Fully Dynamic Betweenness Centrality

Matteo Pontecorvi and Vijaya Ramachandran*

{cavia,vlr}@cs.utexas.edu

Abstract. We present fully dynamic algorithms for maintaining betweenness centrality (BC) of vertices in a directed graph $G = (V, E)$ with positive edge weights. BC is a widely used parameter in the analysis of large complex networks. We achieve an amortized $O(\nu^{*2} \cdot \log^3 n)$ time per update with our basic algorithm, and $O(\nu^{*2} \cdot \log^2 n)$ time with a more complex algorithm, where $n = |V|$, and ν^* bounds the number of distinct edges that lie on shortest paths through any single vertex. For graphs with $\nu^* = O(n)$, our algorithms match the fully dynamic all pairs shortest paths (APSP) bounds of Demetrescu and Italiano [8] and Thorup [28] for unique shortest paths, where $\nu^* = n - 1$. Our first algorithm also contains within it, a method and analysis for obtaining fully dynamic APSP from a decremental algorithm, that differs from the one in [8].

1 Introduction

Betweenness centrality (BC) is a widely-used measure in the analysis of large complex networks, and is defined as follows. Given a directed graph $G = (V, E)$ with $|V| = n$, $|E| = m$ and positive edge weights, let σ_{xy} denote the number of shortest paths (SPs) from x to y in G, and $\sigma_{xy}(v)$ the number of SPs from x to y in G that pass through v, for each pair $x, y \in V$. Then, $BC(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

The measure $BC(v)$ is often used as an index that determines the relative importance of v in G, and is computed for all $v \in V$. Some applications of BC include analyzing social interaction networks [13], identifying lethality in biological networks [20], and identifying key actors in terrorist networks [6, 15]. In the static case, the widely used algorithm by Brandes [5] runs in $O(mn+n^2 \log n)$ on weighted graphs. Several approximation algorithms are available: [1, 24] for static computation and, recently, [4, 3] for dynamic computation. Heuristics for dynamic betweenness centrality with good experimental performance are given in [10, 16, 26], but none provably improve on Brandes. The only earlier exact dynamic BC algorithms that provably improve on Brandes on some classes of graphs are the recent separate incremental and decremental[1] algorithms in [18, 19]. Table 1 contains a summary of these results.

* Computer Science Department, University of Texas, Austin, TX 78712. This work was supported in part by NSF grants CCF-0830737 and CCF-1320675.

[1] Incremental/decremental refer to the insertion/deletion of a vertex or edge; the corresponding weight changes that apply are weight decreases/increases, respectively.

In this paper, we present two results for fully dynamic exact betweenness centrality: a basic algorithm that provably improves over Brandes for dense graphs (where m is close to n^2) with succinct single-source SP dags, and a faster algorithm that is considerably more complicated.

Our techniques recompute the BC scores using certain data structures related to shortest paths extensions (see Section 2), which are generalizations of similar ones introduced by Demetrescu and Italiano in [8] for fully dynamic all pairs shortest paths (APSP) (the `DI` method), where only one SP is maintained for each pair of vertices. To compute BC, however, we need *all* the SPs for each pair of vertices (*all pairs all shortest paths – APASP*). Our fully dynamic algorithms build on our recent work (with Nasre) [19] on decremental APASP (the `NPRdec` method), which generalizes the `DI` data structures to represent all of the multiple SPs for every pair of vertices using a *tuple-system* (see Section 3.1.1).

Paper	Year	Time	Weights	Update Type	DR/UN	Result
Brandes [5]	2001	$O(mn)$	NO	Static Alg.	Both	Exact
Brandes [5]	2001	$O(mn + n^2 \log n)$	YES	Static Alg.	Both	Exact
Geisberger et al. [9]	2007	Heuristic	YES	Static Alg.	Both	Approx.
Riondato et al. [24]	2014	depends on ϵ	YES	Static Alg.	Both	ϵ-Approx.
Semi Dynamic						
Green et al. [10]	2012	$O(mn)$	NO	Edge Inc.	Both	Exact
Kas et al. [12]	2013	Heuristic	YES	Edge Inc.	Both	Exact
NPR [18]	2014	$O(\nu^* \cdot n)$	YES	Vertex Inc.	Both	Exact
NPRdec [19]	2014	$O(\nu^{*2} \cdot \log n)$	YES	Vertex Dec.	Both	Exact
Bergamini et al. [4]	2015	depends on ϵ	YES	Batch (edges) Inc.	Both	ϵ-Approx.
Fully Dynamic						
Lee et al. [16]	2012	Heuristic	NO	Edge Update	UN	Exact
Singh et al. [26]	2013	Heuristic	NO	Vertex Update	UN	Exact
Kourtellis+ [14]	2014	$O(mn)$	NO	Edge Update	Both	Exact
Bergamini et al. [3]	2015	depends on ϵ	YES	Batch (edges)	UN	ϵ-Approx.
This paper (Basic)	2015	$O(\nu^{*2} \cdot \log^3 n)$	YES	Vertex Update	Both	Exact
This paper (FFD)	2015	$O(\nu^{*2} \cdot \log^2 n)$	YES	Vertex Update	Both	Exact

