Arc-Flags in Dynamic Graphs*

Emanuele Berrettini[1], Gianlorenzo D'Angelo[1], and Daniel Delling[2]

[1] Department of Electrical and Information Engineering, University of L'Aquila, Italy. surreale@gmail.com gianlorenzo.dangelo@univaq.it
[2] Faculty of Informatics, Universität Karlsruhe (TH), delling@informatik.uni-karlsruhe.de

Abstract. Computation of quickest paths has undergoing a rapid development in recent years. It turns out that many high-performance route planning algorithms are made up of several basic ingredients. However, not all of those ingredients have been analyzed in a *dynamic* scenario where edge weights change after preprocessing. In this work, we present how one of those ingredients, i.e., Arc-Flags can be applied in dynamic scenarios.

Keywords: Shortest Path, Speed-Up Technique, Dynamic Graph Algorithm

1 Introduction

Finding best connections in transportation networks is a problem familiar to everybody who ever travelled. In general, Dijkstra's algorithm can find the quickest path between two points s and t if a proper model is applied. For transportation networks, this can be achieved by assigning travel times to the edges of the graph representing the transportation network. Unfortunately, transportation networks deriving from real-world applications tend to be huge yielding query times of several seconds. Hence, over the last decade, research focused on accelerating Dijkstra's algorithm on typical instances, e.g., road or railway networks (cf. [3] for a recent overview). Such so called speed-up techniques compute additional data during a preprocessing phase in order to accelerate the queries during the online phase. As we observed in [1], most of recent high-performance rely on basic ingredients.

Unfortunately, not all of those ingredients are proven to work in dynamic scenarios, i.e., edge weights change due to traffic jams or delays of trains. In other words, correctness of the techniques relies on the fact

* Work partially supported by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

that the graph does *not* change between two queries. Unfortunately, such situations arise frequently in practice. In this work, we show how to use one of those ingredients, called Arc-Flags, in such scenarios.

Related Work. As already mentioned, a lot of speed-up techniques have been introduced over the last years. Due to space limitations, we direct the interested reader to [3], which gives a recent overview on *static* route planning algorithms. For the rest of related work, we focus on published results on *dynamic* speed-up techniques.

Geometric containers [14], which can be interpreted as a predecessor of Arc-Flags, also attach a label to each edge that represents all nodes to which a shortest path starts with this particular edge. A dynamization has been published in [15] yielding suboptimal containers if edge weights decrease. In [12], ideas from highway hierarchies [11] and overlay graphs [13] are combined yielding very good query times in dynamic road networks. Moreover, the ALT algorithm, introduced in [7] works considerably well in dynamic scenarios as well [4]. A combination of ALT with contraction, called Core-ALT, even works in time-dependent dynamic road networks [2]. However, to the best of our knowledge, there are no published results on Arc-Flags in dynamic scenarios.

Our Contribution. In this paper, we propose a first approach to cope with Arc-Flags in dynamic graphs. In particular, we propose an algorithm that is able to update Arc-Flags in graphs subject to weight increase operations. Each time that a weight increasing occurs, the algorithm is able to efficiently update all relevant Arc-Flags without recomputation from scratch. In comparison to a from-scratch approach, our algorithm yields a faster update of the arc-flags for the price of a loss in query performance. However, our experimental evalutions (on real world road networks) shows that the decrease in query performance is minor compared to the speed-up gained in the update phase.

The methods developed here are related to [15] since Geometric Containers can be interpreted as predecessor of Arc-Flags. Like for Arc-Flags, preprocessing of Geoemtric Containers is time-consuming. Hence, in [15], the authors present methods how to update the containers in case of weight changes without recomputating all containers from scratch. Like the methods presented here, the main idea is to settle for suboptimal containers in case of delays. By this, query performance decreases after a certain number of updates. However, it turns out that this decrease is acceptable as long as the number of updates stays little.

Outline. In Section 2 we introduce the notation used in the paper; in Section 3 we present the dynamic algorithm for updating Arc-Flags; in Section 4 we experimentally analyze the performances of the algorithm; and in Section 5 we outline the conclusion of the paper.

2 Preliminaries

In this paper, a road network is modeled by *directed weighted graphs* $G = (V, E, w)$, where nodes in V represent road crossings, edges in E represent road segments between two crossings and the weight function $w : E \rightarrow \mathbb{R}^+$ represents an estimate of the travel time needed for traversing road segments.

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. The total weight of a shortest path between nodes s and t is called *distance* from s to t and it is denoted as $d(s, t)$.

A partition of the node set V is a family $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of subsets of V, such that each node $v \in V$ is contained in exactly one set $R_k \in \mathcal{R}$. An element of a partition is called a region. Given a node v in a region R_k, v is a *boundary node* of region R_k if there exists an edge $(u, v) \in E$ or $(v, u) \in E$ such that $u \notin R_k$. The set of boundary nodes of a region R_k is denoted as $B(R_k)$

Given a graph G, the *reverse graph* $\bar{G} = (V, \bar{E})$ of G is the graph where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$.

