Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study*

Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, I-00198 Roma, Italy.
{demetres, frigioni, alberto, nanni}@dis.uniroma1.it

Abstract. We present the first experimental study of the fully dynamic single-source shortest paths problem in digraphs with arbitrary (negative and non-negative) arc weights. We implemented and tested several variants of the theoretically fastest fully dynamic algorithms proposed in the literature, plus a new algorithm devised to be as simple as possible while matching the best worst-case bounds for the problem. According to experiments performed on randomly generated test sets, all the considered dynamic algorithms are faster by several orders of magnitude than recomputing from scratch with the best static algorithm. The experiments also reveal that, although the simple dynamic algorithm we suggest is usually the fastest in practice, other dynamic algorithms proposed in the literature yield better results for specific kinds of test sets.

1 Introduction

The problem of finding efficient dynamic solutions for shortest paths has attracted a lot of interest in the last years, motivated by theoretical as well as practical applications. The problem is the following: we are given a graph G and we want to answer queries on the shortest paths of G, while the graph is changing due to insertions, deletions and weight updates of arcs. The goal is to update the information on the shortest paths more efficiently than recomputing everything from scratch after each update. If all the arc operations above are allowed, then we refer to the *fully dynamic* problem; if only insertions and weight decreases (deletions and weight increases) of arcs are supported, then we refer to the *partially dynamic incremental* (*decremental*) problem. The stated problem is interesting on its own and finds many important applications, including network optimization, document formatting, routing in communication systems, robotics. For a comprehensive review of the application settings for the static and dynamic shortest paths problem, we refer to [1] and [15], respectively.

Several theoretical results have been provided in the literature for the dynamic maintenance of shortest paths in graphs with positive arc weights (see,

* Partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT).

e.g., [7, 9, 15, 16]). We are aware of few efficient fully dynamic solutions for updating shortest paths in general digraphs with arbitrary (positive and non-positive) arc weights [10, 16].

Recently, an equally important research effort has been done in the field of *algorithm engineering*, aiming at bridging the gap between theoretical results on algorithms and their implementation and practical evaluation. Many papers have been proposed in this field concerning the practical performances of *static* algorithms for shortest paths (see e.g. [4, 5, 13]), but very little is known for the experimental evaluation of *dynamic* shortest paths algorithms: [8] considers the fully dynamic single source shortest paths problem in digraphs with positive real arc weights. We are not aware of any experimental study in the case of arbitrary arc weights. On the other hand, several papers report on experimental works concerning different dynamic graph problems (see e.g., [2, 3, 11]).

In this paper we make a step toward this direction and we present the first experimental study of the fully dynamic single-source shortest paths problem in digraphs with arbitrary (negative and non-negative) arc weights. We implemented and experimented several algorithms for updating shortest paths in digraphs with arbitrary arc weights that undergo sequences of weight-increase and weight-decrease operations. Our main goal was that of identifying with experimental evidence the more convenient algorithm to use in practice in a fully dynamic setting. The starting points of our experimental study were the classical Bellman-Ford-Moore's algorithm (e.g., see [1]) and the fully dynamic algorithms proposed by Ramalingam and Reps in [15, 16] and by Frigioni *et al.* in [10].

The solution in [15, 16] requires that all the cycles in the digraph before and after any input update have positive length. It runs in $O(||\delta|| + |\delta| \log |\delta|)$ per update, where $|\delta|$ is the number of nodes affected by the input change δ, and $||\delta||$ is the number of affected nodes plus the number of arcs having at least one affected endpoint. This gives $O(m + n \log n)$ time in the worst case.

The algorithm in [10] has a worst case complexity per update that depends on the *output complexity* of the update operation and on a structural parameter of the graph called *k-ownership*. Weight-decrease operations require $O(\min\{m, kn_a\} \log n)$ worst case time, while weight-increase operations require $O(\min\{m \log n, k(n_a + n_b) \log n + n\})$ worst case time. Here n_a is the number of affected nodes, and n_b is the number of nodes considered by the algorithm and maintaining both the distance and the parent in the shortest paths tree.

