Taylor & Francis
Taylor & Francis Group

APPROXIMATING BETWEENNESS CENTRALITY IN FULLY DYNAMIC NETWORKS

Elisabetta Bergamini and Henning Meyerhenke

Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Abstract *Betweenness is a well-known centrality measure that ranks the nodes of a network according to their participation in shortest paths. Because exact computations are prohibitive in large networks, several approximation algorithms have been proposed. Besides that, recent years have seen the publication of dynamic algorithms for efficient recomputation of betweenness in networks that change over time.*

In this article, we propose the first betweenness centrality approximation algorithms with a provable guarantee on the maximum approximation error for dynamic networks. Several new intermediate algorithmic results contribute to the respective approximation algorithms: (i) new upper bounds on the vertex diameter, (ii) the first fully dynamic algorithm for updating an approximation of the vertex diameter in undirected graphs, and (iii) an algorithm with lower time complexity for updating single-source shortest paths in unweighted graphs after a batch of edge actions.

Using approximation, our algorithms are the first to make in-memory computation of betweenness in dynamic networks with millions of edges feasible. Our experiments show that our algorithms can achieve substantial speedups compared to recomputation, up to several orders of magnitude. Moreover, the approximation accuracy is usually significantly better than the theoretical guarantee in terms of absolute error. More importantly, for reasonably small approximation error thresholds, the rank of nodes is well preserved, in particular for nodes with high betweenness.

1. INTRODUCTION

The algorithmic analysis of complex networks has become a highly active research area. One important task in network analysis is to rank nodes by their structural importance using centrality measures. *Betweenness centrality* (BC) is among the widely used measures; it ranks the importance of nodes based on their participation in the *shortest paths* of the network. More formally, let the graph G represent a network with n nodes and m edges. Naming σ_{st} the number of shortest paths from a node s to a node t and $\sigma_{st}(v)$, the number of shortest paths from s to t that go through v, the (normalized) BC of v is defined as [12]: $c_B(v) = \frac{1}{n(n-1)} \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Nodes with high betweenness can be important in routing, spreading processes, and mediation of interactions. Depending on the context, this

can mean, for example, finding the most influential persons in a social network, the key infrastructure nodes in the internet, or super spreaders of a disease.

Because BC depends on *all* shortest paths, its exact computation is expensive: the best known algorithm [7] is quadratic in the number of nodes for sparse networks and cubic for dense networks, which is prohibitive for networks with hundreds of thousands of nodes. Many graphs of interest, however, such as web graphs or social networks, have millions or even billions of nodes and edges. Thus, recent years have seen the publication of several *approximation algorithms* that aim to reduce the computational effort, while finding BC values that are as close as possible to the exact ones. Good results have been obtained in this regard; in particular, a recent algorithm by Riondato and Kornaropoulos (RK) [25] gives probabilistic guarantees on the quality of the approximation. It is described in more detail in Section 3 because we build our algorithms on this method.

Motivation. Large graphs of interest, such as the Web and social networks, change continuously. Thus, in addition to the identification of important nodes in a static network, an issue of great interest is the dynamic behavior of centrality values in networks that change over time. So far, there have been no approximation algorithms that efficiently update BC scores rather than recomputing them from scratch. Several methods have been proposed to update the BC values after a graph modification, which for some of the algorithms can only be one edge insertion and for others can also be one edge deletion. However, all of these approaches are exact and have a worst-case time complexity, which is the same as Brandes's algorithm (BA) [7] on general graphs and a memory footprint of $\Omega(n^2)$.

Contribution. In this article, as our main contribution, we present the first approximation algorithms for BC in graphs that change over time. Such graphs may be directed or undirected, weighted or unweighted. We consider two dynamic scenarios, an incremental one (i.e., only edge insertions or weight decreases are allowed) and a fully dynamic one, which also handles edge deletions or weight increase operations. After each batch of edge actions, we assert the same guarantee as the static RK algorithm: the approximated BC values differ by, at most, ϵ from the exact values with probability at least $1 - \delta$, where ϵ and δ can be arbitrarily small constants. Running time and memory required depend on how tightly the error should be bounded.

Besides resampling as few shortest paths as possible, several new intermediate algorithmic results contribute to the speed of the respective new approximation algorithms: (i) In Section 4 we propose new upper bounds on the *vertex diameter* (*VD*) (i.e., number of nodes in the shortest path(s) with the maximum number of nodes). These bounds vary depending on the graph type (weighted vs. unweighted, directed vs. undirected). Their usefulness stems from the fact that the new bounds can often improve the one used in the RK algorithm [25] and thereby significantly reduce the number of samples necessary for the error guarantee. (ii) In Section 5, besides detailing the BC approximation algorithms, we also present the first fully dynamic algorithms for updating an approximation of *VD* in undirected graphs. (iii) As part of the BC approximation algorithms, we propose an algorithm with lower time complexity for updating single-source shortest paths in unweighted graphs after a batch of edge actions.

Our experimental study shows that our algorithms are the first to make in-memory computation of a betweenness ranking practical for large dynamic networks. With approximation, we achieve a much improved scaling behavior compared to exact approaches (also dynamic ones), enabling us to update approximate betweenness scores in a network with 36 million edges in a few seconds on typical workstation hardware. Moreover, processing batches of edge actions, our algorithms yield significant speedups (several orders of

magnitude) compared to restarting the approximation with RK. Regarding accuracy, our experiments show that the estimated absolute errors are always lower than the guaranteed ones. For nodes with high betweenness, the rank of nodes is also well preserved, even when relatively few shortest paths are sampled.