Bidirectional Dijkstra's Algorithm for Shortest Paths. Minimal routes in road networks can be computed by shortest paths algorithm such as *Dijkstra's algorithm* [5]. In order to perform an *s*-*t* query, the algorithm grows a shortest path tree starting from the source node s and greedily visiting the graph. The algorithm stops as soon as it visits the target node t. A simple variation of Dijkstra's algorithm is the *bidirectional Dijkstra's algorithm* which grows two shortest path trees starting from both nodes s and t. In detail, the algorithm starts a visit of G starting from s and a visit of the reverse graph $\bar{G}$ starting from t. The algorithm stops as soon the two visits meet at some node in the graph.

Static Arc-Flags. The classic *Arc-Flags* approach, introduced in [9, 10], divides the computation of shortest paths into two phases: a preprocessing phase which is performed off-line and a query phase which is performed on-line. The aim of the preprocessing phase is to compute in advance

some information about shortest paths. This information is used to speed up the shortest path computation which is performed in the query phase.

The preprocessing phase first computes a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V and then associates a *label* to each edge e in E. A label contains, for each region $R_k \in \mathcal{R}$, a *flag* $A_k(e)$ which is true if and only if a shortest path in G towards a node in R_k starts with e. The set of flags of an edge e is called *Arc-Flags* label of e. Furthermore, the preprocessing phase associates (backward) Arc-Flags labels to edges in the reverse graph $\bar{G}$. The query phase consists in 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 that have a set 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 all boundary nodes of each region yielding preprocessing times of several weeks for instances like the Western European road network. This results in practical inapplicability in dynamic scenarios where, in order to keep correctness of queries, the preprocessing phase has to be performed after each edge weight modification. Note that by investing much more memory consumption during preprocessing, the preprocessing time can be decreased to approximately one day [8]. Due to the high memory consumption, we settle for the boundary approach in this work. Still, all insights gained here can also applied to the centralized approach due to [8].

3 Dynamic Algorithm

In this section, we present an algorithm which is able to update the Arc-Flags of a graph G in order to correctly answer to shortest path queries when weight-increase operations occur on G.

The goal is to update arc labels without recomputation from scratch. Arc-Flags are set considering all shortest path trees rooted at each boundary node, hence a possible approach is to maintain shortest path trees for all the boundary nodes of the graph by using the dynamic algorithm in [6]. Given the huge number of boundary nodes in large graphs, this approach is impractical due to its memory overhead and time complexity. However, this method would guarantee optimal query performance (compared to a

full recomputation) since it maintains exact shortest paths and changes flags only where needed.

Our goal is to update Arc-Flags without storing too much additional data. Therefore, we accept a small efficiency loss in the query phase. The main idea is to define a threshold for each edge of the graph and compare it with the edge weight increase when it occurs. In this way, we can determine whether an edge becomes the starting edge of a shortest path to some boundary nodes after a weight-increase operation. However, we cannot determine whether an edge belonging to a shortest path before a weight-increase operation is still on a shortest path after the operation. Thus, we can keep correctness of Arc-Flags in dynamic scenarios without maintaining shortest path trees. On the other hand, we keep unnecessarily true flags which leads to an efficiency loss in the query phase.

In the remainder of the section, we consider only Arc-Flags on graph the G as the inferred properties do not change for the reverse graph $\bar{G}$. In the next section, the following results will be used on both G and $\bar{G}$.

Given a weighted graph $G = (V, E, w)$, and a partition $\mathcal{R} = \{R_1, R_2, \dots, R_r\}$ of V, let us suppose that G is subject to a set of weight-increase operations $C = (c_1, c_2, \dots, c_c)$. Let us denote as $G_i = (V, E, w_i)$ the graph obtained after i weight increase operations, $0 \leq i \leq c$, $G_0 \equiv G$. Each operation c_i increases the weight of one edge e_i in E 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.

Given an edge $e = (u, v)$ and a region R_k, the *minimum threshold* $\delta_{k,i}(e)$ of e in G_i with respect to R_k is defined as $w_i(u, v)$ plus the minimum difference between the distance from v to b and the distance from u to be b among all boundary nodes b of R_k, formally,

$$\delta_{k,i}(e) = \min \{w_i(u, v) + d_i(v, b) - d_i(u, b) \mid b \in B(R_k)\} .$$

In other words, $\delta_{k,i}(e)$ is the minimum weight increase which has to occur to edge e_i in order to make e lie on a shortest path towards region R_k.

