Karlsruhe Reports in Informatics 2016,4
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Faster Incremental All-pairs Shortest Paths

Arie Slobbe, Elisabetta Bergamini, Henning Meyerhenke

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Faster Incremental All-pairs Shortest Paths

Arie Slobbe[1], Elisabetta Bergamini[2], and Henning Meyerhenke[2]

[1] Macalester College, Saint Paul, MN, USA, aslobbe@macalester.edu
[2] Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Germany,
{elisabetta.bergamini, meyerhenke}@kit.edu

Abstract. Finding the shortest paths between pairs of nodes is a fundamental graph problem. Due to the dynamic nature of many today's networks, algorithms that quickly update shortest paths have become a necessity. Several dynamic algorithms have been proposed over the last fifty years, focusing on different update types (incremental- and decremental-only, fully-dynamic). Although an $O(n^2)$-algorithm for the incremental problem is known, previous experimental analysis has shown that algorithms with worse complexities perform significantly better and are therefore preferred in practice. In this paper, we propose a new incremental algorithm with optimal worse-case complexity of $O(n^2)$, which is also faster in practice than all existing methods. We prove that after the insertion of an edge or a node, newly created shortest paths have certain properties which allow us to significantly reduce the amount of work needed to recompute them. In our experimental study on several real-world graphs, we show that our approach outperforms existing methods, on average by a factor 10 and up to a factor 30 for edge insertion and 150 for node insertion.

1 Introduction

One of the fundamental problems in graph theory is the computation of shortest paths between node pairs. It is quite easy to imagine applications for which one can be interested in shortest paths and distances: transportation networks (where weights might represent travel times), social network analysis (for which several shortest-paths based centrality measures exist), but also database systems and document formatting are just a few examples [18]. In the literature, finding shortest paths from a single node is referred to as Single-Source Shortest Path (SSSP), whereas finding shortest paths between all pairs of nodes in the graph is named All-Pairs Shortest Paths (APSP). The SSSP is solved using Dijkstra's algorithm (or a Breadth-First Search, BFS, in unweighted graphs). By computing an SSSP rooted in each node, the APSP can therefore be computed in $O(nm)$ operations in unweighted graphs and $O(n(m + n \log n))$ in weighted graphs. An alternative approach [23] solves the APSP problem in $O(n^{2.373} \log n)$ using fast matrix multiplication. However, this approach only computes distances (problem sometimes referred to as All-Pairs Distances) and does not allow us to retrieve shortest paths in linear time in the length of the path. Also, since real-world networks are often sparse, the SSSP-based approach is often used in practice.

Motivation Networks such as the Web and social networks continuously undergo changes. Dynamic APSP algorithms update the paths after a change in the graph and are divided among incremental (handle only edge/node insertions and edge weight decreases), decremental (handle only edge/node deletions and edge weight increases) and fully-dynamic (handle both). Whereas the best fully-dynamic APSP algorithm by Thorup [22] requires $O(n^2(\log n + \log^2((m+n)/n))$ amortized time per update, the incremental problem can be solved in $O(n^2)$ worst-case time. Although it might seem reductive to only consider insertions and weight decreases, it is important to note that many real-world dynamic networks evolve only this way and do not shrink. Think, for example, about co-authorship networks: a new author (node) might be added to the network, or a new edge (paper), but an existing paper or author will not disappear.

Our contribution We present two new incremental algorithms for the APSP problem, one for edge insertion/weight decrease and one for node insertion. Like existing algorithms, the worst-case complexity of our methods is $O(n^2)$. However, compared to these algorithms, our approaches reduce significantly the number of operations in the average case. Instead of comparing distances between each pair of nodes in the graph, we efficiently identify the affected node pairs, based on properties of the newly-created shortest paths. Our experimental study on real-world graphs shows that our two algorithms outperform existing methods, on average by a factor 10 and up to a factor 30 for edge insertion and 150 for node insertion.

2 Preliminaries

2.1 Notation and problem specification

Let $G = (V, E, \omega)$ be the initial graph (directed or undirected), where V is the set of vertices, E the set of edges and $\omega : E \to \mathbb{R}_{\geq 0}$ is the edge weight function. We define G transposed as the graph $G^t = (V, E^t, \omega)$, where $E^t = \{(v, u) | (u, v) \in E\}$). In case of single-edge update, we denote the update by $(u, v, \omega'(u, v))$ where $(u, v),\ u, v \in V$ is either a new edge of weight $\omega'(u, v)$ or an existing edge for which we are setting the new weight to $\omega'(u, v)$ (smaller than the old weight $\omega(u, v)$). In case of node insertion, we represent the update with $(z, Z^{(\text{in})}, Z^{(\text{out})})$, where z is the new node and $Z^{(\text{in})} = \{(u_1, z), ..., (u_k, z)\},\ u_1, ..., u_k \in V$ is a set of new incoming edges and $Z^{(\text{out})} = \{(z, v_1), ..., (z, v_j)\},\ v_1, ..., v_j \in V$ is a set of new outgoing edges.

Let $G' = (V', E', \omega')$ be the graph after the update. For an edge update, $V' = V$ and $E' = E \cup \{(u, v)\}$, for a node update $V' = V \cup \{z\}$ and $E' = E \cup Z^{(\text{in})} \cup Z^{(\text{out})}$. For any two nodes $x, y \in V'$, we denote by $d(x, y)$ the shortest-path distance from x to y before the edge update, and we denote by $d'(x, y)$ the shortest-path distance from x to y after the edge update. Since the length of a shortest path can only decrease after an edge insertion or decrease of an edge weight, it is always true for all node pairs (x, y) that $d'(x, y) \leq d(x, y)$. On a shortest path from s to t in G, we say w is a predecessor of t when $(w, t) \in E$ and

$d(s,w)+\omega(w,t)=d(s,t)$. For example, if $[s,x,y,t]$ is a shortest path between s and t, then y is the predecessor of t. We can say equivalently that t is a successor of y for s.