2. RELATED WORK

Static BC algorithms—exact and approximation. The fastest existing method for the exact BC computation, BA, requires $\Theta(nm)$ operations for unweighted graphs and $\Theta(nm + n^2 \log n)$ for graphs with positive edge weights [7]. BA computes for every node $s \in V$ a slightly modified version of a single-source shortest-path tree (SSSP tree), producing for each s the directed acyclic graph (DAG) of *all* shortest paths starting at s. Using the information contained in the DAGs, BA computes the *dependency* $\delta_s(v)$ for each node v, that is, the sum over all nodes t of the fraction of shortest paths between s and t to which v is internal. The betweenness of each node v is simply the *sum* over all sources $s \in V$ of the dependencies $\delta_s(v)$. Therefore, we can see the dependency $\delta_s(v)$ as a *contribution* that s gives to the computation of $c_B(v)$. Based on this concept, some algorithms for an *approximation* of BC have been developed. It was proposed [8] propose to approximate $c_B(v)$ by extrapolating it from the contributions of a *subset* of source nodes, also called *pivots*. Selecting the pivots uniformly at random, the approximation can be proven to be an unbiased estimator for $c_B(v)$ (i.e., its expectation is equal to $c_B(v)$). If the number of samples is $O(\log(n/\delta)/\epsilon^2)$, the algorithm guarantees that all estimates are within ϵ from their actual values, with probability at least $1 - \delta$. In a subsequent work, [14] noticed that the approach of [8] can overestimate BC scores of nodes close to the pivots. To limit this bias, they introduce a *scaling function*, which gives less importance to contributions from pivots that are close to the node. However, no theoretical guarantee has been proven for this approach. The BC of a specific node only was approximated [1], based on an adaptive sampling technique that reduces the number of pivots for nodes with high centrality and [9] proposes alternative sampling techniques that try to minimize the number of samples, when the betweenness of a single node has to be estimated. Different from the previous approaches is the approximation algorithm by [25], which samples a *single* random shortest path at each iteration. This approach allows a theoretical guarantee on the quality of approximation (see Section 3). Because of this guarantee, we use it as a building block of our new approach and refer to it as RK.

Exact dynamic algorithms. Dynamic algorithms update the betweenness values of all nodes in response to a modification on the graph, which might be an edge/node insertion, an edge/node deletion, or a change in an edge's weight. The first published approach is QUBE [20], which relies on the decomposition of the graph into biconnected components. The approach has been extended to node updates in [15]. The approach proposed by [16] for unweighted graphs maintains a structure with the previously calculated BC values and additional information, such as the distance of each node v from every source $s \in V$ and the list of *predecessors*, i.e., the nodes immediately preceding v in all shortest paths from s to v. Using this information, it tries to limit the recomputations to the nodes whose betweenness has actually been affected. The approach by [16] win order to reduce the memory requirements. Instead of storing the predecessors of each node v from each possible source, they recompute them every time the information is required.

An existing algorithm for the dynamic APSP problem by [24] was extended by [17] to also update BC scores. The recent work by [22] contains the first dynamic algorithm for
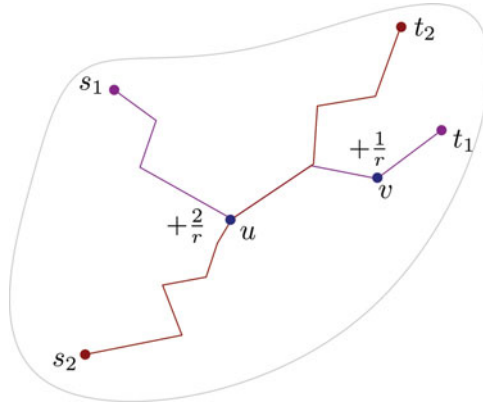
Figure 1 Sampled paths and score update in the RK algorithm.

BC (NPR) which is asymptotically faster than recomputing from scratch on certain inputs. In particular, when only edge insertions are allowed and the considered graph is sparse and weighted, their algorithm takes $O(n^2)$ operations, whereas BA requires $O(n^2 \log n)$ on sparse weighted graphs. Existing fully dynamic APSP algorithms have been extended [23] with new data structures to update *all* shortest paths (APASP) and then recompute dependencies as in BA.

All dynamic algorithms mentioned perform better than recomputation on certain inputs. Yet, none of them has a worst-case complexity better than BA on *all graphs* because all require an update of an APSP problem. For this problem, no algorithm exists that has better worst-case running time than recomputation [26]. In addition, the problem of updating BC seems even more difficult than the dynamic APSP problem. Indeed, the dependencies (and therefore BC) might need to be updated even on nodes whose distance from the source has not changed, because they could be part of new shortest paths or not be part of old shortest paths any longer.

Batch dynamic SSSP algorithms. Dynamic SSSP algorithms recompute distances from a source node after a single edge update or a batch of edge updates. Algorithms for the batch problem have been published [24, 13, 3] and compared in experimental studies [3, 10]. The experiments show that the tuned algorithm T-SWSF [3] performs well on many types of graphs and edge updates. Therefore, we use T-SWSF as a building block in our fully dynamic BC approximation algorithm.

3. RK ALGORITHM

In this section, we briefly describe the static BC approximation algorithm RK [25], the foundation for our incremental approach. The idea of RK is to sample a set $S = \{p_{(1)}, \dots, p_{(r)}\}$ of r shortest paths between randomly sampled source-target pairs (s, t). Then, RK computes the approximated betweenness centrality $\tilde{c}_B(v)$ of a node v as the fraction of sampled paths $p_{(k)} \in S$ that v is internal to by adding $\frac{1}{r}$ to the node's score for each of these paths. Figure 1 illustrates an example in which the sampling of two shortest paths leads to $\frac{2}{r}$ and $\frac{1}{r}$ being added to the score of u and v, respectively. Each possible shortest path p_{st} has the following probability of being sampled in each of the r iterations: $\pi_G(p_{st}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma_{st}}$ (Lemma 7 of [25]).

The number r of samples required to approximate BC scores with the given error guarantee is calculated as

$$r = \frac{c}{\epsilon^2}\left(\lfloor\log_2(VD-2)\rfloor + 1 + \ln\frac{1}{\delta}\right), \tag{3.1}$$

where ϵ and δ are constants in $(0, 1)$, $c \approx 0.5$ and VD is the vertex diameter of G, i.e., the number of nodes in the shortest path of G with maximum number of nodes. In unweighted graphs, VD coincides with $\mathsf{diam}+1$, where diam is the number of edges in the longest shortest path. In weighted graphs, VD and the (weighted) diameter diam (i.e., the length of the longest shortest path) are unrelated quantities. The following error guarantee holds:

Lemma 3.1. [25] *If r shortest paths are sampled according to the above-defined probability distribution π_G, then with probability at least $1 - \delta$ the approximations $\tilde{c}_B(v)$ of the betweenness centralities are within ϵ from their exact value:* $\Pr(\exists v \in V\, s.t.\, |c_B(v) - \tilde{c}_B(v)| > \epsilon) < \delta$.

