Soc. Netw. Anal. Min. (2017) 7:36
https://doi.org/10.1007/s13278-017-0457-y

ORIGINAL ARTICLE

Incremental maintenance of all-pairs shortest paths in relational DBMSs

Sergio Greco[1] · **Cristian Molinaro**[1] · **Chiara Pulice**[2] · **Ximena Quintana**[2]

Received: 6 February 2017 / Revised: 6 July 2017 / Accepted: 26 July 2017 / Published online: 1 August 2017
© Springer-Verlag GmbH Austria 2017

Abstract Computing shortest paths is a classical graph theory problem and a central task in many applications. Although many algorithms to solve this problem have been proposed over the years, they are designed to work in the main memory and/or with static graphs, which limits their applicability to many current applications where graphs are highly *dynamic*, that is, subject to frequent updates. In this paper, we propose novel efficient incremental algorithms for maintaining all-pairs shortest paths and distances in dynamic graphs. We experimentally evaluate our approach on several real-world datasets, showing that it significantly outperforms current algorithms designed for the same problem.

Keywords Shortest paths · Shortest distances · Incremental algorithms · Relational databasesn

This paper was invited as an extended version of Greco et al. (2016b) presented at the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), August 18–21, 2016, San Francisco, USA.

✉ Cristian Molinaro
cmolinaro@dimes.unical.it

Sergio Greco
greco@dimes.unical.it

Chiara Pulice
cpulice@umiacs.umd.edu

Ximena Quintana
x.quintana@dimes.unical.it

[1] University of Calabria, Rende, Italy

[2] University of Maryland, College Park, MD, USA

1 Introduction

Computing shortest path and distances has a wide range of applications in social network analysis (Khopkar et al. 2014; Kang et al. 2012, 2016; Shakarian et al. 2013), road networks Wu et al. (2012), graph pattern matching (Fan et al. 2010), biological networks (Rahman et al. 2005), and many others. It is both an important task in its own right (e.g., if we want to know how close people are in a social network) and a fundamental subroutine for many advanced tasks.

For instance, in social network analysis, many significant network metrics (e.g., eccentricity, diameter, radius, and girth) and centrality measures (e.g., closeness and betweenness centrality) require knowing the shortest paths or distances for all pairs of vertices. There are many other domains where it is needed to compute the shortest paths (or distances) for *all* pairs of vertices, including bioinformatics (Przulj et al. 2004) (where all-pairs shortest distances are used to analyze protein-protein interactions), planning and scheduling (Planken et al. 2012) (where finding the shortest paths for all pairs of vertices is a central task to solve binary linear constraints on events), and wireless sensor networks (Cuzzocrea et al. 2012) (where different topology control algorithms need to compute the shortest paths or distances for all pairs of vertices).

Though several techniques have been proposed to efficiently compute shortest paths/distances, they are designed to work in the main memory and/or with static graphs. This severely limits their applicability to many current applications where graph is dynamic and the graph along with the shortest paths/distances does not fit in the main memory. When graphs are subject to frequent updates, it is impractical to recompute shortest paths or distances from scratch every time a modification occurs.

To overcome these limitations, in this paper we propose efficient algorithms for the incremental maintenance of all-pairs shortest paths and distances in dynamic graphs stored in relational DBMSs.

Contributions We consider the setting where graphs and shortest paths are stored in relational DBMSs and propose novel algorithms to incrementally maintain all-pairs shortest paths and distances after vertex/edge insertions, deletions, and updates. The proposed approach aims at reducing the time needed to update the shortest paths by identifying only those that need to be updated (it is often the case that small changes affect only few shortest paths).

To the best of our knowledge, Pang et al. (2005) is the only disk-based approach in the literature for incrementally maintaining all-pairs shortest distances. In particular, like our approach, Pang et al. (2005) relies on relational DBMSs. We experimentally compare our algorithms against (Pang et al. 2005) on five real-world datasets, showing that our approach is significantly faster. It is worth noticing that our approach is more general than (Pang et al. 2005) in that we keep track of both shortest paths and distances, while Pang et al. (2005) maintain shortest distances only (thus, there is no information on the actual paths).

Organization In Sect. 2, we discuss related work. In Sect. 3, we introduce notation and terminology used throughout the paper. In Sect. 4, we present new efficient algorithms for the incremental maintenance of all-pairs shortest paths/distances. In Sect. 5, we report on an experimental evaluation. Conclusions are reported in Sect. 6.

2 Related work

Different variants of the shortest path problem have been investigated over the years: *single-pair shortest path* (SPSP)—find a shortest path from a given source vertex to a given destination vertex; *single-source shortest paths* (SSSP)—find a shortest path from a given source vertex to each vertex of the graph; *all-pairs shortest paths* (APSP)—find a shortest path from x to y for every pair of vertices x and y.

Variants of these problems where we are interested only in the shortest distances, rather than the actual paths, have been studied as well. We will use SPSD, SSSD, and APSD to refer to the *single-pair shortest distance*, *single-source shortest distances*, and *all-pairs shortest distances* problems, respectively.

In this section, we discuss the approaches in the literature to solve the above problems. We first consider non-incremental algorithms and then incremental ones.

Non-incremental algorithms Since the introduction of the Dijkstra's algorithm (Dijkstra 1959), a plethora of algorithms have been proposed to improve on its performance (see Sommer (2014) for a recent survey).

Most of the recently proposed methods require a preprocessing step for building index structures to support the fast computation of shortest paths and distances (Chang et al. 2012; Jin et al. 2012; Akiba et al. 2013; Goldberg and Werneck 2005; Zhu et al. 2013; Qi et al. 2013; Qiao et al. 2012; Zhu et al. 2013; Cheng et al. 2012; Fu et al. 2013).

Specifically, Chang et al. (2012), Jin et al. (2012) and Akiba et al. (2013) address the SPSD problem, and a similar approach for the SPSP problem has been proposed in Goldberg and Werneck (2005). Zhu et al. (2013) considers both the SPSP and SPSD problems. There have been also proposals addressing the approximate computation of SPSD (Qi et al. 2013; Qiao et al. 2012). These methods are based on the selection of a subset of vertices as "landmarks" and the offline computation of distances from each vertex to those landmarks. Zhu et al. (2013) and Cheng et al. (2012) propose disk-based index structures for solving the SSSP and SSSD problems, while disk-based index structure for the SPSP and SPSD problems have been proposed in Fu et al. (2013).

Gao et al. (2014) addresses the SPSP problem and, like our approach, relies on relational DBMSs.

Besides the fact that most of the aforementioned techniques assume that graphs, shortest paths/distances, and auxiliary index structures fit in the main memory (which is not realistic in many current applications), the main limit of all the approaches mentioned above is that they need to recompute a solution from scratch every time the graph is modified, even if we make small changes affecting a few shortest paths/distances. In many cases, this requires an expensive preprocessing phase to build the index structures that are necessary to answer queries. Then, of course, shortest paths/distances have to be computed again for all pairs. Thus, these methods work well if the graph is static. In contrast, if the graph is dynamic, the expensive preprocessing phase and the recomputation of all shortest paths/distances have to be done every time a change is made to the graph, and this is impractical for large graphs subject to frequent changes.

Incremental algorithms Several approaches have been proposed to incrementally maintain shortest paths and distances when the graph is modified.

Italiano (1988) introduces data structures to support APSP maintenance after edge deletions in directed acyclic graphs. Demetrescu and Italiano (2004) considers both edge insertions and deletions. Baswana et al. (2003) presents a hierarchical scheme for efficiently maintaining all-pairs approximate shortest paths in undirected unweighted

graphs and in the presence of edge deletions. King (1999) proposes an algorithm for maintaining all-pairs shortest paths in directed graphs with positive integer weights bounded by a constant. Another approach for the incremental maintenance of APSPs is Khopkar et al. (2014). Crecelius and Schenkel (2012) proposes an algorithm for maintaining nearest neighbor lists in weighted graphs under vertex insertions and decreasing edge weights. The approach is tailored for scenarios where queries are much more frequent than updates.

Approximate APSP maintenance in unweighted undirected graphs has been addressed in Baswana et al. (2003), Roditty and Zwick (2012) and Henzinger et al. (2013), while Bernstein (2013) considered edge deletions and weight increases in weighted directed graphs.

All the techniques above share the use of specific complex data structures and work in the main memory, which limits their applicability. In fact, as pointed out in Demetrescu et al. (2004) and Demetrescu and Italiano (2006), memory usage is an important issue of current approaches, limiting substantially the graph size that they can handle (for instance, the experimental studies in Demetrescu et al. (2004) and Demetrescu and Italiano (2006) could not go beyond graphs of 10,000 vertices).

Algorithms to incrementally maintain APSDs for graphs stored in relational DBMSs have been proposed in Pang et al. (2005). These algorithms improve on Gupta et al. (1993) (which addresses the more general view maintenance problem in relational databases) by avoiding unnecessary joins and tuple deletions. However, Pang et al. (2005) can deal only with the APSD maintenance problem while Gupta et al. (1993) works with general views (both in SQL and Datalog).

To the best of our knowledge, Pang et al. (2005) is the only disk-based approach for maintaining APSDs (incremental algorithms to find a path between two vertices in disk-resident route collections have been proposed in Bouros et al. (2009), but paths are not required to be shortest ones). We are not aware of disk-based approaches to incrementally maintain APSPs. Pang et al. (2005) is the closest work to ours, but with significant differences.

First, our algorithms maintain both shortest paths and shortest distances, while Pang et al. (2005) is able to maintain shortest distances only—knowing the actual (shortest) paths is necessary in different applications.

Second, while both (Pang et al. 2005) and our algorithms proceed by first identifying "affected" shortest paths (i.e., those whose distance might need to be updated after the graph is changed) and then acting on them, the two approaches differ in *how* this is done. Specifically, the strategy employed by our approach to identify affected shortest paths is different and more effective. For both edge insertions and deletions, we have two distinct phases, one looking forward and the other looking backward. This allows us to filter out earlier shortest paths that do not play a role in the maintenance process. On the other hand, Pang et al. (2005) identifies affected shortest paths in a more blind way: for both edge insertions and deletions, all shortest distances starting at the inserted/deleted edge are combined with all shortest distances ending in the inserted/deleted edge, leading to computationally expensive joins. Moreover, in our deletion algorithm, the recomputation of deleted shortest distances is done in an incremental way, as opposed to Pang et al. (2005), which performs this step by combining all shortest distances that are left after the deletion phase.

Third, as shown in our experimental evaluation (cf. Sect. 5), our algorithms are significantly faster than those of Pang et al. (2005).