The common idea behind these algorithms is to use a technique of Edmonds and Karp [6], which allows it to transform the weight of each arc in a digraph into a non-negative real without changing the shortest paths, and to apply an adaptation of Dijkstra's algorithm to the modified graph. Differently from the case where all arc weights are non-negative (for which no efficient dynamic worst-case solution is known), with this technique it is possible to reduce from $O(mn)$ to $O(m + n \log n)$ the worst-case time of updating a shortest paths tree after a change of the weight of an arc in a graph with n nodes and m arcs.

As a first contribution of the paper, we confirm this claim from an experimental point of view. In particular, we observed that on randomly generated test sets, dynamic algorithms based on the technique of Edmonds and Karp are experimentally faster by several orders of magnitude than recomputing from scratch using the best static algorithm.

The paper also suggests a simple dynamic algorithm that hinges upon the technique of Edmonds and Karp without using complex data structures. The algorithm was devised to be as simple as possible while matching the $O(m + n \log n)$ bound of the best previous dynamic algorithms for the problem.

We implemented and experimentally evaluated all the aforementioned algorithms with the goal of improving their performance in practice. Experiments performed on randomly generated test sets showed that, though our simple dynamic algorithm is usually the fastest in practice, both the algorithms of Ramalingam and Reps and a simplified version of the algorithm of Frigioni *et al.* yield better results for specific kinds of test sets, e.g., where the range of values of arc weights is small. Our implementations were written in `C++` with the support of `LEDA` [14]. The experimental platform including codes, test sets generators and results can be accessed over the Internet at the URL: `ftp://www.dis.uniroma1.it/pub/demetres/experim/dsplib-1.1/` and was designed to make experiments easily repeatable.

2 Algorithms Under Evaluation

Let $G = (N, A, w)$ be a weighted directed graph with $n = |N|$ nodes and $m = |A|$ arcs, where w is function that associates to each $(x, y \in A$ a real weight $w_{x,y}$, and let $s \in N$ be a fixed *source* node. If G does not contain negative cycles, then, for each $x \in N$, we denote as $d(x)$ the minimum distance of x from s, and as $T(s)$ a shortest paths tree of G rooted at s. For each $x \in N$, $T(x)$ denotes the subtree of $T(s)$ rooted at x, $p(x)$ denotes the parent of x in $T(s)$, and $\text{IN}(x)$ and $\text{OUT}(x)$ denote the arcs of A incoming and outgoing x, respectively. The well known optimality condition of the distances of the nodes of a digraph $G = (N, A)$ states that, for each $(z, q) \in A$, $d(q) \leq d(z) + w_{z,q}$ (see, e.g., [1]). The new shortest paths tree in the graph G', obtained from G after an arc update, is denoted as $T'(s)$, while $d'(x)$ and $p'(x)$ denote the distance and the parent of x after the update, respectively.

We assume that the digraph G before an arc update does not contain negative cycles, and consider digraphs that undergo sequences of *decrease* and *increase* operations on the weights of arcs (*insert* and *delete* operations, respectively, can be handled analogously). We say that a node is *affected* by an input update if it changes the distance from the source due to that update.

Every time a dynamic change occurs in the digraph, we have two possibilities to update the shortest paths: either we recompute everything from scratch by using the best static algorithm, or we apply dynamic algorithms. In the following we analyze in detail these possibilities.

2.1 Static Algorithms

The best static algorithm for solving the shortest paths problem in the case of general arc weights is the classical Bellman-Ford-Moore's algorithm [1, 14] (in short, `BFM`). Many different versions of `BFM` have been provided in the literature (see [1] for a wide variety). The worst case complexity of all these variants is $O(mn)$. In [5] the authors show that the practical performances of **BFM** can be improved by using simple heuristics. In particular, they show that the heuristic improvement of **BFM** given in [13] is the fastest in practice. However, from a theoretical point of view, nothing better than the $O(mn)$ worst case bound is known. In our experiments, we considered the LEDA implementation of **BFM**.

2.2 Fully Dynamic Algorithms

We implemented the following fully dynamic algorithms: 1) the algorithm in [10], referred as **FMN**; 2) the algorithm in [16], referred as **RR**; 3) a simple variant of **FMN**, denoted as **DFMN**; 4) a new simple algorithm we suggest, denoted as **DF**.