Note that, for $0 \leq i \leq c$, for each region R_k, and for each edge e, $\delta_{k,i}(e) \geq 0$. In fact, if by contradiction we suppose that $\delta_{k,i}(e) < 0$, then it follows that for a boundary node b of R_k, $w_i(u, v) + d_i(v, b) < d_i(u, b)$, which contradicts the minimality of $d_i(u, b)$. Moreover $\delta_{k,0}(e) = 0$ if and only if $A_k(e) = TRUE$. In fact, by definition of Arc-Flags, $A_k(e) = TRUE$ if and only if $w_0(e) = d_0(u, b') - d_0(v, b')$ for some boundary nodes b' of R_k. It follows that

$$\delta_{k,0}(e) = \min \{w_0(u, v) + d_0(v, b) - d_0(u, b) \mid b \in B(R_k)\} \leq$$

$$\leq w_0(u,v) + d_0(u,b') - d_0(v,b') = 0.$$

The following lemma gives us a necessary condition to check whether the Arc-Flags of an edge needs to be set to TRUE.

Lemma 1. *Given a region R_k, then an edge e is on a shortest path towards R_k in G_i only if $\gamma_i \geq \delta_{k,i-1}(e)$.*

Proof. If $e = (u,v)$ is on a shortest path towards R_k already in G_{i-1}, then $\delta_{k,i-1}(e) = 0$ as $d_{i-1}(u,b) = w_{i-1}(u,v) + d_{i-1}(v,b)$ for a boundary node b in R_k. Thus the statement holds. Otherwise, edge $e = (u,v)$ is on a shortest path towards region R_k in G_i and it is not on a shortest path towards region R_k in G_{i-1}, which means that the weight increase operation occurred on an edge (u,w) outgoing from node u, that is $u \equiv u_i$ and $w \equiv v_i$. In this case, we prove the statement by contradiction, that is, we show that if $\gamma_i < \delta_{k,i-1}(e)$ then edge e in not on a shortest path towards R_k in G_i. Let b be the boundary node of R_k such that

$$\delta_{k,i-1}(e) = w_{i-1}(u,v) + d_{i-1}(v,b) - d_{i-1}(u,b),$$

then $\gamma_i < \delta_{k,i-1}(e)$ implies that

$$\gamma_i < w_{i-1}(u,v) + d_{i-1}(v,b) - d_{i-1}(u,b).$$

It follows that

$$w_{i-1}(u,v) + d_{i-1}(v,b) > d_{i-1}(u,b) + \gamma_i.$$

The last inequality implies that edge (u,v) is not on a shortest path towards b. ■

Minimum thresholds can be computed in the preprocessing phase, during the Arc-Flags computation. Hence, the computation of minimum thresholds does not increase the computational complexity of the preprocessing. For each region R_k, we store the minimum threshold of an edge e with respect to R_k in a data structure $\delta_k(e)$ which is updated each time that an edge weight modification occurs. Hence, storing minimum thresholds requires $O(m \cdot r)$ instead of $O(m \cdot \log r)$ required by Arc-Flags.

When a weight increase operation c_i occurs, we update Arc-Flags and minimum thresholds by using Algorithm UPDATE-ARC-FLAGS in Figure 1.

In detail, Algorithm UPDATE-ARC-FLAGS performs a breadth-first search for each region R_k in $\mathcal{R}$. For each visited edge e it checks

```
1  Algorithm: UPDATE-ARC-FLAGS
   input  : Graph G_{i-1}, weight increase operation c_i, 1 ≤ i ≤ c
   output: Arc-Flags A and minimum thresholds δ
2  foreach region R_k do
3      visit G_{i-1} by performing a breadth-first search
4      foreach visited edge e do
5          if A_k(e) == FALSE then
6              if γ_i ≥ δ_k(e) then
7                  A_k(e) = TRUE
8                  δ_k(e) = 0
9              else
10                 δ_k(e) = δ_k(e) − γ_i
```

Fig. 1. Algorithm UPDATE-ARC-FLAGS

whether it is not on a shortest path towards region k, that is $A_k(e) == FALSE$ (Line 5). In the affirmative case, it applies Lemma 1 by setting $A_k(e)$ to $TRUE$ and $\delta_k(e)$ to 0 if $\gamma_i \geq \delta_k(e)$ or by updating $\delta_k(e)$ to $\delta_k(e) - \gamma_i$ otherwise (Lines 6–10).

It is easy to see that Algorithm UPDATE-ARC-FLAGS requires $O((n+m) \cdot r)$ computational time as it performs r times a breadth-first search of graph G_{i-1}.

The next theorem shows the correctness of algorithm UPDATE-ARC-FLAGS and it follows from Lemma 1 and from the discussion above.

Theorem 1. *After weight increase operation c_i, for each region R_k and for each edge e, if e is on a shortest path towards region R_k in G_i then $A_k(e) = TRUE$.*

4 Experimental study

In this section, we experimentally analyze the algorithm presented. We first report the computational time of the preprocessing phase of Arc-Flags in order to compare it with the computational time of UPDATE-ARC-FLAGS. Then, we present query performances by comparing query time after the execution of UPDATE-ARC-FLAGS against the one obtained after the from scratch recomputation. We also compare the two algorithms by performing mixed sequences of preprocessing and query phases. Finally, we compare our approach with the traditional use of bidirectional Dijkstra to evaluate the speed-up gained by our technique.