The algorithms proposed in this paper generalize the ones presented in Greco et al. (2016b) in that the former are able to incrementally maintain both the shortest paths and the shortest distances, while the latter can maintain shortest distances only. Clearly, keeping track of actual paths (rather than distances only) requires storing and properly maintaining additional information. In this regard, the approach proposed in this paper maintains also the predecessor of the destination vertex in a shortest path in order to reconstruct the actual path (details are provided in Sect. 4). In contrast, the approach in Greco et al. (2016b) keeps track of distances only.

This paper is also related to Greco et al. (2016a), where incremental algorithms to maintain all-pairs shortest distances are proposed. There are two main differences between the two works. On the one hand, the approach in Greco et al. (2016a) is more general in that it can be instantiated both in main memory and on disk, while in this paper, we consider relational DBMSs only (however, it is worth noting that the experimental evaluation in Greco et al. (2016a) showed that the instantiation on relational DBMSs is the most efficient one). On the other hand, the approach in Greco et al. (2016a) can maintain shortest distances only, while in this paper we propose algorithms to maintain both shortest distances and shortest paths.

3 Preliminaries

In this section, we introduce the notation and terminology used in the rest of the paper.

A *graph* is a pair (V, A), where V is a finite set of *vertices* and $A \subseteq V \times V$ is a finite set of pairs called *edges*. A graph is *undirected* if A is a set of *unordered* pairs; otherwise (edges are *ordered* pairs) it is *directed*. A (directed or undirected) *weighted graph* consists of a graph

$G = (V, A)$ and a function $\varphi : A \to \mathbb{R}^+$ assigning a *weight* (or *distance*) to each edge in A.[1] We use $\varphi(u, v)$ to denote the weight assigned to edge (u, v) by φ.

A sequence $v_0, v_1, \ldots, v_n$ $(n > 0)$ of vertices of G is a *path* from v_0 to v_n iff $(v_i, v_{i+1}) \in A$ for every $0 \leq i \leq n-1$. The *weight* (or *distance*) of a path $p = v_0, v_1, \ldots, v_n$ is defined as $\varphi(p) = \sum_{i=0}^{n-1} \varphi(v_i, v_{i+1})$. Obviously, there can be multiple paths from v_0 to v_n, each having a distance. A path from v_0 to v_n with the lowest distance (over the distances of all paths from v_0 to v_n) is called a *shortest path* from v_0 to v_n (there can be multiple shortest paths from v_0 to v_n) and its distance is called the *shortest distance* from v_0 to v_n. We assume that the shortest distance from a vertex to itself is 0, and the shortest distance from a vertex v_0 to a distinct vertex v_n is not defined if there is no path from v_0 to v_n.

In the rest of the paper we consider directed weighted graphs and call them simply graphs—the extension of the proposed algorithms to undirected graphs is trivial. Without loss of generality, we assume that graphs do not have self-loops, i.e., edges of the form (v, v) (the reason is that self-loops can be disregarded for the purpose of finding shortest distances).

We consider the case where graphs and shortest paths are stored in relational databases. Notice that this allows us to take advantage of full-fledged optimization techniques provided by relational DBMSs.

Specifically, the set of edges of a graph is stored in a ternary relation E containing a tuple (a, b, w) iff there is an edge in the graph from a to b with weight w. We call E an *edge relation*. Vertices without incident edges can be ignored for our purposes.

A relation SP of arity 4 is used to represent the shortest paths for all pairs of reachable vertices of the graph (obviously, for a pair of reachable vertices we keep track of only one shortest path). Specifically, a shortest path $v_0, v_1, \ldots, v_n$ $(n \geq 1)$ from v_0 to v_n with (shortest) distance d_{0n} is represented with the set of tuples $\{(v_0, v_i, d_{0i}, v_{i-1}) \mid 1 \leq i \leq n\}$ in SP, where d_{0i} denotes the shortest distance from v_0 to v_i. Clearly, it is always possible to build the whole path from v_0 to v_n by following the chain of predecessors in SP starting from v_n. Analogous approaches are employed by different algorithms for shortest path computation, such as the well-known Dijkstra's algorithms. Thus, a tuple (x, y, d, z) in SP can be read as follows: there is a shortest path from x to y with (shortest) distance d and z is the predecessor of y along the path. We also say that SP is *a shortest path relation* for E.

Notice that an edge relation can admit different shortest path relations (depending on which shortest path is stored for a pair of reachable vertices in case of multiple shortest paths for that pair). We assume we are given an arbitrary shortest path relation for an edge relation and are interested in computing an arbitrary shortest path relation for the updated edge relation. More precisely, we consider the shortest path maintenance problem defined below.

Problem (All-pairs shortest paths maintenance) Given an edge relation E, a shortest path relation SP for E, and an edge e, compute a shortest path relation for $E \cup \{e\}$ (or $E \backslash \{e\}$).

The case where we want to compute a shortest path relation for $E \cup \{e\}$ (resp. $E \backslash \{e\}$) corresponds to the scenario where the original edge relation E is modified by adding a new edge (resp. deleting an edge). Indeed, we will also consider the case where the weight of an edge is updated; however, as shown in Sect. 4.3, our algorithms to deal with insertion and deletion can be used to address this case too. We are interested in solving the problem in an efficient *incremental* fashion, i.e., avoiding to compute the new shortest path relation from scratch.

Clearly, a solution to the all-pairs shortest paths maintenance problem immediately gives the all-pairs shortest distances.

In addition to the edge relation for a graph and the corresponding shortest path relation, our algorithms will use auxiliary relations of arity 4 to store tuples of the form (x, y, d, z), which we also call *distance tuples*, whose meaning is that there is a path from x to y with distance d, and z is the predecessor of y along the path.

We will use the relational algebra operators π (projection), $\bowtie$ (join), $\ltimes$ (left semi-join), $\times$ (Cartesian product), $\cup$ (union), and $\setminus$ (difference).

We will refer to the i-th attribute of a relation as $\$i$. For instance, the projection of a relation R on the first and third attribute is written as $\pi_{\$1,\$3} R$. We will use the generalized projection so that we can write expressions like $\pi_{a,\$1} R$, which is equivalent to $\{a\} \times \pi_{\$1} R$.

Given a tuple $t = (t_1, \ldots, t_n)$, the i-th element of t is denoted as $t[i]$. Given two tuples t_1 and t_2, we say that t_1 and t_2 are *similar*, denoted $t_1 \sim t_2$, iff $t_1[1] = t_2[1]$ and $t_1[2] = t_2[2]$. Intuitively, in our setting, two tuples are similar when they refer to (possibly different) paths between the same pair of vertices.

Below we define the operators *min*, *prune*, $\oplus$, $\ominus$, and $\sqcap$, which will be used in the proposed algorithms. Let R and S be relations containing distance tuples (and thus of arity 4).

The *min* operator is defined as follows:

$$min(R) = \{t \in R \mid t[1] \neq t[2] \wedge \nexists t' \in R \text{ s.t. } t' \sim t \wedge t'[3] < t[3]\}.$$

Thus, $min(R)$ returns all the distance tuples t in R with $t[1] \neq t[2]$ (i.e., t refers to a path whose endpoints are distinct vertices) and s.t. R does not contain a similar distance tuple with a strictly lower distance.

[1] In this paper, we consider positive real weights.

The binary operator *prune* is defined as follows:

$$prune(R,S) = \{t \in R \mid \nexists t' \in S \text{ s.t. } t' \sim t \wedge t'[3] \leq t[3]\}.$$

Thus, $prune(R,S)$ returns all the distance tuples t in R for which there is no similar distance tuple in S with a lower distance.

The binary operator $\oplus$ is defined as follows:

$$R \oplus S = R \cup \{t \in S \mid \nexists t' \in R \text{ s.t. } t' \sim t\}.$$

Thus, $R \oplus S$ returns a relation obtained by adding to R the distance tuples of S that are not similar to any of the distance tuples in R.

The binary operator $\ominus$ is defined as follows:

$$R \ominus S = R \backslash \{t \in R \mid \exists t' \in S \text{ s.t. } t' \sim t \wedge t'[3] = t[3]\}.$$

Thus, $R \ominus S$ returns the set obtained from R by deleting every distance tuple for which there exists a similar distance tuple in S with the same distance.

Finally, the binary operator $\sqcap$ is defined as follows:

$$R \sqcap S = \{t \in R \mid \exists t' \in S \text{ s.t. } t' \sim t \wedge t[3] = t'[3]\}.$$

Thus, $R \sqcap S$ returns all the distance tuples of R having a similar distance tuple in S with the same distance.

Notice that none of the above binary operators is symmetric. Also, all the operators above can be expressed in the relational algebra as well as in SQL.

4 Incremental maintenance of all-pairs shortest paths

In this section, we present algorithms for the incremental maintenance of all-pairs shortest paths (as already mentioned, they immediately provide all-pairs shortest distances too).

We first propose an algorithm to handle edge insertions (Sect. 4.1) and then address edge deletions (Sect. 4.2). After that, we show how such algorithms can be used to handle edge updates too (Sect. 4.3). It is worth noting that insertions and deletions of *vertices* can be straightforwardly reduced to our setting and thus be handled by our algorithms too: vertex insertions (resp. deletions) are handled by inserting (deleting) all edges that are incident from/to the inserted (resp. deleted) vertices. *Therefore, our algorithms can handle arbitrary sequences of edge insertions/deletions/updates and vertex insertions/deletions.*

4.1 Edge insertion

Algorithm 1 deals with edge insertions. We point out that the algorithm (as well as Algorithm 2 presented in Sect. 4.2) is written in a form that eases presentation, without applying optimizations to relational algebra expressions. However, relational DBMSs have full-fledged query optimization techniques to easily optimize the code—indeed, this is one of the advantages of relying on a relational DBMS.

The basic idea of our insertion algorithm is as follows. First of all, notice that the insertion of a new edge could improve shortest paths, because adding a new edge yields new paths, which might be shorter than the original ones. Thus, the algorithm looks at the paths going through the inserted edge to see whether they improve on the ones in the original shortest path relation. This is achieved by looking at paths having the inserted edge as the first, the last, or an intermediate edge. If shorter paths are found, they replace the original ones.

More specifically, given a shortest path relation SP for an edge relation E, and an edge $e = (a,b,w)$ to be added to E, Algorithm 1 computes a shortest path relation for $E \cup \{e\}$. The precondition $\nexists (a,b,d,z) \in SP$ with $d \leq w$ is imposed just because if it does not hold, then the insertion of e has no effect on the shortest path relation, and thus there is no need to recompute it.

The algorithm performs the following four steps.

1. (*Forward step*) First, the algorithm looks at paths (in the new graph) having e as the first edge to see whether new paths improving on the current ones can be obtained (lines 1–2).
2. (*Backward step*) Then, the algorithm looks at paths (in the new graph) having e as the last edge to see whether new paths improving on the current ones can be obtained (lines 3–4).
3. (*Combination step*) After that, the algorithm "combines" the paths obtained from the forward and backward steps (line 5)—more details on how the combination is done are provided in the following. The aim is to build paths having e as an intermediate edge, and that might improve on the current ones.
4. (*Final step*) Finally, a path consisting only of the inserted edge is built (line 6), and all the paths built so far that improve on the original ones are incorporated into the shortest path relation (lines 7–9).

