Decremental All-Pairs ALL Shortest Paths
and Betweenness Centrality *

Meghana Nasre\textsuperscript{1}, Matteo Pontecorvi\textsuperscript{2}, and Vijaya Ramachandran\textsuperscript{2}

\textsuperscript{1} Indian Institute of Technology Madras, India
meghana@cse.iitm.ac.in
\textsuperscript{2} University of Texas at Austin, USA
\{cavia,vlr\}@cs.utexas.edu

Abstract. We consider the all pairs all shortest paths (APASP) problem, which maintains the shortest path dag rooted at every vertex in a directed graph $G = (V, E)$ with positive edge weights. For this problem we present a decremental algorithm (that supports the deletion of a vertex, or weight increases on edges incident to a vertex). Our algorithm runs in amortized $O(\nu^* \cdot \log n)$ time per update, where $n = |V|$, and $\nu^*$ bounds the number of edges that lie on shortest paths through any given vertex. Our APASP algorithm can be used for the decremental computation of betweenness centrality (BC), a graph parameter that is widely used in the analysis of large complex networks. No nontrivial decremental algorithm for either problem was known prior to our work. Our method is a generalization of the decremental algorithm of Demetrescu and Italiano [3] for unique shortest paths, and for graphs with $\nu^* = O(n)$, we match the bound in [3]. Thus for graphs with a constant number of shortest paths between any pair of vertices, our algorithm maintains APASP and BC scores in amortized time $O(n^2 \cdot \log n)$ under decremental updates, regardless of the number of edges in the graph.

1 Introduction

Given a directed graph $G = (V, E)$, with a positive real weight $w(e)$ on each edge $e$, we consider the problem of maintaining the shortest path dag rooted at every vertex in $V$ (we will refer to these as the SP dags). We use the term all-pairs ALL shortest paths (APASP) to denote the collection of SP dags rooted at all $v \in V$, since one can generate all the (up to exponential number of) shortest paths in $G$ from these dags. These dags give a natural structural property of $G$ which is of use in any application where several or all shortest paths need to be examined. A particular application that motivated our work is the computation of betweenness centrality (BC) scores of vertices in a graph [4].

In this paper we present a decremental algorithm for the APASP problem, where each update in $G$ either deletes or increases the weight of some edges incident on a vertex. Our method is a generalization of the method developed by Demetrescu and Italiano [3] (the ‘DI’ method) for decremental APSP where only one shortest path is needed. The DI decremental algorithm [3] runs in $O(n^2 \cdot \log n)$ amortized time per update, for a sufficiently long update sequence. This decremental algorithm is also extended to a fully dynamic algorithm in [3] that runs in $O(n^2 \cdot \log^3 n)$ time, and this result was improved to $O(n^2 \cdot \log^2 n)$ amortized time by Thorup [14]; both algorithms have within them essentially the same decremental algorithm. We briefly discuss the fully dynamic case, for which APASP results have been obtained recently by two of the authors, at the end of the paper.

In [3,14] the goal was to compute all pairs shortest path distances, and hence these algorithms preprocess the graph in order to have a unique shortest path between every pair of vertices. The unique shortest paths assumption, although not restrictive in their case, is crucial to the correctness and time complexity of their algorithms. We are interested in the more general problem of APASP, and this poses several challenges in generalizing the approach in [3].

In addition to APASP, our method gives decremental algorithms for the following two problems.

\begin{itemize}
\item This work was supported in part by NSF grant CCF-0830737. The first author was also supported by CSE/14-15/824/NFIG/MEGA. The second and third authors were also supported by NSF grant CCF-1320675.
\end{itemize}
Locally Shortest Paths (LSPs). For a path $\pi_{xy} \in G$, we define the $\pi_{xy}$ distance from $x$ to $y$ as $w(\pi_{xy}) = \sum_{e \in \pi_{xy}} w(e)$, and the $\pi_{xy}$ length from $x$ to $y$ as the number of edges on $\pi_{xy}$. For any $x, y \in V$, $d(x, y)$ denotes the shortest path distance from $x$ to $y$ in $G$. A path $\pi_{xy}$ in $G$ is a locally shortest path (LSP) if either $\pi_{xy}$ contains a single vertex, or every proper subpath of $\pi_{xy}$ is a shortest path in $G$. As noted in [3], every shortest path (SP) is an LSP, but an LSP need not be an SP (e.g., every single edge is an LSP).

The DI method maintains all LSPs in a graph with unique shortest paths, and these are key to efficiently maintaining shortest paths under decremental and fully dynamic updates. The decremental method we present here maintains all LSPs for all (multiple) shortest paths in a graph, using a compact tuple representation.

Betweenness Centrality (BC). Betweenness centrality is a widely-used measure in the analysis of large complex networks, and is defined as follows. For any pair $x, y$ in $V$, let $\sigma_{xy}$ denote the number of shortest paths from $x$ to $y$ in $G$, and let $\sigma_{xy}(v)$ denote the number of shortest paths from $x$ to $y$ in $G$ that pass through $v$. Then, $BC(v) = \sum_{x \neq v, t \neq v} \frac{\sigma_{xt}(v)}{\sigma_{xt}}$. This measure is often used as an index that determines the relative importance of $v$ in the network. Some applications of BC include analyzing social interaction networks [7], identifying lethality in biological networks [11], and identifying key actors in terrorist networks [24]. Heuristics for dynamic betweenness centrality with good experimental performance are given in [5,9,13], but none of these algorithms provably improve on the widely used static algorithm by Brandes [1], which runs in $O(mn + n^2 \cdot \log n)$ time on any class of graphs, where $m = |E|$.

Recently, the authors gave a simple incremental BC algorithm [10], that provably improves on Brandes’ on sparse graphs, and also typically improves on Brandes’ in dense graphs (e.g., in the setting of Theorem 2 below). In this paper, we complement the results in [10]; however, decremental updates are considerably more challenging (similar to APSP, as noted in [3]).

The key step in the recent incremental BC algorithm [10] is the incremental maintenance of the APASP dags (achieved there using techniques unrelated to the current paper). After the updated dags are obtained, the BC scores can be computed in time linear in the combined sizes of the APASP dags (plus $O(n^3)$). Thus, if we instead use our decremental APASP algorithm in the key step in [10], we obtain a decremental algorithm for BC with the same bound as APSP.

Our Results. Let $n^*$ be the maximum number of edges that lie on shortest paths through any given vertex in $G$; thus, $n^*$ also bounds the number of edges that lie on any single-source shortest path dag. Let $m^*$ be the number of edges in $G$ that lie on shortest paths (see, e.g., Karger et al. [6]). Our main result is the following theorem, where we have assumed that $n^* = \Omega(n)$.

**Theorem 1.** Let $\Sigma$ be a sequence of decremental updates on $G = (V, E)$. Then, all SP dags, all LSPs, and all BC scores can be maintained in amortized time $O(n^*^2 \cdot \log n)$ per update when $|\Sigma| = \Omega(m^*/n^*)$.

**Discussion of the Parameters.** As noted in [6], it is well-known that $m^* = O(n \log n)$ with high probability in a complete graph where edge weights are chosen from a large class of probability distributions. Since $n^* \leq m^*$, our algorithms will have an amortized bound of $O(n^2 \cdot \log^3 n)$ on such graphs. Also, $n^* = O(n)$ in any graph with only a constant number of shortest paths between every pair of vertices, even though $m^*$ can be $\Theta(n^2)$ in the worst case even in graphs with unique shortest paths. In fact $n^* = O(n)$ in some graphs that have an exponential number of shortest paths between some pairs of vertices. In all such cases, and more generally, when the number of edges on shortest paths through any single vertex is $O(n)$, our algorithm will run in amortized $O(n^2 \cdot \log n)$ time per decremental update. Thus we have:

**Theorem 2.** Let $\Sigma$ be a sequence of decremental updates on graphs where the number of edges on shortest paths through any single vertex is $O(n)$. Then, all SP dags, all LSPs, and all BC scores can be maintained in amortized time $O(n^2 \cdot \log n)$ per update when $|\Sigma| = \Omega(m^*/n)$.