Our experiments are performed with a Dual-Core AMD opteron Processor 2218 clocked at 2.6 GHz with 32 GB of main memory. The program was compiled with GNU g++ compiler 4.2 under SuSE Linux 10.3 (Kernel 2.6.22.17).

We consider three graphs that represent the Luxembourg, Dutch and German road networks. In each graph, nodes represent crossings, edges represent links between two crossings and the weights correspond to an estimate of the travel times needed to traverse links. Edges are classified into four categories according to their speed limits: motorways, national roads, regional roads and urban streets. The main characteristics of the graphs are reported in Table 1.

graph	**n. of nodes**	**n. of edges**	**%mot**	**%nat**	**%reg**	**%urb**
road network of Luxembourg	30 647	75 576	0.6	1.9	14.8	82.7
road network of Netherlands	892 027	2 278 824	0.4	0.6	5.1	93.9
road network of Germany	4 375 381	10 967 664	0.3	1.5	15.5	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 distribution of edges into categories: motorways (mot), national roads (nat), regional roads (reg), and urban streets (urb).

Preprocessing. Regarding the preprocessing phase, in Table 2 we report the computational time and the average percentage of TRUE flags of each edge obtained by partitioning the graph into 64 or 128 regions.

graph	**n. of regions**	**preprocessing time (sec.)**	**% TRUE flags**
road network of Luxembourg	64	27	46.2
road network of Netherlands	128	6369	42.7
road network of Germany	128	80981	42.8

Table 2. Preprocessing time. The first column shows the graph; the second one shows the number of regions; the third one shows the preprocessing time; and the last one shows the average percentage of TRUE flags.

To evaluate the performances of UPDATE-ARC-FLAGS, we execute, for each considered graphs and for each road category, random sequences made of a different number c of update operations ranging from 1 to 30. The edge-increase amount for each of them is chosen at random

in $[600, 1200]$, i.e., between 10 and 20 minutes. As performance indicator, we chose the average time (in seconds) used by the algorithm to complete a single update during the execution of a sequence. Experimental results for the Luxembourg, Dutch and German road networks are given in Figures 2, 3 and 4, respectively. In particular, each figure shows four diagrams related to the four road categories considered. Each diagram shows the average time needed by UPDATE-ARC-FLAGS to perform a single update operation, as a function of the number c of weight increase operations occurred in the sequence.

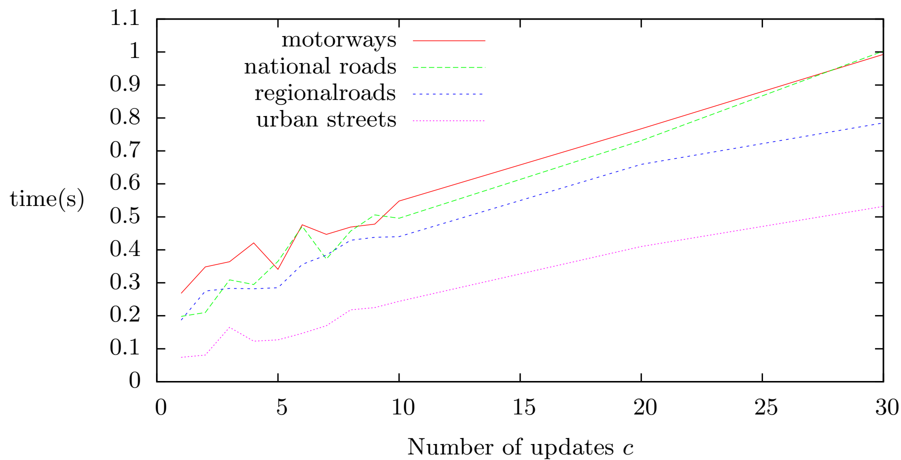

Fig. 2. Average time in seconds (y-axis) needed by UPDATE-ARC-FLAGS to complete a single operation during the execution of a different number of updates per sequence (x-axis) on the road network of Luxembourg. The weight increase is randomly selected in the interval $[600, 1200]$.

As one can see, the UPDATE-ARC-FLAGS is considerably faster than the preprocessing in all the tested graphs. As an example, performing 30 updates on motorways of the German network, using a from-scratch recomputation, would last 80980.8 seconds per update, which means that it would require 28 days, 2 hours, 50 minutes and 24 seconds overall time to perform 30 updates. Algorithm UPDATE-ARC-FLAGS needs only 215.8 seconds per update yielding 1 hour, 47 minutes and 55 seconds overall time. Thus, the speed-up achieved by UPDATE-ARC-FLAGS in this case is about 375. Table 3 shows the speed-up gained by