Table 1. Related results (DR stands for Directed and UN for Undirected)

Our Results. Let ν^* be the maximum number of distinct edges that lie on shortest paths through any given vertex in G; we assume $\nu^* = \Omega(n)$. Both of our BC algorithms are obtained through fully dynamic all pairs all shortest paths. The first APASP algorithm FULLY-DYNAMIC matches the `DI` APSP bound (which computes unique SPs) for graphs with $\nu^* = O(n)$; FULLY-DYNAMIC generalizes `DI`, though it is somewhat different from `DI` even for unique SPs, and its analysis is quite different from `DI`. The second APASP algorithm FFD is a generalization of Thorup [28] (the `Thorup` method) for APASP and matches its bound for APSP when $\nu^* = O(n)$; the main challenge here is to generalize the 'level graphs' of `Thorup` to the case when SPs for a given vertex pair can be distributed across multiple levels. Both APASP algorithms lead to fully dynamic BC algorithms as follows:

Theorem 1. *Let Σ be a sequence of $\Omega(n)$ fully dynamic vertex updates on a directed n-node graph $G = (V, E)$ with positive edge weights. Let ν^* bound the number of distinct edges that lie on shortest paths through any single vertex in any of the updated graphs or their vertex induced subgraphs. Then, all BC scores (and APASP) can be maintained in amortized time:*

(1) $O(\nu^{*2} \cdot \log^3 n)$ *per update with algorithm* FULLY-DYNAMIC*,*
(2) $O(\nu^{*2} \cdot \log^2 n)$ *per update with algorithm* FFD*.*

Discussion of the parameters m^* and ν^*. Let m^* be the number of distinct edges in G that lie on shortest paths; ν^*, defined above, is the maximum number of distinct edges on shortest paths through a single vertex. Clearly, $\nu^* \leq m^* \leq m$.
- m^* vs m: In many cases, $m^* \ll m$: as noted in [11], in a complete graph ($m = \Theta(n^2)$) where edge weights are chosen from a large class of probability distributions, $m^* = O(n \log n)$ with high probability.

- ν^* vs m^*: Clearly, $\nu^* = O(n)$ in any graph with only a constant number of SPs between every pair of vertices. These graphs are called k-geodetic [23] (when at most k SPs exists between two nodes), and are well studied in graph theory [27, 2, 17]. In fact $\nu^* = O(n)$ even in some graphs that have an exponential number of SPs between some pairs of vertices. In contrast, m^* can be $\Theta(n^2)$ even in some graphs with unique SPs, for example the complete unweighted graph K_n.

Another type of graph with $\nu^* \ll m^*$ is one with large clusters of nodes (e.g., as described by the *planted ℓ-partition model* [7, 25]). Consider a graph H with k clusters of size n/k (for some constant $k \geq 1$) with $\delta < w(e) \leq 2\delta$, for some constant $\delta > 0$, for each edge e in a cluster; between the clusters is a sparse interconnect. Then $m^* = \Omega(n^2)$ but $\nu^* = O(n)$.

For the above classes of graphs, both of our BC algorithms will run in amortized $\tilde{O}(n^2)$ time per update ($\tilde{O}$ hides polylog factors). More generally we have:

Theorem 2. *Let Σ be a sequence of $\Omega(n)$ updates on graphs with $O(n)$ distinct edges on shortest paths through any single vertex in any vertex-induced subgraph. Then, all BC scores (and APASP) can be maintained in amortized time $O(n^2 \cdot \log^2 n)$ per update.*

In this extended abstract, we present the key features of our results; details and full proofs are on arXiv [21, 22]. Our algorithms use $\widetilde{O}(m \cdot \nu^*)$ space, extending the $\widetilde{O}(mn)$ result in `DI` for APSP. Brandes uses only linear space, but all known dynamic algorithms require at least $\Omega(n^2)$ space.