**Corollary 1.** If the number of shortest paths for any vertex pair is bounded by a constant, then decremental APASP, LSPs, and BC have amortized cost $O(n^2 \cdot \log n)$ per update when the update sequence has length $\Omega(m^*/n)$. 

2
Fig. 1. Graph $G$

The DI method. Here we will use an example to give a quick review of the DI approach [3], which forms the basis for our method. Consider the graph $G$ in Fig. 1 where all edges have weight 1 except for the ones with explicit weights.

As in DI, let us assume here that $G$ has been pre-processed to identify a unique shortest path between every pair of vertices. In $G$ the shortest path from $a_1$ to $b_1$ is $\langle a_1, v, b_1 \rangle$ and has weight 2, and by definition, the paths $p_1 = \langle a_1, b_1 \rangle$ and $p_2 = \langle a_1, v_1, b_1 \rangle$ of weight 4 are both LSPs. Now consider a decremental update on $v$ that increases $w(a_1, v)$ to 10 and $w(a_2, v)$ to 5, and let $G'$ be the resulting graph (see Fig. 3). In $G'$ both $p_1$ and $p_2$ become shortest paths. Furthermore, a left extension of the path $p_1$, namely $p_3 = \langle x, a_1, b_1 \rangle$ becomes a shortest path from $x$ to $b_1$ in $G'$. Note that the path $p_3$ is not even an LSP in the graph $G$; however, it is obtained as a left extension of a path that has become shortest after the update.

The elegant method of storing LSPs and creating longer LSPs by left and right extending shortest paths is the basis of the DI approach [3]. To achieve this, the DI approach uses a succinct representation of SPs, LSPs and their left and right extensions using suitable data structures. It then uses a procedure cleanup to remove from the data structures all the shortest paths and LSPs that contain the updated vertex $v$, and a complementary procedure fixup that first adds all the trivial LSPs (corresponding to edges incident on $v$), and then restores the shortest paths and LSPs between all pairs of vertices. The DI approach thus efficiently maintains a single shortest path between all pairs of vertices under decremental updates.

Roadmap. In this paper we are interested in maintaining all shortest paths for all vertex pairs and this requires several enhancements to the DI method [3]. In Section 2 we present a new tuple system which succinctly represents all LSPs in a graph with multiple shortest paths and in Section 3 we present our incremental algorithm for maintaining this tuple system, and hence for maintaining APASP and BC scores.

2 A System of Tuples

In this section we present an efficient representation of the set of SPs and LSPs for an edge weighted graph $G = (V, E)$. We first define the notions of tuple and triple.

**Tuple.** A tuple, $\tau = (xa, by)$, represents the set of LSPs in $G$, all of which use the same first edge $(x, a)$ and the same last edge $(b, y)$. The weight of every path represented by $\tau$ is $w(x, a) + d(a, b) + w(b, y)$. We call $\tau$ a locally shortest path tuple (LST). In addition, if $d(x, y) = w(x, a) + d(a, b) + w(b, y)$, then $\tau$ is a shortest path tuple (ST). Fig. 5(a) shows a tuple $\tau$.

**Triple.** A triple $\gamma = (\tau, wt, count)$, represents the tuple $\tau = (xa, by)$ that contains $count > 0$ number of paths from $x$ to $y$, each with weight $wt$. In Fig. 1 the triple $(xa_2, by), 4, 2)$ represents two paths from $x$ to $y$, namely $p_1 = (x, a_2, v, b, y)$ and $p_2 = (x, a_2, v_2, b, y)$ both having weight 4.

| Set                  | $G$ (before update on $v$)                      |
|----------------------|------------------------------------------------|
| $P(x, y)$            | $\{(xa_1, by), (xa_2, by), (xa_3, by)\}$       |
| $P(x, b_1)$          | $\{(xa_1, vb_1), (xa_2, vb_1)\}$              |
| $P^*(x, b_1)$        | $\{(xa_1, vb_1), (xa_2, vb_1)\}$              |
| $L^*(v, y_1)$        | $\{a_1, a_2\}$                                |
| $L(v, b_1y_1)$       | $\{a_1, a_2\}$                                |
| $R^*(x, v)$          | $\{b, b_1\}$                                  |
| $R(xa_2, v)$         | $\{b, b_1\}$                                  |

Fig. 2. A subset of the tuple-system for $G$ in Fig. 1
Storing Locally Shortest Paths. We use triples to succinctly store all LSPs and SPs for each vertex pair in $G$. For $x, y \in V$, we define:

$$P(x, y) = \{(xa, by, wt, count) : (xa, by) \text{ is an LST from } x \text{ to } y \text{ in } G\}$$

$$P^*(x, y) = \{(xa, by, wt, count) : (xa, by) \text{ is an ST from } x \text{ to } y \text{ in } G\}.$$ 

Note that all triples in $P^*(x, y)$ have the same weight. We will use the term LST to denote either a locally shortest tuple or a triple representing a set of LSPs, and it will be clear from the context whether we mean a triple or a tuple.

![Graph G’](image)

**Fig. 3.** Graph $G’$

![Set G’ (with w(a1, v) = 10, w(a2, v) = 5)](image)

| Set | $G’$ (with $w(a_1, v) = 10, w(a_2, v) = 5$) |
|-----|------------------------------------------|
| $P(x, y)$ | $\{(xa_2, by), (xa_3, by), (4, 1)\}$ |
| $P^*(x, y)$ | $\{(xa_2, by), (xa_3, by), (4, 1)\}$ |
| $P(x, b_1)$ | $\{(xa_1, v_1, b_1), (xa_2, v_1, b_1), (7, 1), ((xa_1, a_1, b_1), 5, 1)\}$ |
| $P^*(x, b_1)$ | $\{(xa_1, v_1, b_1), (xa_3, a_1, b_1), (5, 1)\}$ |
| $L^*(v, y_1)$ | $\{a_2\}$ |
| $L(v, b_1y_1)$ | $\{a_2\}$ |
| $R^*(x, v)$ | $\emptyset$ |
| $R(xa_2, v)$ | $\{b_1\}$ |

**Fig. 4.** A subset of the tuple-system for $G’$

Left Tuple and Right Tuple. A left tuple (or $\ell$-tuple), $\tau_\ell = (xa, by)$, represents the set of LSPs from $x$ to $y$, all of which use the same first edge $(x, a)$. The weight of every path represented by $\tau_\ell$ is $w(x, a) + d(a, y)$. If $d(x, y) = w(x, a) + d(a, y)$, then $\tau_\ell$ represents the set of shortest paths from $x$ to $y$, all of which use the first edge $(x, a)$. A right tuple ($r$-tuple) $\tau_r = (x, by)$ is defined analogously. Fig. 5(b) and Fig. 5(c) show a left tuple and a right tuple respectively. In the following, we will say that a tuple (or $\ell$-tuple or $r$-tuple) contains a vertex $v$, if at least one of the paths represented by the tuple contains $v$. For instance, in Fig. 5 the tuple $(xa_2, by)$ contains the vertex $v$ as well as the vertex $v_2$.