We define the *incremental SSSP* problem as follows: given a (source) node $s \in V$ and the old distances $d(s,\cdot)$ in the shortest paths from s to all other nodes, find the distances $d'(s,\cdot)$ (and new shortest paths) in G'. In the *incremental APSP* problem, we have to find all distances $d'(s,t)$ (and new shortest paths) in the shortest path between each pair of nodes $s,t \in V$.

In the following, we will describe only how to compute the new distances d'. However, the update of the shortest paths is straightforward and can be obtained by simply updating the predecessors in the new shortest paths while updating distances.

2.2 Related Work

The problem of dynamically updating shortest paths in a graph has a long history. Papers on the topic date back almost fifty years [17]. The first algorithm with $O(n^2)$ worst-case complexity for edge insertions was proposed in 1985 [7]. The basic idea is simple: naming (u,v) the newly-inserted edge, for each pair of nodes (x,y), we take the minimum between the old distance $d(x,y)$ and the sum $d(x,u)+\omega(u,v)+d(v,y)$, where $\omega(u,v)$ is the weight of edge (u,v).

In subsequent years, several algorithms have been proposed for special classes of graphs. Ausiello et al. [1] proposed an incremental shortest path algorithm for graphs with integer edge weights less than a constant value C. The amortized running time of their algorithm is $O(Cn\log n)$ per edge insertion. The algorithm by Henzinger et al. [10] works on fully-dynamic planar graphs with integer weights and its running time is $O(n^{9/7}\log(nC))$ per edge update. Also King [14] presented a fully-dynamic algorithm for maintaining APSPs in graphs with integer weights less than C: the running time in this case is $O(n^{2.5}\sqrt{C\log n})$ per edge update. Ramalingam and Reps [19,20] proposed dynamic shortest path algorithms for graphs with arbitrary real weights, both for the SSSP and the APSP problem. Their worst-case running times are the same as recomputing from scratch. However, they express the complexity of their algorithms according to a new model, which will be described in Section 3.3. Frigioni et al. [8,9] designed fast algorithms for graphs with bounded genus, bounded degree graphs, and bounded tree width. Also in this case, in the worst case the running times of the algorithms are as bad as recomputing from scratch. The first to propose an asymptotically faster algorithm for fully-dynamic graphs with real edge weights were Demetrescu and Italiano [5]. Their algorithm requires $O(n^2\log^3 n)$ amortized time per update. This bound was then improved by Thorup [22], whose algorithm achieves $O(n^2(\log n+\log^2((m+n)/n))$ amortized time per update. An experimental evaluation [6] studied the behavior of the different approaches on real-world networks. Despite the worse worst-case running time, the dynamic APSP algorithm by Ramalingam and Reps [20] (which we refer to as RR) was shown to be the best-performing in practice. For this reason, it has recently

been used also as a basis for algorithms that dynamically update centrality measures [11,12,3,4].

Regarding node insertions, a very simple method has been proposed recently [13] (we refer to it as KNNB). The algorithm has a $\Theta(n^2)$ complexity and is basically composed of two steps. Naming z the newly-inserted node and $(u_1, z), ..., (u_k, z)$ the new incoming edges edges of z, KNNB sets the distance $d(x, z),\ x \neq z$ to $\min_{i=1,...,k} d(x, u_i)+\omega(u_i, z)$ (analogously, naming $(z, v_1), ..., (z, v_j)$ the outgoing edges, it sets $d(z, x)$ to $\min_{i=1,...,j} \omega(z, v_i) + d(v_i, x)$). Then, KNNB compares the distance between each node pair (x, y) with $d(x, z) + d(z, y)$ and, in case this is smaller, sets $d(x, y)$ to $d(x, z) + d(z, y)$.

2.3 RR algorithm

Since our edge-update algorithm is an improvement of RR [20] (both in terms of worst-case running time and practical performance, as we will see in Section 4), we briefly describe it in this section. To make the understanding easier, we start from the incremental SSSP algorithm and then show how this has been extended by Ramalingam and Reps to the incremental APSP problem [20].

Incremental SSSP Let $s \in V$ be the source node from which we want to find the new distances $d'(s, \cdot)$ and let $(u, v, \omega'(u, v))$ be the edge update. We define all the nodes $t \in V$ such that $d'(s, t) < d(s, t)$ as *affected targets*. Notice that in this case $d'(s, t) = d(s, u) + \omega'(u, v) + d(v, t)$. In fact, the only new edge in G' is (u, v), therefore all shortcuts must go through node u and node v. Since we know $d(s, \cdot)$, the only thing we need to know to find $d'(s, t)$ for the affected nodes t is $d(v, t)$. Also, Ramalingam and Reps [20] proved that a node t can be affected only if all the nodes in the shortest path from v to t are also affected. Therefore, to find the affected nodes we can *start* an SSSP from t (either Dijkstra or a BFS, depending on wether G is weighted or not). If a node y is an affected target, then we update its distance and traverse its outgoing edges. Otherwise, we do not need to continue the search from y. Algorithm 2 in the Appendix (Section A) describes the SSSP update. The algorithm works basically like a Dijkstra rooted in v, with the difference that only the affected nodes are inserted into the priority queue (Line 8). We refer to it as truncated Dijkstra.