To sample the shortest paths according to π_G, RK first chooses a node pair (s, t) uniformly at random and performs an SSSP search from s, also keeping track of the number σ_{sv} of shortest paths from s to v and of the list of predecessors $P_s(v)$ for any node v. Then, one shortest path is selected: starting from t, a predecessor $z \in P_s(t)$ is selected with probability $\sigma_{sz}/\sum_{w \in P_s(t)} \sigma_{sw} = \sigma_{sz}/\sigma_{st}$. The sampling is repeated iteratively until node s is reached. Algorithm 1 is the pseudocode for RK. Function `computeExtendedSSSP` is an SSSP algorithm that keeps track of the number of shortest paths and of the list of predecessors while computing distances, as in BA. Because we are interested only in the paths from s to t, we can stop the computation of the SSSP once t is reached.

Approximating the vertex diameter. RK uses two upper bounds on VD that can both be computed in $O(n+m)$. For unweighted undirected graphs, it samples a source node s_i for each connected component of G, computes a BFS from each s_i, and sums the two shortest paths with maximum length starting in s_i. The VD approximation is the maximum of these sums over all components. For directed or weighted graphs, RK approximates VD with the size of the largest weakly connected component, which can be a significant overestimation for complex networks, possibly of orders of magnitude. In this article, we present new approximations for directed and for weighted graphs, described in Section 4.

4. NEW UPPER BOUNDS ON THE VERTEX DIAMETER

4.1. Directed Unweighted Graphs

Let G be a directed unweighted graph. For now, let us assume G is strongly connected. Let s be any node in G and let u be the node with maximum forward distance from s (i.e., $d(s, u) \geq d(s, x)\ \forall x \in V$). Analogously, let v be the node with maximum backward distance (i.e., $d(v, s) \geq d(x, s)\ \forall x \in V$). Then, naming $\tilde{VD}_{SC}$ the sum $d(s, u) + d(v, s) + 1$:

Lemma 4.1. $VD \leq \tilde{VD}_{SC} < 2VD$.

Algorithm 1: RK algorithm

Input : Graph $G = (V, E), \epsilon, \delta \in (0, 1)$
Output: Approximated BC values $\forall v \in V$

1 **foreach** *node* $v \in V$ **do**
2     $\tilde{c}_B(v) \leftarrow 0$;
3 **end**
4 $\mathsf{VD}(G) \leftarrow \texttt{getVertexDiameter}(G)$;
5 $r \leftarrow (c/\epsilon^2)(\lfloor \log_2(\mathsf{VD}(G) - 2) \rfloor + \ln(1/\delta))$;
6 **for** $i \leftarrow 1$ **to** r **do**
7     $(s_i, t_i) \leftarrow \texttt{sampleUniformNodePair}(V)$;
8     $(d_{s_i}, \sigma_{s_i}, P_{s_i}) \leftarrow \texttt{computeExtendedSSSP}(G, s_i)$;
    // Now one path from s_i to t_i is sampled uniformly at random
9     $v \leftarrow t_i$;
10     $p_{(i)} \leftarrow$ empty list;
11   **while** $P_{s_i}(v) \neq \{s_i\}$ **do**
12   sample $z \in P_{s_i}(v)$ with $\Pr = \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
13        $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
14   add $z \rightarrow p_{(i)}$; $v \leftarrow z$;
15   **end**
16 **end**
17 **return** $\{(v, \tilde{c}_B(v)), \ v \in V\}$

Proof. Let x and y be two nodes such that the number of nodes in the shortest path from x to y is equal to VD. Due to the triangle inequality, $d(x, y) \leq d(x, s) + d(s, y)$. Therefore, $d(x, y) \leq d(v, s) + d(s, u)$. Since in unweighted graphs $d(x, y) = VD - 1$, the first inequality holds. By definition of $VD, 2 \cdot VD \geq (d(v, s) + 1) + (d(s, u) + 1) > \tilde{V}D_{\mathrm{SC}}$. $\square$

The upper bound $\tilde{V}D_{\mathrm{SC}}$ can be computed in $O(n + m)$, simply by running a forward and a backward BFS from any source node s.

Let us now consider any directed unweighted graph G. We can define the directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of strongly connected components (SCCs) (sometimes referred to as a *shrunken graph* in the literature) similarly to [6]. In $\mathcal{G}$, the set of vertices is the set of SCCs of G and there is an edge from $C \in \mathcal{V}$ to $C' \in \mathcal{V}$ if and only if there is an edge in E from a node in C to a node in C'. (Notice that this also means that all the nodes in C' are reachable from the nodes in C.) More generally, the set of nodes reachable from any node in C is exactly the set of nodes in the SCCs reachable from C in $\mathcal{G}$.

We can now define an algorithm that computes an upper bound on VD for G. For each C in $\mathcal{V}$, we compute an upper bound $\tilde{V}D_{\mathrm{SC}}(C)$ on VD in C (i.e., considering only paths that are contained in C) as described before for strongly connected graphs. This can be done in linear time by running a backward and a forward BFS from a random source node s for each SCC and stopping the search when a visited node belongs to a different SCC from that of s. For each C_i, we know that the nodes reachable from nodes in C_i are only those in the SCCs C_j such that there is a path in $\mathcal{G}$ from C_i to C_j. We can compute a topological sorting $\{C_1, \ldots, C_k\}$ of $\mathcal{V}$ (that is, $(C_i, C_j) \in \mathcal{E} \Rightarrow i < j$). Let C_k be the last component in the topological ordering. Then, we know that no path exists from a node in C_k to any node that is not in C_k, which means that the node C_k in $\mathcal{G}$ has outdegree 0.