Algorithm 1 Edge-Insertion-Maintenance (EIM)

Input: Edge relation E,
Shortest path relation SP,
Edge $e = (a,b,w)$ s.t. $\nexists (a,b,d,z) \in SP$ with $d \leq w$.
Output: Shortest path relation for $E \cup \{e\}$.

1: $\Delta P_r = \pi_{a,\$2,\$3+w,\$4}(\sigma_{\$1=b} SP)$;
2: $\Delta SP_r = prune(min(\Delta P_r), SP)$;
3: $\Delta P_\ell = \pi_{\$1,b,\$3+w,a}(\sigma_{\$2=a} SP)$;
4: $\Delta SP_\ell = prune(min(\Delta P_\ell), SP)$;
5: $\Delta P_i = \pi_{\$1,\$6,\$3+\$7-w,\$8}(\Delta SP_\ell \times \Delta SP_r)$;
6: $e' = (a,b,w,a)$;
7: $\Delta SP = prune(min(\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_r \cup \{e'\}), SP)$;
8: $SP^* = \Delta SP \oplus SP$;
9: **return** SP^*;

We now go into the details of each of the steps above.

(*Forward step*) First, the algorithm computes the set ΔP_r of all the distance tuples obtained by concatenating e with shortest paths starting from vertex b (line 1), that is, tuples of the form (a, y, d, z) s.t. there is a path from a to y with distance d (in the updated graph), the predecessor of y in the path is z, the first edge along the path is e, and the path from b to y is a shortest one (w.r.t. the original edge relation).

Then, among these distance tuples, the algorithm selects only those that improve on current shortest distances, that is, those tuples (a, y, d, z) in ΔP_r s.t. either there is no shortest path from a to y in SP or there is one with distance greater than d (line 2).

(*Backward step*) Next, two analogous steps are performed. First, shortest paths ending in vertex a are concatenated with e (line 3), yielding a set ΔP_ℓ of tuples of the form (x, b, d, a) s.t. there is a path from x to b with distance d (in the updated graph), the predecessor of b in the path is a, the last edge along the path is e, and the path from x to a is a shortest one (w.r.t. the original edge relation). Then, among the tuples in ΔP_ℓ, the algorithm selects only those that improve on the shortest distances in SP (line 4).

(*Combination step*) Then, the distance tuples obtained at lines 2 and 4 (which correspond to path whose first or last edge is e, respectively) are combined via a Cartesian product (line 5). Specifically, this step computes a relation ΔP_i by combining each tuple (x, b, d_1, z_1) in ΔSP_ℓ with each tuple (a, y, d_2, z_2) in ΔSP_r so as to get a tuple $(x, y, d_1 + d_2 - w, z_2)$. Notice that the distance of tuples in ΔP_i is diminished of w because e is taken into account both in tuples of ΔSP_ℓ and in tuples of ΔSP_r.

(*Final step*) Then, a distance tuple e' representing the path that consists only of e is built (line 6). Finally, the algorithm selects those tuples in $\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_r \cup \{e'\}$ that improve on the shortest paths in SP (line 7) and incorporates them into SP (line 8).

The following example illustrates how Algorithm 1 works.

Example 1 Consider the graph in Fig. 1 (top). A shortest path relation SP for the graph is represented in Fig. 1 (bottom) as follows: an edge from x to y labeled with (d, z) means that there is a tuple (x, y, d, z) in SP. Thus, for instance, the edge from b to a with label $(2, c)$ means that $(b, a, 2, c) \in SP$, that is, there is a shortest path from b to a with distance 2 and c is the predecessor of a along that path. Suppose we add the edge $(a, d, 1)$ to the graph.

Figure 2 shows the path computed by the forward step (dotted edge), that is, we have $\Delta SP_r = \{(a, e, 2, d)\}$.

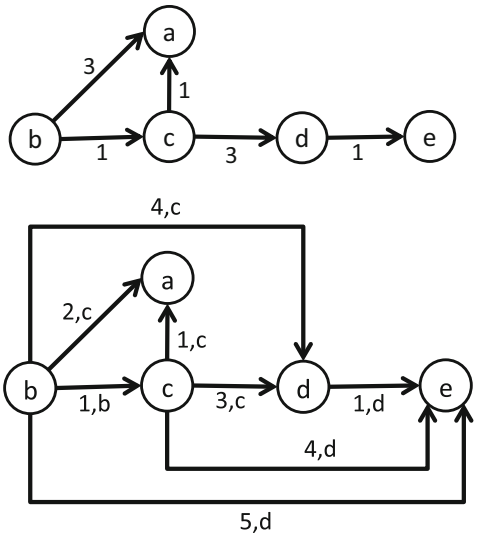


Fig. 1 A graph (*top*) and its shortest paths (*bottom*)

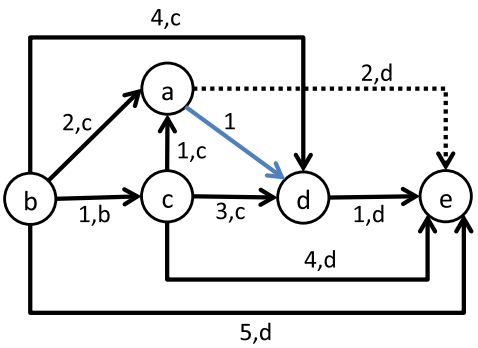


Fig. 2 Forward step of Algorithm 1

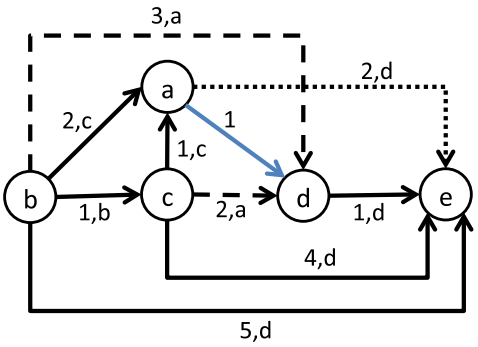


Fig. 3 Backward step of Algorithm 1

The paths computed by the backward step are shown in Fig. 3 (dashed edges), that is, we have $\Delta SP_\ell = \{(c, d, 2, a), (b, d, 3, a)\}$.

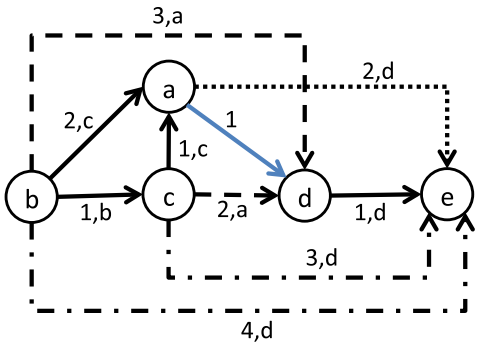


Fig. 4 Combination step of Algorithm 1

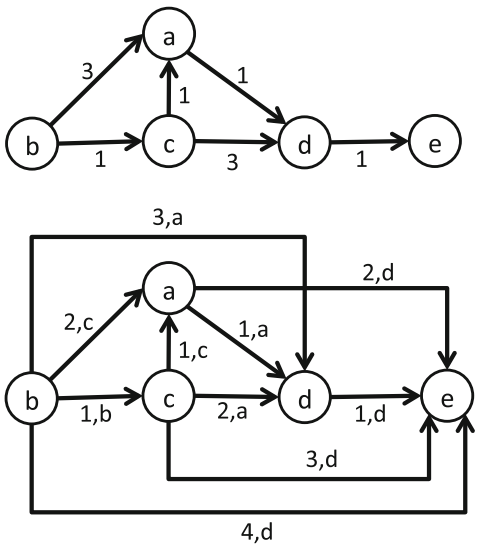


Fig. 5 Updated graph (*top*) and its shortest paths (*bottom*)

Figure 4 shows the paths computed by the combination step (dotted-and-dashed edges), that is, we have $\Delta P_i = \{(c, e, 3, d), (b, e, 4, d)\}$.

As all the distance tuples computed in the previous steps correspond to shortest paths (in the updated graph) improving on the original shortest paths, they are incorporated into the shortest path relation at the final step. Notice that also $(a, d, 1, a)$ is incorporated into the shortest path relation, as d was not reachable from a in the original graph.

Figure 5 shows the updated graph and the new shortest path relation.

The following theorem states the correctness of Algorithm 1, that is, our algorithm exactly computes the all-pairs shortest paths/distances of a new graph obtained by adding a new edge. We recall that approximation algorithms have been proposed in the literature (see Sect. 2); however, in this paper, our goal is to compute *exact* solutions—in an incremental fashion, of course.

Theorem 1 Given an edge relation E, a shortest path relation SP for E, and an edge e, Algorithm 1 computes a shortest path relation for $E \cup \{e\}$.

Proof Let $E_n = E \cup \{e\}$ and SP_n be a shortest path relation for E_n. Before proving the claim (i.e., showing that $SP^* = SP_n$), we make the following two observations, which will be used later on in the proof.

First, if a distance tuple (x, y, d, z) is in SP^* (resp. ΔP_r, ΔSP_r, ΔP_ℓ, ΔSP_ℓ, ΔP_i), then there is a path from x to y in E_n whose distance is d and where z is the predecessor of y along the path. Indeed, this property holds also for ΔP_r, ΔSP_r, ΔP_ℓ, ΔSP_ℓ, ΔP_i, and ΔSP.

Second, SP^* does not contain two distance tuples (x, y, d_1, z_1) and (x, y, d_2, z_2) s.t. $d_1 \neq d_2$ or $z_1 \neq z_2$ (to see why, it suffices to look at lines 7–8 and the definitions of *min*, *prune* and $\oplus$).

In the following, whenever the fourth element of a distance tuple (x, y, d, z) is irrelevant, we simply write the distance tuple as $(x, y, d, -)$.

Soundness ($SP^* \subseteq SP_n$). Let $(x, y, d, z) \in SP^*$. As noticed above, this means that there is a path from x to y in E_n; thus, there must be a shortest one too, which implies that a distance tuple $(x, y, d', -)$ is in SP_n. We show that $d = d'$. One of the following two cases must occur.