![Tuple τ = (xa, by)](image)

(a) tuple $\tau = (xa, by)$

![Ell-tuple τ_ell = (xa, y)](image)

(b) $\ell$-tuple $\tau_\ell = (xa, y)$

![R-tuple τ_r = (x, by)](image)

(c) $r$-tuple $\tau_r = (x, by)$

**Fig. 5.** Tuples

ST and LST Extensions. For a shortest path $r$-tuple $\tau_r = (x, by)$, we define $L(\tau_r)$ to be the set of vertices which can be used as pre-extensions to create LSTs in $G$. Similarly, for a shortest path $\ell$-tuple $\tau_\ell = (xa, y)$, $R(\tau_\ell)$ is the set of vertices which can be used as post-extensions to create LSTs in $G$. We do not define $R(\tau_r)$.
and \( L(\tau) \). So we have:
\[
L(x, by) = \{x': (x', x) \in E(G) \text{ and } (x'x, by) \text{ is an LST in } G\}
\]
\[
R(xa, y) = \{y': (y, y') \in E(G) \text{ and } (xa, yy') \text{ is an LST in } G\}.
\]

For \( x, y \in V \), \( L^*(x, y) \) denotes the set of vertices which can be used as pre-extensions to create shortest path tuples in \( G \); \( R^*(x, y) \) is defined symmetrically:
\[
L^*(x, y) = \{x': (x', x) \in E(G) \text{ and } (x'x, y) \text{ is a } \ell\text{-tuple representing SPs in } G\}
\]
\[
R^*(x, y) = \{y': (y, y') \in E(G) \text{ and } (x, yy') \text{ is an } r\text{-tuple representing SPs in } G\}.
\]

Fig. 2 shows a subset of these sets for the graph \( G \) in Fig. 1.

**Key Deviations from DI [3].** The assumption of unique shortest paths in [3] ensures that \( \tau = (xa, by) \), \( \tau_\ell = (xa, y) \), and \( \tau_r = (x, by) \) all represent exactly the same (single) locally shortest path. However, in our case, the set of paths represented by \( \tau \) and \( \tau_\ell \) and \( \tau_r \) can be different, and \( \tau \) is a subset of paths represented by \( \tau_\ell \) and \( \tau_r \). Our definitions of ST and LST extensions are derived from the analogous definitions in [3] for SP and LSP extensions of paths. For a path \( \pi = x \rightarrow a \leadsto b \rightarrow y \), DI defines sets \( L, L^*, R \) and \( R^* \). In our case, the analog of a path \( \pi = x \rightarrow a \leadsto b \rightarrow y \) is a tuple \( \tau = (xa, by) \), but to obtain efficiency, we define the set \( L \) only for an \( r\)-tuple and the set \( R \) only for an \( \ell\)-tuple. Furthermore, we define \( L^* \) and \( R^* \) for each pair of vertices.

In the following two lemmas we bound the total number of tuples in the graph and the total number of tuples that contain a given vertex \( v \). These bounds also apply to the number of triples since there is exactly one triple for each tuple in our tuple system.

**Lemma 1.** The number of LSTs in \( G = (V, E) \) is bounded by \( O(m^* \cdot \nu^*) \).

**Proof.** For any LST \((xa, xx)\), for some \( a \in V \), the first and last edge of any such tuple must lie on a shortest path containing \( a \). Let \( E^*_a \) denote the set of edges that lie on shortest paths through \( a \), and let \( I_a \) be the set of incoming edges to \( a \). Then, there are at most \( \nu^* \) ways of choosing the last edge in \((xa, xx)\) and at most \( E^*_a \cap I_a \) ways of choosing the first edge in \((xa, xx)\). Since \( \sum_{a \in V} |E^*_a \cap I_a| = m^* \), the number of LSTs in \( G \) is at most \( \sum_{a \in V} \nu^* \cdot |E^*_a \cap I_a| \leq m^* \cdot \nu^* \). \( \square \)

**Lemma 2.** The number of LSTs that contain a vertex \( v \) is \( O(\nu^*^2) \).

**Proof.** We distinguish three different cases:

1. Tuples starting with \( v \): for a tuple that starts with edge \((v, a)\), the last edge must lie on \( a \)'s SP dag, so there are at most \( \nu^* \) choices for the last edge. Hence, the number of tuples with \( v \) as start vertex is at most \( \sum_{a \in V} \nu^* \leq n \cdot \nu^* \).
2. Similarly, the number of tuples with \( v \) as end vertex is at most \( n \cdot \nu^* \).
3. For any tuple \( \tau = (xa, by) \) that contains \( v \) as an internal vertex, both \( (x, a) \) and \( (b, y) \) lie on a shortest path through \( v \), hence the number of such tuples is at most \( \nu^*^2 \). \( \square \)

### 3 Decremental Algorithm

Here we present our decremental APASP algorithm. Recall that a decremental update on a vertex \( v \) either deletes or increases the weights of a subset of edges incident on \( v \). We begin with the data structures we use.

**Data Structures.** For every \( x, y, x \neq y \) in \( V \), we maintain the following:

1. \( P(x, y) \) – a priority queue containing LSTs from \( x \) to \( y \) with weight as key.
2. \( P^*(x, y) \) – a priority queue containing STs from \( x \) to \( y \) with weight as key.
3. \( L^*(x, y) \) – a balanced search tree containing vertices with vertex ID as key.
4. $R^\ast(x, y)$ — a balanced search tree containing vertices with vertex ID as key.

For every $\ell$-tuple we have its right extension, and for every $r$-tuple its left extension. These sets are stored as balanced search trees (BSTs) with the vertex ID as a key. Additionally, we maintain all tuples in a BST $dict$, with a tuple $\tau = (xa, by)$ having key $[x, y, a, b]$. We also maintain pointers from $\tau$ to $R(xa, y)$ and $L(x, by)$, and to the corresponding triple containing $\tau$ in $P(x, y)$, (and in $P^\ast(x, y)$ if $(xa, by)$ is an ST). Finally, we maintain a sub-dictionary of $dict$ called Marked-Tuples (explained below). Marked-Tuples, unlike the other data structures, is specific only to one update.

The Decremental Algorithm. Given the updated vertex $v$ and the updated weight function $w'$ over all the incoming and outgoing edges of $v$, the decremental algorithm performs two main steps cleanup and fixup, as in DI. The cleanup procedure removes from the tuple system every LSP that contains the updated vertex $v$. The following definition of a new LSP is from DI.

Definition 1. A path that is shortest (locally shortest) after an update to vertex $v$ is new if either it was not an SP (LSP) before the update, or it contains $v$.

The fixup procedure adds to the tuple system all the new shortest and locally shortest paths. In contrast to DI, recall that we store locally shortest paths in $P$ and $P^\ast$ as triples. Hence removing or adding paths implies decrementing or incrementing the count in the relevant triple; thus a triple is removed or added only if its count goes down to zero or up from zero. Moreover, new tuples may be created through combining several existing tuples. Some of the updated data structures for the graph $G'$ in Fig. obtained after a decremental update on $v$ in the graph $G$ in Fig. are schematized in Fig. 2.

3.1 The Cleanup Procedure

Algorithm (cleanup) uses an initially empty heap $H_c$ of triples. It also initializes the empty dictionary Marked-Tuples. The algorithm then creates the trivial triple corresponding to the vertex $v$ and adds it to $H_c$ (Step 2 Algorithm 1). For a triple $((xa, by), wt, count)$ the key in $H_c$ is $[wt, x, y]$. The algorithm repeatedly extracts min-key triples from $H_c$ (Step 4 Algorithm 1) and processes them. The processing of triples involves left-extending (Steps 5–11 Algorithm 1) and right-extending triples (Step 12 Algorithm 1) and removing from the tuple system the set of LSPs thus formed. This is similar to cleanup in DI. However, since we deal with a set of paths instead of a single path, we need significant modifications, of which we now highlight two: (i) Accumulation used in Step 4 and (ii) use of Marked-Tuples in Step 7 and Step 11.