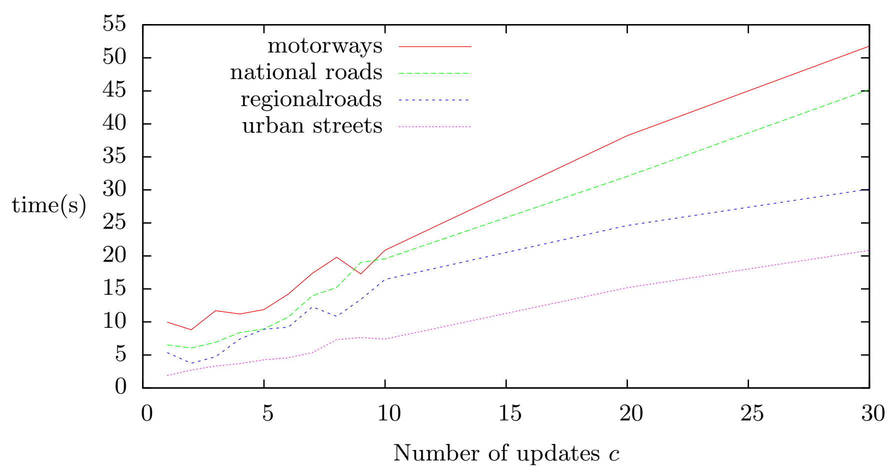

Fig. 3. Average time in seconds (y-axis) needed by UPDATE-ARC-FLAGS to complete a single operation during the execution of a different number of updates per sequence (x-axis) on the road network of Netherlands. The weight increase is randomly selected in the interval $[600, 1200]$.

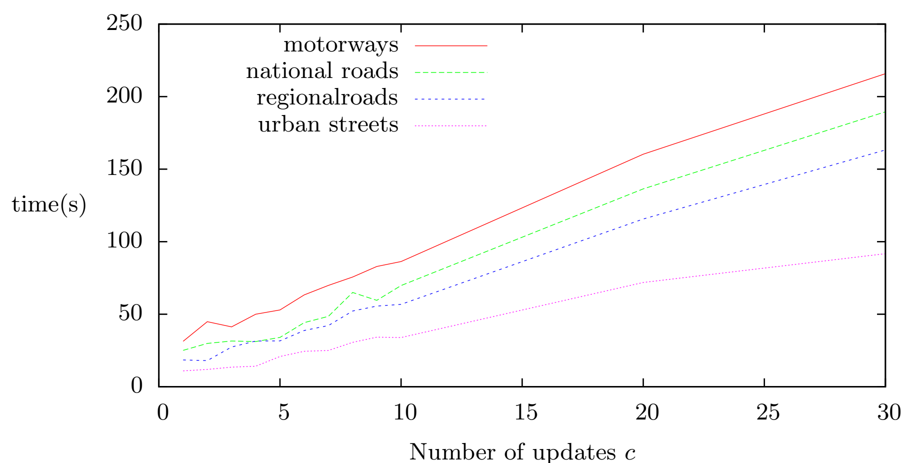

Fig. 4. Average time in seconds (y-axis) needed by UPDATE-ARC-FLAGS to complete a single operation during the execution of a different number of updates per sequence (x-axis) on the road network of Germany. The weight increase is randomly selected in the interval $[600, 1200]$.

UPDATE-ARC-FLAGS in the case of a sequence made of 30 weight increase operations.

Graph	Road category	speed-up
road network of Luxembourg	mot	26.89
	nat	26.62
	reg	34.01
	urb	50.19
road network of Netherlands	mot	123.01
	nat	140.86
	reg	211.43
	urb	305.58
road network of Germany	mot	375.17
	nat	427.31
	reg	496.06
	urb	882.87

Table 3. Speed-up gained by UPDATE-ARC-FLAGS in the case of a sequence made of 30 weight increase operations. The first column shows the graph; the second one shows the road category: motorways (mot), national roads (nat), regional roads (reg), and urban streets (urb); and the third one shows the speed-up gained by UPDATE-ARC-FLAGS with respect to a from-scratch approach.

Query Performance. In order to evaluate query performances, we run queries using source-target pairs that are picked uniformly at random. For each update sequence, first we update flags using UPDATE-ARC-FLAGS and then we run queries to evaluate the average query time. To measure the performance loss, we execute the same queries by using Arc-Flags updated by a from-scratch approach. Hence, we execute the preprocessing from-scratch on the modified graph, we perform the same sequence of queries and we compute the average query time. The parameter chosen to evaluate performances is the ratio between the average query time after the execution of UPDATE-ARC-FLAGS and the one obtained with the from-scratch recomputation. This value is referred at as *query performance loss* (*qpl*). In our experiments, we pick sequences of 10000 random source-target pairs. Figures 5, 6 and 7 show results about query performance loss on the three considered graphs. Each figure shows four diagrams which represents the query performance loss related to the four road categories considered.

