Dynamic Arc-Flags in Road Networks

Gianlorenzo d'Angelo, Daniele Frigioni, Camillo Vitale

To cite this version:

Gianlorenzo d'Angelo, Daniele Frigioni, Camillo Vitale. Dynamic Arc-Flags in Road Networks. 10th International Symposium, SEA 2011, May 2011, Kolimpari, Chania, Crete, Greece. pp.88-99, 10.1007/978-3-642-20662-7_8 . hal-00644054

HAL Id: hal-00644054

https://inria.hal.science/hal-00644054

Submitted on 23 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dynamic Arc-Flags in Road Networks

Gianlorenzo D'Angelo, Daniele Frigioni, and Camillo Vitale

Department of Electrical and Information Engineering, University of L'Aquila,
Via Gronchi, 18, I–67100, L'Aquila, Italy.
{gianlorenzo.dangelo,daniele.frigioni}@univaq.it,
camillo.vitale@gmail.com

Abstract. In this work we introduce a new data structure, named *Road-Signs*, which allows us to efficiently update the Arc-Flags of a graph in a dynamic scenario. Road-Signs can be used to compute Arc-Flags, can be efficiently updated and do not require large space consumption for many real-world graphs like, e.g., graphs arising from road networks. In detail, we define an algorithm to preprocess Road-Signs and an algorithm to update them each time that a weight increase operation occurs on an edge of the network. We also experimentally analyze the proposed algorithms in real-world road networks showing that they yields a significant speed-up in the updating phase of Arc-Flags, at the cost of a very small space and time overhead in the preprocessing phase.

1 Introduction

Great research efforts have been done over the last decade to accelerate Dijkstra's algorithm on typical instances of transportation networks, such as road or railway networks (see [3] and [4] for recent overviews). This is motivated by the fact that transportation networks tend in general to be huge yielding unsustainable times to compute shortest paths. These research efforts have lead to the development of a number of so called speed-up techniques, whose aim is to compute additional data in a preprocessing phase in order to accelerate the shortest paths queries during an on-line phase. However, most of the speed-up techniques developed in the literature do not work well in dynamic scenarios, when edge weights changes occur to the network due to traffic jams or delays of trains. In other words, the correctness of these speed-up techniques relies on the fact that the network does not change between two queries. Unfortunately, such situations arise frequently in practice. In order to keep the shortest paths queries correct, the preprocessed data needs to be updated. The easiest way is to recompute the preprocessed data from scratch after each change to the network. This is in general infeasible since even the fastest methods need too much time.

Related Works. Geometric Containers [17], was the first technique studied in a dynamic scenario [18]. The key idea is to allow suboptimal containers after a few updates. However, this approach yields quite a loss in query performance. The same holds for the dynamic variant of Arc-Flags proposed in [1], where,

after a number of updates, the query performances get worse yielding only a low speed-up over Dijkstra's algorithm. In [15], ideas from highway hierarchies [14] and overlay graphs [16] are combined yielding very good query times in dynamic road networks. In [2], a theoretical approach to correctly update overlay graphs has been proposed, but the proposed algorithms have not been shown to have good practical performances in real-world networks. The ALT algorithm, introduced in [8] works considerably well in dynamic scenarios where edge weights can increase their value that is, when delays or traffic jams increase travel times. Also in this case, query performances get worse if too many edges weights change [5]. Summarizing, all above techniques work in a dynamic scenario as long as the number of updates is small. As soon as the number of updates is greater than a certain value, it is better to repeat the preprocessing from scratch.

Contribution. In this paper we introduce a new data structure, named *Road-Signs*, which allows us to efficiently update the Arc-Flags of a graph in a dynamic scenario. Road-Signs can be used to compute Arc-Flags, they can be efficiently updated and do not require large space consumption for many real-world graphs like, e.g., graphs arising from road networks. In detail, we define an algorithm to preprocess Road-Signs and an algorithm to update them each time that a weight increase operation occurs on an edge of the graph. As the updating algorithm is able to correctly update Arc-Flags, there is no loss in query performance. To our knowledge, the only dynamic technique known in the literature with no loss in query performance is that in [15].