Algorithm 1 cleanup($v$)

1: $H_c \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: $\gamma \leftarrow ((v, v), 0, 1)$; add $\gamma$ to $H_c$
3: while $H_c \neq \emptyset$ do
4: extract in $S$ all the triples with min-key $[wt, x, y]$ from $H_c$
5: for every $b$ such that $(xa, by) \in S$ do
6: let $fcount = \sum c_t$ such that $((xa, by), wt, c_t) \in S$
7: for every $x' \in L(x, by)$ such that $(x', x, by) \notin$ Marked-Tuples do
8: $wt' \leftarrow wt + w(x', x)$; $\gamma' \leftarrow ((x', x), wt', fcount')$; add $\gamma'$ to $H_c$
9: remove $\gamma'$ in $P(x', y)$ // decrements count by $fcount$
10: if a triple for $(x', x, by)$ exists in $P(x', y)$ then
11: insert $(x', x, by)$ in Marked-Tuples
12: else
13: delete $x'$ from $L(x, by)$ and delete $y$ from $R(x', x, b)$
14: if a triple for $(x', x, by)$ exists in $P^\ast(x', y)$ then
15: remove $\gamma'$ in $P^\ast(x', y)$ // decrements count by $fcount$
16: if $P^\ast(x, y) = \emptyset$ then delete $x'$ from $L^\ast(x, y)$
17: if $P^\ast(x', b) = \emptyset$ then delete $y$ from $R^\ast(x', b)$
18: perform symmetric step 5–11 for right extensions

Accumulation. In Step 4 we extract a collection $S$ of triples all with key $[wt, x, y]$ from $H_c$ and process them together in that iteration of the while loop. Assume that for a fixed last edge $(b, y)$, $S$ contains triples of the form $(xa, by)$, for $t = 1, \ldots, k$. Our algorithm processes and left-extends all these triples with the
same last edge together. This ensures that, for any \( x' \in L(x, by) \), we generate the triple \((x'x, by)\) exactly once. The accumulation is correct because any valid left extension for a triple \((xa_i, by)\) is also a valid left extension for \((xa_j, by)\) when both triples have the same weight.

**Marked-Tuples.** The dictionary of Marked-Tuples is used to ensure that every path through the vertex \( v \) is removed from the tuple system exactly once and therefore counts of paths in triples are correctly maintained. Note that a path of the form \((xa, by)\) can be generated either as a left extension of \((a, by)\) or by a right extension of \((xa, b)\). This is true in DI as well. However, due to the assumption of unique shortest paths they do not need to maintain counts of paths, and hence do not require the book-keeping using Marked-Tuples.

See sections A.1 and A.2 in the Appendix for more details.

### 3.1.1 Correctness and Complexity

We establish the correctness of cleanup in Lemma 3 and an upper bound on its worst case time in Lemma 4.

**Lemma 3.** After Algorithm 7 is executed, the counts of triples in \( P \) (\( P^* \)) represent counts of LSPs (SPs) in \( G \) that do not pass through \( v \). Moreover, the sets \( L, L^*, R, R^* \) are correctly maintained.

**Proof.** To prove the lemma statement we show that the while loop in Step 5 of Algorithm 1 maintains the following invariants.

**Loop Invariant:** At the start of each iteration of the while loop in Step 5 of Algorithm 1 assume that the min-key triple to be extracted and processed from \( H_e \) has key \([wt, x, y]\). Then the following properties hold about the tuple system and \( H_e \). We assert the invariants about the sets \( P, L, R \). Similar arguments can be used to establish the correctness of the sets \( P^*, L^*, \) and \( R^* \).

\( I_1 \) For any \( a, b \in V \), if \( G \) contains \( c_{ab} \) number of locally shortest paths of weight \( wt \) of the form \((xa, by)\) passing through \( v \), then \( H_e \) contains a triple \( \gamma = ((xa, by), wt, c_{ab}) \). Further, \( c_{ab} \) has been decremented from the initial count in the triple for \((xa, by)\) in \( P(x, y) \).

\( I_2 \) Let \([\hat{wt}, \hat{x}, \hat{y}]\) be the key extracted from \( H_e \) and processed in the previous iteration. For any key \([wt_1, x_1, y_1] \leq \hat{wt}, \hat{x}, \hat{y}]\), let \( G \) contain \( c > 0 \) number of LSPs of weight \( wt \) of the form \((x_1a_1, b_1y_1)\). Further, let \( c_v \) (resp. \( c_\hat{v} \)) denote the number of such LSPs that pass through \( v \) (resp. do not pass through \( v \)). Here \( c_v + c_\hat{v} = c \).

Then, (a) if \( c > c_v \) there is a triple in \( P(x_1, y_1) \) of the form \((x_1a_1, b_1y_1)\) and weight \( wt \) representing \( c - c_v \) LSPs. If \( c = c_v \) there is no such triple in \( P(x_1, y_1) \).
(b) \( x_1 \in L(a_1, b_1y_1) \), \( y_1 \in R(x_1a_1, b_1) \), and \((x_1a_1, b_1y_1) \in \) Marked-Tuples iff \( c_v > 0 \).
(c) For every \( x' \in L(x_1, b_1y_1) \), a triple corresponding to \((x'x_1, b_1y_1)\) with weight \( wt' = wt + w(x', x_1) \) and the appropriate count is in \( H_e \) if \([wt', x', y_1] \geq [wt, x, y]\). A similar claim can be stated for every \( y' \in R(x_1a_1, y_1) \).

\( I_3 \) For any key \([wt_2, x_2, y_2] \geq [wt, x, y]\), let \( G \) contain \( c > 0 \) LSPs of weight \( wt_2 \) of the form \((x_2a_2, b_2y_2)\). Further, let \( c_v \) (resp. \( c_\hat{v} \)) denote the number of such LSPs that pass through \( v \) (resp. do not pass through \( v \)). Here \( c_v + c_\hat{v} = c \). Then the tuple \((x_2a_2, b_2y_2) \in \) Marked-Tuples, iff \( c_v > 0 \) and a triple for \((x_2a_2, b_2y_2)\) representing \( c_v \) LSPs is present in \( H_e \).

See section A.3 in the Appendix for the details of the proof.

**Lemma 4.** For an update on a vertex \( v \), Algorithm 7 takes \( O(v^2 \cdot \log n) \) time.

**Proof.** The cleanup procedure examines a triple \( \gamma \) only if the tuple in \( \gamma \) contains the updated vertex \( v \). It removes each such \( \gamma \) from a constant number of data structures \( (P, P^*, L, L^*, R, R^*) \), each with an \( O(\log n) \) cost. In addition, each triple is inserted into \( H_e \) and extracted from it exactly once. Since the number of tuples containing \( v \) is bounded by Lemma 2, the lemma follows.
3.2 The Fixup Procedure

The goal of the fixup procedure is to add to the tuple system all new shortest and locally shortest paths (recall Definition 1).

The fixup procedure (pseudo-code in Algorithm 2) works with a heap of triples \((H_f)\) here, which is initialized with a candidate shortest path triple for each pair of vertices. Recall that for a pair \(x, y\), there may be several triples of a given weight \(wt\) in \(P(x, y)\). Instead of inserting all min-weight triples (which are candidates for shortest path triples), our algorithm inserts exactly one triple for every pair of vertices into \(H_f\). This ensures that the number of triples examined during fixup is not too large. Once \(H_f\) is suitably initialized, the fixup algorithm repeatedly extracts the set of triples with minimum key and processes them. The main invariant for the algorithm (similar to DI [3]) is that for a pair \(x, y\), the weight of the first set of triples extracted from \(H_f\) gives the distance from \(x\) to \(y\) in the updated graph. Thus, these triples are all identified as shortest path triples, and we need to extend them if in fact they represent new shortest paths.

To readily identify triples containing paths through \(v\) we use some additional book-keeping; for every triple \(\gamma\) we store the update number \((\text{update-num}(\gamma))\) and a count of the number of paths in that triple that pass through \(v\) \((\text{paths}(\gamma, v))\). Finally, similar to cleanup, the fixup procedure also left and right extends triples to create triples representing new locally shortest paths.

We now describe the steps of the algorithm.

Algorithm 2 initializes \(H_f\) in Steps [2, 3] as follows. (i) For every edge incident on \(v\), it creates a trivial triple \(\gamma\) which is inserted into \(H_f\) and \(P\). It also sets \(\text{update-num}(\gamma)\) and \(\text{paths}(\gamma, v)\) for each such \(\gamma\); (ii) For every \(x, y \in V\), it adds a candidate min-weight triple from \(P(x, y)\) to \(H_f\) (even if \(P(x, y)\) contains several min-weight triples; this is done for efficiency).