We call $\tilde{VD}_{\text{DIR}}(C)$ the upper bound on VD restricted only to those nodes that *start* in C (but may end somewhere else). For C_k, $\tilde{VD}_{\text{DIR}}(C_k)$ is equal to $\tilde{VD}_{\text{SC}}(C_k)$. For the other components, we can compute it in the following way: starting from C_k, we process all the components in reverse topological order and set

$$\tilde{VD}_{\text{DIR}}(C) = \max_{(C,C')\in\mathcal{E}} \tilde{VD}_{\text{DIR}}(C') + \tilde{VD}_{\text{SC}}(C).$$

To get an upper bound on the whole graph, we can take the maximum over all upper bounds $\tilde{VD}_{\text{DIR}}(C)$, i.e., we set $\tilde{VD}_{\text{DIR}} := \max_{C\in\mathcal{V}} \tilde{VD}_{\text{DIR}}(C)$. In other words,

$$\tilde{VD}_{\text{DIR}} = \max_{p\in\mathcal{P}(\mathcal{G})} \sum_{C_i\in p} \tilde{VD}_{\text{SC}}(C_i).$$

where $\mathcal{P}(G)$ is the set of paths in $\mathcal{G}$.

Proposition 4.2. $VD \leq \tilde{VD}_{DIR} < 2 \cdot VD^2$.

Proof. Let us prove the first inequality. Let $p = (u_1, \ldots, u_{VD})$ be a shortest path in G whose number $|p|$ of nodes is equal to VD. Say p traverses l SCCs $(C_1, \ldots, C_l)$. Then, p can be partitioned in l subpaths p_i, $i = 1, .., l$, such that $p_i \subseteq C_i$ and p_i is a shortest path in C_i. By Lemma 4.1, $|p_i| \leq \tilde{VD}_{\text{SC}}(C_i)$, $i = 1, \ldots, l$. Therefore, $|p| = \sum_{i=1,\ldots,l} |p_i| \leq \sum_{i=1,\ldots,l} \tilde{VD}_{\text{SC}}(C_i) \leq \tilde{VD}_{\text{DIR}}$ (this last inequality holds by definition of $\tilde{VD}_{\text{DIR}}$).

How "bad" can $\tilde{VD}_{\text{DIR}}$ be in the worst case? Now, let $(C_1, \ldots, C_l)$ denote the path in $\mathcal{G}$ such that $\tilde{VD}_{\text{DIR}} = \sum_{i=1,\ldots l} \tilde{VD}_{\text{SC}}(C_i)$. Let l be the number of components in this path. How large can l be? Because there is at least one node of G in each C_i, there must be at least a shortest path of size l in G that goes through the components $C_1, \ldots, C_l$. Therefore, $l \leq VD$. Also, let $k = \max_{C\in\mathcal{V}} \tilde{VD}_{\text{SC}}(C)$. By Lemma 4.1, $k < 2 \cdot VD(C_k)$, where C_k is the SCC whose upper bound is equal to k. Clearly, $k < 2 \cdot VD$. Then, by definition, $\tilde{VD}_{\text{DIR}} = \sum_{i=1,\ldots l} \tilde{VD}_{\text{SC}}(C_i) \leq l \cdot k < 2 \cdot VD^2$. $\square$

Figure 2 clarifies the new upper bound with an example. The upper bound can be computed in $O(n+m)$. Indeed, $\mathcal{G}$ can be computed in $O(n+m)$ by finding the SCCs of G and scanning the edges in E. Then, the topological sorting and the accumulation of the $\tilde{VD}_{\text{DIR}}(C)$ of the different components can be done in $O(|\mathcal{V}| + |\mathcal{E}|) = O(n + m)$. Notice that our new upper bound is never larger than the size of the largest weakly connected component, the previous bound used in RK. Also, although the upper bound can be as bad as $2 \cdot VD^2$ in theory, our experimental results on real-world complex networks show that it is within a factor 4 from the exact VD and several orders of magnitude smaller than the size of the largest weakly connected components.

4.2. Undirected Weighted Graphs

Let G be an undirected graph. For simplicity, let G be connected for now. If it is not, we compute an approximation for each connected component and take the maximum over all the approximations. Let $T \subseteq G$ be an SSSP tree from any source node $s \in V$. Let p_{xy} denote a shortest path between x and y in G and let p_{xy}^T denote a shortest path between x and y in T. Let $|p_{xy}|$ be the number of nodes in p_{xy} and $d(x, y)$ be the distance between
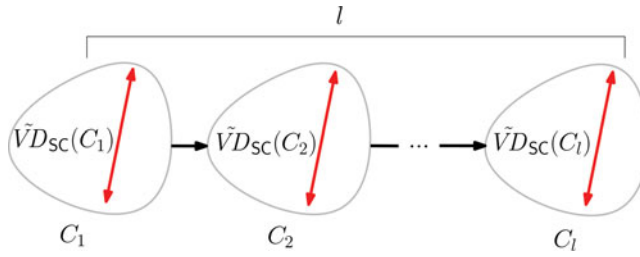
Figure 2 Path $(C_1, \ldots, C_l)$ of the DAG $\mathcal{G}$ of SCCs. Each SCC C_i has its own upper bound $\tilde{V}D_{\text{SC}}(C_i)$ and $\tilde{V}D_{\text{DIR}}$ is computed as $\sum_{i=1,\ldots,l} \tilde{V}D_{\text{SC}}(C_i)$.

x and y in G, and analogously for $|p_{xy}^T|$ and $d^T(x, y)$. Let $\overline{\omega}$ and $\underline{\omega}$ be the maximum and minimum edge weights in G, respectively. Let u and v be the nodes with maximum distance from s, i.e., $d(s, u) \geq d(s, v) \geq d(s, x) \; \forall x \in V, x \neq u$. We define the VD approximation $\tilde{V}D_{\text{W}} := 1 + \frac{d(s,u)+d(s,v)}{\underline{\omega}}$. Then,

Proposition 4.3. $VD \leq \tilde{V}D_W < 2 \cdot VD \cdot \overline{\omega}/\underline{\omega}$.

Proof. To prove the first inequality, we can notice that $d^T(x, y) \geq d(x, y)$ for all $x, y \in V$, since all the edges of T are contained in those of G. Also, since every edge has weight at least $\underline{\omega}$, $d(x, y) \geq (|p_{xy}| - 1) \cdot \underline{\omega}$. Therefore, $d^T(x, y) \geq (|p_{xy}| - 1) \cdot \underline{\omega}$, which can be rewritten as $|p_{xy}| \leq 1 + \frac{d^T(x,y)}{\underline{\omega}}$, for all $x, y \in V$. Thus, $VD = \max_{x,y} |p_{xy}| \leq 1 + (\max_{x,y} d^T(x, y))/\underline{\omega} \leq 1 + \frac{d^T(s,u)+d^T(s,v)}{\underline{\omega}} = 1 + \frac{d(s,u)+d(s,v)}{\underline{\omega}}$, where the last expression equals $\tilde{V}D_{\text{W}}$ by definition.

