. Montemanni and L. Gambardella. The robust shortest path problem with interval data via Benders decomposition. *4OR – Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 3:315–328, 2005.

[7] I. Murthy and S.-S. Her. Solving min-max shortest-path problems on a network. *Naval Research Logistics*, 39:669–683, 1992.

[8] L. R. Nielsen, K. A. Andersen, and D. Pretolani. Bicriterion a priori route choice in stochastic time-dependent networks. Technical Report 10, Logistics/SCM, Research Group, Aarhus School of Business, 2006.

[9] M. Pascoal and M. Resende. Dynamic preprocessing for the minmax regret robust shortest path problem with finite multi-scenarios. Technical Report 10, INESC-Coimbra, Coimbra, September 2014. Retrieved from `http://www.uc.pt/en/org/inescc/res_reports_docs/rr_10_2014`.

[10] M. Pascoal and M. Resende. Minmax regret robust shortest path problem in a finite multi-scenario model. *Applied Mathematics and Computation*, 241:88–111, 2014.

$w = 10$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,10}$	1	1.909	1.820	1.703	1.877	1.769
	2		1.803	1.687	1.870	**1.765**
$K^{3,100}_{30,10}$	1	1.984	1.843	1.730	1.917	1.820
	2		1.750	1.697	1.843	1.807
	3		1.747	1.660	1.856	**1.790**
$K^{2,100}_{60,10}$	1	2.025	1.940	1.785	2.048	**1.919**
	2		1.940	1.750	2.073	1.921
$K^{3,100}_{60,10}$	1	2.263	2.140	2.120	2.293	2.288
	2		2.120	2.100	2.291	2.312
	3		2.120	2.010	2.323	**2.244**
$K^{2,100}_{90,10}$	1	2.389	2.040	2.010	2.225	**2.213**
	2		2.030	1.990	2.222	2.238
$K^{3,100}_{90,10}$	1	**2.475**	2.390	2.280	2.535	2.557
	2		2.285	2.200	2.572	2.546
	3		2.240	2.110	2.532	2.520
$w = 20$	M	NP	AP_s	AP_d	TP_s	TP_d
$K^{2,100}_{30,20}$	1	2.016	1.740	1.685	1.817	1.740
	2		1.680	1.635	1.762	**1.706**
$K^{3,100}_{30,20}$	1	2.077	1.690	1.705	1.758	1.797
	2		1.650	1.625	1.743	1.721
	3		1.635	1.605	1.738	**1.711**
$K^{2,100}_{60,20}$	1	2.207	2.050	1.910	2.153	2.037
	2		2.020	1.705	2.154	**1.863**
$K^{3,100}_{60,20}$	1	2.213	1.960	1.845	2.087	2.003
	2		1.945	1.805	2.104	1.987
	3		1.910	1.745	2.092	**1.957**
$K^{2,100}_{90,20}$	1	2.233	2.105	2.055	2.269	2.267
	2		2.080	1.830	2.299	**2.081**
$K^{3,100}_{90,20}$	1	**2.481**	2.380	2.345	2.597	2.603
	2		2.360	2.225	2.598	2.542
	3		2.210	2.210	2.511	2.577

Table 16: Average CPU time (in seconds) for CPLEX with and without preprocessing

[11] M. Pascoal and M. Resende. Reducing the minmax regret robust shortest path problem with finite multi-scenarios. In P. Bourguignon, R. Jeltsch, A. Pinto, and M. Viana, editors, *CIM Series in Mathematical Sciences: Dynamics, Games and Science III*. Springer-Verlag, to appear, 2014.

[12] G. Yu and J. Yang. On the robust shortest path problem. *Computers Operations Research*, 25:457–468, 1998.