Algorithm 2 executes Steps [10–17] when for a pair \(x, y\), the first set of triples \(S^*\), all of weight \(wt\), are extracted from \(H_f\). We claim (Invariant 3) that \(wt\) denotes the shortest path distance from \(x\) to \(y\) in the updated graph. The goal of Steps [10–17] is to create a set \(S\) of triples that represent new shortest paths, and this step is considerably more involved than the corresponding step in DI. In DI [3], only a single path \(p\) is extracted from \(H_f\) possibly resulting in a new shortest path from \(x\) to \(y\). If \(p\) is new then it is added to \(P^*\) and the algorithm extends it to create new LSP. In our case, we extract not just multiple paths but multiple shortest path triples from \(x\) to \(y\), and some of these triples may not be in \(H_f\). We now describe how the algorithm generates the new shortest paths in Steps [10–17].

Steps [10–17] Algorithm 2 – As mentioned above, Steps [10–17] create a set \(S\) of triples that represent new shortest paths. There are two cases.

- \(P^*(x, y)\) is empty: Here, we process the triples in \(S^*\), but in addition, we may be required to process triples of weight \(wt\) from the set \(P(x, y)\). To see this, consider the example in Fig. 1 and consider the pair \(a_1, b_1\). In \(G\), there is one shortest path \((a_1, v, b_1)\) which is removed from \(P(a_1, b_1)\) and \(P^*(a_1, b_1)\) during cleanup. In \(G'\), \(d(a_1, b_1) = 4\) and there are 2 shortest paths, namely \(p_1 = (a_1, b_1)\) and \(p_2 = (a_1, v_1, b_1)\). Note that both of these are LSPs in \(G\) and therefore are present in \(P(a_1, b_1)\). In Step 5, Algorithm 2 we insert exactly one of them into the heap \(H_f\). However, both need to be processed and also left and right extended to create new locally shortest paths. Thus, under this condition, we examine all the min-weight triples present in \(P(a_1, b_1)\).

- \(P^*(x, y)\) is non-empty: After a decremental update, the distance from \(x\) to \(y\) can either remain the same or increase, but it cannot decrease. Further, cleanup removed from the tuple system all paths that contain \(v\). Hence, if \(P^*(x, y)\) is non-empty at this point, it implies that all paths in \(P^*(x, y)\) avoid \(v\). In this case, we can show (Invariant 4) that it suffices to only examine the triples present in \(H_f\). Furthermore, the only paths that we need to process are the paths that pass through the vertex \(v\).

Steps [19–29] Algorithm 2 – These steps left-extend and right-extend the triples in \(S\) representing new shortest paths from \(x\) to \(y\).
Fixup maintains the following two invariants. The invariant below (Invariant 3) shows that for any pair \( x, y \), the weight of the first set of the triples extracted from \( H_f \) determines the shortest path distance from \( x \) to \( y \). The proof of the invariant is similar to the proof of Invariant 3.1 in [3].

**Invariant 3** If the set \( S' \) in Step 4 of Algorithm 2 is the first extracted set from \( H_f \) for \( x, y \), then the weight of each triple in \( S' \) is the shortest path distance from \( x \) to \( y \) in the updated graph.

**Proof.** Assume for the sake of contradiction that the invariant is violated at some extraction. Thus, the first set of triples \( S' \) of weight \( \hat{w} \) extracted for some pair \((x, y)\) do not represent the set of shortest paths from \( x \) to \( y \) in the updated graph. Consider the earliest of these events and let \( \gamma = ((x'a', b'y), \hat{w}, \text{count}'') \) be a triple in the updated graph that represents a set of shortest paths from \( x \) to \( y \) with \( \hat{w} < \hat{w}' \). The triple \( \gamma \) cannot be present in \( H_f \), else it would have been extracted before any triple of weight \( \hat{w} \) from \( H_f \). Moreover, \( \gamma \) cannot be in \( P(x, y) \) at the beginning of fixup otherwise \( \gamma \) (or some other triple of weight \( w_t \)) would have been inserted into \( H_f \) during Step 5 of Algorithm 2. Thus \( \gamma \) must be a new LST generated by the algorithm. Since all edges incident on \( v \) are added to \( H_f \) during Step 2 of Algorithm 2 and \( \gamma \) is not present in \( H_f \), implies that \( \gamma \) represents paths which have at least two or more edges. We now define \( \text{left}(\gamma) \) as the set of LSTs of the form \(((x, c, b), \hat{w} - w(b, y), \text{count}_i)\) that represent all the LSPs in the left tuple \((x, b);\) similarly we define \( \text{right}(\gamma) \) as the set of LSTs of the form \(((a, b, d), \hat{w} - w(x, a), \text{count}_j)\) that represent all the LSPs in the right tuple \((a, b).\) Note that since \( \gamma \) is a shortest path tuple, all the paths represented by LSTs in \text{left}(\gamma) \) and \text{right}(\gamma) \) are also shortest paths. All of the paths in either \text{left}(\gamma) \) or in \text{right}(\gamma) \) are new shortest paths and therefore are not present in \( P^* \) at the beginning of fixup. Since edge weights are positive \((\hat{w} - w(b, y)) < \hat{w} < \hat{w}' \) and \((\hat{w} - w(x, a)) < \hat{w} < \hat{w}' \). As we extract paths from \( H_f \) in increasing order of weight, and all extractions before the wrong extraction were correct, the triples in \text{left}(\gamma) \) and \text{right}(\gamma) \) should have been extracted from \( H_f \) and added to \( P^* \). Thus, the triple corresponding to \((x, a, by)\) of weight \( \hat{w} \) should have been generated during left or right extension and inserted in \( H_f \). Hence, some triple of weight \( \hat{w} \) must be extracted from \( H_f \) for the pair \((x, y)\) before any triple of weight \( \hat{w} \) is extracted from \( H_f \). This contradicts our assumption that the invariant is violated. \( \square \)
Lemma 5. Any new SP from \(x, y\). Recall that all the new shortest paths for a pair need not be present in \(H_f\) and we may be required to consider min-weight triples present in \(P(\cdot)\) as well.

**Invariant 4** The set \(S\) of triples constructed in Steps 10–17 of Algorithm 2 represents all of the new shortest paths from \(x\) to \(y\).

**Proof.** Any new SP from \(x\) to \(y\) is of the following three types:

1. a single edge containing the vertex \(v\) (such a path is added to \(P(x, y)\) and \(H_f\) in Step 2)
2. a path generated via left/right extension of some shortest path (such a path is added to \(P(x, y)\) and \(H_f\) in Step 24 and an analogous step in right-extend).
3. a path that was an LSP but not SP before the update and is an SP after the update.

In (1) and (2) above any new SP from \(x\) to \(y\) which is added to \(H_f\) is also added to \(P(x, y)\). However, amongst the several triples representing paths of the form (3) listed above, only one candidate triple will be present in \(H_f\). Thus we conclude that for a given \(x, y\) and when we extract from \(H_f\) triples of weight \(wt\), \(P(x, y)\) contains a superset of the triples that are present in \(H_f\). We now consider the two cases that the algorithm deals with.

- \(P^*(x, y)\) is empty when the first set of triples for \(x, y\) is extracted from \(H_f\). In this case, we process all the min-weight triples in \(P(x, y)\). By the above argument, we know that all new SPs from \(x\) to \(y\) are present in \(P(x, y)\). Therefore it suffices to argue that all of them are new. Assume for the sake of contradiction, some path \(p\) represented by them is not new. By definition, \(p\) does not contain \(v\) and \(p\) was a SP before the update. Therefore, clearly \(p\) was in \(P^*(x, y)\) before the update. However, since cleanup only removes paths that contain \(v\), the path \(p\) remains untouched during cleanup and hence continues to exist in \(P^*(x, y)\). This contradicts the fact that \(P^*(x, y)\) is empty.
- \(P^*(x, y)\) is not empty when the first set of triples for \(x, y\) is extracted from \(H_f\). Let the weight of triples in \(P^*(x, y)\) be \(wt\). This implies that the shortest path distance from \(x\) to \(y\) before and after the update is \(wt\). Recall that we are dealing with decremental updates. We first argue that it suffices to consider triples in \(H_f\). This is observed from the fact that any new SP of the form (1) and (2) listed above is present in \(H_f\). Furthermore, note that any path of form (3) above has a weight strictly larger than \(wt\) since it was an LSP and not SP before the update. Thus in the presence of paths of weight \(wt\), none of the paths of form (3) are candidates for shortest paths from \(x\) to \(y\). This justifies considering triples only in \(H_f\).