As Figures 5, 6 and 7 show, using UPDATE-ARC-FLAGS to update flags after a weight-increase operation leads to a decrease of query per-

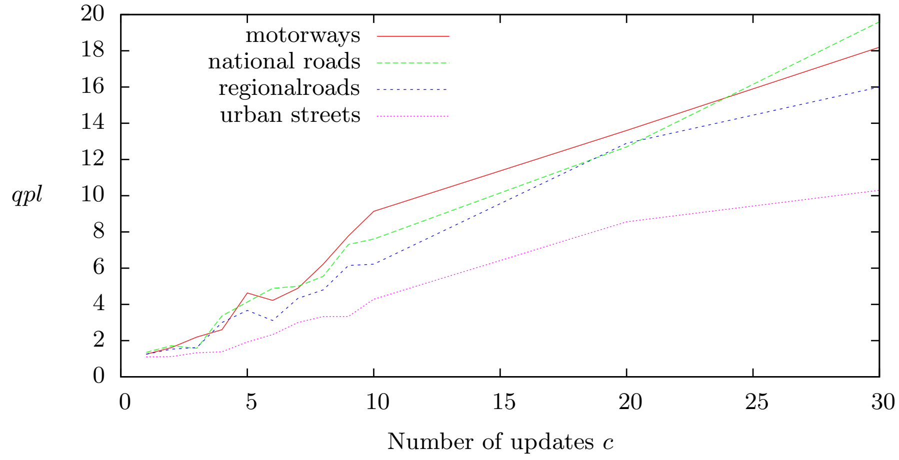

Fig. 5. Query performances after the execution of UPDATE-ARC-FLAGS on the road network of Luxembourg. The x-axis represents the number c of updates in the sequence, the y-axis represents the query performance loss (qpl).

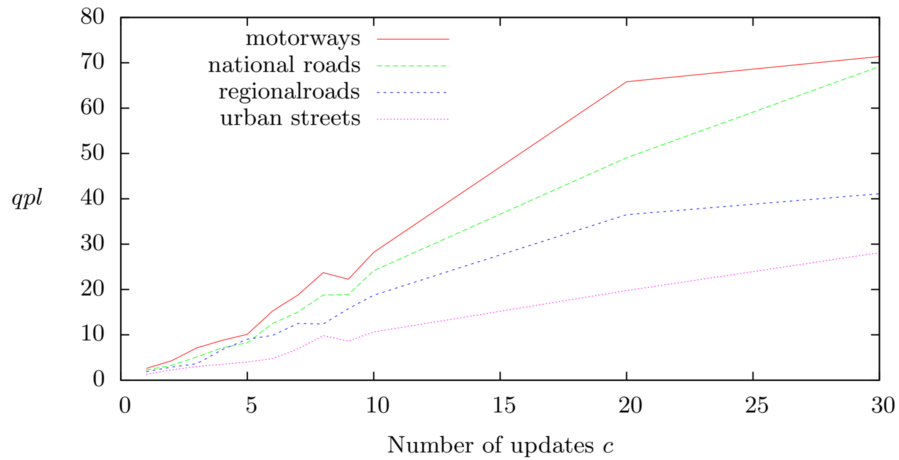

Fig. 6. Query performances after the execution of UPDATE-ARC-FLAGS on the road network of Netherlands. The x-axis represents the number c of updates in the sequence, the y-axis represents the query performance loss (qpl).

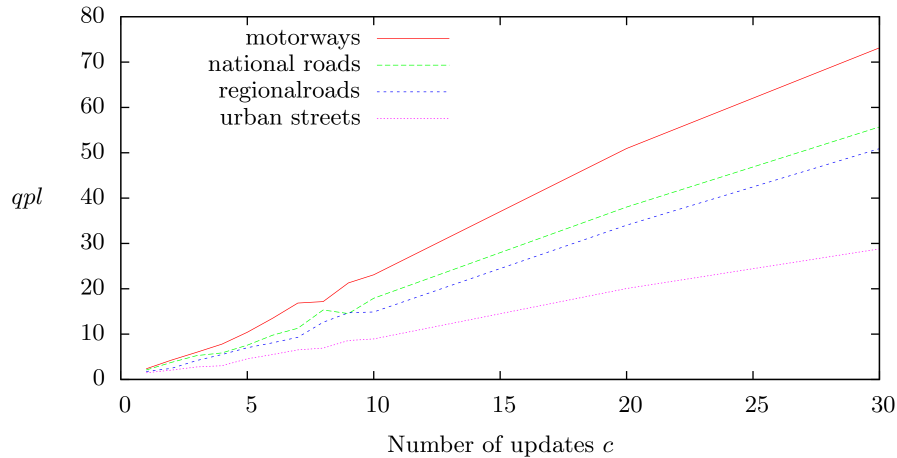

Fig. 7. Query performances after the execution of UPDATE-ARC-FLAGS on the road network of Germany. The x-axis represents the number c of updates in the sequence, the y-axis represents the query performance loss (qpl).

formances. Moreover, the query performance loss grows linearly with the number of updates. This is obvious, because UPDATE-ARC-FLAGS only changes flags from FALSE to TRUE. In this way, an Arc-Flag search would consider more edges as the number of updates become bigger leading to an increase of query time. It is also important to consider the information provided by Table 1: urban edges represents more than 80% in the road network of Luxembourg and in the German road network and more than 90% in the road network of Netherlands. For this category of edges, the use of UPDATE-ARC-FLAGS leads to a very small query performance loss. As an example, in the German network, after twenty updates on urban edges, queries are twenty times slower than after a from-scratch recomputation. This is due to the fact that urban streets mainly represent starting or ending edges of shortest paths and hence updates on these edges do not influences many Arc-Flags. Thus, if we consider a small number of updates, the use of UPDATE-ARC-FLAGS leads to query times that are comparable with those of pure Arc-Flags.