We experimentally analyze the proposed algorithms in real-world road networks showing that, in comparison to the recomputation from-scratch of Arc-Flags, they yield a significant speed-up in the updating phase of Arc-Flags, at the cost of a little space and time overhead in the preprocessing phase. In detail, we experimentally show that our algorithm updates the Arc-Flags at least 62 times faster than the recomputation from scratch in average, considering the graph where the new algorithm performs worse. Moreover it performs better when the network is big, hence it can be effectively used in real-world scenarios. In order to compute and store the Road-Signs, we need an overhead in the preprocessing phase and in the space occupancy. However, we experimentally show that such an overhead is very small compared to the speed-up gained in the updating phase. In fact, considering the graph where the new algorithm performs worse, the preprocessing requires about 2.45 and 2.88 times the time and the space required by Arc-Flags, respectively.

2 Preliminaries

A road network is modelled by a weighted directed graph $G = (V, E, w)$, called *road graph*, where nodes in V represent road crossings, edges in E represent road segments between two crossings and the weight function $w : E \to \mathbb{R}^+$ represents an estimate of the travel time needed for traversing road segments. Given G, we denote as $\bar{G} = (V, \bar{E})$ the *reverse graph* of G where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$.

A *minimal travel time route* between two crossings S and T in a road network corresponds to a *shortest path* from the node s representing S and the node t representing T in the corresponding road graph. The total weight of a shortest path between nodes s and t is called *distance* and it is denoted as $d(s,t)$. A partition of V is a family $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of subsets of V called *regions*, such that each node $v \in V$ is contained in exactly one region. Given $v \in R_k$, v is a *boundary node* of R_k if there exists an edge $(u,v) \in E$ such that $u \notin R_k$.

Minimal routes in road networks can be computed by shortest paths algorithm such as Dijkstra's algorithm [6]. In order to perform an s-t query, the algorithm grows a shortest path tree starting from the source node s and greedily visits the graph. The algorithm stops as soon as it visits the target node t. A simple variation of Dijkstra's algorithm is *bidirectional Dijkstra* which grows two shortest path trees starting from both nodes s and t. In detail, the algorithm performs a visit of G starting from s and a visit of $\bar{G}$ starting from t. The algorithm stops as soon the two visits meet at some node in the graph.

A widely used approach to speed up the computation of shortest paths is *Arc-Flags* [9, 11], which consists of two phases: a preprocessing phase which is performed off-line and a query phase which is performed on-line. The preprocessing phase of Arc-Flags first computes a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V and then associates a *label* to each edge (u,v) in E. A label contains, for each region $R_k \in \mathcal{R}$, a *flag* $A_k(u,v)$ which is true if and only if a shortest path in G towards a node in R_k starts with (u,v). The set of flags of an edge (u,v) is called *Arc-Flags* label of (u,v). The preprocessing phase associates also Arc-Flags labels to edges in the reverse graph $\bar{G}$. The query phase consists of a modified version of bidirectional Dijkstra's algorithm: the forward search only considers those edges for which the flag of the target node's region is true, while the backward search only follows those edges having a true flag for the source node's region. The main advantage of Arc-Flags is its easy query algorithm combined with an excellent query performance. However, preprocessing is very time-consuming. This is due to the fact that the preprocessing phase grows a full shortest path tree from each boundary node of each region yielding a huge preprocessing time. This results in a practical inapplicability of Arc-Flags in dynamic scenarios where, in order to keep correctness of queries, the preprocessing phase has to be performed from scratch after each edge weight modification.

3 Dynamic Arc-Flags

Given a road graph $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V in regions, we consider the problem of updating the Arc-Flags of G in a dynamic scenario where a sequence of weight-increase operations $C = (c_1, c_2, \ldots, c_h)$ occur on G. We denote as $G_i = (V, E, w_i)$ the graph obtained after i weight increase operations, $0 \leq i \leq h$, $G_0 \equiv G$. Each operation c_i increases the weight of one edge $e_i = (x_i, y_i)$ of an amount $\gamma_i > 0$, i.e. $w_i(e_i) = w_{i-1}(e_i) + \gamma_i$ and $w_i(e) = w_{i-1}(e)$, for each edge $e \neq e_i$ in E.

Since Arc-Flags of G are computed by considering shortest paths trees rooted at each boundary node induced by $\mathcal{R}$, a possible approach for dynamic Arc-Flags is to maintain these trees by using e.g. the dynamic algorithm in [7]. As the number of boundary nodes in large graphs is high, this approach is impractical.