Incremental APSP (RR) Clearly, to update the distances between all pairs of nodes one could just run Algorithm 2 from each node $s \in V$. However, Ramalingam and Reps showed that we can do better than this [20]. If G is directed, the distance from v to any other node cannot decrease as a consequence of the edge update $(u, v, \omega'(u, v))$. Therefore, $d(v, \cdot) = d'(v, \cdot)$. Since in this case we know $d(v, \cdot)$, we do not need to recompute (part of) the SSSP from v like in Algorithm 2. To find the affected nodes with respect to a given source, we can just walk down the nodes reachable from v, inserting them into a queue as we visit them and therefore sidestepping the use of a priority queue. In other words, we are basically performing a Breadth-First Search (BFS) rooted in v instead of Dijkstra. Notice that doing this we might not visit the nodes in order of increasing

distance from v, but this is not necessary since we already know all the distances from v. This optimized version of Algorithm 2 for the APSP is described in Algorithm 3 in the Appendix (Section A). As for the incremental SSSP, we stop the traversal of the outgoing edges of a node when the node is not affected. We call this traversal a truncated BFS. Instead of running Algorithm 3 from each node in V, RR further optimizes the update by first identifying the *affected sources*, i. e. the nodes s such that $d(s,v) > d'(s,v)$ (equivalently, $d(s,v) > d(s,u) + \omega'(u,v)$). These are the only nodes for which the condition in Line 1 is true and therefore the only ones for which we actually have to update something. The affected sources can be identified by running Algorithm 3 rooted in u on G transposed. This basically means that instead of scanning the outgoing edges of u and its affected successors, we scan the incoming edges. Algorithm 4 in the Appendix (Section A) shows the pseudocode.

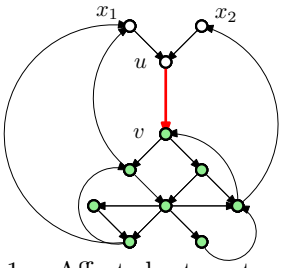

Fig. 1: Affected targets (in green) and affected sources (x_1, x_2, u) after the insertion of edge (u, v).

To summarize, RR finds all the affected sources with Algorithm 4 first and then for each affected source updates the distances to its affected targets with Algorithm 3.

3 Faster Incremental APSP

3.1 Edge insertion

To explain our new algorithm, let us start with a simple example. In Figure 1, a new edge has been inserted between node u and node v. This decreases the distance from nodes x_1, x_2, u to all the nodes represented in green. RR would first identify the affected source nodes (i. e. x_1, x_2, u) and, for each of them, run a (truncated) BFS rooted in v to identify the affected targets. This means we are repeating almost exactly the same procedure each time, namely once for each of the affected sources. It is true that we have to update the distances between each affected source and each one of its affected targets (and this cannot be avoided). However, RR also goes through the outgoing edges of each affected target multiple times (leading to a worse-case running time of $O(mn)$).

The basic idea of our algorithm is to avoid this recomputation. Instead of starting a BFS from v for each one of the affected sources, we run the BFS only once for the affected source u, updating at the same time also the distances from the other affected sources. The reason why we can do this is based on a property of the affected nodes, stated in Lemma 1. Naming $S(y)$ the set of affected sources of node y, i. e. $S(y) := \{x \in V : d'(x,y) < d(x,y)\}$, the following holds (proof in Section B of the Appendix):

Lemma 1. *Let a directed graph $G = (V, E)$, a single edge update $(u, v, \omega'(u, v))$ and a node $y \in V$ be given. Then: for each z that is predecessor of y in a shortest path from v, $S(y) \subseteq S(z)$.*

Naming $T(x) := \{y \in V : d'(x,y) < d(x,y)\}$ the set of affected targets of node x, we can similarly prove that $T(x) \subseteq T(s)$ if s is the successor of x in a shortest path to u. This implies that $T(x) \subseteq T(u),\ \forall x \in V$.

Corollary 1. *Let a directed graph $G = (V, E)$, a single edge update $(u, v, \omega'(u, v))$ and a node $x \in V$ be given. Then: $T(x) \subseteq T(u)$.*

Algorithm 1 shows the pseudocode for our APSP update. First, we identify all the affected sources of node v with Algorithm 4 (Line 2). Then, we basically run Algorithm 3 with source node $s = u$, with a few differences. First, when inserting an affected target w into Q, we also keep track of the predecessor y in (one of) the shortest path(s) from v (Line 22). Then, we do not only update the distance from u to y, but also those from each node in $S(y)$ to y. Since by Lemma 1 the set of affected targets of y is contained in that of z, we can just go through $S(z)$ (Line 10) to find out which of the $x \in S(z)$ are also affected sources for y i. e. $d(x,y) > d'(x,y)$. In this case, we add x to $S(y)$ (which will be used by the successors of y) and set $d(x,y)$ to $d'(x,y)$ (Lines 12 - 14).

Then, we loop over the outgoing edges of y to find new affected targets for u (Lines 18 - 23). Differently from Algorithm 3, here in Line 19 we also have to make sure that y is the predecessor for w in a shortest path from v, i. e. $d(v,w) = d(v,y) + \omega(y,w)$. Note that, by Corollary 1, $T(x) \subseteq T(u),\ \forall x \in V$, therefore Algorithm 1 will correctly update the distance between each affected source x and its affected targets.

Proposition 1. *Given a graph $G = (V, E)$, an edge update $(u, v, \omega'(u, v))$, and pairwise distances $d(\cdot, \cdot)$, Algorithm 1 correctly computes the distances $d'(x, y)$ for each $x, y \in V$ in $G' = (V, E \cup \{(u, v)\}$ (proof in Section B of the Appendix).*

Notice that if we are also interested in the actual shortest paths and not only in the distances, we can set the predecessor in the shortest path from x to y to $P(y)$, when updating the distances in Line 3 .