In conclusion, UPDATE-ARC-FLAGS is able to rapidly update Arc-Flags with a speed-up between 26 and 882 with respect to a from-scratch recomputation (see Table 3), and to achieve still good performances in the query phase with a performance loss of at most 73. Table 4 shows the relation between the speed-up gained by

UPDATE-ARC-FLAGS in the update phase and the query performance loss in the case of a sequence made of 30 weight increase operations. As one can see, the query performance loss is always much smaller than the speed-up.

Graph	**Road category**	**speed-up**	*qpl*
road network of Luxembourg	mot	26.89	18.2
	nat	26.62	19.6
	reg	34.01	16.0
	urb	50.19	10.3
road network of Netherlands	mot	123.01	71.39
	nat	140.86	69.18
	reg	211.43	41.13
	urb	305.58	28.11
road network of Germany	mot	375.17	73.15
	nat	427.31	55.71
	reg	496.06	50.9
	urb	882.87	28.78

Table 4. Relation between the speed-up gained by UPDATE-ARC-FLAGS in the update phase and the query performance loss in the case of a sequence made of 30 weight increase operations. The first column shows the graph; the second one shows the road category: motorways (mot), national roads (nat), regional roads (reg), and urban streets (urb); the third one shows the speed-up gained by UPDATE-ARC-FLAGS; and the last one shows the query performance loss (*qpl*).

Comparison. In order to evaluate the speed-up gained by our approach against the simple use of bidirectional Dijkstra, we perform mixed sequences of edge weight update and query operations. Each sequence is made of 1000 operations. In particular, we run a different number c of update operations ranging from 1 to 30, with a random edge-increase amount in $[600, 1200]$, and $1000-c$ queries using source-target pairs picked uniformly at random.

When the current operation in the sequence is an edge weight update, our approach performs UPDATE-ARC-FLAGS in order to run Arc-Flags when a subsequent query operation occurs. A traditional approach just stores the edge weight changes in $O(1)$ and runs bidirectional Dijkstra for all the subsequent query operations. As a performance meter, we choose the ratio r_{seq} between the overall time required by the traditional approach to perform the entire sequence of operations and that required by our approach. Results for the considered graphs and road categories are reported in Figures 8, 9 and 10.

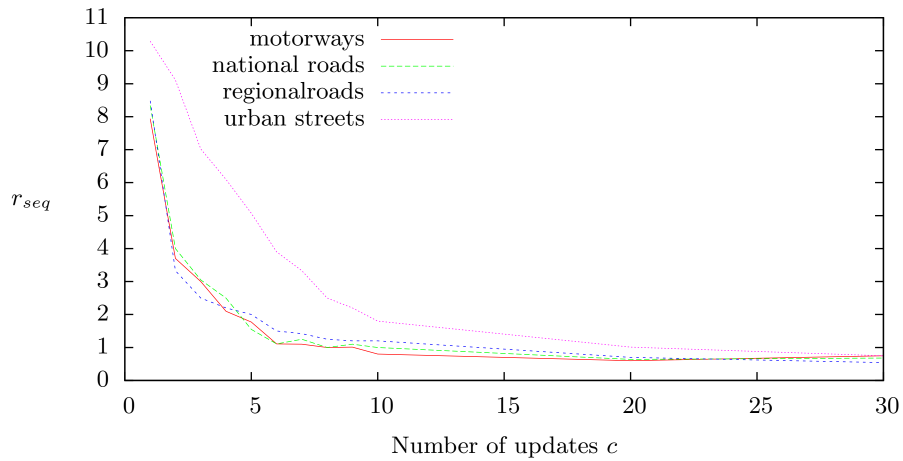

Fig. 8. Performances of our approach to perform mixed sequences of updates and queries on the road network of Luxembourg. The x-axis represents the number c of edge weight updates in the sequence, the y-axis represents the ratio r_{seq} between the time required by the traditional approach (bidirectional Dijkstra) and that required by our approach.