In what follows, for sake of simplicity, we consider only Arc-Flags on the graph G as the inferred properties do not change for the reverse graph $\bar{G}$. Moreover, we assume that there exists a unique shortest path for any pair of nodes in G. The extension of the data structure and algorithms to the case of multiple shortest paths is straightforward as it is enough to break ties arbitrarily during the preprocessing and updating phases. The experimental study given in the next section considers such extension.

This section is organized as follows. First, we introduce the new data structure, which we call *Road-Signs* (denoted as S) and we show how to compute Road-Signs during the preprocessing phase of Arc-Flags. Then, we give an algorithm that uses Road-Signs in order to update the Arc-Flags. Finally, as Road-Signs result to be space expensive, we give a method to store them in a compact way, by obtaining a technique which is efficient for any kind of sparse graphs as, for instance, the road graphs used in the experimental study of the next section.

Data structure. Given an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the Road-Sign $S_k(u, v)$ of (u, v) to R_k is the subset of boundary nodes b of R_k, such that there exists a shortest path from u to b that contains (u, v). The Road-Signs of (u, v) are represented as a boolean vector, whose size is the overall number of boundary nodes in the network, where the i-th element is true if the i-th boundary node is contained in $S_k(u, v)$, for some region R_k. Hence, such a data structure requires $O(|E| \cdot |B|)$ memory, where B is the set of boundary nodes of G induced by $\mathcal{R}$.

The Road-Signs of G can be easily computed by using the preprocessing phase of Arc-Flags, which builds a shortest path tree from each boundary node on $\bar{G}$. Given an edge (u, v) and a region R_k, $A_k(u, v)$ is set to true if and only if (u, v) is an edge in at least one of the shortest path trees grown for the boundary nodes of R_k. Therefore, such a procedure can be easily generalized to compute also Road-Signs. In fact, it is enough to add the boundary node b to $S_k(u, v)$ if (u, v) is an edge in the tree grown for b.

Updating algorithm. Our algorithm to update Arc-Flags is based on the following Proposition, which gives us a straightforward method to compute the Arc-Flags of a graph given the Road-Signs of that graph.

Proposition 1. *Given $G = (V, E, w)$, a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the following conditions hold:*

- *if $u, v \in R_k$, then $A_k(u, v) = true$;*
- *if $S_k(u, v) \neq \emptyset$, then $A_k(u, v) = true$;*
- *if u or v is not in R_k and $S_k(u, v) = \emptyset$, then $A_k(u, v) = false$.*

In what follows, we hence give an algorithm to update Road-Signs. Let us consider a weight increase operation c_i on edge (x_i, y_i). The algorithm, denoted

Phase 1: DetectAffectedNodes(G_{i-1}, c_i, R_k)
Input : Graph G_{i-1}, operation c_i on edge (x_i, y_i) and region $R_k \in \mathcal{R}$
Output: Sets $B_k(u)$, for each $u \in V$

```
1  foreach u ∈ V do
2  └ B_k(u) := ∅;
3  B_k(x_i) := S_k(x_i, y_i);
4  Q.push(x_i, y_i);
5  repeat
6  │ (u, v) = Q.pop();
7  │ B_old := B_k(u);
8  │ B_k(u) := B_k(u) ∪ (B_k(v) ∩ S_k(u, v));
9  │ if B_k(u) \ B_old ≠ ∅ then
10 │ │ foreach z ∈ V such that (z, u) ∈ E do
11 │ └ └ Q.push(z, u);
12 until Q ≠ ∅;
```

Fig. 1. First phase of algorithm DynamicRoadSigns.

as DynamicRoadSigns, is based on the fact that if the shortest paths from a node u to a region R_k do not contain the edge (x_i, y_i), then the Road-Signs to R_k of the edges outgoing from u do not change as a consequence of c_i. Therefore, DynamicRoadSignsworks in two phases: the first phase, named DetectAffectedNodes, detects the set of nodes u such that a shortest path from u to b changes as a consequence of c_i (i.e. a shortest path from u to b contains edge (x_i, y_i)), where b is a boundary node in some region R_k s. t. $u \notin R_k$; the second phase, named UpdateRoadSigns, updates $S_k(u, v)$ for each region R_k and edge (u, v) where u is one of the nodes detected in the first phase.