Finally, we note that for any triple considered, our algorithm only processes paths through \(v\). This again follows from the fact that only paths through \(v\) were removed by cleanup and possibly need to be restored if the distance via them remains unchanged after the update.

\[\square\]

The following lemma establishes the correctness of fixup.

**Lemma 5.** After execution of Algorithm 2 for any \((x, y) \in V\), the counts of the triples in \(P(x, y)\) and \(P^*(x, y)\) represent the counts of LSPs and SPs from \(x\) to \(y\) in the updated graph. Moreover, the sets \(L, L^*, R, R^*\) are correctly maintained.

**Proof.** We prove the lemma statement by showing the invariants are maintained by the while loop in Step 6 of Algorithm 2.

**Loop Invariant:** At the start of each iteration of the while loop in Step 6 of Algorithm 2 let the min-key triple to be extracted and processed from \(H_f\) have key = \(|wt, x, y|\). We claim the following about the tuple system and \(H_f\).

\(\mathcal{I}_1\) For any \(a, b \in V\), if \(G'\) contains \(c_{ab}\) number of LSPs of weight \(wt\) of the form \((xa, by)\). Further, a triple \(\gamma = ((xa, by), wt, c_{ab})\) is present in \(P(x, y)\) (note that \(H_f\) can also contain other triples from \(x\) to \(y\) with weight \(wt\)).
Let \( [\hat{w}, \hat{x}, \hat{y}] \) be the last key extracted from \( H_f \) and processed before \( [w, x, y] \). For any key \( [w_1, x_1, y_1] \leq [w, x, y] \), let \( G' \) contain \( c > 0 \) number of SPs of weight \( w_1 \) of the form \( (x_1a_1, b_1y_1) \). Further, let \( c_{\text{new}} \) (resp. \( c_{\text{old}} \)) denote the number of such SPs that are new (resp. not new). Here \( c_{\text{new}} + c_{\text{old}} = c \). Then, (a) the triple for \( (x_1a_1, b_1y_1) \) with weight \( w_1 \) in \( P^*(x_1, y_1) \) represents \( c \) SPs. (b) \( x_1 \in L(a_1, b_1y_1) \), \( x_1 \in L^*(a_1, y_1) \), and \( y_1 \in R(x_1a_1, b_1) \), \( y_1 \in R^*(x_1, b_1) \). Further, \( (x_1a_1, b_1y_1) \) ∈ Marked-Tuples iff \( c_{\text{old}} > 0 \). (c) If \( c_{\text{new}} > 0 \), for every \( x' \in L(x_1, b_1y_1) \), a triple corresponding to \( (x'x_1, b_1y_1) \) with weight \( w_x' = w_c + w(x'x_1) \) and the appropriate count is in \( P(x_1, y_1) \) and in \( H_f \) if \( [w_c, x', y_1] \geq [w_c, x, y] \). A similar claim can be stated for every \( y' \in R(x_1a_1, y_1) \).

For any key \( [w_2, x_2, y_2] \geq [w, x, y] \), let \( G' \) contain \( c > 0 \) number of LSPs of weight \( w_2 \) of the form \( (x_2a_2, b_2y_2) \). Further, let \( c_{\text{new}} \) (resp. \( c_{\text{old}} \)) denote the number of such SPs that are new (resp. not new). Here \( c_{\text{new}} + c_{\text{old}} = c \). Then the tuple \( (x_2a_2, b_2y_2) \) ∈ Marked-Tuples, iff \( c_{\text{old}} > 0 \) and \( c_{\text{new}} \) paths have been added to \( H_f \) by some earlier iteration of the while loop.

The proof that these invariants hold at initialization and termination and are maintained at every iteration of the while loop is similar to the proof of Lemma \( 3 \).

3.2.1 Complexity of Fixup. As in DI, we observe that shortest paths and LSPs are removed only in cleanup and are added only in fixup. In a call to fixup, accessing a triple takes \( O(\log n) \) time since it is accessed on a constant number of data structures. So, it suffices to bound the number of triples accessed in a call to fixup, and then multiply that bound by \( O(\log n) \).

We will establish an amortized bound. The total number of LSTs at any time, including the end of the update sequence, is \( O(m^* \cdot \nu^*) \) (by Lemma \( 3 \)). Hence, if fixup accessed only \( new \) triples outside of the \( O(n^2) \) triples added initially to \( H_f \), the amortized cost of fixup (for a sufficiently long update sequence) would be \( O(\nu^2 \cdot \log n) \), the cost of a cleanup. This is in fact the analysis in DI, where fixup satisfies this property. However, in our algorithm fixup accesses several triples that are already in the tuple system: In Steps \( 11 \)–\( 13 \) we examine triples already in \( P \), in Steps \( 15 \)–\( 17 \) we could increment the count of an existing triple in \( P \), and in Steps \( 19 \)–\( 28 \) we increment the count of an existing triple in \( P \). We bound the costs of these steps in Lemma \( 6 \) below by classifying each triple \( \gamma \) as one of the following disjoint types:

- **Type-0** (*contains-v*): \( \gamma \) represents at least one path containing vertex \( v \).
- **Type-1** (*new-LST*): \( \gamma \) was not an LST before the update but is an LST after the update, and no path in \( \gamma \) contains \( v \).
- **Type-2** (*new-ST-old-LST*): \( \gamma \) is an ST after the update, and \( \gamma \) was an LST but not an ST before the update, and no path in \( \gamma \) contains \( v \).
- **Type-3** (*new-ST-old-ST*): \( \gamma \) was an ST before the update and continues to be an ST after the update, and no path in \( \gamma \) contains \( v \).
- **Type-4** (*new-LST-old-LST*): \( \gamma \) was an LST before the update and continues to be an LST after the update, and no path in \( \gamma \) contains \( v \).

The following lemma establishes an amortized bound for fixup which is the same as the worst case bound for cleanup.

**Lemma 6.** The fixup procedure takes time \( O(\nu^2 \cdot \log n) \) amortized over a sequence of \( \Omega(m^*/\nu^*) \) decremental-only updates.

**Proof.** We bound the number of triples examined; the time taken is \( O(\log n) \) times the number of triples examined due to the data structure operations performed on a triple. The initialization in Steps \( 1 \)–\( 5 \) takes \( O(n^2) \) time. We now consider the triples examined after Step \( 5 \). The number of Type-0 triples is \( O(\nu^2) \) by Lemma \( 2 \). The number of Type-1 triples is addressed by amortizing over the entire update sequence as described in the paragraph below. For Type-2 triples we observe that since updates only increase the weights on edges, a shortest path never reverts to being an LSP. Further, each such Type-2 triple is examined only
a constant number of times (in Steps 10–13). Hence we charge each access to a Type-2 triple to the step in which it was created as a Type-1 triple. For Type-3 and Type-4, we note that for any \(x, y\) we add exactly one candidate min-weight triple from \(P(x, y)\) to \(H_f\), hence initially there are at most \(n^2\) such triples in \(H_f\). Moreover, we never process an old LST which is not an ST so no additional Type-4 triples are examined during fixup. Finally, triples in \(P^*\) that are not placed initially in \(H_f\) are not examined in any step of fixup, so no additional Type-3 triples are examined. Thus the number of triples examined by a call to fixup is \(O(\nu^*^2)\) plus \(O(X)\), where \(X\) is the number of new triples fixup adds to the tuple system. (This includes an \(O(1)\) credit placed on each new LST for a possible later conversion to an ST.)