The common idea behind all these algorithms is to use a technique of Edmonds and Karp [6], which allows it to transform the weight of each arc in a digraph into a non-negative real without changing the shortest paths. This is done as follows: after an input update, for each $(z, v) \in A$, replace $w_{z,v}$ with the *reduced weight* $r_{z,v} = d(z) + w_{z,v} - d(v)$, and apply an adaptation of Dijkstra's algorithm to the modified graph. The computed distances represent changes to the distances since the update. The actual distances of nodes after the update can be easily recovered from the reduced weights. This allows it to reduce from $O(mn)$ to $O(m + n \log n)$ the worst-case time of updating a shortest paths tree after a change of the weight of an arc in a digraph with n nodes and m arcs.

In what follows we give the main idea of the implemented algorithms to handle *decrease* and *increase* operations. For more details we refer to [10, 15, 16].

Weight decrease operations. Concerning the case of a *decrease* operation on arc (x, y), all the implemented algorithms basically update the shortest paths information by a Dijkstra's computation performed starting from node y, according to the technique of Edmonds and Karp. In Dijkstra's computation, when a node z is permanently labeled, *all* arcs (z, h) are traversed and the priority of h in the priority queue is possibly updated.

The only exception concerns **FMN**, where the following technique is exploited to bound the number of traversed arcs. For each node z, the sets $\textsc{in}(z)$ and $\textsc{out}(z)$ are partitioned into two subsets as follows. For each $x \in N$, $\textsc{in-own}(x)$ denotes the subset of $\textsc{in}(x)$ containing the arcs owned by x, and $\overline{\textsc{in-own}}(x) = \textsc{in}(x) - \textsc{in-own}(x)$ denotes the set of arcs in $\textsc{in}(x)$ not owned by x. Analogously, $\textsc{out-own}(x)$ and $\overline{\textsc{out-own}}(x)$ represent the arcs in $\textsc{out}(x)$ owned and not owned by x, respectively. Digraph G admits a k-ownership if, for all nodes x, both $\textsc{in-own}(x)$ and $\textsc{out-own}(x)$ contain at most k arcs (see [9] for more details). Finally, the arcs in $\overline{\textsc{in-own}}(x)$ ($\overline{\textsc{out-own}}(x)$) are stored in a min-based (max-based) priority queue where the priority of arc (y, x) ((x, y)) is the quantity $d(y) + w_{y,x}$ ($d(y) - w_{x,y}$). When the new distance of a node z is computed the above partition allows it to traverse only the arcs (z, h) in $\textsc{out-own}(z)$ and those in $\overline{\textsc{out-own}}(z)$, such that h is affected as well. This is possible by exploiting the priority of the arcs in $\overline{\textsc{out-own}}(z)$.

Weight increase operations. In the case of an *increase* of the weight of on arc (x, y) of a positive quantity ϵ, the implemented algorithms work in two phases.

First they find the affected nodes and then compute the new distances for the affected nodes. The second phase is essentially the same for all the algorithms, and consists of a Dijkstra's computation on the subgraph of G induced by the affected nodes, according to the technique of Edmonds and Karp. The main differences concern the first phase. As we will see, the only exception concerns `DF`, which avoids computing the first phase.

- `FMN`. The first phase of `FMN` is performed by collecting the nodes in a set M, extracting them one by one, and searching an alternative shortest path from s. To this aim, for each affected node z considered, only the arcs (h, z) in $\text{IN-OWN}(z)$ and those in $\overline{\text{IN-OWN}}(z)$, such that h is affected as well, are traversed. This is possible by exploiting the priority of the arcs in $\overline{\text{IN-OWN}}(z)$. This phase is quite complicated since it also handles zero cycles in an output bounded fashion.

- `DFMN`. The main difference of `DFMN` with respect to `FMN` is the elimination of the partition of arcs in owned and not-owned, that increases the number of arcs traversed (wrt `FMN`), but allows us to obtain a simpler and faster code.

