Decremental All-Pairs ALL Shortest Paths and Betweenness Centrality ⋆

Meghana Nasre[1], Matteo Pontecorvi[2], and Vijaya Ramachandran[2]

[1] Indian Institute of Technology Madras, India
`meghana@cse.iitm.ac.in`
[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 $O(\nu^{*2} \cdot \log n)$ time per update, where $n = |V|$, and ν^* 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), which 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 $\mathbf{w}(e)$ 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

⋆ 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.

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 algorithm [3] runs in $O(n^2 \cdot \log n)$ amortized time per update, for a sufficiently long update sequence. In [3] the result is extended to a fully dynamic algorithm 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]. We briefly discuss the fully dynamic case at the end of our paper. In these earlier algorithms, the unique shortest paths assumption is crucial.

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

Locally Shortest Paths (LSPs). For a path $\pi_{xy} \in G$, we define the π_{xy} *distance* from x to y as $\mathbf{w}(\pi_{xy}) = \sum_{e \in \pi_{xy}} \mathbf{w}(e)$, and the π_{xy} *length* from x to y as the number of edges in π_{xy}. For any $x, y \in V$, $d(x, y)$ denotes the shortest path distance from x to y in G. A path π_{xy} in G is a *locally shortest path (LSP)* [3] if either π_{xy} contains a single vertex, or every proper subpath of π_{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.

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 σ_{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, $BC(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$. 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 [2, 8]. 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, we 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 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 $O(n^2)$). 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 APASP.

Our Results. Let ν^* be the maximum number of edges that lie on shortest paths through any given vertex in G; thus, ν^* 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 $\nu^* = \Omega(n)$.

Theorem 1. *Let Σ 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 $O({\nu^*}^2 \cdot \log n)$ per update when $|\Sigma| = \Omega(m^*/\nu^*)$.*

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 $\nu^* \leq m^*$, our algorithms will have an amortized bound of $O(n^2 \cdot \log^3 n)$ on such graphs. Also, $\nu^* = O(n)$ 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 $\nu^* = O(n)$ even 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 Σ 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 $\Omega(m^*/n)$.*

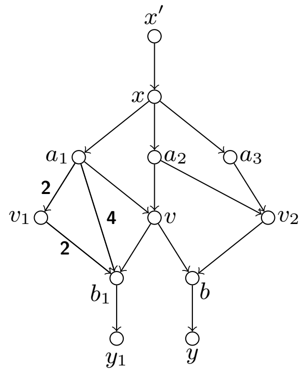


Fig. 1. Graph G

Set	G (before update on v)
$P(x,y) = P^*(x,y)$	$\{((xa_1, by), 4, 1), ((xa_2, by), 4, 2), ((xa_3, by), 4, 1)\}$
$P(x, b_1)$	$\{(xa_1, vb_1), 3, 1), ((xa_2, vb_1), 3, 1)\}$
$P^*(x, b_1)$	$\{((xa_1, vb_1), 3, 1), ((xa_2, vb_1), 3, 1)\}$
$L^*(v, y_1)$	$\{a_1, a_2\}$
$L(v, b_1y_1)$	$\{a_1, a_2\}$
$R^*(x, v)$	$\{b, b_1\}$
$R(xa_2, v)$	$\{b, b_1\}$

Fig. 2. A subset of the tuple-system for G in Fig. 1

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 $\mathbf{w}(a_1, v)$ to 10 and $\mathbf{w}(a_2, v)$ 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.

In this paper we are interested in maintaining *all* shortest paths for all vertex pairs and this requires several enhancements to the methods in [3]. In Section 2 we present a new *tuple system* which succinctly represents all LSPs in a graph with multiple shortest paths. In the rest of the paper we present our decremental 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 τ is $\mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$. We call τ a *locally shortest path tuple (LST)*. In addition, if $d(x, y) = \mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$, then τ is a *shortest path tuple (ST)*. Fig. 5(a) shows a tuple τ.

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 $p_1 = \langle x, a_2, v, b, y\rangle$ and $p_2 = \langle x, a_2, v_2, b, y\rangle$ both having weight 4.

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:

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

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.