1. There is a shortest path p' from x to y in E_n (its distance is d') which does not go through e. Since p' goes only through edges in E, then there is a distance tuple (x, y, d', z') in SP. Since ΔSP contains distance tuples corresponding to paths in E_n that strictly improve on shortest distances in SP (line 7), then there is no distance tuple $(x, y, d'', -)$ in ΔSP. Since $SP^* = \Delta SP \oplus SP$ (line 8), then $(x, y, d', z') \in SP^*$. Hence, $d = d'$ (and $z = z'$).
2. Every shortest path from x to y in E_n (whose distance is d') goes through e. Let p' be one of such shortest paths. Then, e is either *(i)* the first edge of p', or *(ii)* the last edge of p', or *(iii)* an intermediate edge of p'. Notice that in case *(i)* the subpath of p' that goes from b to y is a shortest path in E_n and also in E (because p' does not go though e twice); in case *(ii)* the subpath of p' that goes from x to a is a shortest path in E_n and also in E; in case *(iii)* the subpath of p' that goes from x to a and the subpath of p' that goes from b to y are shortest paths in E_n and also in E. Now it is easy to see that a distance tuple for p' is computed at line 1 (resp. 3 and 5) when case *(i)* (resp. *(ii)* and *(iii)*) occurs. In particular, in case *(iii)*, since every shortest path from x to y in E_n goes through e, it must be the case that every shortest path from x to b in E_n has e as last edge, and every shortest path from a to y in E_n has e as first

edge, and thus a distance tuple for p' is computed at line 5. Notice that since every shortest path from x to y in E_n goes through e, this is the case where the insertion of e strictly improves the shortest distance from x to y. Thus, a distance tuple $(x, y, d', -)$ belongs to ΔSP (line 7) and SP^* (line 8). Hence, $d = d'$.

Completeness ($SP^* \supseteq SP_n$). Consider a distance tuple (x, y, d, z) in SP_n. If there is a shortest path p from x to y in E_n (its distance is d) that does not go through e, then $(x, y, d, z) \in SP$. This means that ΔSP does not contain a distance tuple $(x, y, d', -)$ because distance tuples in $\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_r \cup \{e\}$ correspond to paths in SP_n and thus do not strictly improve on p (see line 7). Hence, $(x, y, d, z) \in SP^*$ (see line 8). If every shortest path from x to y in E_n goes through e, then it can be verified that $(x, y, d, z) \in SP^*$ by applying the same reasoning used in part *(2)* above. □

4.2 Edge deletion

We now turn our attention to edge deletions. The basic idea of our algorithm is as follows. The algorithm consists of two phases: a *deletion phase*, which deletes some shortest paths (roughly speaking, they are shortest paths possibly going through the deleted edge), and a *recalculate phase*, which recomputes a new shortest path (if any) for the pairs of vertices deleted in the first phase.

As for the deletion phase, since the deletion of an edge can make shortest paths worse (i.e., shortest distances might become higher or two vertices might not be reachable anymore), the algorithm looks at the paths going through the deleted edge to see whether alternative paths with the same distance exist. If a shortest path goes through the deleted edge and there is no alternative path with the same distance, it is deleted from the shortest path relation. More specifically, this phase is achieved in three steps that look at paths having the deleted edge as the first, the last, or an intermediate edge along the path, respectively.

After affected shortest paths have been deleted, the recalculate phase tries to recompute new shortest paths for the pairs of vertices deleted in the deletion phase. This is achieved through an iterative process that proceeds one hop at a time trying to reconstruct deleted paths.

Algorithm 2 handles edge deletions. The deletion phase is carried out in lines 5–20, and the recalculate phase is performed in lines 21–28. The precondition on e is imposed because if it does not hold, that is there is no distance tuple of the form (a, b, w, z) in SP, then there is a path from a to b not using e and with a distance strictly lower than w, which means that the deletion of e does not affect the shortest path relation, and thus there is no need to recompute it (recall that we consider positive weights).

Algorithm 2 Edge-Deletion-Maintenance (EDM)

Input: Edge relation E,
Shortest path relation SP,
Edge $e{=}(a, b, w)$ s.t. $\exists z.(a, b, w, z) \in SP$.
Output: Shortest path relation for $E_n = E \setminus \{e\}$.

1: $AP = \pi_{\$1,\$5,\$3+\$6,\$7}(\sigma_{\$1=a} E_n \underset{\$2=\$1}{\bowtie} \sigma_{\$2=b} SP)$;
2: **if** $\exists (a, b, w, z') \in AP$ **then**
3: $\quad SP^* = \{(a, b, w, z')\} \oplus SP$;
4: $\quad$ **return** SP^*;
5: $\Delta P_r = \pi_{a,\$2,\$3+w,\$4}(\sigma_{\$1=b} SP)$;
6: $AP_r = \pi_{\$1,\$2,\$3,\$1}(\sigma_{\$1=a} E_n) \cup (\pi_{\$1,\$5,\$3+\$6,\$7}(\sigma_{\$1=a} E_n \underset{\$2=\$1}{\bowtie} SP))$;
7: $\Delta SP_r = (SP \sqcap \Delta P_r) \ominus AP_r$;
8: $NP_r = (AP_r \sqcap (SP \sqcap \Delta P_r))$;
9: $SP = NP_r \oplus SP$;
10: $\Delta P_\ell = \sigma_{\$2=b \wedge \$4=a} SP$;
11: $AP_\ell = \pi_{\$1,\$2,\$3,\$1}(\sigma_{\$2=b} E_n) \cup (\pi_{\$1,\$6,\$3+\$7,\$5}(SP \underset{\$2=\$1}{\bowtie} \sigma_{\$2=b} E_n))$;
12: $\Delta SP_\ell = \Delta P_\ell \ominus AP_\ell$;
13: $NP_\ell = (AP_\ell \sqcap \Delta P_\ell)$;
14: $SP = NP_\ell \oplus SP$;
15: $NP_i = (\pi_{\$1,\$6,\$3+\$7,\$8}((\sigma_{\$2=a} SP) \underset{\$2=\$1}{\bowtie} NP_r)) \sqcap SP$;
16: $SP = NP_i \oplus SP$;
17: $\Delta P_i = \pi_{\$1,\$6,\$3+\$7-w,\$8}(\Delta SP_\ell \times \Delta SP_r)$;
18: $\Delta SP_i = SP \sqcap \Delta P_i$;
19: $SP^- = \Delta SP_\ell \cup \Delta SP_r \cup \Delta SP_i \cup \{(a, b, w, z)\}$;
20: $SP = SP \setminus SP^-$;
21: $\Delta P = (\pi_{\$1,\$2,\$3,\$1} E_n \cup (\pi_{\$1,\$5,\$3+\$6,\$7}(E_n \underset{\$2=\$1}{\bowtie} SP))) \underset{\substack{\$1=\$1 \wedge \\ \$2=\$2}}{\ltimes} SP^-$;
22: $\Delta SP = min(\Delta P)$;
23: $SP^+ = \Delta SP$;
24: **while** $\Delta SP \neq \emptyset$ **do**
25: $\quad \Delta P = \pi_{\$1,\$5,\$3+\$6,\$7}(E_n \underset{\$2=\$1}{\bowtie} \Delta SP) \underset{\$1=\$1 \wedge \$2=\$2}{\ltimes} SP^-$;
26: $\quad \Delta SP = prune(min(\Delta P), SP^+)$;
27: $\quad SP^+ = \Delta SP \oplus SP^+$;
28: $SP^* = SP \cup SP^+$;
29: **return** SP^*;

First of all, the algorithm checks whether there is an alternative path in the updated graph from a to b with distance w (lines 1–2). If so, the distance tuple in SP for the shortest path from a to b is updated into (a, b, w, z'), where z' is the predecessor of b along the alternative path (line 3). Since there is an alternative path from a to b with the same distance as the weight of the deleted edge, the remaining shortest paths are not affected by the deletion, and the algorithm terminates (line 4).

If no alternative path is found, handling the edge deletion is much more involved, and the algorithm proceeds with the two aforementioned phases as follows.

Deletion phase. Similar to Algorithm 1, the deletion phase consists of different steps:

1. a *forward step*, looking at paths having e as the first edge (lines 5–9);
2. a *backward step*, looking at paths having e as the last edge (lines 10–14);

3. a *combination step*, combining paths of the previous two steps, whose aim is to look at paths having e as an intermediate edge (lines 15–18); and
4. a *deletion step*, (lines 19–20), where paths computed at the previous steps are deleted from the shortest path relation.

We now go into the details of each of the steps above.

(*Forward step*) First, the algorithm computes all the distance tuples obtained by concatenating e with shortest paths starting from vertex b (line 5), that is, tuples of the form (a, y, d, z) s.t. there is a path from a to y with distance d, the first edge along such a path is e, the path from b to y is a shortest one (w.r.t. the original edge relation), and z is the predecessor of y. Such distance tuples are stored in ΔP_r—intuitively, they correspond to paths from a, using e as the first edge, and that might be affected by the deletion of e.

Then, alternative paths in the new graph from a to other vertices are computed and stored in AP_r (line 6).

After that, the distance tuples in ΔP_r that correspond to shortest paths (w.r.t. the original graph) and for which there are no alternative paths (in AP_r) with the same distance are stored in ΔSP_r (line 7). These distance tuples will be later deleted from SP.

On the other hand, the distance tuples in ΔP_r corresponding to shortest paths (w.r.t. the original graph) and for which there is an alternative path with the same distance are incorporated into SP (lines 8–9). This is done to properly update the predecessor of the destination vertex (i.e., the last element of the distance tuples).

(*Backward step*) Then, shortest paths having e as the last edge are similarly processed. Specifically, the algorithm selects all the distance tuples corresponding to shortest paths (w.r.t. the original edge relation) having b as destination vertex and e as the last edge (line 10). Such distance tuples are stored in ΔP_ℓ and correspond to paths to b, using e as the last edge, and that might be affected by the deletion of e.

After that, alternative paths to b (from other vertices) in the new graph are computed and stored in AP_ℓ (line 11).

Then, the distance tuples in ΔP_ℓ (recall that they correspond to shortest paths in the original graph) for which there is no alternative path (in AP_ℓ) with the same distance are stored in ΔSP_ℓ (line 12)—they will be later deleted from SP.

On the other hand, the distance tuples in ΔP_ℓ for which there is an alternative path with the same distance are incorporated into SP (lines 13–14). This is done to properly update the predecessor of the destination vertex (i.e., the last element of the distance tuples).

(*Combination step*) Then, the algorithm computes distance tuples obtained as the concatenation of distance tuples in SP ending in a with distance tuples in NP_r (the latter correspond to shortest paths from a). Among them, only those that correspond to shortest paths are kept in NP_i (line 15) and incorporated into SP (line 16). This step is done so that shortest paths in the original graph that possibly had e as an intermediate edge are replaced by alternative paths with the same distance in the updated graph.

After that, the distance tuples in ΔSP_ℓ computed in line 12 (corresponding to paths where the last edge is e) are combined with the tuples in ΔSP_r computed in line 7 (corresponding to paths where the first edge is e) via a Cartesian product (line 17). Analogous to line 5 of Algorithm 1, since edge e is taken into account twice, the distance of the tuples obtained from the Cartesian product is diminished of w. Among the so-obtained distance tuples, only those that correspond to shortest paths are kept (line 18).