Overview of the Paper. In Section 2 we describe our fully dynamic BC algorithm that uses the data structures maintained by our APASP algorithms. In Section 3 we review the `NPRdec` and `DI` algorithms. In Section 4 we describe our fully dynamic approach, and in Section 5 we present our first algorithm FULLY-DYNAMIC and establish its amortized time bound of $O(\nu^{*2} \cdot \log^3 n)$. In Section 6 we briefly describe our faster algorithm FFD, and we conclude with Section 7.

2 The Fully Dynamic Betweenness Centrality Algorithm

The static Brandes algorithm [5] computes BC scores in a two phase process. The first phase (implicitly) computes the SP out-dag for every source through n applications of Dijkstra's algorithm. The second phase uses an 'accumulation' technique that computes all BC scores using these SP dags in $O(n \cdot \nu^*)$ time.

In our fully dynamic algorithm, we will leave the second phase unchanged. For the first phase, we will use the approach in the incremental BC algorithm in [18], which maintains the SP dags using a very simple and efficient incremental algorithm. For decremental and fully dynamic updates, the corresponding dynamic APASP algorithms to maintain the SP dags are more involved. Neither the decremental nor our new fully dynamic APASP algorithms maintain the SP dags explicitly, instead they maintain data structures to update a collection of *tuples* (see Section 3.1.1). We now describe a very simple method to construct the SP dags from these data structures (this step is not addressed in the decremental APASP algorithm in [19]).

For every vertex pair x, y, the following sets $R^*(x, y)$, $L^*(x, y)$ are maintained in `NPRdec`, and in both of our fully dynamic algorithms (a restricted version of these sets was introduced for APSP in `DI`) :

- $R^*(x, y)$ contains all nodes y' such that every shortest path $x \rightsquigarrow y$ in G can be extended with the edge (y, y') to generate another shortest path $x \rightsquigarrow y \rightarrow y'$.
- $L^*(x, y)$ contains all nodes x' such that every shortest path $x \rightsquigarrow y$ in G can be extended with the edge (x', x) to generate another shortest path $x' \rightarrow x \rightsquigarrow y$.

These sets allow us to construct the SP dag for each source s using the following algorithm BUILD-DAG. In our fully dynamic algorithms R^* and L^* will be supersets of the exact collections of nodes defined above, but the check in Step 3 will ensure that only the correct SP dag edges are included. The combined sizes of these R^* and L^* sets is $O(n \cdot \nu^* \cdot \log n)$ in our fully dynamic algorithms, hence the amortized time bound for the overall fully dynamic BC algorithm is dominated by the time bound for fully dynamic APASP.

Algorithm 1 BUILD-DAG$(G, s, \mathbf{w}, D)$ ($\mathbf{w}$ is the weight function; D is the distance matrix)

1: **for** each $t \in V$ **do**
2: **for** each $u \in R^*(s, t)$ **do**
3: **if** $D(s, t) + \mathbf{w}(t, u) = D(s, u)$ **then** add the edge (t, u) to dag(s)

3 Background

3.1 The NPR Decremental APASP Algorithm [19]

The decremental algorithm `NPRdec` for APASP builds on the key concept of a *locally shortest path (LSP)* in a graph, introduced in the `DI` method [8]. A path p in G is an LSP if the path p' obtained by removing the first edge from p and the path p'' obtained by removing the last edge from p are both SPs in G. For APASP, we need to maintain all shortest paths, and G can have an exponential (in n) number of SPs. Thus the `DI` method is not feasible for APASP since it maintains each SP (and LSP) separately. In order to succinctly maintain all SPs and LSPs in a manner suitable for efficient decremental updates, `NPRdec` developed the tuple-system described below.

3.1.1 A System of Tuples. Since a graph could have an exponential number of shortest paths, `NPRdec` introduced the compact tuple-system described below.