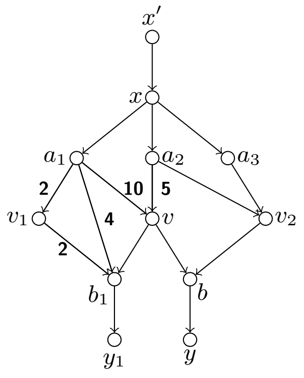
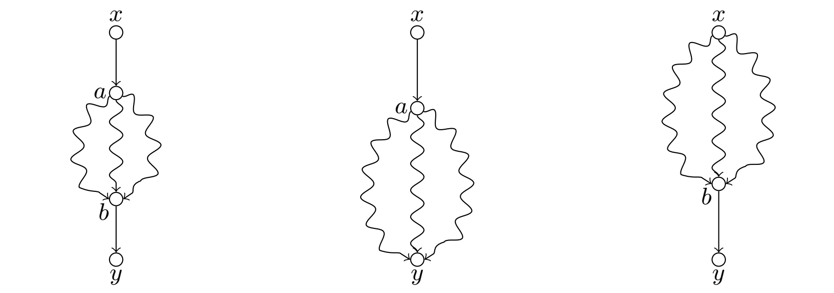


Fig. 3. Graph G'

Set	G' (with $\mathbf{w}(a_1, v) = 10$, $\mathbf{w}(a_2, v) = 5$)
$P(x, y)$ $= P^*(x, y)$	$\{((xa_2, by), 4, 1), ((xa_3, by), 4, 1)\}$
$P(x, b_1)$	$\{((xa_1, v_1b_1), 5, 1), ((xa_2, vb_1), 7, 1), ((xa_1, a_1b_1), 5, 1)\}$
$P^*(x, b_1)$	$\{((xa_1, v_1b_1), 5, 1), ((xa_1, a_1b_1), 5, 1)\}$
$L^*(v, y_1)$	$\{a_2\}$
$L(v, b_1y_1)$	$\{a_2\}$
$R^*(x, v)$	$\emptyset$
$R(xa_2, v)$	$\{b_1\}$

Fig. 4. A subset of the tuple-system for G'

Left Tuple and Right Tuple. A left tuple (or ℓ-tuple), $\tau_\ell = (xa, y)$, 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 τ_ℓ is $\mathbf{w}(x, a) + d(a, y)$. If $d(x, y) = \mathbf{w}(x, a) + d(a, y)$, then τ_ℓ 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 ℓ-tuple or r-tuple) *contains* a vertex v, if at least one of the paths represented by the tuple contains v.



(a) tuple $\tau = (xa, by)$ (b) ℓ-tuple $\tau_\ell = (xa, y)$ (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 ℓ-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 $L(\tau_\ell)$. 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 τ_ℓ and τ_r can be different, and τ is a subset of paths represented by τ_ℓ and τ_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 $\pi = x \to a \leadsto b \to y$, DI defines sets L, L^*, R and R^*. In our case, the analog of a path $\pi = x \to a \leadsto b \to y$ 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 ℓ-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 $(\times a, \times\times)$, 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 ν^* ways of choosing the last edge in $(\times a, \times\times)$ and at most $E^*_a \cap I_a$ ways of choosing the first edge in $(\times a, \times\times)$. Since $\sum_{a \in V} |E^*_a \cap I_a| = m^*$, the number of LSTs in G is at most $\sum_{a \in V} \nu^* \cdot |E^*_a \cap I_a| \leq m^* \cdot \nu^*$. □

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 ν^* choices for the last edge. Hence, the number of tuples with v as start vertex is at most $\sum_{a \in V \setminus v} \nu^* \leq n \cdot \nu^*$.

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 ${\nu^*}^2$. □

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. $R^*(x, y)$ – a balanced search tree containing vertices with vertex ID as key.

For every ℓ-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 τ to $R(xa, y)$ and $L(x, by)$, and to the corresponding triple containing τ in $P(x, y)$, (and in $P^*(x, y)$ 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 Algorithm. Given the updated vertex v and the updated weight function $\mathbf{w}'$ 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 [3].

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^* 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. 3, obtained after a decremental update on v in the graph G in Fig. 1, are schematized in Fig. 4.

3.1 The Cleanup Procedure

Alg. 1 (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, Alg. 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, Alg. 1) and *processes* them. The processing of triples involves left-extending (Steps 5–17, Alg. 1) and right-extending triples (Step 18, Alg. 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: $\gamma \leftarrow ((vv, vv), 0, 1)$; 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 $(x\times, by) \in S$ do
6:         let $fcount' = \sum_i ct_i$ such that $((xa_i, by), wt, ct_i) \in S$
7:         for every $x' \in L(x, by)$ such that $(x'x, by) \notin$ Marked-Tuples do
8:             $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$; 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 $P^*(x', y)$ then
15:                remove $\gamma'$ in $P^*(x', y)$ // decrements $count$ by $fcount$
16:                if $P^*(x, y) = \emptyset$ then delete $x'$ from $L^*(x, y)$
17:                if $P^*(x', b) = \emptyset$ then delete $y$ from $R^*(x', b)$
18:     perform symmetric steps 5 – 17 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_t, 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.

3.1.1 Complexity and Correctness.

Lemma 3 establishes the correctness of Alg. 1 and can be proved using a suitable loop invariant for the while loop in Step 3. The time bound in Lemma 4 follows from Lemma 2 since every triple examined in cleanup has at least one path that contains v.