(*Deletion step*) The distance tuples computed in lines 7, 12 and 18, together with the distance tuple (a, b, w, z), are stored in relation SP^- (line 19) and deleted from SP (line 20).

Recalculate phase The second phase of Algorithm 2 (lines 21–28) computes the new shortest paths (when they exist) for the endpoints of the distance tuples in SP^-, and eventually adds them to SP.

More specifically, the algorithm first adds to E_n all the distance tuples obtained by concatenating edges in E_n with shortest paths in the updated shortest path relation SP (line 21). Indeed, only those distance tuples whose endpoints are in a tuple of SP^- are kept and stored in ΔP. Among these distance tuples, the minimum ones are selected and stored in ΔSP (line 22). These distance tuples are then added to SP^+ (line 23), which is the set that will be eventually added to SP.

Then, SP^+ is iteratively updated as follows (lines 24–27). The algorithm computes all the distance tuples obtained by concatenating edges in E_n with tuples in ΔSP and keeps only those whose endpoints are in a tuple of SP^- (line 25). Among these, only distance tuples that improve on shortest distances in SP^+ are kept (line 26) and incorporated into SP^+ (line 27). When no more better distance tuples can be obtained, SP^+ is added to SP (line 28) and the result is returned (line 29).

It is important to notice that this way of computing the new shortest paths allows the algorithm to prune the search space, as computed paths that are not shortest ones are disregarded and no further explored.

An example illustrating how Algorithm 2 works is provided below.

Example 2 Consider the graph of Fig. 6 (top) and the shortest path relation SP in Fig. 6 (bottom). Once again, the shortest path relation is represented as follows: an edge from x to y labeled with (d, z) means that there is a tuple

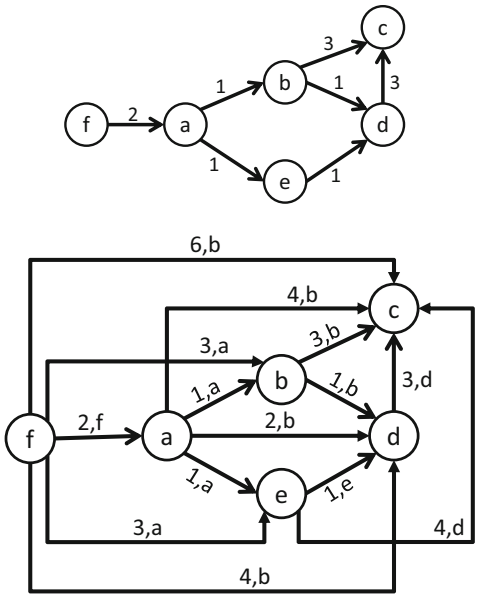


Fig. 6 A graph (*top*) and its shortest paths (*bottom*)

(x, y, d, z) in SP, that is, there is a shortest path from x to y with distance d and z is the predecessor of y.

Suppose we delete the edge $(a, b, 1)$ from the graph.

Since there is no alternative path in the updated graph from a to b, then Algorithm 2 performs the deletion phase.

The forward step identifies the two dotted paths in Fig. 7 as possibly affected by the edge deletion.

Indeed, for the path from a to d there is an alternative one with the same distance, namely the one going through e. Thus, the predecessor of d is updated into e (i.e., $(a, d, 2, b)$ is updated into $(a, d, 2, e)$), see Fig. 8. Since there is no alternative path from a to c with distance 4, the path from a to c will be later deleted by the algorithm (i.e., $(a, c, 4, b) \in \Delta SP_r$).

Then, the backward step identifies the dashed path in Fig. 9 as possibly affected by the edge deletion. Since there is no alternative path from f to b, such a path will be later deleted by the algorithm (i.e., $(f, b, 3, a) \in \Delta SP_\ell$).

After that, the combination step realizes that, for the path from f to d, the predecessor of d has to be updated into e (Fig. 10).

Moreover, the combination step identifies the dashed-and-dotted path in Fig. 11 as possibly affected by the deletion. Indeed, since there is no alternative path with the same distance, this path will be later deleted (i.e., $(f, c, 6, b) \in \Delta SP_i$).

In the deletion step of the deletion phase, all the non-solid paths in Fig. 11 are deleted from the shortest path relation.

Then, the algorithm tries to recompute the deleted shortest paths. Figure 12 shows the new shortest paths

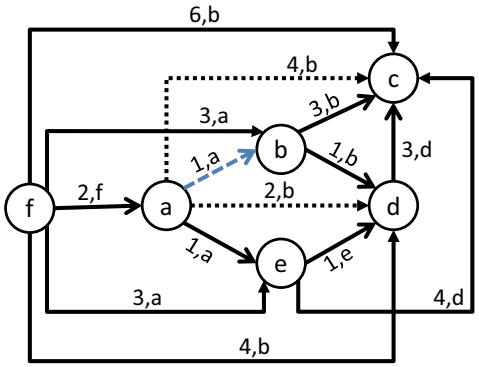


Fig. 7 Forward step of Algorithm 2 (1 of 2)

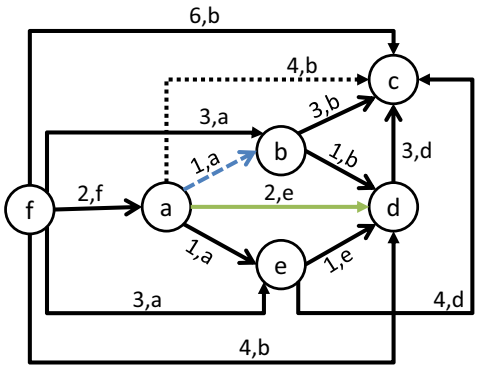


Fig. 8 Forward step of Algorithm 2 (2 of 2)

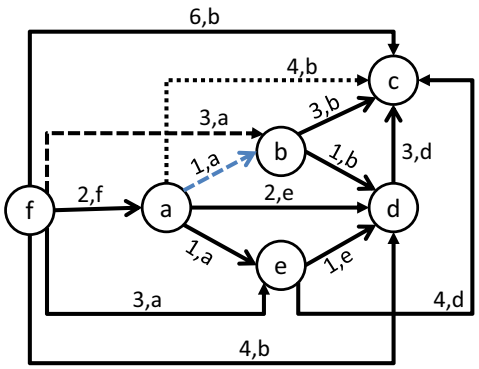


Fig. 9 Backward step of Algorithm 2

computed by the recalculate phase. At this point, Algorithm 2 terminates.

Figure 13 shows the shortest path relation for the updated graph.

The following theorem states the correctness of Algorithm 2, that is, it is an algorithm to *exaclty* compute the all-pairs shortest paths/distances of a graph after deleting an edge.

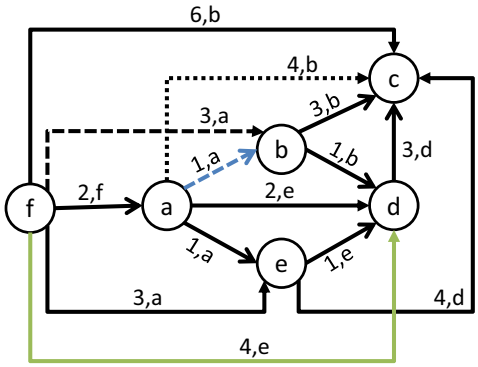


Fig. 10 Combination step of Algorithm 2 (1 of 2)

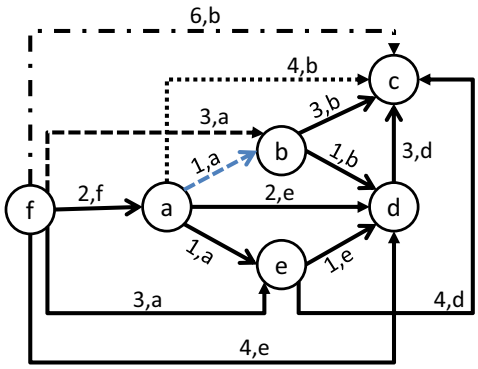


Fig. 11 Combination step of Algorithm 2 (2 of 2)

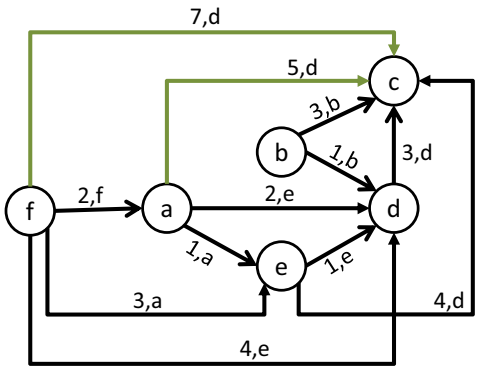


Fig. 12 Recalculate phase of Algorithm 2

Theorem 2 Given an edge relation E, a shortest path relation SP for E, and an edge e, Algorithm 2 computes a shortest path relation for $E\backslash\{e\}$.

Proof Let $E_n = E - \{e\}$ and SP_n be a shortest path relation for E_n.

First of all, notice that each iteration of the **while** loop in lines 24–27 computes shortest paths that strictly improve on the currently computed ones. As we consider positive edge weights, the number of iterations is bounded by the number of simple paths (i.e., paths without multiple occurrences of the same vertex), which is finite, and thus the algorithm terminates.

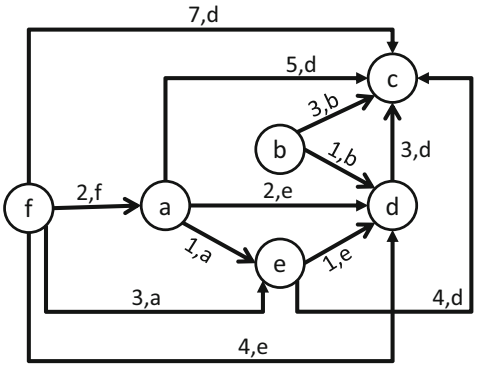


Fig. 13 Shortest paths of the updated graph

In the following, whenever the fourth element of a distance tuple (x, y, d, z) is irrelevant, we simply write it as $(x, y, d, -)$. Likewise, we sometimes write $(x, y, -, -)$ in place of (x, y, d, z).

If a distance tuple (x, y, d, z) is in SP and there exists a distance tuple $(x, y, d, -)$ in SP_n, then there is a path from x to y with distance d in the updated graph. One of the following two cases occurs:

- None of the shortest paths from x to y in the original graph goes through e. Then, it is easy to see that (x, y, d, z) remains in SP and thus is in SP^*.
- At least one shortest path from x to y goes though e in the original graph. Since there is an alternative path with the same distance in the new graph, then (x, y, d, z) is updated into a distance tuple (x, y, d, z') either at line 9 (when an original shortest path from x to y has e as the first edge) or line 14 (when an original shortest path from x to y has e as the last edge) or line 16 (when an original shortest path from x to y has e as an intermediate edge).