DetectAffectedNodes consists of a modified breadth first search of the reverse graph $\bar{G}$, for each region R_k, which starts from node x_i and prunes when a node with no shortest paths to region R_k containing (x_i, y_i) is extracted. In this search, a node can be visited at most once for each boundary node of R_k. The output of this phase is a set $B_k(u)$, for each region $R_k \in \mathcal{R}$ and for each node $u \in V$, which contains the boundary nodes b of region R_k such that a shortest path from u to b contains edge (x_i, y_i). Note that, only edges (u, v) such that $B_k(u) \neq \emptyset$ for some region $R_k \in \mathcal{R}$ could change some of their Road-Signs and Arc-Flags towards region R_k, while edges (u, v) such that $B_k(u) = \emptyset$ for each $R_k \in \mathcal{R}$ do not change neither their Road-Signs nor their Arc-Flags. The pseudo-code of DetectAffectedNodes for a region $R_k \in \mathcal{R}$ is given in Fig. 1, where Q is the queue of the modified breadth first search. Operation $Q.push(x, y)$ inserts node x into Q and stores also the predecessor y of x in the visit. Operation $Q.pop()$ extracts a pair (x, y) where x is a node and y is the predecessor of x in the visit at the time when x is pushed into Q. At lines 1–3, $B_k(u)$ is initialized as $S_k(x_i, y_i)$ for $u = x_i$ and as the empty set for any other node. At lines 4–12, the graph search of $\bar{G}$ is performed, starting from

Phase 2: UpdateRoadSigns(G_{i-1}, c_i, R_k, B_k)
Input : Graph G_{i-1}, modification c_i on edge (x_i, y_i), region $R_k \in \mathcal{R}$, and sets $B_k(u)$, for each $u \in V$
Output: Updated Road-Signs

```
1  foreach b ∈ S_k(x_i, y_i) do
2      BinaryHeap.Clear();
3      foreach u : b ∈ B_k(u) do
4          D[u, b] := ∞;
5          foreach v such that (u, v) ∈ E and b ∉ B_k(v) do
6              Compute the distance from v to b and store it in D[v, b];
7          D[u, b] := min{w(u, v) + D[v, b] | (u, v) ∈ E and b ∉ B_k(v)};
8          if D[u, b] ≠ ∞ then
9              find the node z such that (u, z) ∈ E and b ∈ S_k(u, z);
10             S_k(u, z) := S_k(u, z) \ {b};
11             z' := argmin{w(u, v) + D[v, b] | (u, v) ∈ E and b ∉ B_k(v)};
12             S_k(u, z') := S_k(u, z') ∪ {b};
13         BinaryHeap.Push(u, D[u, b]);
14     while BinaryHeap ≠ ∅ do
15         (v, D[v, b]) :=BinaryHeap.Pop_Min();
16         foreach u such that (u, v) ∈ E and b ∈ B_k(u) do
17             if w(u, v) + D[v, b] < D[u, b] then
18                 D[u, b] := D[v, b] + w(u, v);
19                 BinaryHeap.node(u).Decrease(u, D[u, b]);
20                 find the node z such that (u, z) ∈ E and b ∈ S_k(u, z);
21                 S_k(u, z) := S_k(u, z) \ {b};
22                 S_k(u, v) := S_k(u, v) ∪ {b};
```

Fig. 2. Second phase of algorithm DynamicRoadSigns.

node x_i. When a node u is extracted for the first time from Q, $B_k(u)$ is set to $B_k(v) \cap S_k(u, v)$ at line 8, where v is the predecessor of u in the visit at the time when u is pushed into Q. If a node u is extracted more than once from Q (that is, if u reaches R_k using different paths for different boundary nodes of R_k), $B_k(u)$ is updated to $B_k(u) \cup (B_k(v) \cap S_k(u, v))$ at line 8 Finally, only nodes z such that $(z, u) \in E$ and some boundary nodes have been added to $B_k(u)$ at line 8 (i.e. $B_k(v) \cap S_k(u, v) \neq \emptyset$) are inserted in Q (lines 9–11). In this way a boundary node b of region R_k is inserted in $B_k(u)$ if and only if b is contained in all the Road-Signs in some path from u to x_i in G and hence, if and only if there exists a shortest path from u to b containing (x_i, y_i).