Let $\mathbf{w}$ be the edge weight function in G, and let $d(x, y)$ denote the shortest path length from x to y. A *tuple*, $\tau = (xa, by)$, represents a set of paths in G, all with the same weight, and all of which use the same first edge (x, a) and the same last edge (b, y). If the paths in τ are LSPs, then τ is an LST (locally shortest tuple), and the weight of every path in τ is $\mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$. If $d(x, y) = \mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$, then τ is a *shortest path tuple (ST).*

A *triple* $\gamma = (\tau, wt, count)$ represents the tuple $\tau = (xa, by)$ that contains *count* paths from x to y, each with weight wt. We use triples to succinctly store all LSPs and SPs for each vertex pair in G. For $x, y \in V$, we define:

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

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). A *right tuple* (r-tuple) $\tau_r = (x, by)$ is defined analogously. For a shortest path r-tuple $\tau_r = (x, by)$, $L(\tau_r)$ is the set of vertices which can be used as pre-extensions to create LSTs in G, and 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. Hence:

$$L(x, by) = \{x' : (x', x) \in E(G) \text{ and } (x'x, by) \text{ is an LST in } G\}$$
$$R(xa, y) = \{y' : (y, y') \in E(G) \text{ and } (xa, yy') \text{ is an LST in } G\}.$$

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:

$$L^*(x, y) = \{x' : (x', x) \in E(G) \text{ and } (x'x, y) \text{ is a } \ell\text{-tuple representing SPs in } G\}$$
$$R^*(x, y) = \{y' : (y, y') \in E(G) \text{ and } (x, yy') \text{ is an } r\text{-tuple representing SPs in } G\}.$$

Data Structures. The `NPRdec` algorithm uses priority queues for P and P^*, and balanced search trees for L^*, L, R^* and R, as well as for a set Marked-Tuples that is specific only to one update. It also uses priority queues H_c and H_f for the cleanup and fixup procedures, respectively.

Lemma 1. *[19] Let $G = (V, E)$ be a directed graph with positive edge weights. The number of LSTs (or triples) that contain a vertex v in G is $O({\nu^*}^2)$, and the total number of LSTs (or triples) in G is bounded by $O(m^* \cdot \nu^*)$.*

The `NPRdec` algorithm maintains all STs and LSTs in the current graph, and for each tuple, it maintains the L, R, L^* and R^* sets. To execute a new update to a vertex v, `NPRdec` (similar to `DI`) first calls an algorithm cleanup on v which removes all STs and LSTs that contain v. This is followed by a call to algorithm fixup on v which computes all STs and LSTs in the updated graph that are not already present in the system. The overall algorithm update consists of cleanup followed by fixup. If the updates are all decremental then `NPRdec` maintains exactly all the SPs and LSPs in the graph in $O({\nu^*}^2 \cdot \log n)$

amortized time per update. Several challenges to adapting the techniques in the DI decremental method to the tuple-system are addressed in [19]. The analysis of the amortized time bound is also more involved since with multiple shortest paths it is possible for the dynamic APASP algorithm to examine a tuple and merely change its count; in such a case, the DI proof method of charging the cost of the examination to the new path added to or removed from the system does not apply.

3.2 The DI Fully Dynamic APSP Algorithm [8]

The DI method first gives a decremental APSP algorithm, and shows that this is also a correct, though inefficient, fully dynamic APSP algorithm. The inefficiency arises because under incremental updates the method may maintain some old SPs and their combinations that are not currently SPs or LSPs; such paths are called historical shortest paths (HPs) and locally historical paths (LHPs). To obtain an efficient fully dynamic algorithm, the DI method introduces 'dummy updates' into the update sequence. A dummy update performs cleanup and fixup on a vertex that was updated in the past. Using a strategically chosen sequence of dummy updates, it is established in [8] that the resulting APSP algorithm runs in amortized time $O(n^2 \cdot \log^3 n)$ per real update. The DI method continues to use the notation P^*, L^*, etc., even though these are supersets of the defined sets in a fully dynamic setting. We will do the same in our fully dynamic algorithms.

4 Overview of Our Fully Dynamic APASP Approach

A natural approach to obtain a fully dynamic APASP algorithm would be to convert the NPRdec decremental APASP algorithm to an efficient fully dynamic APASP algorithm by using dummy updates, similar to DI. There are two steps in this process, and each has challenges (the second step is more challenging).

Step 1: *Converting NPRdec to a correct (but inefficient) fully dynamic APASP algorithm.* Recall that the decremental APSP algorithm in DI stores old or 'historical' SPs (i.e., HPs) in the P^* sets if it is used as a fully dynamic algorithm. Historical paths arise due to the following reason: When incremental updates are interleaved with decremental ones, a path placed previously in a P^* may cease to be an SP and become a *historical SP (or HP)* if a shorter path for the same vertex pair is created by an incremental update. However, DI show that their decremental algorithm remains correct if HPs remain in P^*.