- `RR`. Concerning `RR`, observe that it maintains a subset SP of the arcs of G, containing the arcs of G that belong to at least one shortest path from s to the other nodes of G. The digraph with node set N and arc set SP is a dag denoted as $SP(G)$. As a consequence, `RR` works only if all the cycles in the digraph, before and after any input update, have positive length. In fact, if zero cycles are allowed, then all of these cycles that are reachable from the source will belong to $SP(G)$, which will no longer be a dag. The first phase of `RR` finds the affected nodes as follows. It maintains a work set containing nodes that have been identified as affected, but have not yet been processed. Initially, y is inserted in that set only if there are no further arcs in $SP(G)$ entering y after the operation. Nodes in the work set are processed one by one, and when node u is processed, all arcs (u, v) leaving u are deleted from $SP(G)$, and v is inserted in the work set. All nodes that are identified as affected during this phase are inserted in the work set.

- `DF`. Now we briefly describe the main features of `DF`, in the case of an *increase* operation. `DF` maintains a shortest paths tree of the digraph G, and is able to detect the introduction of a negative cycle in the subgraph of G reachable from the source, as a consequence of an insert or a *decrease* operation. Zero cycles do not create any problem to the algorithm. Differently from `RR` and `FMN`, the algorithm has not been devised to be efficient in output bounded sense, but to be fast in practice, and costs $O(m+n \log n)$ in the worst case. The algorithm consists of two phases called `Initializing` and `Updating`. The `Initializing` phase marks the nodes in $T(y)$ and, for each marked node v, finds the best unmarked neighbor p in $\text{IN}(v)$. This is done to find a path (not necessarily a shortest path) from s to v in G' whose length is used to compute the initial priority of v in the priority queue H of the `Updating` phase. If $p \neq nil$ and $d(p) + w_{p,v} - d(v) < \epsilon$ then this priority is computed as $d(p) + w_{p,v} - d(v)$, otherwise it is initialized to ϵ, which is the variation of y's distance. In both cases the initial priority of the node is an upper bound on the actual variation. The `Updating` phase properly updates the information on the shortest paths from s to the marked nodes by performing a computation analogous to Dijkstra's algorithm.

In general, `FMN` and `RR` perform the Dijkstra's computation on a set of nodes which is a subset of the nodes considered by the `Updating` phase of `DF`. The `Initializing` phase of `DF` simply performs a visit of the subgraph of G induced by the arcs in $\textsc{in}(z)$, for each node z in $T(y)$, in order to find a temporary parent of z in the current shortest paths tree. This shows that `DF` is not output bounded.

3 Experimental Setup

In this section we describe our experimental framework, presenting the problem instances generators, the performance indicators we consider, and some relevant implementation details. All codes being compared have been implemented by the authors as `C++` classes using advanced data types of LEDA [14] (version 3.6.1). Our experiments were performed on a SUN Workstation Sparc Ultra 10 with a single 300 MHz processor and 128 MB of main memory running UNIX Solaris 5.7. All `C++` programs were compiled by the GNU `g++` compiler version 1.1.2 with optimization level O4. Each experiment consisted of maintaining both the distance of nodes from the source and the shortest paths tree in a random directed graph by means of algorithms `BFM`, `FMN`, `DFMN`, `RR` and `DF` upon a random mixed sequence of *increase* and *decrease* operations. In the special case of `BFM`, after each update the output structures were rebuilt from scratch.

3.1 Graph and Sequence Generators

We used four random generators for synthesizing the graphs and the sequences of updates:

- `gen_graph(n,m,s,min,max)`: builds a random directed graph with n nodes, m arcs and integer arc weights w s.t. `min`$\leq w \leq$`max`, forming no negative or zero length cycle and with all nodes reachable from the source node `s`. Reachability from the source is obtained by first generating a connecting path through the nodes as suggested in [5]; remaining arcs are then added by uniformly and independently selecting pairs of nodes in the graph. To avoid introducing negative and zero length cycles we use the potential method described in [12].
- `gen_graph_z(n,m,s,min,max)`: similar to `gen_graph`, but all cycles in the generated graphs have exactly length zero.
- `gen_seq(G,q,min,max)`: issues a mixed sequence of `q` *increase* and *decrease* operations on arcs chosen at random in the graph `G` without introducing negative and zero length cycles. Weights of arcs are updated so that they always fit in the range [`min`,`max`]. Negative and zero length cycles are avoided by using the same potentials used in the generation of weights of arcs of `G`. Optionally, the following additional constraints are supported:
  - *Modifying Sequence*: each increase or decrease operation is chosen among the operations that actually modify some shortest path from the source.
  - *Alternated Sequence*: the sequence has the form *increase-decrease-increase-decrease...*, where each pair of consecutive *increase-decrease* updates is performed on the same arc.
- `gen_seq_z(G,q,min,max)`: similar to `gen_seq`, but the update operations in the generated sequences force cycles in the graph `G` to have length zero.