In the second phase, UpdateRoadSigns computes the shortest paths from a node u such that $B_k(u) \neq \emptyset$ to any boundary node in $B_k(u)$, for a given region $R_k \in \mathcal{R}$, and it updates the Road-Signs accordingly. Such shortest paths are computed as follows. First, for each node u such that $b \in B_k(u)$, for a certain boundary node $b \in S_k(x_i, y_i)$, a shortest path from u to b passing only

through neighbors of u whose shortest path to b do not contain (x_i, y_i), i.e. only nodes v such that $(u, v) \in E$ and $b \notin B_k(v)$, are considered. Then, the paths passing through the remaining neighbors of u are considered. The pseudo-code of UpdateRoadSigns is given in Fig. 2. The procedure uses a binary heap which is filled during the first computation of shortest paths (Lines 3–13) and it is used during the second computation (Lines 14–22) to extract the nodes in a greedy order, mimicking Dijkstra's algorithm. The cycle at Lines 1–22 considers only boundary nodes b belonging to the Road-Sign of edge (x_i, y_i). In the cycle at lines 3–13 the shortest paths from u to b through nodes v such that $(u, v) \in E$, $b \in B_k(u)$ and $b \notin B_k(v)$, are considered. In detail, at lines 5–6 the shortest paths from each node v to b are computed and the distances are stored in a data structure called $D[v, b]$. Note that, this step can be done by using Arc-Flags. At line 7 the estimated distance $D[u, b]$ from u to b is computed. At lines 9–12 the Road-Signs are updated according to the new distance: first (line 9) the node z such that $(u, z) \in E$ and $b \in S_k(u, z)$ is found (note that there is only a single node satisfying this condition as we are assuming that there is only one shortest path for each pair of nodes); then (line 10) b is removed from the Road-Sign of (u, z) and it is added to the Road-Sign of (u, z') (line 12), where z' is the neighbor of u giving the new estimated distance (line 11). Finally, at line 13, node u is pushed in the binary heap with priority given by the computed estimated distance. At Lines 14–22 the shortest paths from u to b through nodes v such that $(u, v) \in E$, $b \in B_k(u)$, and $b \in B_k(v)$, are considered. In detail, nodes v are extracted at line 15 in a greedy order, based on the distance to b. Then, for each node u such that $(u, v) \in E$ and $b \in B_k(u)$ (lines 16–22) a relaxation step is performed at lines 17–18, followed by a decrease operation in the binary heap (line 19) and the related update of the Road-Signs at lines 20–22. Each time that the Road-Signs are updated, the related Arc-Flags are updated according to Proposition 1. In detail, given an update on $R_k(u, v)$ for certain region $R_k \in \mathcal{R}$ and edge (u, v), then $A_k(u, v)$ is set to $true$ if $u, v \in R_k$ or $S_k(u, v) \neq \emptyset$, and it is set to $false$ otherwise. For simplicity, this step is not reported in the pseudo-code and it is indeed performed at lines 10, 12, 21, and 22 of UpdateRoadSigns.

Algorithm DynamicRoadSigns consists in calling procedures DetectAffectedNodes and UpdateRoadSigns, for each region $R_k \in \mathcal{R}$. The next theorem states the correctness of DynamicRoadSigns. Due to space limitations, the proof is given in the full paper.

Theorem 1. *Given $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, for each $(u, v) \in E$ and $R_k \in \mathcal{R}$,* DynamicRoadSigns *correctly updates $S_k(u, v)$ and $A_k(u, v)$ after a weight increase operation on an edge of G.*

Compacting Road Signs. Storing Road-Signs is very space consuming. Here, we give a simple method to reduce the memory space needed to store data structure S. Given a region R_k and a node $u \notin R_k$, let us denote as $B(R_k)$ the set of boundary nodes of R_k. By the definition of Road-Signs and the assumption that there exists only one shortest path between u and any boundary node b, the following two observation hold: (i) $B(R_k) = \bigcup_{(u,v) \in E} S_k(u, v)$; (ii) $S_k(u, v_1) \cap$

graph	n. of nodes	n. of edges	%MOT	%NAT	%REG	%URB
NED	892 027	2 278 824	0.4	0.6	5.1	93.9
LUX	30 647	75 576	0.6	1.9	14.8	82.7

Table 1. Tested road graphs. The first column indicates the graph; the second and the third columns show the number of nodes and edges in the graph, respectively; the last four columns show the percentage of edges into categories: motorways (MOT), national roads (NAT), regional roads (REG), and urban streets (URB).