To prove the second inequality, we first notice that $d(s, u) \leq (|p_{su}| - 1) \cdot \overline{\omega}$, and analogously $d(s, v) \leq (|p_{sv}| - 1) \cdot \overline{\omega}$. Consequently, $\tilde{V}D_{\text{W}} \leq 1 + (|p_{su}| + |p_{sv}| - 2) \cdot \overline{\omega}/\underline{\omega} < 2 \cdot |p_{su}| \cdot \overline{\omega}/\underline{\omega}$, supposing that $|p_{su}| \geq |p_{sv}|$ without loss of generality. By definition of VD, $|p_{su}| \leq VD$. Therefore, $\tilde{V}D_{\text{W}} < 2 \cdot VD \cdot \overline{\omega}/\underline{\omega}$. □

To obtain the upper bound $\tilde{V}D_{\text{W}}$, we can simply compute an SSSP search from any node s, find the two nodes with maximum distance and perform the remaining calculations. Notice that $\tilde{V}D_{\text{W}}$ extends the upper bound proposed for RK [25] for unweighted graphs: When the graph is unweighted and thus $\underline{\omega} = \overline{\omega}$, $\tilde{V}D_{\text{W}}$ becomes equal to the approximation used by RK. Complex networks are often characterized by a small diameter and in networks such as coauthorship, friendship, communication networks, VD and $\overline{\omega}/\underline{\omega}$ can be several orders of magnitude smaller than the size of the largest component. In this case, the new bound translates into a substantially improved VD approximation.

4.3. Directed Weighted Graphs

The upper bound for directed weighted graphs can be easily derived from those described in Sections 4.1 and 4.2. If G is strongly connected, we can define $\tilde{V}D_{\text{SCW}} := 1 + \frac{d(s,u)+d(v,s)}{\underline{\omega}}$, where s is any node, u is the node with maximum forward distance from s, v is the node with maximum backward distance, and $\underline{\omega}$ is the minimum edge weight. It can be easily proved that Proposition 4.3 holds also in this case, considering a forward SSSP tree from s (where distances are bounded by $d(s, u)$) and a backward SSSP tree
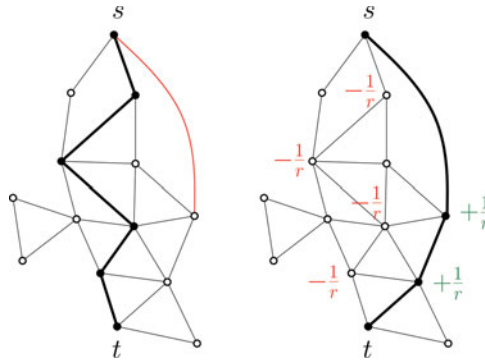
Figure 3 Updating shortest paths and BC scores.

(where distances are bounded by $d(v, s)$). For general directed weighted graphs, we can use the algorithm described in Section 4.1 using $\tilde{V}D_{\text{SCW}}(C) := 1 + \frac{d(s,u)+d(v,s)}{\underline{\omega}_c}$ as an upper bound for each SCC C (where $\underline{\omega}_c$ is the minimum edge weight in C). It is easy to prove that the resulting bound is an upper bound on VD and that it is always smaller than $2 \cdot \max_{C \in \mathcal{G}} \frac{\overline{\omega}_c}{\underline{\omega}_c} \cdot VD^2$, using Propositions 4.2 and 4.3. Since it requires to compute an SSSP tree for each SCC, the complexity of computing the bound is $O(m + n \log n)$ (the other operations can be done in linear time, as described in Section 4.1).

5. FULLY DYNAMIC APPROXIMATION ALGORITHMS

5.1. Path Subsitution

Our algorithms for dynamic BC approximation are composed of two phases: an *initialization* phase, which executes RK on the initial graph, and an *update* phase, which recomputes the approximated BC scores after a sequence of edge updates. Let us consider a batch $\beta = \{e_1, \ldots, e_k\}$ of edge updates $e_i = \{u_i, v_i, \omega(u_i, v_i)\}$ applied to a graph G. Also, let us assume for the moment that β is composed of **edge insertions only** (or weight decreases) and β **does not increase** the vertex diameter of G and, therefore, also the number r of samples required by RK for the maximum error guarantee. We will discuss the general case in Section 5.2. Intuitively, our basic idea is to keep the old sampled paths and update them only when necessary, instead of recomputing r shortest paths from scratch. Figure 3 shows an example to illustrate this idea: Assume several shortest paths between s and t exist, of which one has been sampled (with black nodes). An edge insertion (represented in red) shortens the distance between s and t, creating a new shorter path. Therefore, we simply subtract $1/r$ from each node in the old shortest path and add $1/r$ to each node in the new one.

From this point on, we give a formal description and consider only edge insertions. We suppose the graph is undirected, but in this restricted semidynamic setting, our results can be easily extended to weight decreases and directed graphs. Let $G' = (V, E \cup \beta)$ be the new graph, let $d_s'(t)$ denote the new distance between any node pair (s, t), and let σ_{st}' be the new number of shortest paths between s and t. Let $\mathcal{S}_{st}$ and $\mathcal{S}_{st}'$ be the old and the new set of shortest paths between s and t, respectively. A new set $S' = \{p_{(1)}', \ldots, p_{(r)}'\}$ of shortest paths has to be sampled now in order to let Lemma 3.1 hold for the new configuration;

in particular, the probability $\Pr(p'_{(k)} = p'_{st})$ of each shortest path p'_{st} to be sampled must be equal to $\pi_{G'}(p'_{st}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_{st}}$. Clearly, one could rerun RK on the new graph, but we can be more efficient: we recall that we are assuming that the VD and, therefore, also the number of samples for G' is smaller than or equal to that of G. Given any old sampled path p_{st}, we can *keep* p_{st} if the set of shortest paths between s and t has not changed and *replace* it with a new path between s and t, otherwise. Then, the following lemma holds:

Lemma 5.1. *Let S be a set of shortest paths of G sampled according to π_G. Let $\mathcal{P}$ be the procedure that creates S' by substituting each path $p_{st} \in S$ with a path p'_{st} according to the following rules: 1. $p'_{st} = p_{st}$ if $d'_s(t) = d_s(t)$ and $\sigma'_{st} = \sigma_{st}$. 2. p'_{st} selected uniformly at random among $\mathcal{S}'_{st}$ otherwise. Then, p'_{st} is a shortest path of G' and the probability of any shortest path p'_{xy} of G' to be sampled at each iteration is $\pi_{G'}(p'_{xy})$, i.e., $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_{xy}}, k = 1, \ldots, r$.*

Proof. To see that p'_{st} is a shortest path of G', it is sufficient to notice that, if $d'_s(t) = d_s(t)$ and $\sigma'_s(t) = \sigma_s(t)$, then all the shortest paths between s and t in G are shortest paths also in G'.

Let p'_{xy} be a shortest path of G' between nodes x and y. Basically, there are two possibilities for p'_{xy} to be the kth sample. Naming e_1 the event $\{\mathcal{S}_{xy} = \mathcal{S}'_{xy}\}$ (the set of shortest paths between x and y does not change after the edge insertion) and e_2 the complementary event of e_1, we can write $\Pr(p'_{(k)} = p'_{xy})$ as $\Pr(p'_{(k)} = p'_{xy} \cap e_1) + \Pr(p'_{(k)} = p'_{xy} \cap e_2)$.

Using conditional probability, the first addend can be rewritten as $\Pr(p'_{(k)} = p'_{xy} \cap e_1) = \Pr(p'_{(k)} = p'_{xy}|e_1)\Pr(e_1)$. As the procedure $\mathcal{P}$ keeps the old shortest path when e_1 occurs, then $\Pr(p'_{(k)} = p'_{xy}|e_1) = \Pr(p_{(k)} = p'_{xy}|e_1) = \frac{1}{n(n-1)}\frac{1}{\sigma_x(y)}$, which is also equal to $\frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$, since $\sigma_x(y) = \sigma'_x(y)$ when we condition on e_1. Therefore, $\Pr(p'_{(k)} = p'_{xy} \cap e_1) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot \Pr(e_1)$.

Analogously, $\Pr(p'_{(k)} = p'_{xy} \cap e_2) = \Pr(p'_{(k)} = p'_{xy}|e_2)\Pr(e_2)$. In this case, $\Pr(p'_{(k)} = p'_{xy}|e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)}$, since this is the probability of the node pair (x, y) to be the kth sample in the initial sampling and of p'_{xy} to be selected among other paths in $\mathcal{S}'_{xy}$. Then, $\Pr(p'_{(k)} = p'_{xy} \cap e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)} \cdot \Pr(e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)} \cdot (1 - \Pr(e_1))$. The probability $\Pr(p'_{(k)} = p'_{xy})$ can, therefore, be rewritten as $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot \Pr(e_1) + \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot (1-\Pr(e_1)) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$. □

Since the set of paths is constructed according to $\pi_{G'}$, Theorem 5.2 follows directly from Lemma 3.1.

Theorem 5.2. *Let $G = (V, E)$ be a graph and let $G' = (V, E \cup \beta)$ be the modified graph after the the batch β. Let $VD(G) \geq VD(G')$. Let S be a set of r shortest paths of G sampled according to π_G and $r = \frac{c}{\epsilon^2}\left(\lfloor \log_2(VD(G) - 2)\rfloor + 1 + \ln\frac{1}{\delta}\right)$ for some constants $\epsilon, \delta \in (0, 1)$. Then, if a new set S' of shortest paths of G' is built according to procedure $\mathcal{P}$ and the approximated values of betweenness centrality $\tilde{c}'_B(v)$ of each node v are computed as the fraction of paths of S' to which v is internal, then*

$$\Pr(\exists v \in V \text{ s.t. } |c'_B(v) - \tilde{c}'_B(v)| > \epsilon) < \delta,$$

where $c'_B(v)$ is the new exact value of betweenness centrality of v after the edge insertion.

Notice that our guarantee is only probabilistic and it is possible (although with probability smaller than δ) that at some time step the approximated betweenness of a node diverges from the exact one for more than ϵ. Because there is dependency between the approximated values at different time steps, it is possible that this error propagates over the following time steps. However, it is also possible that the error decreases as a consequence of some modifications in the graph. Theorem 5.2 tells us that, *at each single time step*, the absolute error on the betweenness of each node is smaller than ϵ, with probability at least $1 - \delta$. We would also like to point out that, although cases where the maximum error exceeds ϵ are theoretically possible, this actually never happened in our experiments, where the measured errors were often orders of magnitude smaller than ϵ (see Section 6). Algorithm 2 shows the update procedure based on Theorem 5.2. For each sampled node pair (s_i, t_i), $i = 1, \ldots, r$, we first update the SSSP from s_i, a step that will be discussed in Sections 5.3 and 5.4. In the case that the distance or the number of shortest paths between s_i and t_i has changed, a new shortest path is sampled uniformly, as in RK. This means that $\frac{1}{r}$ is subtracted from the score of each node in the old shortest path and the same quantity is added to the nodes in the new shortest path. Contrarily, if both distances and number of shortest paths between s_i and t_i have not changed, nothing needs to be updated. Considering edges in a batch allows us to recompute the BC scores only once instead of doing it after each single-edge update. Moreover, this allows us to use specific batch algorithms for the update of the SSSP DAGs, which process the nodes affected by multiple edges of β only once, instead of for each single edge.

Unlike RK, with our dynamic algorithm we scan the neighbors every time we need the predecessors instead of storing them (Line 11). This allows us to use $\Theta(n)$ memory per sample (i.e., $\Theta(r \cdot n)$ in total) instead of $\Theta(m)$ per sample, and our experiments show that the running time is hardly influenced. The number of samples depends on ϵ, so, in theory, this can be as large as $|V|$. However, our experiments show that relatively large values of ϵ (e.g., $\epsilon = 0.05$) lead to a good ranking of nodes with high BC and, for such values, the number of samples is typically much smaller than $|V|$, making the memory requirements of our algorithms significantly less demanding than those of the dynamic exact algorithms ($\Omega(n^2)$) for many applications.