For the APASP case, the decremental NPRdec algorithm is not correct when used with fully dynamic updates. To see this, let us extend the notion of historical paths to historical tuples (HT and LHT) in the natural way. Using NPRdec, we could have an HT τ in P^* which is no longer an ST, but is an LST. Now, if additional paths are added to τ in the next update, then NPRdec will treat this tuple as an LST and update its count in P but not in P^*. If later, τ is restored as an ST (through a decremental update), it will have an incorrect lower count in P^*

which will not be detected (this can never happen in DI since it assumes unique SPs). Additional issues occur in NPRdec that need to be addressed (see [21]).

Our first step in developing a fully dynamic APASP algorithm is to update the NPRdec algorithm so that the resulting algorithm FULLY-UPDATE remains correct under fully dynamic updates. This algorithm and its analysis are available in [21]; the details are technical and are omitted here. Algorithm FULLY-UPDATE matches the amortized bound in NPRdec for decremental updates while being correct for fully dynamic updates. However, as with DI, it is inefficient as a fully dynamic algorithm.

Lemma 2. *Consider a sequence of r calls to* FULLY-UPDATE *on a graph with n vertices. Let C be the maximum number of tuples in the tuple-system that can contain a path through a given vertex, and let D be the maximum number of tuples that can be in the tuple-system at any time. Then* FULLY-UPDATE *executes the r updates in $O((r \cdot (n^2 + C) + D) \cdot \log n)$ time.*

Lemma 3. *Suppose every HT in the tuple-system is an ST in one of z different n-node graphs, and every LHT is formed from these HTs. Then,*

1. *The number of LHTs in G's tuple-system is at most $O(z \cdot m \cdot \nu^*)$.*
2. *If all HTs that contain a given vertex u lie within $z' \leq z$ of the z graphs, then the number of LHTs that contain u is $O((z + z'^2) \cdot \nu^{*2})$.*

The proof of Lemma 2 adapts the NPRdec analysis to FULLY-UPDATE; Lemma 3 follows from basic properties of the tuple-system (see [21] for both proofs).

Step 2. *Obtaining a good dummy sequence for efficient fully dynamic APASP.* The DI method uses 'dummy updates', where a vertex updated at time t is also given a 'dummy' update at steps $t + 2^i$, for each $i > 0$ (this update is performed along with the real update at step $t + 2^i$). The effect of a dummy update on a vertex v is to remove any HP or LHP that contains v, thereby streamlining the collection of paths maintained. Further, with unique SPs, each HP in $P^*(x, y)$ for a given pair x, y will have a different weight. An $O(\log n)$ bound on the number of HPs in a $P^*(x, y)$ is established in DI as follows. Let the current time step be t, and consider an HP τ last updated at $t' < t$. Let us denote the smallest i such that $t' + 2^i > t$ as the dummy-index for τ. By observing that different HPs for x, y must have different dummy-indices, it follows that their number is $O(\log t)$, which is $O(\log n)$ since the data structure is reconstructed after $O(n)$ updates.

If we try to apply the DI dummy sequence to APASP, we are faced with the issue that a new ST for x, y (with the same weight) could be created at each update in a long sequence of successive updates. Then, an incremental update could transform all of these STs into HTs. If this happens, then several HTs for x, y, all with the same weight, could have the same dummy-index (in DI only one HP can be present for this entire collection due to unique SPs). Thus, the DI approach of obtaining an $O(\log n)$ bound for the number of HPs for each vertex pair does not work for HTs in our tuple-system.

Our method for Step 2 is to use a different dummy sequence, and a completely different analysis that obtains an $O(\log n)$ bound for the number of different 'PDGs' (a PDG is a type of derived graph defined in Section 5) that

can contain the HTs. Our new dummy sequence is inspired by the 'level graph' method introduced in Thorup [28] to improve the amortized bound for fully dynamic APSP to $O(n^2 \cdot \log^2 n)$, saving a log factor over DI. The Thorup method is complex because it maintains $O(\log n)$ levels of data structures for suitable 'level graphs'. Our first algorithm FULLY-DYNAMIC does not maintain these level graphs (though our second algorithm FFD does). Instead, FULLY-DYNAMIC performs exactly like the fully dynamic algorithm in DI, except that it uses this alternate dummy update sequence, and it calls FULLY-UPDATE for APASP instead of the DI update algorithm for APSP. Our change in the update sequence requires a completely new proof of the amortized bound which we sketch in the next section (Section 5). We consider this to be a contribution of independent interest: If we replace FULLY-UPDATE by the DI update algorithm in FULLY-DYNAMIC, we get a new fully dynamic APSP algorithm which is as simple as DI, with a new analysis. The full details of algorithm FULLY-DYNAMIC are in [21].