Regarding distance tuples (x, y, d, z) is in SP s.t. there does not exist a distance tuple $(x, y, d, -)$ in SP_n, we now show the following property: if $(x, y, d, z) \in SP$ and there is no $(x, y, d, -)$ in SP_n, then $(x, y, d, z) \in SP^-$. Suppose $(x, y, d, z) \in SP$ and there is no $(x, y, d, -)$ in SP_n. Then, one of the following cases occurs.

(i) There is a shortest path from x to y in E whose first edge is e and there is no shortest path from x to y in E_n with distance d. In such a case $(x, y, d, z) \in \Delta SP_r$ (see lines 5–7) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(ii) There is a shortest path from x to y in E whose last edge is e and there is no shortest path from x to y in E_n with distance d. In such a case $(x, y, d, z) \in \Delta SP_\ell$ (see lines 10–12) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(iii) There is a shortest path from x to y in E where e is an intermediate edge and for the two subpaths starting from and ending in e conditions (i) and ii apply, respectively. In such a case $(x, y, d, z) \in \Delta SP_i$ (see lines 17–18) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(iv) $e = (x, y, d, z)$. Then, $(x, y, d, z) \in SP^-$ (see line 19).

Because of the property above, if the shortest distance from x to y changes in the updated graph, then $(x, y, -, -)$ is deleted from SP. Then, it can be easily verified that a shortest path from x to y (if any) in the updated graph is correctly computed by the recalculate phase on lines 21–28. □

4.3 Edge update

In this section, we show how our algorithms can be easily used to handle an edge update (i.e., the edge weight update).

Given an edge relation E, a shortest path relation SP for E, and an edge (a, b, w) whose weight w must be changed into w', we can compute a shortest path relation for $(E \backslash \{(a, b, w)\}) \cup \{(a, b, w')\}$ as follows:

1. if $w' < w$, then we call Algorithm EIM with input $E \backslash \{(a, b, w)\}$, SP, and (a, b, w');
2. if $w' > w$, then we call Algorithm EDM with input $(E \backslash \{(a, b, w)\}) \cup \{(a, b, w')\}$, SP, and (a, b, w).

Thus, when the new weight w' "improves" on the old one w, we reduce the problem to an edge insertion, otherwise we reduce the problem to an edge deletion. As stated in the following theorem, this way of processing edge weight updates is correct—we point out that the following theorem, like Theorems 1 and 2, guarantees the *exact* computation of the new all-pairs shortest paths/distances.

Theorem 3 Given an edge relation E, a shortest path relation SP for E, an edge $e = (a, b, w)$ in E, and a weight w', then

1. if $w' < w$, EIM$(E \backslash \{(a, b, w)\}, SP, (a, b, w'))$ returns a shortest path relation for $(E \backslash \{(a, b, w)\}) \cup \{(a, b, w')\}$;
2. if $w' > w$, EDM$((E \backslash \{(a, b, w)\}) \cup \{(a, b, w')\}, SP, (a, b, w))$ returns a shortest path relation for $(E \backslash \{(a, b, w)\}) \cup \{(a, b, w')\}$.

Proof Item 1 can be proved by proceeding in the same way as in the proof of Theorem 1. Likewise, Item 2 can be proved by proceeding in the same way as in the proof of Theorem 2. □

5 Experimental evaluation

In this section, we report on an experimental evaluation we carried out to assess the validity of our approach.

All experiments were run on an Intel i7 3770K 3.5 GHz (single CPU), 12 GB of memory, SSD storage (no network-based storage), running Linux Mint 17.1 and MySQL 5.5.43 (default settings).

5.1 Experimental setup

Below we describe the algorithms and datasets used in the experimental evaluation.

Algorithms We compared our algorithms against the ones proposed by Pang et al. (2005), which are, to the best of our knowledge, the only disk-based approach for the incremental maintenance of all-pairs shortest distances—in particular, they rely on relational DBMSs too. Indeed, we also compared the algorithms presented in this paper with our previous proposal Greco et al. (2016b), and the running times of the two approaches were nearly the same. Thus, in the following, we do not report the execution time of the algorithms proposed in Greco et al. (2016b). Also, recall that while the algorithms presented in this paper maintain both shortest paths and distances, the approach of Greco et al. (2016b) is able to maintain shortest distances only. To sum up, in this section we will consider the EIM algorithm (Algorithm 1), the EDM algorithm (Algorithm 2), and the insertion and deletion algorithms of Pang et al. (2005) (denoted PDR).

Datasets Experiments were carried out on the following real-world networks.

- *DIMES public data repository*[2] This dataset provides monthly snapshots of Autonomous Systems on the Internet. Vertices represent Autonomous Systems and edges represent direct links between Autonomous Systems that were found for a given month. Since the original graphs were unweighted, we assigned unitary weight to every edge.
- *Road Network of North America (RNNA)*[3] This dataset is a road network of North America; thus edge weights stand for road lengths.
- *Twitter social network*[4] This network contains information about who follows whom on Twitter. Vertices represents users and edges represent follow relationships. We assigned unitary weight to every edge, as the original graph was unweighted.

[2] http://www.netdimes.org/new/.

[3] http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

[4] http://konect.uni-koblenz.de/networks/munmun_twitter_social.

Table 1 Properties of the considered datasets

Datasets	# Vertices	# Edges	# Shortest distances
DIMES	17,148	53,833	$\approx 1.02 \times 10^8$
RNNA	175,813	179,179	$\approx 2.76 \times 10^8$
Twitter	465,017	834,797	$\approx 8.03 \times 10^8$
Gnutella	62,586	147,892	$\approx 8.84 \times 10^8$
Instagram	54,017	963,881	$\approx 9.3 \times 10^8$

- *Gnutella peer-to-peer network*[5] This dataset stores the Gnutella peer-to-peer file sharing network. Vertices represent hosts in the Gnutella network, and edges represent connections between the Gnutella hosts. As the original graph was unweighted, we assigned unitary weight to every edge.
- *Instagram social network*[6] This dataset was originally collected in 2014 and introduced in Tagarelli and Interdonato (2015). Vertices represent users, and edges represent follow relationships. We assigned unitary weight to every edge (connecting a pair follower-followee).

The datasets' features are reported in Table 1. The number of shortest distances is the number of pairs of reachable vertices (which is also the number of stored shortest paths, as we store one shortest path for each pair of reachable vertices).[7]

A couple of remarks on the size of the datasets are in order.

(1) As we deal with the problem of incrementally maintaining all-pairs shortest paths, the input of the algorithms consists of both a graph *and the corresponding shortest paths*,[8] with the size of the latter being much bigger than the size of the former. The datasets we considered have up to hundreds of millions of shortest distances—indeed, the biggest dataset (namely, the Instagram network) has nearly one billion shortest distances.

(2) We also run the algorithms proposed in Demetrescu et al. (2004) and Khopkar et al. (2014), which are state-of-the-art algorithms for the APSD maintenance working in the main memory, over the aforementioned datasets. We used the implementations of the algorithms provided by their authors, which work in the main memory. Demetrescu et al. (2004) run out of memory on all datasets, not being able to handle the graph, the shortest distances, and the employed data structures. Khopkar et al. (2014) showed higher running times than our approach on the DIMES dataset, while run out of memory on the remaining datasets. Indeed, it was already as pointed out in Demetrescu et al. (2004) and Demetrescu and Italiano (2006) that memory usage is an important issue of current approaches, limiting substantially the graph size that they can handle (for instance, the experimental studies in Demetrescu et al. (2004) and Demetrescu and Italiano (2006) could not go beyond graphs of 10,000 vertices). Moreover, as shown in the following, the datasets are already too large for PDR, but can be efficiently maintained by our algorithms.

5.2 Results on the DIMES dataset

In this section, we discuss the experimental results for the DIMES dataset. As the DIMES dataset is an evolving graph and its repository provides different snapshots of it, we could run experiments with *real* edge insertions/deletions—we considered the first two snapshots (namely, January and February 2007).

Runtimes for the insertion of 90 edges are reported in Fig. 14a. PDR shows an unstable behavior, as its running times are around 200 secs (i.e., three orders of magnitude slower than EIM) for about half of the edges and are similar to EIM for the remaining edges.

Results for the deletion of 90 edges are reported in Fig. 14b. PDR is not reported as it did not terminate within one hour in all cases, that is, it was not able to handle the deletion of a single edge within the time limit. On the other hand, we can see that EDM performs very well.

In order to allow PDR to answer in a reasonable amount of time and compare its behavior with EDM, we considered smaller induced subgraphs of the original DIMES graph. An *induced subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ contains all edges of E connecting two vertices in V'. To generate these graphs, we started from a randomly chosen vertex in the DIMES dataset and visited connected vertices in a breadth-first fashion, adding to the graph in construction all edges of the original graph involving vertices so far encountered, until the desired size was reached.

Runtimes for both PDR and EDM are reported in Fig. 14c (each running time is the average time over 10 edges), which shows that PDR becomes impractical very quickly.

[5] http://snap.stanford.edu/data/p2p-Gnutella31.html.

[6] http://uweb.dimes.unical.it/tagarelli/data/.

[7] The time taken to generate all-pairs shortest paths for the original datasets ranges from around 222 seconds for the smallest dataset, namely DIMES, to around 4800 seconds for the biggest dataset, namely Instagram. It is worth noticing that computing all-pairs shortest paths is a task performed only once at the beginning, provided that *incremental* algorithms are subsequently used to maintain shortest paths and distances.

[8] Clearly, the input of PDR consists of a graph and the corresponding shortest distances, rather than the actual paths.