All our generators are based on the LEDA pseudo-random source of numbers. We initialized the random source with a different odd seed for each graph and sequence we generated.

3.2 Performance Indicators

We considered several performance indicators for evaluating and comparing the different codes. In particular, for each experiment and for each code we measured: (a) the average running time per update operation during the whole sequence of updates; (b) the average number of nodes processed in the distance-update phase of algorithms. Again, this is per update operation during the whole sequence of updates.

Indicator (b) is very important in an output-bounded sense as it measures the actual portion of the shortest paths tree for which the dynamic algorithms perform high-cost operations such as extractions of minima from a priority queue. It is interesting to observe that, if an *increase* operation is performed on an arc (x, y), the value of the indicator (b) measured for both `RR` and `DFMN` reports the number of affected nodes that change their distance from the source after the update, while the value of (b) measured for `DF` reports the number of nodes in the shortest paths tree rooted at y before the update.

Other measured indicators were: (c) the maximum running time per update operation; (d) the average number of scanned arcs per update operation during the whole sequence of updates; (e) the total time required for initializing the data structures.

The running times were measured by the UNIX system call `getrusage()` and are reported in milliseconds. Indicators (b) and (d) were measured by annotating the codes with probes. The values of all the indicators are obtained by averaging over 15 trials. Each trial consists of a graph and a sequence randomly generated through `gen_graph` or `gen_graph_z` and `gen_seq` or `gen_seq_z`, respectively, and is obtained by initializing the pseudo-random generator with a different seed.

3.3 Implementation Details

We put effort to implementing algorithms `DFMN`, `RR` and `DF` in such a way that their running times can be compared as fairly as possible. In particular, we avoided creating "out of the paper" implementations of algorithms `DFMN` and `RR`. For example, in `RR` we do not explicitly maintain the shortest paths dag SP so as to avoid additional maintenance overhead that may penalize `RR` when compared with the other algorithms. Instead, we tried to keep in mind the high-level algorithmic ideas while devising fast codes.

For these reasons, we used just one code for performing *decrease* and we focused on hacking and tweaking codes for *increase*. We believe that the effect of using LEDA data structures does not affect the relative performance of different algorithms. More details about our codes can directly be found in our experimental package distributed over the Internet. In the remainder of this paper, we refer to `ALL-DECR` as the *decrease* code and to `DFMN`, `RR` and `DF` as the *increase* codes.

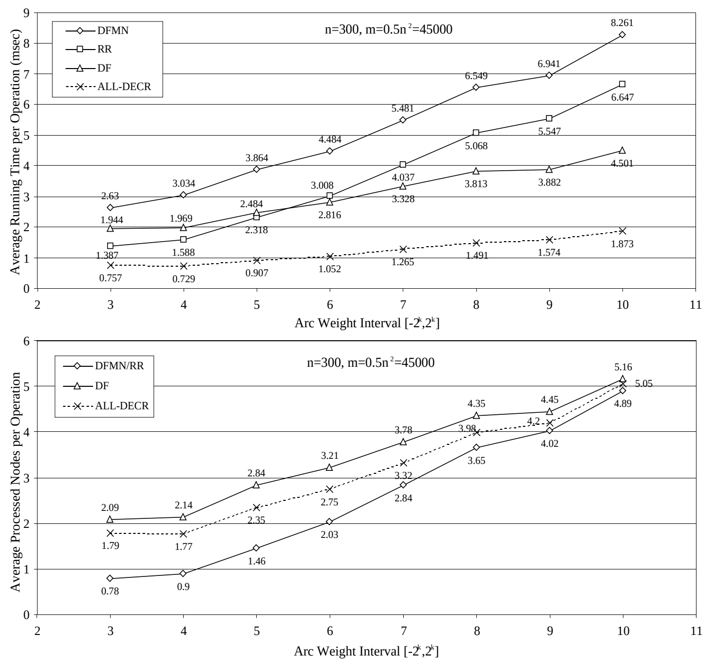

Fig. 1. Experiments performed with $n = 300$ and $m = 0.5n^2 = 45000$ for arc weight intervals increasing from $[-8, 8]$ to $[-1024, 1024]$.