$S_k(u, v_2) = \emptyset$, for each $v_1 \neq v_2$ such that $(u, v_1) \in E$ and $(u, v_2) \in E$. It follows that we can derive the Road-Sign of an edge (u, v), for an arbitrary v by the Road-Signs of other edges $(u, v') \in E$, $v' \neq v$, as $S_k(u, v) = B(R_k) \setminus \bigcup_{(u,v')\in E, v'\neq v} S_k(u, v')$. In this way, we do not store the Road-Sign of edge (u, v) and we simply compute it when it is needed, by using the above formula. As we can apply this method for each node $u \in V$, we avoid to store $|V|$ Road-Signs and hence the compacted data structure requires $O((|E|-|V|)\cdot|B|)$ space, where Road-Signs are represented as $|E| - |V|$ bit-vectors. Since in sparse graphs, like e.g. road networks $|E| \approx |V|$ the space requirement of Road-Signs is very small, as it is experimentally confirmed in the next section.

4 Experimental study

In this section, we first compare the performances of DynamicRoadSigns against the recomputation from scratch of Arc-Flags. Then, we analyze the preprocessing performances by comparing the time and space required to compute Arc-Flags against the time and space required to compute Arc-Flags and Road-Signs. The best query performances for Arc-Flags are achieved when partitions are computed by using arc-separator algorithms [12]. In this paper we used arc-separators obtained by the METIS library [10] and the implementation of Arc-Flags of [1].

Our experiments are performed on a workstation equipped with a 2.66 GHz processor (Intel Core2 Duo E6700 Box) and 8Gb of main memory. The program has been compiled with GNU g++ compiler 4.3.5 under Linux (Kernel 2.6.36).

We consider two road graphs available from PTV [13] representing the Netherlands and Luxembourg road networks, denoted as NED and LUX, respectively. In each graph, edges are classified into four categories according to their speed limits: motorways (MOT), national roads (NAT), regional roads (REG) and urban streets (URB). The main characteristics of the graphs are reported in Table 1. Due to the space requirements of Arc-Flags, we were unable to perform experiments on bigger networks.

Evaluation of the updating phase. To evaluate the performances of DynamicRoadSigns, we execute, for each graph considered and for each road category, random sequences of 50 weight-increase operations. That is, given a graph and a road category, we perform 50 weight-increase operations on edges

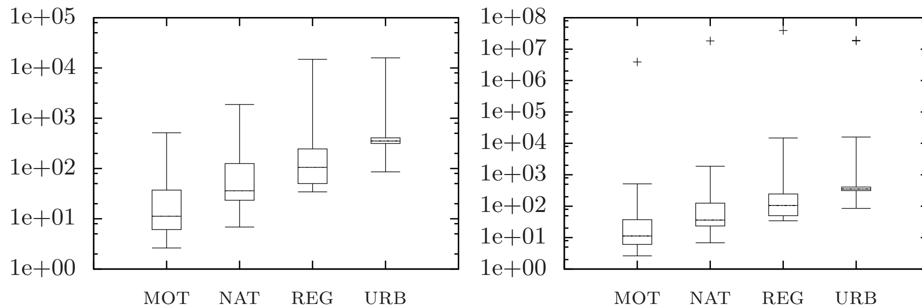

Fig. 3. Speed-up factors for the road network of the Netherlands, without (left) and with (right) outliers. For each road category, we represent minimum value, first quartile, medial value, third quartile, and maximum value.

belonging to the given category. The weight-increase amount for each operation is chosen uniformly at random in $[600, 1200]$, i.e., between 10 and 20 minutes. As performance indicator, we choose the time used by the algorithm to complete a single update during the execution of a sequence. We measure as speed-up factor the ratio between the time required by the recomputation from scratch of Arc-Flags and that required by DynamicRoadSigns. The results are reported in Fig. 3, Fig. 4, and Table 2.