Let \(\sigma\) be the number of updates in the update sequence. Since triples are removed only in cleanup, at most \(O(\sigma \cdot \nu^*^2)\) triples are removed by the cleanups. There can be at most \(O(m^* \cdot \nu^*)\) triples remaining at the end of the sequence (by Lemma 1), hence the total number of new triples added by all fixups in the update sequence is \(O(\sigma \cdot \nu^*^2 + m^* \cdot \nu^*)\). When \(\sigma > m^* / \nu^*\), the first term dominates, and this gives an average of \(O(\nu^*^2)\) triples added per fixup, and the desired amortized time bound for fixup. \(\square\)

### 3.3 Complexity of the Decremental Algorithm.

Lemma 6 establishes that the amortized cost per update of fixup is \(O(\nu^*^2 \cdot \log n)\) when the decremental update sequence is of length \(\Omega(m^* / \nu^*)\). Lemma 4 shows that the worst case cost per update of cleanup is \(O(\nu^*^2 \cdot \log n)\). Since an update operation consists of a call to cleanup followed by a call to fixup, this establishes Theorem 1.

### 4 Discussion

We have presented an efficient decremental algorithm to maintain all-pairs all shortest paths (APASP). The space used by our algorithm is \(O(m^* \cdot \nu^*)\), the worst case number of triples in our tuple system. By using this decremental APASP algorithm in place of the incremental APASP algorithm used in [10], we obtain a decremental algorithm with the same bound for maintaining BC scores.

Very recently, two of the authors have obtained a fully dynamic APASP algorithm [12] that combines elements in the fully dynamic APSP algorithms in [3] and [14], while building on the results in the current paper. When specialized to unique shortest paths (i.e., APSP), this algorithm is about as simple as the one in [3] and matches its amortized bound.

### References

1. U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
2. T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, 2004.
3. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
4. L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
5. O. Green, R. McColl, and D. A. Bader. A fast algorithm for updating betweenness centrality. In *Proc. of 4th PASSAT*, pages 11–20, 2012.
6. D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.
7. N. Kourtellis, T. Alahakoon, R. Simha, A. Iamnitchi, and R. Tripathi. Identifying high betweenness centrality nodes in large social networks. *Social Network Analysis and Mining*, pages 1–16, 2012.
8. V. Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.
9. M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. Qube: a quick algorithm for updating betweenness centrality. In *Proc. of 21st WWW*, pages 351–360, 2012.
10. M. Nasre, M. Pontecorvi, and V. Ramachandran. Betweenness centrality – Incremental and faster. In *Proc. of 39th MFCS*, pages 577–588, 2014.
11. J. W. Pinney, G. A. McConkey, and D. R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. of RECOMB*, 2005.
12. M. Pontecorvi and V. Ramachandran. Fully dynamic all pairs all shortest paths and betweenness centrality. Manuscript, 2014.
13. R. R. Singh, K. Goel, S. Iyengar, and Sukrit. A faster algorithm to update betweenness centrality after node alteration. In *Proc. of 10th WAW*, pages 170–184, 2013.
14. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. of 9th SWAT*, pages 384–396, 2004.
A Details from Section 3.1

A.1 Accumulation

Need for accumulation. In Step 5 of Algorithm 1, we consider every \( (x, y) \) such that \( (x, y) \) belongs to \( S \). We also assume that we have the accumulated count of such triples available in Step 6. An efficient method to accumulate these counts is given below. We use this accumulated count to generate a longer LST for each \( x' \in L(x, y) \) (Step 8). (For the moment, ignore the check of a tuple being present in Marked-Tuples.) Consider our example in Fig. 1 and assume that we have deleted the two paths of the form \( (x, a_1, v, b, y) \) and \( (x', x, a_2, v, b, y) \). Note that both these paths represented by triples of the form \( (x', x, y) \). We further remark that such an implementation is correct because any valid left extension of triples of the form \( (x, y) \) is also a valid left extension of triples of the form \( (x', y) \) when the triples have the same weight. Furthermore, it is efficient since it generates the triple of the form \( (x', y) \) at most once. This is the precise reason for defining the set \( L \) for an \( r \)-tuple \( (x, y) \) instead of defining it for the tuple \( (x_1, y) \).

Accumulation technique. An efficient implementation of getting accumulated counts can be achieved in several ways. For the sake of concreteness, we sketch an implementation by maintaining two arrays \( A \) and \( B \) of size \( n \) each and two linked lists \( L_a \) and \( L_b \). Assume that the arrays are initialized to zero and the linked lists are empty just before any triple with key \( [wt, x, y] \) is extracted from the heap. When a triple \( \gamma = ((x_i, y_j), wt, count_{ij}) \) is extracted from \( H_c \), we add \( count_{ij} \) to \( A[a_i] \) and \( B[b_j] \). The lists \( L_a \) and \( L_b \) maintain pointers to non-zero locations in the arrays \( A \) and \( B \) respectively. Thus, when all triples of weight \( wt \) corresponding to tuples of the form \( (x, y) \) are extracted from \( H_c \), the value in \( A[a_i] \) denotes the number of locally shortest paths of the form \( (x, y) \) to be updated. Similarly, the value in \( B[b_j] \) denotes number of locally shortest paths of the form \( (x, y) \) to be updated. Using the lists \( L_a \) and \( L_b \), we can efficiently access the accumulated counts as well as reinitialize (to zero) all the non-zero values in the two arrays \( A \) and \( B \).

A.2 Need for Marked-Tuples

Consider the example in Fig. 1 and assume that we have deleted the two paths of the form \( (x', y) \) which pass through \( v \). Furthermore, assume that we have generated them via left extending the two triples of the form \( (x_1, y) \) and \( (x_2, y) \). Now note that since path \( (x', x, a_2, v_2, b, y) \) continues to exist in the tuple system, \( x' \in L(x, y) \) and \( y \in R(x', a_2) \). Thus, when we consider the triples of the form \( (x', b, y) \) for right extension, it is possible to generate the same paths again. To avoid such a double generation we use the dictionary Marked-Tuples. In Step 7 of Algorithm 1, just before we create a left extension of a set of triples of the form \( (x, y) \) using the vertex \( x' \in L(x, y) \), we check whether \( (x', y) \) is present in Marked-Tuples. Recall that, Marked-Tuples is empty when the cleanup begins. When a triple for \( (x', y) \) is generated for the first time (either by a left extension or right extension), and there are additional locally shortest paths in \( G \) of the form \( (x', y) \) which do not pass through \( v \), we insert a tuple \( (x', y) \) in Marked-Tuples (Step 11 of Algorithm 1). Thus the data structure Marked-Tuples and the checks in Step 10 of Algorithm 1 ensure that the paths are generated exactly once either as a left extension or as a right extension but not by both. Note that such a marking is not required when there are no additional paths in \( G \) which do not pass through \( v \). In that case, we immediately delete \( x' \) from \( L(x, y) \) and \( y \) from \( R(x', b) \) (Step 13 of Algorithm 1) ensuring that a triple for \( (x', y) \) gets generated exactly once. This is the only case that can occur in DI 3 due to the assumption of unique shortest paths, and therefore this book-keeping with Marked-Tuples is not required in 3.
A.3 Analysis of Cleanup

Lemma 3. After Algorithm 1 is executed, the counts of triples in \( P^* \) represent counts of LSPs (SPs) in \( G \) that do not pass through \( v \). Moreover, the sets \( L, L^*, R, R^* \) are correctly maintained.

Proof. To prove the lemma statement we show that the while loop in Step 3 of Algorithm 1 maintains the following invariants.

Loop Invariant: At the start of each iteration of the while loop in Step 3 of Algorithm 1 assume that the min-key triple to be extracted and processed from \( H \) has key \([wt, x, y]\). Then the following properties hold about the tuple system and \( H \). We assert the invariants about the sets \( P, L, \) and \( R \). Similar arguments can be used to establish the correctness of the sets \( P^*, L^*, \) and \( R^* \).