4 Experimental Results

The goal of this section is to identify with experimental evidence the more convenient algorithm to use in practice in a fully dynamic environment. We performed several experiments on random graphs and random update sequences for different parameters defining the test sets aiming at comparing and separating the performances of the different algorithms.

Preliminary tests proved that, due to its complex data structures, `FMN` is not practical neither for *decrease* nor for *increase* operations, and so we focused on studying the performances of its simplified version `DFMN`.

Our first experiment showed that the time required by an *increase* may significantly depend upon the width of the interval of the arc weights in the graph:

– *Increasing arc weight interval*: we ran our `DFMN`, `RR` and `DF` codes on mixed sequences of 2000 modifying update operations performed on graphs with 300 nodes and $m = 0.5n^2 = 45000$ arcs and with arc weights in the range $[-2^k, 2^k]$ for values of k increasing from 3 to 10. The results of this test for *increase* operations are shown in Figure 1. It is interesting to note that the smaller is the width of the arc weight interval, the larger is the gap between the number of affected nodes considered by `RR` during any *increase* operation on an arc (x, y), and the number of nodes in $T(y)$ scanned by `DF`. In particular, `RR` is faster than `DF` for weight intervals up to $[-32, 32]$, while

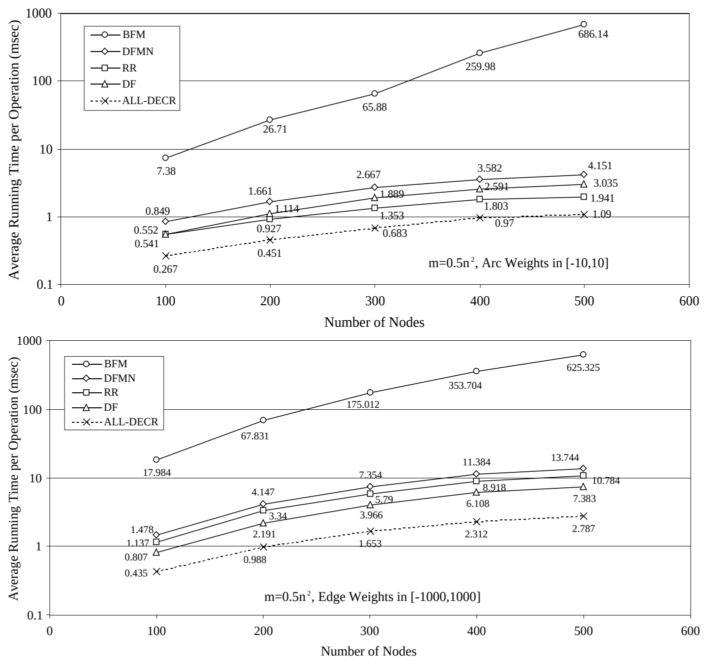

Fig. 2. Experiments performed with $100 \le n \le 500$ and $m = 0.5n^2$ for arc weights in the range $[-10, 10]$ and $[-1000, 1000]$.

DF improves upon RR for larger intervals. This experimental result agrees with the fact that RR is theoretically efficient in output bounded sense, but spends more time than DF for identifying affected nodes. The capacity of identifying affected nodes even in presence of zero cycles penalizes DFMN that is always slower than RR and DF on these problem instances with no zero cycles.

In our second suite of experiments, we ran BFM, DFMN, RR and DF codes on random sequences of 2000 modifying updates performed both on dense and sparse graphs with no negative and zero cycles and for two different ranges of the arcs weights. In particular, we performed two suites of tests:

– *Increasing number of nodes*: we measured the running times on dense graphs with $100 \le n \le 500$ and $m = 0.5n^2$ for arc weights in $[-10, 10]$ and $[-1000, 1000]$. We repeated the experiment on larger sparse graphs with $1000 \le n \le 3000$ and $m = 30n$ for the same arc weights intervals and we found that the performance indicators follow the same trend of those of dense graphs that are shown in Figure 2. This experiment agrees with the first one and confirms that, on graphs with arc density 50%, DF beats RR for large weight intervals and RR beats DF for small weight intervals. Notice that the dynamic codes we considered are better by several orders of magnitude than recomputing from scratch through the LEDA BFM code.