In Section 6 we briefly describe the second algorithm FFD, which achieves an $O(\log n)$ improvement over the amortized bound for FULLY-DYNAMIC. This algorithm overcomes some technical challenges in order to generalize the Thorup method to APASP, and is considerably more complicated than FULLY-DYNAMIC.

5 Algorithm FULLY-DYNAMIC

Algorithm FULLY-DYNAMIC applies FULLY-UPDATE (see Section 4, Step 1) to vertex v with the new weight function $\mathbf{w}'$ for the t-th update. Then it executes dummy updates on a sequence $\mathcal{N}$ of the most recently updated vertices as specified in Steps 2-5. The length of this sequence of vertices is determined by the position k of the lsb set to 1 in the bit representation $B = b_{r-1} \cdots b_0$ of t.

Algorithm 2 FULLY-DYNAMIC$(G, v, \mathbf{w}', t)$

1: FULLY-UPDATE$(v, \mathbf{w}')$
2: $k \leftarrow$ position of the least significant bit set in the representation $b_{r-1} \cdots b_0$ of t
3: $\mathcal{N} \leftarrow$ set of vertices updated at steps $t-1, \cdots, t-(2^k-1)$
4: **for** each $u \in \mathcal{N}$ in decreasing order of update time **do**
5: FULLY-UPDATE$(u, \mathbf{w}')$ (dummy updates)

Let G_t be the graph after the t-th update, with G_0 the initial graph. Thus, $G = G_{t-1}$ in Algorithm 2, and the updated graph is G_t. For each i such that $b_i = 1$, we let $time_t(i)$ be the earlier update step t' whose bit representation matches B in positions $b_{r-1} \cdots b_i$ and has zeros elsewhere. We define $Prior\text{-}times(t) = \{time_t(i) \mid b_i = 1\}$. Note that $|Prior\text{-}times(t)| = O(\log t)$. The following lemma follows from the fact that a vertex updated at $t' \notin Prior\text{-}times(t)$ would have been updated by a more recent dummy update (see [21] for the proof).

Lemma 4. *For every vertex v in G_t, the step t_v of the most recent update to v is in $Prior\text{-}times(t)$.*

The Prior Deletion Graph (PDG). For $t' < t$, let W be the set of vertices that are updated in the interval of steps $[t'+1, t]$. We define the *prior deletion graph (PDG)* $\Gamma_{t',t}$ as the induced subgraph of $G_{t'}$ on the vertex set $V(G_{t'}) - W$. If t is the current update step, then we simply use $\Gamma_{t'}$ instead of $\Gamma_{t',t}$.

We say that a path p *is present in both* $G_{t'}$ *and* G_t if no call to FULLY-UPDATE is made on any vertex in p in the interval $[t'+1, t]$. The following lemma follows from a PDG $\Gamma_{t',t}$ being the result of applying a sequence of decremental updates to $G_{t'}$. Thus, an ST in $G_{t'}$ is an ST in $\Gamma_{t',t}$ if it is present in it.

Lemma 5. *1. If τ is an ST in $G_{t'}$ then τ continues to be an ST in every PDG $\Gamma_{t',t}$ with $t \geq t'$ in which τ is present.*
2. For any $\hat{t} \geq t'$, if τ is an ST in $G_{\hat{t}}$ then τ is an ST in every PDG $\Gamma_{t',t''}$, $t'' \geq \hat{t}$, in which τ is present.

PDGs for Update t: We will associate with the current update step t, the set of PDGs $\Gamma_{t'}$, for $t' \in \mathit{Prior\text{-}times}(t)$. These PDGs are similar to the *level graphs* maintained in `Thorup`, but we choose to give them a different name since we do not maintain these graphs; we only use them here to analyze the performance of our algorithm. We rebuild the tuple-system after $2n$ updates, so $t \leq 2n$.