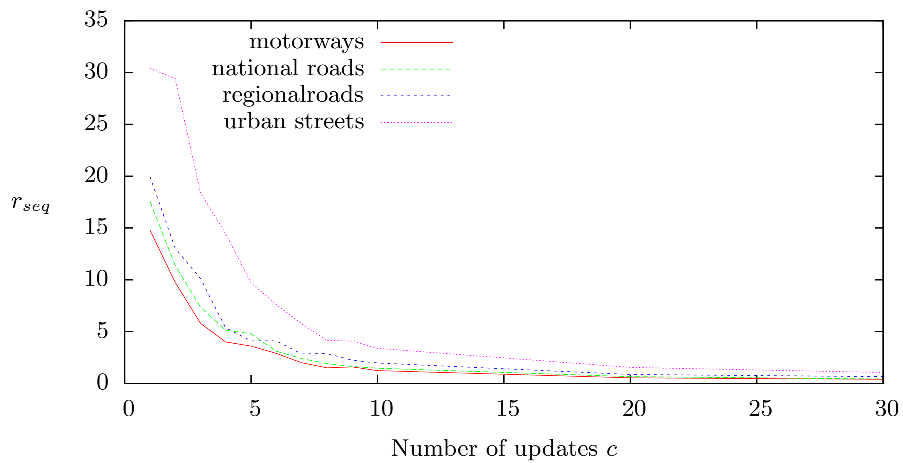

Fig. 9. Performances of our approach to perform mixed sequences of updates and queries on the road network of Netherlands. The x-axis represents the number c of edge weight updates in the sequence, the y-axis represents the ratio r_{seq} between the time required by the traditional approach (bidirectional Dijkstra) and that required by our approach.

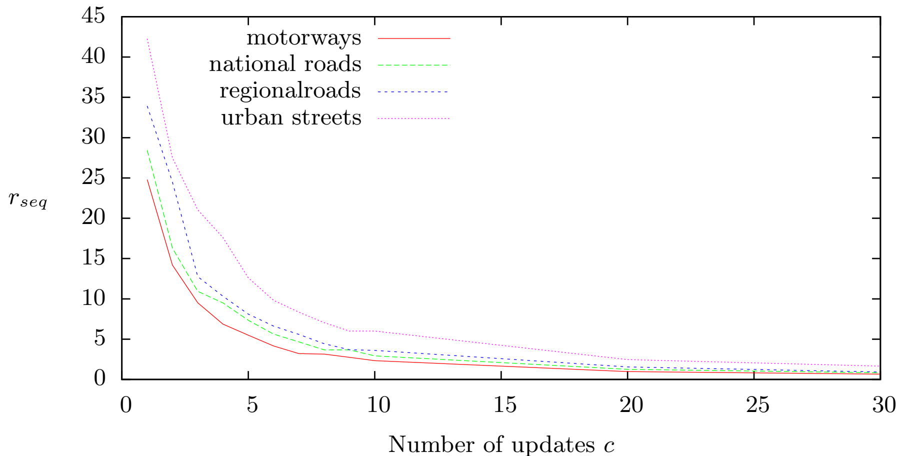

Fig. 10. Performances of our approach to perform mixed sequences of updates and queries on the road network of Germany. The x-axis represents the number c of edge weight updates in the sequence, the y-axis represents the ratio r_{seq} between the time required by the traditional approach (bidirectional Dijkstra) and that required by our approach.

As expected, r_{seq} tends to decrease with c. In particular it is bigger than 1 only when $c < 20$. This is due to the fact that the traditional approach does not perform any update phase while our approach performs UPDATE-ARC-FLAGS. This is slower than a simple bidirectional Dijkstra's query algorithm, even if it is faster than any other preprocessing algorithms as shown above. When the number c of weight increase operations in the sequence is high, this time overhead becomes evident, yielding to a value of r_{seq} which is smaller than 1. In addition to that, query performances decrease with the increase of c. This is due to the query performance loss induced by UPDATE-ARC-FLAGS. However, when c is less than 20 we can see that our approach leads to a significant speed-up especially in the bigger graph.

5 Conclusion

Despite the great interest dedicated during the last years to speed-up techniques for shortest paths, there are only few published algorithms which are proven to work in dynamic graphs. In this paper, we proposed a first approach to cope with Arc-Flags in dynamic graphs subject to weight increase operations.

The main idea is to define a threshold for each edge of the graph and compare it with the edge weight increase when it occurs. In this way, we are able to determine whether an edge label should be set to TRUE but we are not able to determine whether an it should be set to FALSE. Thus, we can keep correctness of Arc-Flags in dynamic scenarios in linear time without maintaining shortest path trees. On the other hand, we keep unnecessarily true flags which leads to efficiency loss in the query phase. Nevertheless, we experimentally show that such an efficiency loss is very small compared to the speed-up gained in the update phase.

References

1. D. Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.
2. D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.
3. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.
4. D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.
5. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
6. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.
7. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
8. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Path Computations: Ninth DIMACS Challenge*, volume 24 of *DIMACS Book*. American Mathematical Society, 2009. To appear.
9. U. Lauther. Slow preprocessing of graphs for extremely fast shortest path calculations. In *Workshop on Computational Integer Programming at ZIB*, 1997.
10. U. Lauther. An extremely fast, exact algorithm for finding shortest paths. *Static Networks with Geographical Background*, 22:219–230, 2004.
11. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
12. D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*,

volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

13. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.
14. D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.
15. D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10:1.3, 2005.