Fig. 3 shows two box-plot diagrams representing the values of the speed-up factors obtained for the road network of Netherlands, for each road category. In detail, the diagram on the left side does not represent outlier values while the diagram on the right side do. These outlier values occur when DynamicRoadSigns performs much better than Arc-Flags because the number of Road-Signs changed is very small. Here, we consider a test as outlier if the overall number of boundary nodes involved in the computation is less than 15 i.e. $| \cup_{u \in V, R_k \in \mathcal{R}} B_k(u)| \leq 15$. Even without considering outliers, the speed-up gained by DynamicRoadSigns is high in most of the cases, reaching the value of 10 000 in some cases. It is worth noting that it reaches the highest values when update operations occur on urban edges while it is smaller when they occur on motorway edges. This is due to the fact that, when an update operation occurs on urban edges, the number of shortest paths that change as a consequence of such operation is small compared to the case that an update operation occurs on motorways edges. This implies that DynamicRoadSigns, which selects the nodes that change such shortest paths and focus the computation only on such nodes, performs better than the recomputation from-scratch of the shortest paths from any boundary node. Fig. 4 is similar to Fig. 3 but it is referred to the road network of Luxembourg. The properties highlighted for ned hold also for lux. We note that, for ned, the speed-up factors achieved are higher than that achieved for lux. This can be explained by the different sizes of the networks. In fact, when an edge update operation occurs, it affects only a part of the graph, hence only a subset of the edges in the graph need to update their Arc-Flags or Road-Signs. In most of the

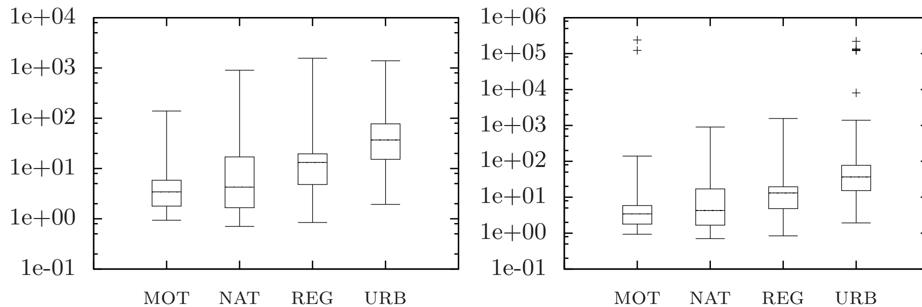

Fig. 4. Speed-up factors for the road network of Luxembourg. Without (left) and with (right) outliers. For each road category, we represent minimum value, first quartile, medial value, third quartile, and maximum value.

cases this part is small compared to the size of the network and, with high probability, it corresponds to the subnetwork close to the edge increased or closely linked to it. In other words, it is unlike that a traffic jam in a certain part of the network affects the shortest paths of another part which is far or not linked to the first one. Clearly, this fact is more evident when the road network is big and this explains the different performances between NED and LUX. Moreover, this allows us to state that DynamicRoadSigns would perform better if applied in networks bigger than those used in this paper, as continental networks.

As a further measure of the performances of DynamicRoadSigns against the recomputation from-scratch of Arc-Flags, we report the *average* computational time and speed-up factors in Table 2. It is evident here that DynamicRoadSigns outperforms the recomputation from-scratch by far and that it requires reasonable computational time which makes Road-Signs a technique suitable to be used in practice.

Evaluation of the preprocessing phase. Regarding the preprocessing phase, in Tables 3 and 4 we report the computational time and the space occupancy required by Arc-Flags and DynamicRoadSigns. Table 3 shows that, for computing Road-Signs along with Arc-Flags, we need about 2 times the computational time required for computing only Arc-Flags, which is a very small overhead compared to the speed-up gained in the updating phase. The same observation can be done regarding the space occupancy. In fact, Table 4 shows that the space required for storing both Road-Signs and Arc-Flags is between 1.77 and 2.88 that required to store only Arc-Flags. It is worth noting that without the compact storage of data structure S described in the previous section, S would require 12.78 and 4.13 times more space for NED and LUX, respectively.