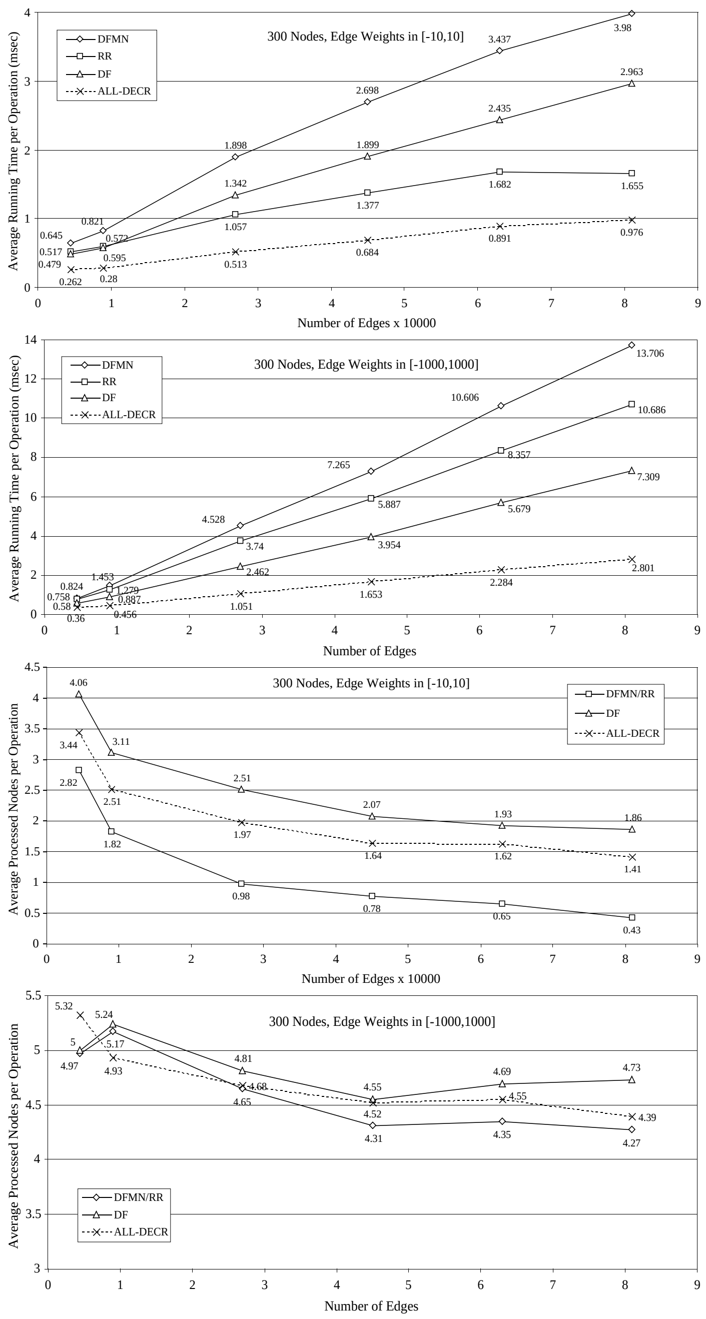

Fig. 3. Experiments performed with $n = 300$, $0.05n^2 \leq m \leq 0.9n^2$ for arc weights in the range $[-10, 10]$ and $[-1000, 1000]$.

– *Increasing number of arcs*: we retained both the running times and the number of nodes processed in the distance-update phase of algorithms on dense graphs with $n = 300$ and $0.05n^2 \leq m \leq 0.9n^2$ for arc weights in $[-10, 10]$ and $[-1000, 1000]$. We repeated the experiment on larger sparse graphs with $n = 2000$ and $10n \leq m \leq 50n$ for the same arc weights intervals and again we found similar results. Performance indicators for this experiment on dense graphs are shown in Figure 3 and agree with the ones measured in the first test for what concerns the arc weight interval width. However, it is interesting to note that even for small weight ranges, if the arc density is less than 10%, the running time of `DF` slips beneath that of `RR`.