3.2 Node insertion

Let us now consider the node insertion problem: a new node z and a set of incoming edges $Z^{(\text{in})} = \{(u_1, z), ..., (u_k, z)\},\ u_1, ..., u_k \in V$ and a set of outgoing edges $Z^{(\text{out})} = \{(z, v_1), ..., (z, v_j)\},\ v_1, ..., v_j \in V$ are added to the graph G. We assume that before the insertion $d(x,z) = d(z,x) = \infty,\ \forall x \in V$. Clearly, one could just apply the Algorithm described in Section 3.1 to each edge in $Z^{(\text{in})}$ and $Z^{(\text{out})}$. However, this might imply that we update the distance between a certain node pair or visit a certain edge multiple times. Algorithm 5 in the Appendix (Section A) describes our optimized algorithm for the APSP update after a node insertion. First (Lines 1 - 13), we identify the set $S(z)$ of affected sources, i. e. the set of nodes x such that $d'(x,z) < d(x,z)$. Notice that all the new shortest paths have to go through z, therefore if $d'(x,y) < d(x,y)$ for two nodes x and y, then $x \in S(z)$. To identify the nodes in $S(z)$, we basically do the same as Algorithm 4, with the difference that here we need a priority queue, since we cannot use

Algorithm 1: APSP update - edge insertion

```
   Input  : Graph G = (V, E), edge update (u, v, ω'(u, v)), pairwise distances
            d(·, ·)
   Output : Updated pairwise distances
   Assume: boolean vis(v) is false, ∀v ∈ V
 1 if ω'(u, v) < d(u, v) then
 2     S(v) ← findAffectedSources(G, (u, v, ω'(u, v)), d);
 3     d(u, v) ← ω'(u, v);
 4     Q ← ∅;
 5     P(v) ← v;
 6     Q.push(v);
 7     vis(v) ← true;
 8     while Q.length() > 0 do
 9         y = Q.front();
           /* update distances for source nodes                          */
10         foreach x ∈ S(P(y)) do
11             if d(x, y) > d(x, u) + ω'(u, v) + d(v, y) then
12                 d(x, y) ← d(x, u) + ω'(u, v) + d(v, y);
13                 if y ≠ v then
14                     S(y).insert(x);
15                 end
16             end
17         end
           /* enqueue all neighbors that get closer to u                 */
18         foreach w s.t. (y, w) ∈ E do
19             if not vis(w) and d(u, w) > ω'(u, v) + d(v, w) and
               d(v, w) = d(v, y) + ω(y, w) then
20                 Q.push(y, w);
21                 vis(w) ← true;
22                 P(w) ← y;
23             end
24         end
25     end
26 end
```

previously-computed distances (in the case of Algorithm 4, the distances to node u). While doing this, we update the distances to node z (Line 8). Then (Lines 14 - 32), we update the distances between each affected target node y and each one of its affected sources $S(y) = \{x \in V | d'(x, y) < d(x, y)\}$. Notice that in this case, $d'(x, y)$ is equal to $d'(x, z) + d'(z, y)$. Similarly to Lemma 1, we can prove that $S(y) \subseteq S(q)$ if q is the predecessor of y in a shortest path from z. Therefore, we can basically execute Algorithm 1, also in this second part replacing the queue with a priority queue, since we also have to compute the SSSP from node z. Since $z \notin S(z)$, the distances from z are updated separately (Line 29).

Proposition 2. *Given a graph $G = (V, E)$, a node update $(z, Z^{(in)}, Z^{(out)})$, and pairwise distances $d(\cdot, \cdot)$, Algorithm 5 correctly computes the distances $d'(x, y)$ for each $x, y \in V$ in $G' = (V \cup \{z\}, E \cup Z^{(in)} \cup Z^{(out)})$ (proof in Section B of the Appendix).*

3.3 Time Complexity

As defined in Sections 3.1 and 3.2, let $T(x)$ of a node $x \in V$ be the set $\{y \in V : d'(x, y) < d(x, y)\}$ (and, analogously, $S(y) = \{x \in V : d'(x, y) < d(x, y)\}$). Since the distances between each x and each node in $T(x)$ must be updated, the complexity of any algorithm for the APSP update is clearly $\Omega(\sum_{x \in V} |T(x)|)$, where $|T(x)|$ is the cardinality of $T(x)$. This quantity can be $\Theta(n^2)$, in case for example of an edge insertion connecting two disconnected components of equal size. However, if the update affects only a small portion of the graph, this can be much smaller than n^2. The simple algorithm by Even and Gazit [7] (compare each pair of nodes in V and update the distances when necessary) requires $\Theta(n^2)$, whatever the set of affected nodes is. In [20], the complexity of RR is expressed as a function of the set of affected nodes. Let us define the *extended size* $||A||$ of a set of nodes A as the sum of the number of nodes in A and the number of edges that have a node of A as their endpoint. Using a Fibonacci heap-based priority queue, the complexity of the incremental SSSP algorithm described in Section 2.3 is $\Theta(||T(s)|| + |T(s)| \log |T(s)|)$, since each node in $|T(s)|$ is inserted and extracted from the priority queue and since each of its outgoing edges is visited. The complexity of RR is $\Theta(||S(v)||)$ for the initial identification of affected sources (Algorithm 4) and $\Theta(\sum_{x \in S(v)} ||T(x)||)$, because the priority queue has been replaced with a queue. If we express this in terms of worst-case complexity, this can be $\Theta(nm)$. However, in most cases, the number of operations is actually much smaller than $n \cdot m$, and also than n^2. Let us now consider the complexity of our new algorithms. Algorithm 1 first identifies the set of affected sources with Algorithm 4, which takes $\Theta(||S(v)||)$. Then, we run a truncated BFS from v, which takes $||T(u)||$. For each node in $y \in T(u)$, we also scan the set of affected sources of the predecessor $P(y)$ of y in a shortest path from v (Line 10). Therefore, the following proposition holds.

Proposition 3. *The running time of Algorithm 1 for updating distances after an edge update $(u, v, \omega'(u, v))$ is $\Theta(||S(v)|| + ||T(u)|| + \sum_{y \in T(u)} |S(P(y))|)$.*

Notice that, since $S(P(y))$ is $O(n)$, the worst-case complexity of Algorithm 1 is $O(n^2)$. Also, notice that $\sum_{y \in T(u)} |S(P(y))| = O(\sum_{x \in V} ||T(x)||)$, the running time of RR. In fact, $||T(x)|| \geq |T(x)| + |N(T(x))|$, where $N(T(x))$ is the set of nodes that have at least one neighbor in $T(x)$. If a node x is in $S(P(y))$, then $P(y) \in T(x)$ and therefore $y \in N(T(x))$. This means that for each node $x \in S(P(y)$ that is increasing by one the cardinality of $\sum_{y \in T(u)} |S(P(y))|$, we are increasing by at least one also the cardinality of $\sum_{x \in V} ||T(x)||$.