Lemma 6. *Each HT in the tuple-system for G_t is an ST in at least one of the $\Gamma_{t',t}$ for $t' \in \mathit{Prior\text{-}times}(t)$. Further $z = O(\log n)$ in Lemma 3 for G_t.*

Proof. Consider an HT $\tau = (xa, by)$ in G_t. Let the most recently updated vertex in τ be v, and let its update step be $t_v \leq t$. By definition of HT, τ is an ST in some t' in $[t_v, t]$, hence by Lemma 5, part 2, using $\hat{t} = t' = t_v$ and $t'' = t$, we have τ an ST in Γ_{t_v}. Further, by Lemma 4, $t_v \in \mathit{Prior\text{-}times}(t)$. Finally, since $|\mathit{Prior\text{-}times}(t)| = O(\log t) = O(\log n)$ for any t, $z = O(\log n)$ in Lemma 3. ∎

We will now use the above lemma to establish the amortized time bound.

Lemma 7. *Algorithm 2 executes a sequence Σ of n real updates on an n-node graph in $O(\nu^{*2} \cdot \log^3 n)$ amortized time per update.*

Proof. (Sketch) We apply Lemma 2. By Lemma 6 we have $z = O(\log n)$ in Lemma 3, hence $D = O(m \cdot \nu^* \cdot \log n)$ in Lemma 2. Let C_1 and C_2 be the cost of a cleanup for a real and dummy update, respectively. Then, we use $z' = z$ in Lemma 3 for the real updates, so $C_1 = O(\nu^{*2} \cdot \log^2 n)$.

It is readily seen that there are $O(n \log n)$ dummy updates performed during the n real updates. At the real update step t, when a dummy update is performed on vertex u (last updated at time t_u), only PDGs Γ_t and Γ_{t_u} contain u, hence $z' = 2$ in Lemma 3. Thus $C_2 = O(\nu^{*2} \cdot \log n)$. Hence, by Lemma 2, the total time for the n real updates and $n \log n$ dummy updates is $O(\ (n \cdot (n^2 + C_1) + n \log n \cdot (n^2 + C_2) + D) \cdot \log n) = O(n \cdot \nu^{*2} \cdot \log^3 n + m \cdot \nu^* \cdot \log^2 n)$. Since we assume $\nu^* = \Omega(n)$, we have $m = O(n \cdot \nu^*)$, and we obtain the desired amortized cost for each of the n real updates. ∎

6 Algorithm FFD

We give a very brief overview of Algorithm FFD, deferring the details to [22].

Background. For unique SPs, `Thorup` uses a *level system* of decremental-only graphs, with updates being insertion or deletion of a node with incident edges. The PDGs in Section 5 are an abstract representation of the graphs maintained in Thorup's level system. Every path maintained by `Thorup` is an SP or LSP in some level graph (i.e., PDG), and when a node is removed from the current graph, it is also removed from every PDG that contains it. This saves a log factor in the amortized time bound over the DI bound.

Algorithm FFD. In our algorithm FFD for fully dynamic APASP, we explicitly maintain the PDGs of Section 5 using 'local' data structures (see [22] for a detailed description of the structures). A level i PDG is *active* at time t if i is the lsb set in some $t' \in \mathit{Prior\text{-}times}(t)$; we say $\mathit{level}(t') = i$. Each path p in a tuple is centered in level $k = \mathit{level}(t')$, where t' is the most recent step in which p entered the tuple system in a fixup step. Thus, the paths represented by a tuple are spread across the active levels at which these paths are centered; this avoids copying over all data structures each time a new level is activated, which would be very expensive. To keep track of this distribution of paths, we associate an $O(\log n)$-size array C_γ with each tuple γ that stores the number of paths in γ centered at each level.

We face several challenges when we try to extend the `Thorup` method to APASP. Here we briefly describe a major challenge, which we call the *partial extension problem (PEP)* (see [22] for a detailed example). This arises when a collection of HTs for x, y are restored as STs due to a decremental update. A tuple τ in this collection may have its correct extensions in the local structures L_i^* and R_i^* in level i, but its extensions in a more recent level j may not be in L_j^* and R_j^* if τ is not an ST in that level, and is instead an HT. Thus, when the algorithm processes τ as an ST after the current decremental update, it needs to generate the correct extensions in L_j^* and R_j^* since they are not currently present in these sets, but to maintain efficiency, it should not try to generate extensions in L_i^* and R_i^*, since they are already present there. Neither `Thorup` nor FULLY-DYNAMIC need to distinguish between these two cases. In FULLY-DYNAMIC, algorithm FULLY-UPDATE (called in Steps 1 and 5) creates LHTs by combining every pair of compatible HTs, hence these LHTs will always be available in the corresponding tuple-system. This problem is not an issue in `Thorup` either, due to the assumption of unique SPs: `Thorup` can afford to look at all HPs, since there are only $O(n^2 \cdot \log n)$ of them. Algorithm FFD maintains HTs (since it maintains APASP), and their number can be much larger.