Lemma 3. *After Alg. 1 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.*

Lemma 4. *For an update on a vertex v, Alg. 1 takes $O(\nu^{*2} \cdot \log n)$ time.*

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 Alg. 2) works with a heap of triples (H_f here), which is initialized with a *candidate* shortest path triple for each pair of vertices. The 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 γ we store the update number (update-num(γ)) and a count of the number of paths in that triple that pass through v ($paths(\gamma, v)$). Finally, similar to cleanup, the fixup procedure also left and right extends triples to create triples representing new locally shortest paths.

Alg. 2 initializes H_f in Steps 2–5 as follows. (i) For every edge incident on v, it creates a trivial triple γ which is inserted into H_f and P. It also sets update-num(γ) and paths(γ, v) for each such γ; (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).

Alg. 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 our algorithm generates the new shortest paths in Steps 10–17.
Steps 10–17, Alg. 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 $\langle a_1, v, b_1\rangle$ 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 $p_1 = \langle a_1, b_1\rangle$ and $p_2 = \langle a_1, v_1, b_1\rangle$. Note that both of these are LSPs in G and therefore are present in $P(a_1, b_1)$. In Step 5, Alg. 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, Alg. 2 – These steps left-extend and right-extend the triples in S representing *new* shortest paths from x to y.

Algorithm 2 fixup($v, \mathbf{w}'$)

```
1: $H_f \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: for each edge incident on $v$ do
3:     create a triple $\gamma$; set $paths(\gamma, v) = 1$; set update-num($\gamma$); add $\gamma$ to $H_f$ and to $P()$
4: for each $x, y \in V$ do
5:     add a min-weight triple from $P(x,y)$ to $H_f$
6: while $H_f \neq \emptyset$ do
7:     extract in $S'$ all triples with min-key $[wt, x, y]$ from $H_f$; $S \leftarrow \emptyset$
8:     if $S'$ is the first extracted set from $H_f$ for $x, y$ then
9:         {Steps 10–17: add new STs (or increase counts of existing STs) from $x$ to $y$.}
10:        if $P^*(x,y)$ is empty then
11:            for each $\gamma' \in P(x,y)$ with weight $wt$ do
12:                let $\gamma' = ((xa', b'y), wt, count')$
13:                add $\gamma'$ to $P^*(x,y)$ and $S$; add $x$ to $L^*(a', y)$ and $y$ to $R^*(x, b')$
14:        else
15:            for each $\gamma' \in S'$ containing a path through $v$ do
16:                let $\gamma' = ((xa', b'y), wt, count')$
17:                add $\gamma'$ with $paths(\gamma', v)$ in $P^*(x,y)$ and $S$; add $x$ to $L^*(a', y)$ and $y$ to $R^*(x, b')$
18:        {Steps 19–28: add new LSTs (or increase counts of existing LSTs) that extend SPs from
           $x$ to $y$.}
19:        for every $b$ such that $(x\times, by) \in S$ do
20:            let $fcount' = \sum_i ct_i$ such that $((xa_i, by), wt, ct_i) \in S$
21:            for every $x'$ in $L^*(x, b)$ do
22:                if $(x'x, by) \notin$ Marked-Tuples then
23:                    $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$
24:                    set update-num($\gamma'$); $paths(\gamma', v) \leftarrow \sum_{\gamma=(x\times, by)} paths(\gamma, v)$; add $\gamma'$ to $H_f$
25:                    if a triple for $(x'x, by)$ exists in $P(x', y)$ then
26:                        add $\gamma'$ with $paths(\gamma', v)$ in $P(x', y)$; add $(x'x, by)$ to Marked-Tuples
27:                    else
28:                        add $\gamma'$ to $P(x', y)$; add $x'$ to $L(x, by)$ and $y$ to $R(x'x, b)$
29:        perform steps symmetric to Steps 19 – 28 for right extensions.
```

Fixup maintains the following invariants. Invariant 3 is proved similarly to Invariant 3.1 in [3]. The proof of Invariant 4 requires a careful analysis of various cases to show that indeed all *new* shortest paths are inserted into the set S. Finally, Lemma 5 is proved using a suitable loop invariant for the while loop in Step 6 of Alg. 2.

Invariant 3 *If the set S' in Step 7 of Alg. 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.*

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

Lemma 5. *After execution of Alg. 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.*

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 1). 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 long enough 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 γ as one of the following disjoint types:

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

The following lemma establishes an amortized bound for fixup which is the same as the worst case bound for cleanup. This proves Theorem 1.

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 σ 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. □

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 for maintaining BC scores with the same bound.

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 streaming 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.