Algorithm 5 for node insertion requires $\Theta(||S(z)|| + |S(z)| \log |S(z)|)$ for Lines 1 - 13 (truncated Dijkstra from z on G transposed). Then, for Lines 14

- 32 the running time is $\Theta(||T(z)|| + |T(z)| \log |T(z)|)$ for the truncated Dijkstra from z, plus $\sum_{y \in T(z)} |S(P(y))|)$ for Lines 18 - 23.

Proposition 4. *The running time of Algorithm 5 for updating distances after the insertion of node z is* $\Theta(||S(z)||+|S(z)| \log |S(z)|+||T(z)||+|T(z)| \log |T(z)|+ \sum_{y \in T(z)} |S(P(y))|)$.

Also for Algorithm 5 the worst-case complexity is $O(n^2)$.

4 Experimental Results

Implementation and settings. We refer to our algorithm for edge update as QUINCA (from QUick Incremental APSP) and to the one for node insertion as QUINCA-N. For an experimental comparison with other edge update algorithms, we implement RR [20] and the simple quadratic algorithm by Even and Gazit (EG) [7]. For node insertions, we implement KNNB [13]. All the algorithms are implemented in C++, building on the open-source *NetworKit* framework [21]. We choose to implement RR because it was shown to be the best performing in practice [6], and EG and KNNB for their optimal worst-case asymptotic bounds of $O(n^2)$. The machine used for the experiments has 2 x 8 Intel(R) Xeon(R) E5-2680 cores at 2.7 GHz, of which we use only one core, and 256 GB RAM.

Data sets and experimental design. For our experiments, we consider a set of real-world networks belonging to different domains, taken from SNAP [16], KONECT [15] and the 10th DIMACS Implementation Challenge [2]. The properties of the networks are reported Table 1 (directed) and in Table 2 (undirected) in Section C of the Appendix. In our experiments on single edge insertions, we compare QUINCA with RR and EG, whereas on node insertions we run QUINCA, KNNB and RR. Since RR is a single-edge insertion algorithm, we apply it to each neighboring edge of the new node and report the sum of the running times. We do not do the same for EG because its number of operations is always greater than that of KNNB if the node has more than one neighboring edge (equal otherwise). To simulate real edge insertions, we remove an existing edge from the graph (chosen at random), compute distances on the graph without the edge and then add the edge back, updating distances with the incremental algorithms. For node insertions we basically do the same: we remove all the neighboring edges of a random node and then we insert them back and update distances. Since random updates do not usually affect a large fraction of nodes, we also examine the behavior of the algorithms on a different scenario: we remove and insert back the node with maximum degree instead of a random node. This is likely to create several shortcuts in the shortest paths and therefore affect many pair-wise distances. For all random updates, we consider 20 edge/node insertions and report the average over these 20 runs.

Experimental results. The results for edge insertions are reported in Table 1 and Table 2 in Section C of the Appendix. Although the speedups may vary significantly among the networks (due to their structures and sizes), some general

considerations can be made. First, RR is always faster than EG, which confirms the previous experimental study by Demetrescu and Italiano [6]. Also, our new method QUINCA clearly outperforms the existing approaches and is always faster than both of them. The speedup of our method on RR varies between ≈ 1.5 (on the `faroe-islands` street network) and ≈ 30 (on `GoogleNw` and `as-caida20071105`). Compared to recomputation, QUINCA is on average 17634.2 times faster, RR is on average 1715.1 times faster and EG is on average 39.3 times faster (geometric means). This means that our edge insertion algorithm improves the state of the art by about a factor 10. Table 3 and Table 4 in

Fig. 2: Average speedups on recomputation for the three incremental algorithms, under different types of updates: single edge insertion, random node insertion and maximum-distance node insertion. The bars represent the geometric means of the speedups on all tested directed and undirected networks (see Section C of the Appendix for all exact values).

Section C of the Appendix summarize the results for node insertions on undirected and directed graphs respectively. The first three columns of both tables report the speedups of the incremental algorithms on recomputation for random node insertions, whereas the last three report the speedups for maximum degree node insertion. Also in this case, QUINCA-N is the fastest approach on almost all networks and all update types. Only for the maximum degree insertion and for `oregon2-010526`, the speedup of KNNB is better than that of QUINCA-N, although by a very small margin: 43.34 versus 42.47. This is due to the fact that in this case the node insertion affects almost all the nodes in the graph, making it faster to simply compare each pair of nodes. It is interesting to notice that EG and KNNB behave quite differently depending on the type of update. This is particularly evident looking at Figure 2, which summarizes all the speedups on all types of updates. On random node insertions, performing RR on each inserted edge is significantly faster than KNNB: compared to recomputation, the average speedups of QUINCA-N, EG and KNNB are 5503.2, 604.3 and 64.7 respectively. However, on maximum-degree insertion the speedups are very different: 281.2 for QUINCA-N, 12.4 for EG and 33.2 for KNNB. The reason why KNNB is slower

than EG on random insertions is that often only a small fraction of the graph is affected and therefore it is still faster to run truncated BFSs from each affected node than comparing each pair of nodes in the graph. On the other hand, on maximum-degree insertions, a large fraction of nodes become affected, making KNNB faster. Also, this explains why the speedups of all incremental algorithms are significantly worse for this type of update.