5.2. Sampling New Paths

In the previous section, we assumed that $VD(G) \geq VD(G')$. Although many real-world networks exhibit a shrinking-diameter behavior [21], it is clearly possible that VD increases as a consequence of edge insertions/deletions or weight updates. If this happens, we can still update the old paths as described in Section 5.1, but we also need to sample new additional paths, according to the probability distribution π'_G. The general algorithm to update the BC scores after a batch β could be described as follows: First, we update the old shortest paths as described in Section 5.1. Then, we recompute an upper bound on $VD(G')$ in linear time, using the algorithms described in Section 4. Using $VD(G')$, we compute the number of samples $r(G')$ defined in (3.1). If $r(G') > r(G)$, we sample new $r(G') - r(G)$ additional paths using RK. For undirected graphs, we also propose two fully dynamic algorithms (one for weighted and one for unweighted graphs) to keep track of an upper bound on VD over time (Section 5.5). This saves additional time, allowing for a quick recomputation of the upper bound after the batch instead of recomputing it from scratch.

Algorithm 2: BC update after a batch β of edge insertions/weight decreases

---

    Input : Graph $G = (V, E)$, source node s, number of iterations r, batch β of
            edge insertions/weight decreases
    Output: New approximated BC values $\forall v \in V$

1 **for** $i \leftarrow 1$ **to** r **do**

2 $d_i^{old} \leftarrow d_{s_i}(t_i)$;

3 $\sigma_i^{old} \leftarrow \sigma_{s_i}(t_i)$;

4 $(d_{s_i}, \sigma_{s_i}) \leftarrow$ UpdateSSSP$(G, d_{s_i}, \sigma_{s_i}, \beta)$;

5   **if** $d_{s_i}(t_i) < d_i^{old}$ **or** $\sigma_{s_i}(t_i) \neq \sigma_i^{old}$ **then**

6   **foreach** $w \in p_{(i)}$ **do**

7 $\tilde{c}_B(w) \leftarrow \tilde{c}_B(w) - 1/r$;

8   **end**

9 $v \leftarrow t_i$;

10 $p_{(i)} \leftarrow$ empty list;

11 $P_{s_i}(v) \leftarrow \{u \in V | (u, v) \in E \wedge d_{s_i}(u) + \omega(u, v) = d_{s_i}(v)\}$;

12   **while** $P_{s_i}(v) \neq \{s_i\}$ **do**

13   sample $z \in P_{s_i}(v)$ with $\Pr = \sigma_{s_i}(z)/\sigma_{s_i}(v)$;

14 $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;

15   add z to $p_{(i)}$;

16 $v \leftarrow z$;

17   **end**

18   **end**

19 **end**

20 **return** $\{(v, \tilde{c}_B(v)), \ v \in V\}$

Notice that, if edge deletions are allowed, it is not sufficient to check whether the distance and the number of shortest paths between two nodes s and t has not changed (Line 5 of Algorithm 2), because they might remain unchanged even if the set of shortest paths is actually different. In this case, we always replace the old shortest path with a new one (we basically remove the *if* statement in Line 5).

In the following, we present the fully dynamic algorithms (for weighted and unweighted graphs) needed to update the shortest paths (updateSSSP in Algorithm 2) and the fully dynamic algorithm that recomputes an upper bound on VD for undirected graphs. Finally, we show how these algorithms can be combined to obtain an even faster algorithm (than the one described in this section) for dynamic BC approximation in undirected graphs (Section 5.6).

5.3. SSSP Update in Weighted Graphs

Our SSSP update is based on T-SWSF [3], which recomputes distances from a source node s after a batch β of weight updates (or edge insertions/deletions). For our purposes, we need two extensions of T-SWSF: an algorithm that also recomputes the number of shortest paths between s and the other nodes (updateSSSP-W) and one that also updates a VD approximation for the connected component of s (updateApprVD-W) (the latter is used in the fully dynamic VD approximation for undirected graphs, described in Section 5.5). The VD approximation is computed as described in Sections 3 and 4.2. Thus,

updateApprVD-W keeps track of the two maximum distances d' and d'' from s and the minimum edge weight $\underline{\omega}$.

We call *affected nodes* those nodes whose distance (or also whose number of shortest paths, in updateSSSP-W) from s has changed as a consequence of β. Basically, the idea is to put the set A of affected nodes w into a priority queue Q with priority $p(w)$ equal to the candidate distance of w. When w is extracted, if there is actually a path of length $p(w)$ from s to w, the new distance of w is set to $p(w)$, otherwise w is reinserted into Q with a higher candidate distance. In both cases, the affected neighbors of w are inserted into Q.

Algorithm 3 describes the SSSP update for weighted undirected graphs. The extension to directed graphs is trivial. The pseudocode updates both the VD approximation for the connected component of s and the number of shortest paths from s, so it basically includes both updateSSSP-W and updateApprVD-W. Initially, we scan the edges $e = \{u, v\}$ in β and, for each e, we insert the endpoint with greater distance from s into Q (w.l.o.g., let v be such endpoint). The priority $p(v)$ of v represents the *candidate* new distance of v. This is the minimum between the $d(v)$ and $d(u)$ plus the weight of the edge $\{u, v\}$. Notice that we use the expression "insert v into Q" for simplicity, but this can also mean update $p(v)$ if v is already in Q and the new priority is smaller than $p(v)$. When we extract a node w from Q, we have two possibilities: (i) there is a path of length $p(w)$ and $p(w)$ is actually the new distance, or (ii) there is no path of length $p(w)$ and the new distance is greater than $p(w)$. In the first case (Lines 9–23), we set $d(w)$ to $p(w)$ and insert the neighbors z of w such that $d(z) > d(w) + \omega(\{w, z\})$ into Q (to check if new shorter paths to z that go through w exist). In the second case (Lines 24–40), there is no shortest path between s and w known any longer, so that we set $d(w)$ to ∞. We compute $p(w)$ as $\min_{\{v,w\} \in E} d(v) + \omega(v, w)$ (the new candidate distance for w) and insert w into Q. Also, its neighbors could have lost one (or all of) their old shortest paths, so we insert them into Q as well. The update of $\underline{\omega}$ can be done while scanning the batch and of d' and d'' when we update $d(w)$.