5 Conclusions

We proposed a technique to correctly update Arc-Flags in dynamic graphs. In particular, we introduced the Road-Sign data structure, which can be used to

graph	cat.	avg. time Arc-Flags		avg. time DynamicRoadSigns		ratio		avg. speed-up	
NED	MOT	2 418.09	2 413.99	246.73	92.82	9.80	25.99	51.30	425.32
	NAT	2 397.14		74.71		32.08		169.82	
	REG	2 420.72		27.91		86.73		470.48	
	URB	2 416.22		7.63		316.67		1053.03	
LUX	MOT	8.25	8.28	2.96	2.04	2.79	4.06	11.70	62.87
	NAT	8.24		3.05		2.70		47.07	
	REG	8.32		1.46		5.70		78.06	
	URB	8.32		0.54		15.41		119.39	

Table 2. Average update times and speed-up factors. The first column indicates the graph; the second column indicates the road category where the weight changes occur; the third and fourth columns show the average computational time in seconds for Arc-Flags and for DynamicRoadSigns, respectively; the fifth column shows the ratio between the values reported in the third and the fourth columns, that is the ratio of average computational times; the last column shows the average speed-up factor of DynamicRoadSigns against Arc-Flags, that is the average ratio between the computational times.

graph	n. of regions	prep. time AF (sec.)	prep. time AF + RS (sec.)	ratio
NED	128	2 455.21	4 934.10	2.01
LUX	64	8.29	20.33	2.45

Table 3. Preprocessing time. The first column shows the graph; the second one shows the number of regions; the third one shows the preprocessing time required for computing only Arc-Flags; the fourth column shows the preprocessing time required for computing both Arc-Flags and Road-Signs; and the last column shows the ratio between the values reported in the fourth and the third column.

compute Arc-Flags, can be efficiently updated and does not require large space consumption. Therefore, we gave two algorithms to compute the Road-Signs in the preprocessing phase and to update them each time that a weight increasing occurs. We experimentally analyzed the proposed algorithms and data structures in road networks showing that they yields a significant speed-up in the updating phase, at the cost of a small space and time overhead in the preprocessing phase.

The proposed algorithms are able to cope only with weight increase operations which is the most important case in road networks where the main goal is to handle traffic jams. However, when a weight decrease operation occurs (e.g. when a the traffic jams is over) a recomputation from scratch is needed. Therefore, an interesting open problem is to find efficient algorithms to update Road-Signs after weight decrease operations.

References

1. E. Berrettini, G. D'Angelo, and D. Delling. Arc-flags in dynamic graphs. In *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization,*

graph	n. of regions	space AF (B)	space AF and RS (B)	ratio
NED	128	36 461 184	64 612 836	1.77
LUX	64	604 608	1 744 531	2.88

Table 4. Preprocessing space requirements. The first column shows the graph; the second one shows the number of regions; the third one shows the space required for storing Arc-Flags; the fourth one shows space required for storing both Arc-Flags and Road-Signs by using the compact storage; and the last column shows the ratio between the values reported in the fourth and the third column.

and Systems (ATMOS 2009). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Germany, 2009.

2. F. Bruera, S. Cicerone, G. D'Angelo, G. D. Stefano, and D. Frigioni. Dynamic multi-level overlay graphs for shortest paths. *Mathematics in Computer Science*, 1(4):709–736, 2008.
3. D. Delling, R. Hoffmann, M. Kandyba, and A. Schulze. *Algorithm Engineering*, volume 5971 of *LNCS*, chapter 9, Case Studies, pages 389–445. Springer, 2010.
4. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.
5. D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In *6th Workshop on Exp. Alg. (WEA'07)*, volume 4525 of *LNCS*, pages 52–65, 2007.
6. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
7. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.
8. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
9. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *Shortest Path Computations: Ninth DIMACS Challenge*, volume 24 of *DIMACS Book*. 2009.
10. G. Karypis. METIS - A Family of Multilevel Partitioning Algorithms, 2007.
11. U. Lauther. An extremely fast, exact algorithm for finding shortest paths. *Static Networks with Geographical Background*, 22:219–230, 2004.
12. R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM J. Exp. Algorithmics*, 11:2.8, 2006.
13. PTV AG - Planung Transport Verkehr. http://www.ptv.de, 2008.
14. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *14th European Symp. on Alg. (ESA'06)*, volume 4168 of *LNCS*, pages 804–816. Springer, 2006.
15. P. Sanders and D. Schultes. Dynamic Highway-Node Routing. In *6th Workshop on Exp. Alg. (WEA'07)*, volume 4525 of *LNCS*, pages 66–79, 2007.
16. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *4th Workshop on Alg. Eng. and Experiments (ALENEX'02)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.
17. D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *11th European Symp. on Alg. (ESA'03)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.
18. D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM J. Exp. Algorithmics*, 10:1.3, 2005.