To summarize, the results show that the relative performances of other approaches (when compared to each other) strongly depend on the type of update, whereas our method outperforms them on all tested instances. In particular, QUINCA and QUINCA-N are on average about a factor 10 times faster than the best competitors, and up to a factor 30 for edge insertions (`GoogleNw` and `as-caida20071105`) and up to a factor 150 for node insertions (`subelj-cora-cora`).

5 Conclusions and future work

Finding shortest paths between the nodes of a graph is a problem of great practical relevance. In this paper we have proposed and evaluated new techniques for the APSP update after the insertion (or weight decrease) of an edge and the insertion of a node. Our algorithms for edge and node update have a worst-case complexity of $O(n^2)$ and our experiments on real networks show that they outperform existing methods. Also, we show that for node insertions the performance of existing algorithms strongly depends on the type of the update, whereas our new approach is almost always better than all competitors, on average by one order of magnitude.

Future work might include the use of parallelization to further increase the speedups of our algorithms. Also, our techniques might be used as a basis to design more efficient incremental algorithms for shortest-paths based centrality measure, such as closeness and betweenness.

It would also be interesting to investigate whether our techniques can be applied to the more difficult problem of updating the shortest paths after an edge or a node deletion, for which the best known algorithm has a complexity of $O(n^2(\log n + \log^2((m+n)/n))$ amortized time per update.

Our implementations are based on *NetworKit*[1], the open-source framework for high-performance large-scale network analysis, and we plan to publish our source code in upcoming releases of the package. The code can be inspected for reviewing purposes by accessing the private hg repository at `https://algohub.iti.kit.edu/parco/NetworKit/NetworKit-fork-arie` with login "rev-inc-apsp" and password "InKaP$P".

References

1. G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12(4):615–638, 1991.
2. D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

3. E. Bergamini and H. Meyerhenke. Fully-dynamic approximation of betweenness centrality. In *23rd European Symp. on Algorithms, ESA 2015*, volume 9294 of *LNCS*, pages 155–166. Springer, 2015.
4. E. Bergamini, H. Meyerhenke, and C. Staudt. Approximating betweenness centrality in large evolving networks. In *17th Work. on Algorithm Eng. and Exp., ALENEX 2015*, pages 133–146. SIAM, 2015.
5. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
6. C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.
7. S. Even and H. Gazit. On the computational complexity of dynamic graph problems. *Methods of Operations Research*, 49:371–387, 1985.
8. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semidynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22(3):250–274, 1998.
9. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, 2000.
10. M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.
11. M. Kas, K. M. Carley, and L. R. Carley. Incremental closeness centrality for dynamically changing social networks. In *Advances in Social Networks Analysis and Mining 2013, ASONAM '13*, pages 1250–1258. ACM, 2013.
12. M. Kas, K. M. Carley, and L. R. Carley. An incremental algorithm for updating betweenness centrality and k-betweenness centrality and its performance on realistic dynamic social network data. *Social Netw. Analys. Mining*, 4(1):235, 2014.
13. S. S. Khopkar, R. Nagi, A. G. Nikolaev, and V. Bhembre. Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis. *Social Netw. Analys. Mining*, 4(1):220, 2014.
14. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Symp. on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91. IEEE Computer Society, 1999.
15. J. Kunegis. KONECT: the koblenz network collection. In *22nd Int. World Wide Web Conf., WWW '13*, pages 1343–1350, 2013.
16. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
17. P. Loubal. *A network evaluation procedure*. Highway Research Record 205, 1967.
18. G. Ramalingam. *Bounded Incremental Computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer, 1996.
19. G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
20. G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2):233–277, 1996.
21. C. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. `http://arxiv.org/abs/1403.3005`, 2014.
22. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *9th Scandinavian Workshop on Algorithm Theory, SWAT 2004*, volume 3111 of *LNCS*, pages 384–396. Springer, 2004.
23. V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *44th Symp. on Theory of Computing, STOC 2012*, pages 887–898. ACM, 2012.

A Pseudocodes

Algorithm 2: Incremental SSSP (truncated Dijkstra) [20]

Input : Graph $G = (V, E)$, source node $s \in V$, edge update $(u, v, \omega'(u, v))$, distances $d(s, \cdot)$
Output: Updated distances from s

```
if d(s,v) > d(s,u) + ω'(u,v) then
    d(s,v) ← d(s,u) + ω'(u,v);
    PQ ← ∅;
    PQ.insert(v, d(s,v));
    while PQ.length() > 0 do
        (y, p_sy) ← PQ.extractMin();
        foreach w s.t. (y,w) ∈ E do
            if d(s,w) > p_sy + ω(y,w) then
                d(s,w) ← p_sy + ω(y,w);
                PQ.update(w, d(s,w));
            end
        end
    end
end
```

Algorithm 3: Incremental APSP for one source (truncated BFS) [20]

Input : Graph $G = (V, E)$, source node $s \in V$, edge update $(u, v, \omega'(u, v))$, distances $d(\cdot, \cdot)$
Output : Updated distances from s
Assume: boolean $vis(v)$ is false $\forall v \in V$

```
if d(s,v) > d(s,u) + ω'(u,v) then
    d(s,v) ← d(s,u) + ω'(u,v);
    Q ← ∅;
    Q.insert(v);
    vis(v) ← true;
    while Q.length() > 0 do
        y ← Q.front();
        d(s,y) ← d(s,u) + ω'(u,v) + d(v,y);
        foreach w s.t. (y,w) ∈ E do
            if not vis(w) and d(s,w) > d(s,u) + ω'(u,v) + d(v,w) then
                vis(w) ← true;
                Q.push(w);
            end
        end
    end
end
Reset vis(·) to false;
```

Algorithm 4: Find affected sources [20]