\( I_1 \) For any \( a, b \in V \), if \( G \) contains \( c_{ab} \) number of locally shortest paths of weight \( wt \) of the form \((xa, by)\) passing through \( v \), then \( H \) contains a triple \( \gamma = ((xa, by), wt, c_{ab}) \). Further, \( c_{ab} \) has been decremented from the initial count in the triple for \((xa, by)\) in \( P(x, y) \).

\( I_2 \) Let \([wt, x, y]\) be the key extracted from \( H \) and processed in the previous iteration. For any key \([wt_1, x_1, y_1]\) \( \leq [wt, x, y] \), let \( G \) contain \( c > 0 \) number of LSPs of weight \( wt_1 \) of the form \((x_1a, b_1y_1)\). Further, let \( c_v \) (resp. \( c_0 \)) denote the number of such LSPs that pass through \( v \) (resp. do not pass through \( v \)). Here \( c_v + c_0 = c \).

(a) if \( c > c_v \) there is a triple in \( P(x_1, y_1) \) of the form \((x_1a, b_1y_1)\) and weight \( wt_1 \) representing \( c - c_v \) LSPs. If \( c = c_v \) there is no such triple in \( P(x_1, y_1) \).

(b) \( x_1 \in L(a_1, b_1y_1), y_1 \in R(x_1a_1, b_1), \) and \((x_1a_1, b_1y_1) \in \text{Marked-Tuples} \) iff \( c_0 > 0 \).

(c) For every \( x' \in L(x_1, b_1y_1) \), a triple corresponding to \((x'x_1, b_1y_1)\) with weight \( wt' = wt_1 + w(x', x_1) \) and the appropriate count is in \( H \) if \([wt', x', y_1] \geq [wt, x, y] \). A similar claim can be stated for every \( y' \in R(x_1a_1, y_1) \).

\( I_3 \) For any key \([wt_2, x_2, y_2]\) \( \geq [wt, x, y] \), let \( G \) contain \( c > 0 \) LSPs of weight \( wt_2 \) of the form \((x_2a_2, b_2y_2)\). Further, let \( c_v \) (resp. \( c_0 \)) denote the number of such LSPs that pass through \( v \) (resp. do not pass through \( v \)). Here \( c_v + c_0 = c \). Then the tuple \((x_2a_2, b_2y_2)\) is in \( \text{Marked-Tuples} \) iff \( c_0 > 0 \) and a triple for \((x_2a_2, b_2y_2)\) representing \( c_v \) LSPs is present in \( H \).

Initialization: We show that the invariants hold at the start of the first iteration of the while loop in Step 3 of Algorithm 1. The min-key triple in \( H \) has key \([0, v, v]\). Invariant \( I_1 \) holds since we inserted into \( H \) the trivial triple of weight 0 corresponding to the vertex \( v \) and that is the only triple of such key. Moreover, since we do not represent trivial paths containing the single vertex, no counts need to be decremented. Since we assume positive edge weights, there are no LSPs in \( G \) of weight less than zero. Thus, invariants \( I_2(a), I_2(b) \), and \( I_2(c) \) hold trivially. Invariant \( I_3 \) holds since \( H \) does not contain any triple of weight \( > 0 \) and we initialized Marked-Tuples to empty.

Maintenance: Assume that the invariants are true at the beginning of the \( 1 \leq i \leq k \)-th iteration of the while loop. We now prove that the claims are true at the beginning of the \((k+1)\)-th iteration. Let the min-key triple at the beginning of the \( k \)-th iteration be \([wt_k, x_k, y_k]\). By invariant \( I_1 \), we know that for any \( a_i, b_j \), if there exists LSPs in \( G \) of the form \((x_ka_i, b_jy_k)\) of weight \( wt_k \), they have been inserted into \( H \) and further their counts have been decremented from appropriate triples in \( P(x_k, y_k) \). Now consider the set of triples with key \([wt_k, x_k, y_k]\) which we extract in the set \( S \) (Step 4 Algorithm 1). We consider left-extensions of triples in \( S \); symmetric arguments apply for right-extensions. Consider for a particular \( b \), the set of triples \( S_b \subseteq S \) and let \( fcount' \) denote the sum of the counts of the paths represented by triples in \( S_b \). Let \( x' \in L(x_k, b_ky_k) \); our goal is to generate the paths \((x'x_k, by_k)\) with count \( = fcount' \) and weight \( wt' = wt_k + w(x', x_k) \). However, we generate such paths only if they have not been generated by a right-extension of another set of paths. We note that the paths of the form \((x'x_k, by_k)\) can be generated by right extending the set of triples of the form \((x'x_k, b)\). Without loss of generality assume that the triples of the form \((x'x_k, b)\) have a key which is greater than the key \([wt_k, x_k, y_k]\) and they are not in \( H \). Thus, at the beginning of the \( k \)-th iteration, by invariant \( I_3 \), we know that \((x'x_k, by_k) \notin \text{Marked-Tuples} \). Steps 3-4 Algorithm 1 create a triple of the
form \((x'x_k, y_k)\) of weight \(wt'\) and decrement \(fcount'\) many paths from the appropriate triple in \(P(x', y_k)\) and add it to \(H_c\). This establishes invariants \(I_2(a)\) and \(I_2(c)\) at the beginning of the \((k+1)\)-th iteration. In addition, if there are no LSPs in \(G\) of the form \((x'x_k, y_k)\) which do not pass through \(v\), we delete \(x'\) from \(L(x_k, y_k)\) and delete \(y_k\) from \(R(x'x, b)\) (Step 13 Algorithm 1). On the other hand, if there exist LSPs in \(G\) of the form \((x'x_k, y_k)\), then \(x'\) (resp. \(y_k\)) continues to exist in \(L(x_k, y_k)\) (resp. in \(R(x'x, b)\)). Further, we add the tuple \((x'x_k, y_k)\) to Marked-Tuples and note that the corresponding triple is already present in \(H_c\) (Step 11 Algorithm 1). Since the invariants \(I_2(b)\) and \(I_2(c)\) were true for every key \(\leq [wt_i, x_i, y_i]\) and by the above steps we ensure that these invariants hold for every key \(\leq [wt_k, x_k, y_k]\). Thus, invariant \(I_2(b)\) is true at the beginning of the \((k+1)\)-th iteration. Note that any triple that is generated by a left extension (or symmetrically right extension) is inserted into \(H_c\) as well as into Marked-Tuples. This establishes invariant \(I_3\) at the beginning of the \((k+1)\)-th iteration.

Finally, to see that invariant \(I_1\) holds at the beginning of the \((k+1)\)-th iteration, let the min-key at the \((k+1)\)-th iteration be \([wt_{k+1}, x_{k+1}, y_{k+1}]\). Note that triples with weight \(wt_{k+1}\) starting with \(x_{k+1}\) and ending in \(y_{k+1}\) can be created either by left extending or right extending the triples of smaller weight. And since for each of iteration \(\leq k\) invariant \(I_2(c)\) holds, we conclude that invariant \(I_1\) holds at the beginning of the \((k+1)\)-th iteration.

**Termination:** The exit condition of the while loop is when the heap \(H_c\) is empty. Because Invariant \(I_1\) maintains in \(H_c\) the first triple to be extracted and processed, then \(H_c = \emptyset\) implies that there are no more triples containing the vertex \(v\) that need to be left or right extended and removed from the tuple system. Moreover, since the invariants hold for the last set of triples of weight \(wt\) extracted from the heap, by \(I_2(a)\), all LSPs having weight less than or equal to \(wt\) have been decremented from the appropriate sets \(P(\cdot)\). Finally, due to \(I_2(b)\), the sets \(L\) and \(R\) are also correctly maintained after the while loop terminates. □

In an analogous way we can prove the correctness of the loop invariant for fixup given in the proof of Lemma 5.