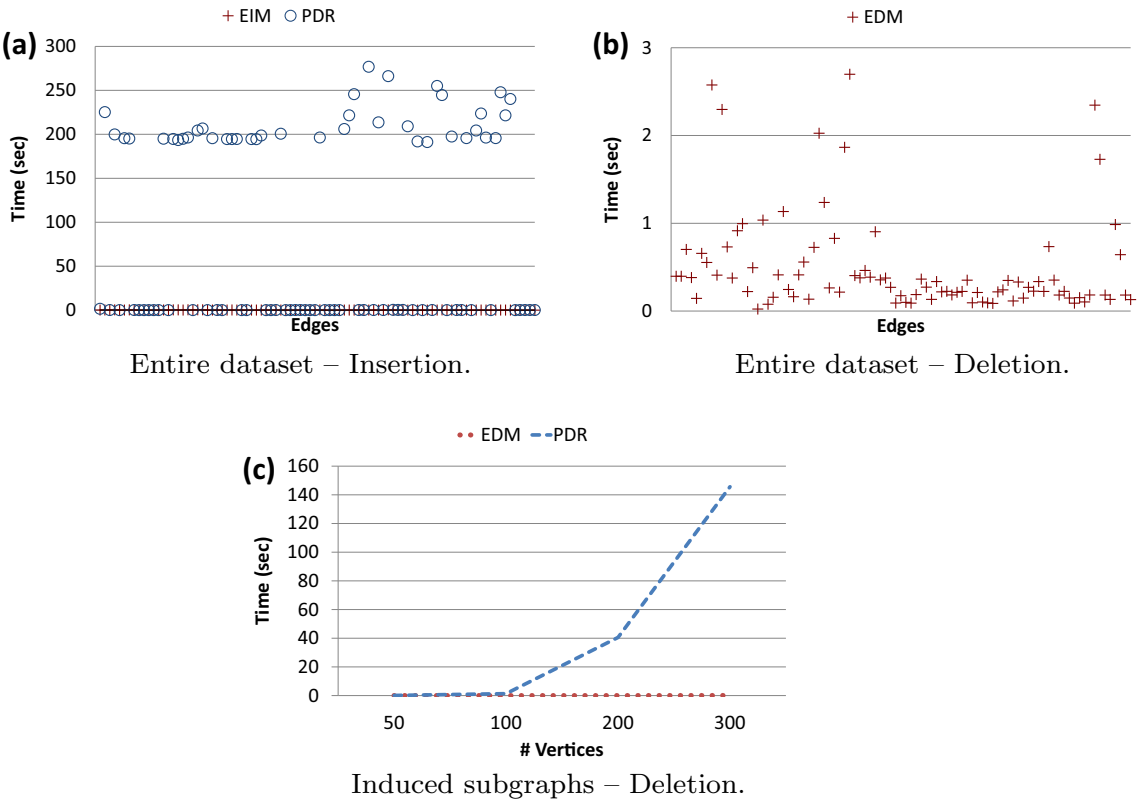


Fig. 14 DIMES dataset

5.3 Results on the RNNA dataset

We now discuss the experimental results for the Road Network of North America.

Figure 15a shows the execution times for the insertion of 150 edges (randomly chosen). Algorithm EIM outperforms PDR being always faster with a difference from one to three orders of magnitude—in most of the cases the difference is at least two orders of magnitude.

Figure 15b shows the execution times for the deletion of 150 edges (randomly chosen). Like for the DIMES dataset, execution times for PDR are not reported as, for every single edge, no answer was given within one hour. The EDM algorithm performs quite well being able to handle deletion in reasonable time.

Once again, to compare PDR with EDM, we also considered smaller induced subgraphs of the original RNNA network, generated in the same way as for the DIMES dataset. However, in this case, the starting vertex was one with highest degree (this choice was not possible for the DIMES dataset because the resulting subgraphs were too large to be handled by PDR). As shown in Fig. 15c, PDR's running times quickly diverge from those of EDM.

5.4 Results on the twitter dataset

In this section, we consider the Twitter network. Figure 16a, b shows running times for the insertion and deletion of 200 randomly chosen edges, respectively.

For insertion, EIM is faster than PDR by one order of magnitude for over 90% of the edges, while for remaining ones they have similar running times.

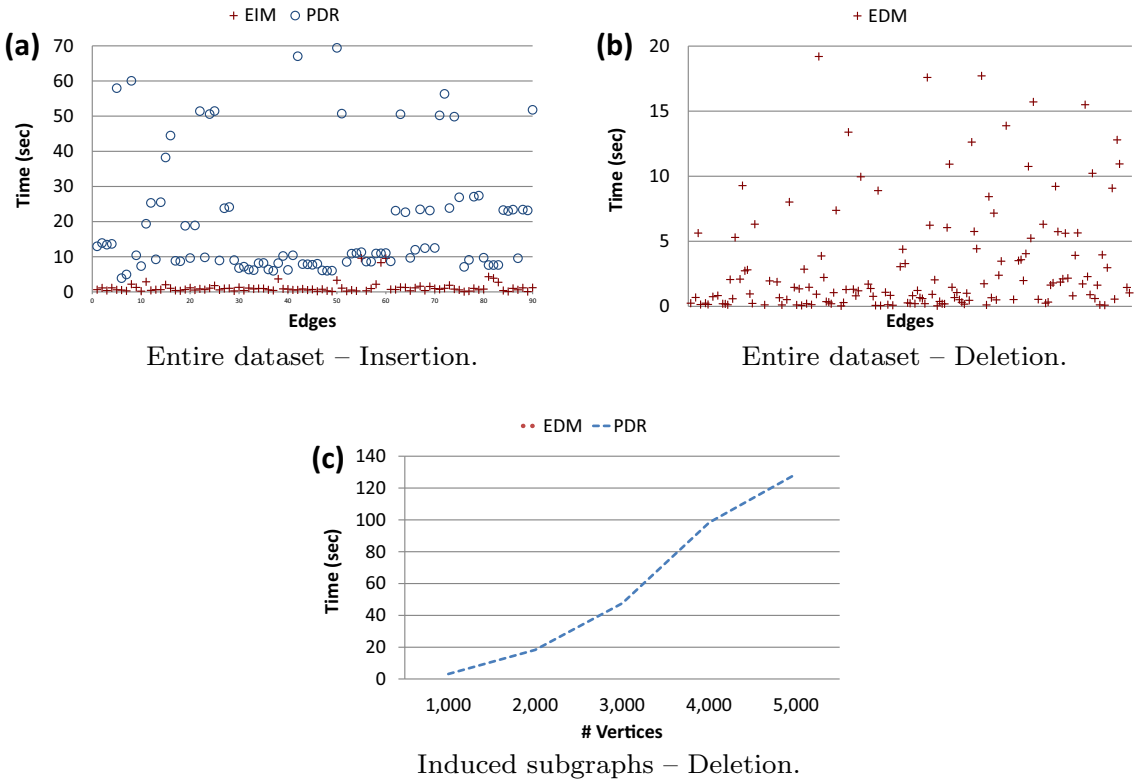
Regarding deletion, EDM was very fast, while PDR did not provided an answer within one hour for each single edge.

5.5 Results on the gnutella dataset

The running times for the insertion and the deletion of 200 randomly chosen edges over the Gnutella network are reported in Fig. 17a, b), respectively.

Similar to the DIMES dataset, PDR shows an unstable behavior, as its running time is over 2500 secs (i.e., PDR is four orders of magnitude slower than EIM) for over one-third of the edges, and is similar to EIM's running time for the remaining edges.

As for deletion, PDR did not terminate within one hour for every edge we considered, while EDM has good performances.

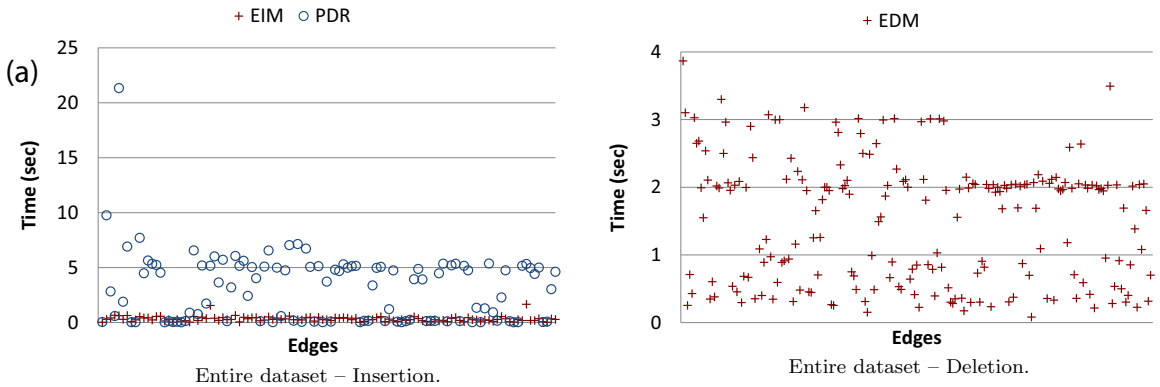


Entire dataset – Insertion.

Entire dataset – Deletion.

Induced subgraphs – Deletion.

Fig. 15 RNNA dataset

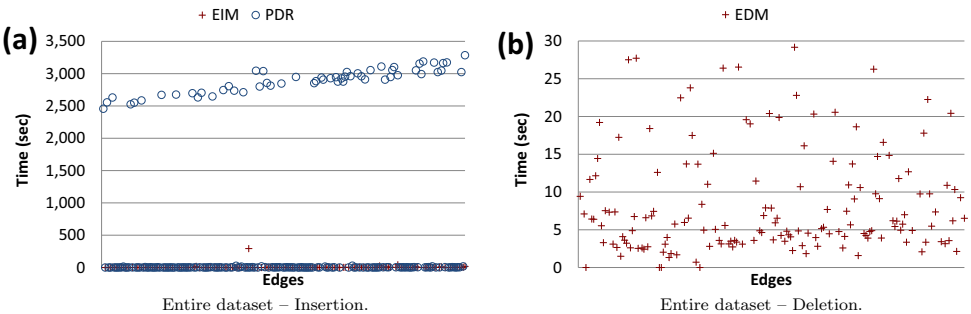


Entire dataset – Insertion.

Entire dataset – Deletion.

Fig. 16 Twitter dataset

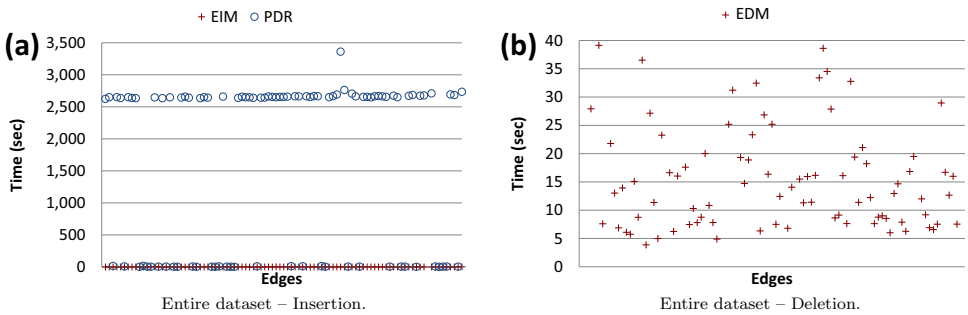
Fig. 17 Gnutella dataset



Entire dataset – Insertion.

Entire dataset – Deletion.

Fig. 18 Instagram dataset



5.6 Results on the Instagram dataset

We now discuss the experimental results over the biggest network we considered, namely Instagram, which has nearly one billion shortest distances.

Figure 18a, b shows running times for the insertion and deletion of 100 randomly chosen edges, respectively.