Input : Graph $G = (V, E)$, edge update $(u, v, \omega'(u, v))$, distances $d(\cdot, \cdot)$
Output : Affected sources S
Assume: $vis(v)$ is false $\forall v \in V$

```
1  S ← ∅;
2  if d(u, v) > ω'(u, v) then
3      Q ← ∅;
4      Q.insert(u);
5      vis(u) ← true;
6      while Q.length() > 0 do
7          x ← Q.front();
8          foreach z s.t. (z, x) ∈ E do
9              if not vis(z) and d(z, v) > d(z, u) + ω'(u, v) then
10                 Q.push(z);
11                 vis(z) ← true;
12                 S.insert(z);
13             end
14         end
15     end
16 end
17 Reset vis(·) to false;
18 return S
```

Algorithm 5: APSP update - node insertion

Input : Graph $G = (V, E)$, node update $(z, Z^{(\text{in})}, Z^{(\text{out})})$, pairwise distances $d(\cdot, \cdot)$

Output : Updated pairwise distances

$S(z) \leftarrow \emptyset$;
$PQ \leftarrow \emptyset$;
$PQ.insert(z, 0)$;
while $PQ.length() > 0$ **do**
    $(x, p_{xz}) \leftarrow PQ.extractMin()$;
    foreach q *s.t.* $(q, x) \in E$ **do**
        if $d(q, z) > \omega(q, x) + p_{xz}$ **then**
            $d(q, z) \leftarrow \omega(q, x) + p_{xz}$;
            $PQ.update(q, d(q, z))$;
            $S(z).insert(q)$;
        end
    end
end
$PQ.push(z, 0)$;
while $PQ.length() > 0$ **do**
    $(y, p_{zy}) \leftarrow PQ.extractMin()$;
    if $y \neq z$ **then**
        foreach $x \in S(P(y))$ **do**
            if $d(x, y) > d(x, z) + d(z, y)$ **then**
                $d(x, y) \leftarrow d(x, z) + d(z, y)$;
                $S(y).insert(x)$;
            end
        end
    end
    foreach w *s.t.* $(y, w) \in E$ **do**
        if $d(z, w) > p_{zy} + \omega(y, w)$ **then**
            $PQ.update(w, d(z, w))$;
            $P(w) \leftarrow y$;
            $d(z, w) \leftarrow p_{zy} + \omega(y, w)$;
        end
    end
end

B Omitted Proofs

B.1 Proof of Lemma 1

Proof. Let x be any node in $S(y)$, i.e. $d'(x,y) < d(x,y)$. We want to show that $x \in S(z)$ (i.e. $d'(x,z) < d(x,z)$). It is true that $d(x,y) \leq d(x,z) + \omega(z,y)$ because either $d(x,z) + \omega(z,y)$ represents the shortest possible distance between x and y in G, or there exists a shorter path. Notice that $\omega'(z,y) = \omega(z,y)$. Thus, if $d(x,z) = d'(x,z)$ then

$$d(x,y) \leq d(x,z) + \omega(z,y) = d'(x,z), + \omega'(z,y) = d'(x,y),$$

which contradicts $x \in S(y)$. Since pairwise distances in G' can only be equal to or shorter than pairwise distances in G, $d(x,z) \neq d'(x,z)$ implies $d(x,z) > d'(x,z)$ and thus $x \in S(z)$. □

B.2 Proof of Proposition 1

Proof. Let (x,y) be a node pair such that $d'(x,y) < d(x,y)$. If $x = u$ and $y = v$, the distance is correctly updated in Line 3. Otherwise, $x \in S(v)$ and $y \in T(u)$, by Lemma 1 and Corollary 1. By correctness of Algorithm 4 (see [20] for proof of correctness), $x \in S(v)$ after Line 2. Then, at some point, node y will be extracted from Q (Line 9). This is true because $y \in T(u)$ and by correctness of Algorithm 3 (see [20] for proof of correctness). Also, by Lemma 1, $x \in S(z)$ for each node in each shortest path from v to y. This means that the condition in Line 11 will be true for each of them and x will be inserted in each of the $S(z)$, including $S(P(y))$. The distance between x and y will therefore be correctly updated in Line 12. □

B.3 Proof of Proposition 2

Proof. Initially, $d(x,z) = d(z,y) = \infty$, $\forall x, y \in V$. Therefore, all nodes x that can reach z in V' will be extracted from PQ in Line 5 and their distance $d'(x,z)$ will be correctly computed, by correctness of Dijkstra on G' transposed. Similarly, all nodes y reachable from z in V' will be extracted from PQ in Line 16 and the distance $d'(z,y)$ correctly computed. Similarly to Lemma 1 and Proposition 1, we can prove that $x \in S(P(y))$ in Line 6, if $d'(x,y) < d(x,y)$. Their distance $d'(x,y)$ will therefore be computed correctly in Line 20. □

C Additional experimental results

Graph	Nodes	Edges	Time Static (s)	QUINCA	EG	RR
polblogs	1 224	16 715	0.37	**5 797.72**	95.95	1 271.29
subelj-jung-j-jung-j	6 120	50 535	0.23	**10 285.91**	2.85	4 421.67
wiki-Vote	7 115	100 762	12.42	**15 593.83**	100.77	1 830.44
elec	7 118	103 617	2.86	**14 489.50**	26.13	1 592.99
p2p-Gnutella09	8 114	26 013	15.08	**20 458.02**	94.32	2 297.68
freeassoc	10 617	63 788	30.19	**18 392.22**	110.31	1 645.95
p2p-Gnutella04	10 876	39 994	28.85	**22 767.26**	100.86	2 192.05
dblp-cite	12 591	49 728	1.76	**12 755.20**	5.19	1 024.75
cfinder-google	15 763	170 335	34.74	**38 443.87**	65.22	2 748.30
p2p-Gnutella25	22 687	54 705	25.92	**22 680.02**	23.55	1 527.33
subelj-cora-cora	23 166	91 500	22.13	**50 104.85**	19.31	2 790.01
ego-twitter	23 370	33 101	1.41	**21 027.60**	1.21	8 173.44
ego-gplus	23 628	39 242	2.44	**29 039.04**	2.04	8 601.35
munmun-digg-reply	30 398	85 247	63.74	**32 484.83**	32.31	1 342.11
linux	30 837	213 424	10.78	**17 808.80**	5.31	954.18
faroe-islands	31 097	31 974	81.51	**65.80**	34.10	43.71

Table 1: Results on directed graphs after a single edge insertion. The table shows the time taken by the static algorithm (SSSP from each node) and the speedups of the incremental algorithms on it. Bold font represents the best speedup for each graph.