In order to maintain both correctness and efficiency in the PEP scenario for APASP, we introduce two new data structures: (1) the historical distance matrices DL that allow us to efficiently determine the most recent level graph in which an HT was an ST, and (2) data structures LN and RN that allow us to efficiently identify exactly those new extensions that need to be performed.

7 Conclusion

We conclude with a possible avenue for improving the amortized bound. Instead of the tuples we maintain in our tuple systems, we could have maintained left and right tuples (see Section 3.1.1). This would reduce the space usage from $\tilde{O}(m \cdot \nu^*)$ to $\tilde{O}(mn)$. This improved space bound is achievable with $\tilde{O}(\nu^{*2})$ amortized time (details omitted). The number of left or right tuples that contain a given vertex is only $\tilde{O}(n \cdot \nu^*)$, but the time bound does not improve with our current method. Is there an improved method that achieves $\tilde{O}(n \cdot \nu^*)$ amortized time?

References

1. D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proc. of the 5th WAW*, pages 124–137, 2007.
2. H.-J. Bandelt and H. M. Mulder. Interval-regular graphs of diameter two. *Discrete Mathematics*, 50(0):117 – 134, 1984.
3. E. Bergamini and H. Meyerhenke. Fully-dynamic approximation of betweenness centrality. arXiv:1504.07091 [cs.DS], 2015.
4. E. Bergamini, H. Meyerhenke, and C. L. Staudt. Approximating betweenness centrality in large evolving networks. In *Proc. of ALENEX 2015*, chapter 11, pages 133–146. SIAM, 2015.
5. U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
6. T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, 2004.
7. A. Condon and R. M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Struct. Algorithms*, 18(2):116–140, Mar. 2001.
8. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
9. R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proc. of ALENEX 2008*, chapter 8, pages 90–100. SIAM, 2008.
10. O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of 4th PASSAT*, pages 11–20, 2012.
11. 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.
12. M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Proc. of ASONAM*, 2013.
13. N. Kourtellis, T. Alahakoon, R. Simha, A. Iamnitchi, and R. Tripathi. Identifying high betweenness centrality nodes in large social networks. *SNAM*, pages 1–16, 2012.
14. N. Kourtellis, G. D. F. Morales, and F. Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Trans. Knowl. Data Eng.*, 27(9):2494–2506, 2015.
15. V. Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.
16. 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. 21st WWW Conference*, pages 351–360, 2012.
17. H. M. Mulder. Interval-regular graphs. *Discrete Mathematics*, 41(3):253 – 269, 1982.
18. M. Nasre, M. Pontecorvi, and V. Ramachandran. Betweenness centrality incremental and faster. In *MFCS 2014*, volume 8635 of *LNCS*, pages 577–588. Springer, 2014.
19. M. Nasre, M. Pontecorvi, and V. Ramachandran. Decremental all-pairs all shortest paths and betweenness centrality. In *ISAAC 2014*, volume 8889 of *LNCS*, pages 766–778. Springer, 2014.
20. J. W. Pinney, G. A. McConkey, and D. R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. of 9th RECOMB*, 2005.
21. M. Pontecorvi and V. Ramachandran. Fully dynamic all pairs all shortest paths. http://arxiv.org/abs/1412.3852v2, 2014.
22. M. Pontecorvi and V. Ramachandran. A faster algorithm for fully dynamic betweenness centrality. http://arxiv.org/abs/1506.05783, 2015.
23. J. S. R. M. Ramos and M. T. Ramos. A generalization of geodetic graphs: K-geodetic graphs. *Inverstigacin Operativa*, 1:85–101, 1998.
24. M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proc. of the 7th ACM WSDM*, pages 413–422. ACM, 2014.
25. S. E. Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, Aug. 2007.
26. R. R. Singh, K. Goel, S. Iyengar, and Sukrit. A faster algorithm to update betweenness centrality after node alteration. In *Proc. of 10th WAW*, 2013.
27. N. Srinivasan, J. Opatrny, and V. Alagar. Bigeodetic graphs. *Graphs and Combinatorics*, 4(1):379–392, 1988.
28. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.