As for insertion, the difference between PDR and EIM is significant, with EIM being four orders of magnitude faster in most of the cases.

Regarding deletion, like for the other datasets, PDR was not able to answer within one hour, while EDM has good running times.

5.7 Discussion

From the experimental results reported in this section, we can draw the following conclusions.

We run state-of-the-art algorithms working in the main memory (namely, Khopkar et al. (2014) and Demetrescu et al. (2004)) over five real-world datasets, and they run out of memory in all cases. This suggests the need of resorting to disk-based approaches for APSD and APSP maintenance.

To the best of our knowledge, Pang et al. (2005) is the state-of-the-art approach for the all-pairs shortest distances maintenance problem working on the secondary memory (in particular, graphs are stored in relational DBMSs as in our approach). In all cases, our algorithms notably outperform the algorithms proposed in Pang et al. (2005), being able to handle both insertions and deletions with very good performances over networks with hundreds of millions of shortest distances—indeed, Pang et al. (2005) was not able to handle the deletion of a single edge within one hour for every dataset.

Our experimental results showed that handling edge deletions is in general more expensive than handling edge insertions. While our algorithms share the same basic idea, that is, they first identify the affected shortest paths and then update them, this is more expensive for deletions—in particular, the update phase is more expensive for deletions. In fact, the insertion algorithm identifies the affected shortest paths and updates them using single operations. In contrast, the deletion algorithm requires multiple operations to identify and then delete the affected shortest paths, and after that, it employs an iterative procedure to recompute new shortest paths, which is more time-consuming.

6 Summary and outlook

Computing shortest paths and distances is a fundamental problem in social network analysis and many other domains.

In several current applications, graphs are subject to frequent updates, and thus it becomes impractical to recompute solutions from scratch every time the graph is changed. To deal with such scenarios, we have proposed efficient algorithms for the incremental maintenance of all-pairs shortest paths and distances.

Our experimental evaluation showed that current main-memory algorithms cannot scale to large graphs, such as common social and road networks, as they run out of memory with all networks we considered—the smallest having tens of thousands of vertices and edges. Our algorithms rely on relational databases and significantly outperform the state-of-the-art algorithms designed for in-memory execution and based on relational DBMSs (Pang et al. 2005). Still, algorithms working in the main memory can be a valid option for handling smaller networks. In this regard, Greco et al. (2016a) reports on an experimental evaluation comparing algorithms working in the main memory with ones based on graph database systems and relying on relational DBMSs, showing that for larger graphs disk-based approaches are needed.

There are different directions for future work.

One interesting issue is to design algorithms that can handle batches of operations at once and leverage the information about a whole batch to improve performance. Analyzing the combined effect of multiple edges and identifying the overall portion of the graph affected by their insertion/deletion can be exploited to reduce the computational effort w.r.t. processing edges individually.

Another interesting issue to be investigated is the development of parallel algorithms. In this regard, a challenging problem is to devise algorithms that analyze a set of insertions/deletions and are able to properly distribute the computation on the basis of the affected portion of the graph minimizing the overlapping computation carried out by different processes.

Finally, the ideas developed in this paper for maintaining shortest distances and paths might be carried over other domains to develop incremental algorithms for other graph-related problems. Some efforts in this direction are Greco et al. (2017a, b), where incremental algorithms for maintaining the maximum-flow in dynamic networks have been proposed.

References

Akiba T, Iwata Y, Yoshida Y (2013) Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceeding of international conference on management of data (SIGMOD), pp 349–360

Baswana S, Hariharan R, Sen S (2003) Maintaining all-pairs approximate shortest paths under deletion of edges. In: Proceedings of ACM-SIAM symposium on discrete algorithms (SODA), pp 394–403

Bernstein A (2013) Maintaining shortest paths under deletions in weighted directed graphs: (extended abstract). In: Proceedings of ACM on symposium on theory of computing (STOC), pp 725–734

Bouros P, Skiadopoulos S, Dalamagas T, Sacharidis D, Sellis T.K (2009) Evaluating reachability queries over path collections. In: Proceedings of international conference on scientific and statistical database management (SSDBM), pp 398–416

Chang L, Yu JX, Qin L, Cheng H, Qiao M (2012) The exact distance to destination in undirected world. VLDB J 21(6):869–888

Cheng J, Ke Y, Chu S, Cheng C (2012) Efficient processing of distance queries in large graphs: a vertex cover approach. In: Proceedings of international conference on management of data (SIGMOD), pp 457–468

Crecelius T, Schenkel R (2012) Pay-as-you-go maintenance of precomputed nearest neighbors in large graphs. In: Proceedings of ACM conference on information and knowledge management (CIKM), pp 952–961

Cuzzocrea A, Papadimitriou A, Katsaros D, Manolopoulos Y (2012) Edge betweenness centrality: a novel algorithm for qos-based topology control over wireless sensor networks. J Netw Comput Appl 35(4):1210–1217

Demetrescu C, Emiliozzi S, Italiano G.F (2004) Experimental analysis of dynamic all pairs shortest path algorithms. In: Proceedings of ACM-SIAM symposium on discrete algorithms (SODA), pp 369–378

Demetrescu C, Italiano GF (2004) A new approach to dynamic all pairs shortest paths. J ACM 51(6):968–992

Demetrescu C, Italiano GF (2006) Experimental analysis of dynamic all pairs shortest path algorithms. ACM Trans Algorithms 2(4):578–601

Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische Mathematik 1(1):269–271

Fan W, Li J, Ma S, Tang N, Wu Y, Wu Y (2010) Graph pattern matching: from intractable to polynomial time. Proc VLDB Endow (PVLDB) 3(1):264–275

Fu AWC, Wu H, Cheng J, Wong RCW (2013) Is-label: an independent-set based labeling scheme for point-to-point distance querying. Proc VLDB Endow (PVLDB) 6(6):457–468

Gao J, Zhou J, Yu JX, Wang T (2014) Shortest path computing in relational dbmss. IEEE Trans Knowl Data Eng 26(4):997–1011

Goldberg, A.V., Werneck, R.F.F. (2005) Computing point-to-point shortest paths from external memory. In: Proceedings of workshop on algorithm engineering and experiments and workshop on analytic algorithmics and combinatorics (ALENEX/ANALCO), pp 26–40

Greco S, Molinaro C, Pulice C (2016) Efficient maintenance of all-pairs shortest distances. In: Proceedings of international conference on scientific and statistical database management (SSDBM), pp 9:1–9:12

Greco S, Molinaro C, Pulice C, Quintana X (2016) All-pairs shortest distances maintenance in relational dbmss. In: Proceedings of IEEE/ACM international conference on advances in social networks analysis and mining (ASONAM), pp 215–222

Greco S, Molinaro C, Pulice C, Quintana X (2017) Efficient maximum flow maintenance on dynamic networks. In: Proceedings of international conference on world wide web companion (WWW), pp 1383–1385

Greco S, Molinaro C, Pulice C, Quintana X (2017) Incremental maximum flow computation on evolving networks. In: Proceedings of the symposium on applied computing (SAC), pp 1061–1067

Gupta A, Mumick I.S, Subrahmanian V.S (1993) Maintaining views incrementally. In: Proceedings of international conference on management of data (SIGMOD), pp 157–166

Henzinger M, Krinninger S, Nanongkai D (2013) Dynamic approximate all-pairs shortest paths: Breaking the O(mn) barrier and derandomization. In: Proceedings of IEEE symposium on foundations of computer science (FOCS), pp 538–547

Italiano GF (1988) Finding paths and deleting edges in directed acyclic graphs. Inf Process Lett 28(1):5–11

Jin R, Ruan N, Xiang Y, Lee V.E. (2012) A highway-centric labeling approach for answering distance queries on large sparse graphs. In: Proceedings of international conference on management of data (SIGMOD), pp 445–456

Kang C, Kraus S, Molinaro C, Spezzano F, Subrahmanian VS (2016) Diffusion centrality: a paradigm to maximize spread in social networks. Artif Intell 239:70–96

Kang C, Molinaro C, Kraus S, Shavitt Y, Subrahmanian V.S (2012) Diffusion centrality in social networks. In: Proceedings of international conference on advances in social networks analysis and mining (ASONAM), pp 558–564

Khopkar SS, Nagi R, Nikolaev AG, Bhembre V (2014) Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis. Soc Netw Anal Min 4(1):220

King, V (1999) Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: Proceedings of IEEE symposium on foundations of computer science (FOCS), pp 81–91

Pang C, Dong G, Ramamohanarao K (2005) Incremental maintenance of shortest distance and transitive closure in first-order logic and SQL. ACM Trans Database Syst 30(3):698–721

Planken L, de Weerdt M, van der Krogt R (2012) Computing all-pairs shortest paths by leveraging low treewidth. J Artif Intell Res 43:353–388

Przulj N, Wigle DA, Jurisica I (2004) Functional topology in a network of protein interactions. Bioinformatics 20(3):340–348

Qi Z, Xiao Y, Shao B, Wang H (2013) Toward a distance oracle for billion-node graphs. Proc VLDB Endow (PVLDB) 7(1):61–72

Qiao M, Cheng H, Chang L, Yu J.X (2012) Approximate shortest distance computing: A query-dependent local landmark scheme.

In: Proceedings of IEEE international conference on data engineering (ICDE), pp 462–473

Rahman SA, Advani P, Schunk R, Schrader R, Schomburg D (2005) Metabolic pathway analysis web service (pathway hunter tool at cubic). Bioinformatics 21(7):1189–1193

Roditty L, Zwick U (2012) Dynamic approximate all-pairs shortest paths in undirected graphs. SIAM J Comput 41(3):670–683

Shakarian P, Broecheler M, Subrahmanian VS, Molinaro C (2013) Using generalized annotated programs to solve social network diffusion optimization problems. ACM Trans Comput Logic 14(2):10:1–10:40

Sommer C (2014) Shortest-path queries in static networks. ACM Comput Surv 46:45:1–45:31

Tagarelli A, Interdonato R (2015) Time-aware analysis and ranking of lurkers in social networks. Soc Netw Anal Min 5(1):46:1–46:23

Wu L, Xiao X, Deng D, Cong G, Zhu AD, Zhou S (2012) Shortest path and distance queries on road networks: an experimental evaluation. Proc VLDB Endow (PVLDB) 5(5):406–417

Zhu A.D, Ma H, Xiao X, Luo S, Tang Y, Zhou S (2013) Shortest path and distance queries on road networks: towards bridging theory and practice. In: Proceedings of international conference on management of data (SIGMOD), pp 857–868

Zhu A.D, Xiao X, Wang S, Lin W (2013) Efficient single-source shortest path and distance queries on large graphs. In: Proceedings of ACM SIGKDD international conference on knowledge discovery and data mining (KDD), pp 998–1006