Graph	Nodes	Edges	Time Static (s)	QUINCA	EG	RR
HC-BIOGRID	4 039	10 321	3.56	**8 226.18**	89.17	1 485.08
Mus-musculus	4 610	5 747	2.86	**6 932.48**	55.27	691.73
Caenor-elegans	4 723	9 842	4.16	**8 974.83**	76.40	949.28
ca-GrQc	5 241	14 484	3.38	**11 172.57**	50.62	1 137.95
advogato	7 418	42 892	6.76	**15 020.19**	50.56	1 784.50
hprd-pp	9 465	37 039	20.99	**56 655.96**	96.76	3 888.49
ca-HepTh	9 877	25 973	18.06	**22 138.33**	76.41	2 494.78
dr-melanogaster	10 625	40 781	26.99	**32 705.07**	98.78	2 554.50
PGPgiantcompo	10 680	24 316	31.82	**11 591.48**	73.91	1 612.05
oregon1-010526	11 174	23 409	19.76	**39 407.76**	65.49	1 551.08
oregon2-010526	11 461	32 730	21.08	**41 509.26**	66.37	1 541.84
Homo-sapiens	13 690	61 130	48.20	**45 548.80**	106.51	2 576.66
GoogleNw	15 763	148 585	44.50	**64 969.18**	74.30	2 172.45
dip20090126	19 928	41 202	87.79	**29 922.43**	91.66	1 566.50
as-caida20071105	26 475	53 381	168.23	**49 022.58**	99.67	1 720.83

Table 2: Results on undirected graphs after a single edge insertion. The table shows the time taken by the static algorithm (SSSP from each node) and the speedups of the incremental algorithms on it. Bold font represents the best speedup for each graph.

Graph	Random			Maximum degree		
	QUINCA-N	KNNB	RR	QUINCA-N	KNNB	RR
polblogs	**835.75**	86.55	126.33	**44.84**	37.44	2.48
subelj-jung-j-jung-j	**2 846.96**	4.00	472.63	**728.90**	3.82	31.16
wiki-Vote	**13 520.33**	183.58	599.99	**211.70**	16.35	5.45
elec	**4 195.41**	57.99	257.66	**144.57**	16.55	5.51
p2p-Gnutella09	**1 302.14**	58.88	133.07	**571.06**	42.69	49.44
freeassoc	**2 633.07**	77.97	187.11	**1 157.70**	74.78	61.54
p2p-Gnutella04	**2 063.15**	56.17	166.03	**243.76**	53.19	37.31
dblp-cite	**2 525.56**	8.25	112.62	**12.75**	5.15	1.40
cfinder-google	**7 745.63**	100.83	444.21	**71.35**	80.14	19.75
p2p-Gnutella25	**2 201.40**	47.77	126.41	**611.83**	34.66	44.16
subelj-cora-cora	**7 494.77**	28.41	435.22	**36 771.79**	27.36	246.39
ego-twitter	**10 774.97**	3.40	713.22	**2 164.25**	3.41	176.58
ego-gplus	**15 821.97**	21.78	127.43	**215.68**	1.30	2.34
munmun-digg-reply	**11 705.47**	47.84	566.61	**138.40**	42.79	14.28
linux	**37 181.37**	8.78	581.75	**19 700.71**	8.81	241.20
faroe-islands	**887.88**	71.96	140.03	**2 151.78**	73.45	118.34

Table 3: Results on directed graphs after a node insertion. The first three columns report the speedups of the incremental algorithms on recomputation for random node insertions, whereas the last three report the speedups for maximum degree node insertion. Bold font represents the best speedup for each graph and type of insertion.

Graph	Random			Maximum degree		
	QUINCA-N	KNNB	RR	QUINCA-N	KNNB	RR
HC-BIOGRID	**3 401.32**	146.19	2 215.51	**227.13**	108.94	21.32
Mus-musculus	**4 245.39**	92.15	1 227.16	**68.66**	46.00	3.49
Caenor-elegans	**4 191.21**	126.17	1 193.00	**527.22**	95.91	61.97
ca-GrQc	**4 028.92**	98.01	609.91	**1 373.53**	78.09	165.24
advogato	**2 916.52**	83.39	699.97	**106.89**	23.67	3.59
hprd-pp	**5 260.68**	174.96	257.80	**414.39**	126.02	17.54
ca-HepTh	**5 818.89**	129.07	1 427.53	**453.95**	114.95	47.66
dr-melanogaster	**5 686.80**	165.55	782.77	**214.47**	129.80	15.27
PGPgiantcompo	**5 850.40**	160.35	949.85	**109.21**	98.33	15.28
oregon1-010526	**14 445.48**	110.43	5 016.66	**39.95**	15.51	0.45
oregon2-010526	**12 522.29**	111.22	2 577.04	42.47	**43.34**	0.46
Homo-sapiens	**8 330.10**	179.43	1 111.09	**409.96**	115.48	10.39
GoogleNw	**9 894.74**	125.74	4 876.25	**8.56**	7.01	0.06
dip20090126	**16 495.29**	156.47	3 224.10	**421.66**	141.25	11.88
as-caida20071105	**25 170.40**	158.33	4 385.44	**68.58**	36.75	0.76

Table 4: Results on undirected graphs after a node insertion. The first three columns report the speedups of the incremental algorithms on recomputation for random node insertions, whereas the last three report the speedups for maximum degree node insertion. Bold font represents the best speedup for each graph and type of insertion.