As from the previous tests our `DFMN` code is always slower than `RR` and `DF`, our third experiment aims at investigating if families of problem instances exist for which `DFMN` is a good choice for a practical dynamic algorithm. As it is able to identify affected nodes even in presence of zero cycles, we were not surprised to see that `DFMN` beats in practice `DF` in a dynamic setting where graphs have many zero cycles. We remark that `RR` is not applicable in this context.

– *Increasing number of arcs and zero cycles*: we ran `DFMN` and `DF` codes on random graphs with 2000 nodes, $10n \leq m \leq 50n$, weights in $[-10, 10]$, all zero cycles, and subject to 2000 random alternated and modifying updates per sequence. We used generators `gen_graph_z` and `gen_seq_z` to build the input samples. Figure 4 shows the measured running times of increase operations for this experiment.

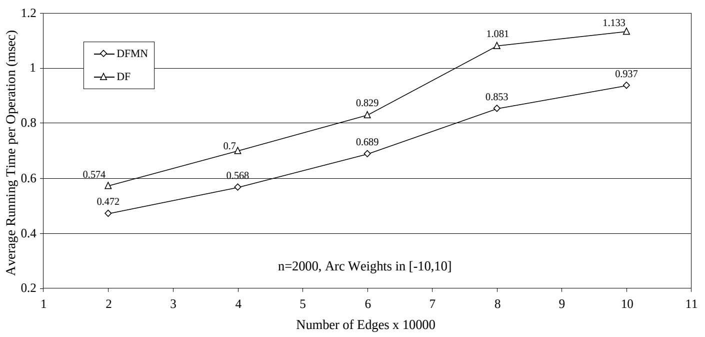

Fig. 4. Experiments performed with $n = 2000$, $10n \leq m \leq 50n$ for arc weights in $[-10, 10]$. All cycles have zero length during updates.

Performance indicators (c), (d) and (e) provided no interesting additional hint on the behavior of the algorithms and therefore we omit them from our discussion: the interested reader can find in the experimental package the detailed results tables of our tests.

5 Future Work

The continuation of the present work is along two directions: (1) Performing experiments on graphs from real life; (2) Reimplementing the algorithms in the `C` language, without the support of LEDA, with the goal of testing algorithms on larger data sets.

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. D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal on Experimental Algorithmics*, 2:Article 5, 1997.
3. G. Amato, G. Cattaneo, and G. F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *ACM-SIAM Symp. on Discrete Algorithms*, pp. 1–10, 1997.
4. B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symp. on Algorithms.* Lect. Notes in Comp. Sc. 1136, pp. 349–363, 1996.
5. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.
6. J. Edmonds, R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.
7. P. G. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic shortest paths and breadth-first search on digraphs. In *Symp. on Theoretical Aspects of Computer Science.* Lect. Notes in Comp. Sc. 1200, pp. 33–46, 1997.
8. D. Frigioni, M. Ioffreda, U. Nanni, G. Pasqualone. Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Path Problem. *ACM Journal on Experimental Algorithmics*, 3:Article 5 (1998).
9. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):351–381, 2000.
10. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *European Symp. on Algorithms.* Lect. Notes in Comp. Sc. 1461, pp. 320–331, 1998.
11. D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Shaefer, C. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *European Symp. on Algorithms.* Lect. Notes in Comp. Sc. 1461, pp. 368–380, 1998.
12. A. V. Goldberg. Selecting problems for algorithm evaluation. In *Workshop on Algorithm Engineering.* Lect. Notes in Comp. Sc. 1668, pp. 1–11, 1999.
13. A. V. Goldberg, and T. Radzik. A heuristic improvement of the Bellman-Ford algorithm. *Applied Math. Letters*, 6:3–6, 1993.
14. K. Mehlhorn and S. Naher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
15. G. Ramalingam. Bounded incremental computation. Lect. Notes in Comp. Sc. 1089, 1996.
16. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.