The pseudocode also updates a global variable $vis(w)$ that keeps track, for each node w, of the number of source nodes from which w is reachable. This is necessary for the fully dynamic VD approximation and will be explained in Section 5.5. In particular, we decrease $vis(w)$ when updating $d(w)$ in the case that the old $d(w)$ was equal to ∞ (i.e., w has become reachable) and we decrease $vis(w)$ when we set $d(w)$ to ∞ (i.e., w has become unreachable). We update the number of shortest paths after updating $d(w)$, as the sum of the shortest paths of the predecessors of w (Lines 16–18). In updateApprVD-W, d' and d'' are recomputed while updating the distances (Line 10) and $\underline{\omega}$ is updated while scanning β (Line 5). Let $|\beta|$ represent the cardinality of β and let $||A||$ represent the sum of the nodes in A and of the edges that have at least one endpoint in A. Then, the following complexity derives from feeding Q with the batch and inserting into/extracting from Q the affected nodes and their neighbors.

Lemma 5.3. *The time required by* updateApprVD-W *(*updateSSSP-W*) to update the distances and* $\tilde{VD}$ *(the number of shortest paths) is* $O(|\beta| \log |\beta| + ||A|| \log ||A||)$.

Proof. In the initial scan of the batch (Lines 2–4), we scan the nodes of the batch and insert the affected nodes into Q (or update their value). This requires at most one heap operation (insert or decrease-key) for each element of β, therefore, $O(|\beta| \log |\beta|)$ time. When we extract a node w from Q, we have two possibilities: (i) $con(w) = p(w)$ (Lines 9–23) or (ii) $con(w) > p(w)$ (Lines 24–40). In the first case, we scan the neighbors of w and perform at

most one heap operation for each of them (Lines 19–21). In the second case, we scan only
if $d(w) \neq \infty$. Therefore, we can perform up to one heap operation per incident edge of w,
for each extraction of w in which $d(w) \neq \infty$ or $con(w) = p(w)$. How many times can an
affected node w be extracted from Q with $d(w) \neq \infty$ or $con(w) = p(w)$? If the first time
we extract w, $con(w)$ is equal to $p(w)$ (case (i)), then the final value of $d(w)$ is reached
and w is not inserted into Q anymore. If the first time we extract w, $con(w)$ is greater than
$p(w)$ (case (ii)), then w can be inserted into the queue again. However, its distance is set to

---

Algorithm 3: SSSP update for weighted graphs (updateSSSP-W)

    Input : Graph $G = (V, E)$, vector d of distances from s, vector σ of number of shortest
             paths from s, batch β
    Output: New values of $d(v)$ and $\sigma(v)$ $\forall v \in V$
1   $Q \leftarrow$ empty priority queue;
2   **foreach** $e = \{u, v\} \in \beta, d(u) < d(v)$ **do**
3 | $Q \leftarrow \texttt{insertOrDecreaseKey}(v, p(v) = \min\{d(u) + \omega(\{u, v\}), d(w)\})$;
4   **end**
5   $\underline{\omega} \leftarrow \min\{\underline{\omega}, \ \omega(e) : e \in \beta\}$;
6   **while** *there are nodes in Q* **do**
7 | $\{w, p(w)\} \leftarrow \texttt{extractMin}(Q)$;
8 | $con(w) \leftarrow \min_{z:(z,w) \in E} d(z) + \omega(z, w)$;
9 | **if** $con(w) = p(w)$ **then**
10 | | update d' and d'';
11 | | **if** $d(w) = \infty$ **then**
12 | | | $vis(w) \leftarrow vis(w) + 1$;
13 | | **end**
14 | | $d(w) \leftarrow p(w); \sigma(w) \leftarrow 0$;
15 | | **foreach** *incident edge* (z, w) **do**
16 | | | **if** $d(w) = d(z) + \omega(z, w)$ **then**
17 | | | | $\sigma(w) \leftarrow \sigma(w) + \sigma(z)$;
18 | | | **end**
19 | | | **if** $d(z) \geq d(w) + \omega(z, w)$ **then**
20 | | | | $Q \leftarrow \texttt{insertOrDecreaseKey}(z, p(z) = d(w) + \omega(z, w))$;
21 | | | **end**
22 | | **end**
23 | **end**
24 | **else**
25 | | **if** $d(w) \neq \infty$ **then**
26 | | | $vis(w) \leftarrow vis(w) - 1$;
27 | | | **if** *vis(w)=0* **then**
28 | | | | insert w into U;
29 | | | **end**
30 | | | **if** $con(w) \neq \infty$ **then**
31 | | | | $Q \leftarrow \texttt{insertOrDecreaseKey}(w, p(w) = con(w))$;
32 | | | | **foreach** *incident edge* (z, w) **do**
33 | | | | | **if** $d(z) = d(w) + \omega(w, z)$ **then**
34 | | | | | | $Q \leftarrow \texttt{insertOrDecreaseKey}(z, p(z) = d(w) + \omega(z, w))$;
35 | | | | | **end**
36 | | | | **end**
37 | | | | $d(w) \leftarrow \infty$;
38 | | | **end**
39 | | **end**
40 | **end**
41   **end**

∞ and, therefore, no additional operations are performed, until $d(w)$ becomes less than ∞. But this can happen only in case (i), after which $d(w)$ reaches its final value. To summarize, each affected node w can be extracted from Q with $d(w) \neq \infty$ or $con(w) = p(w)$ at most twice and, every time this happens, at most one heap operation per incident edge of w is performed. The complexity is therefore $O(|\beta| \log |\beta| + ||A|| \log ||A||)$. $\qquad\square$

5.4. SSSP Update in Unweighted Graphs

For unweighted graphs, we basically replace the priority queue Q of updateApprVD-W and updateSSSP-W with a list of queues. Each queue represents a *level* (distance from s) from 0 (to which only the source belongs) to the maximum distance d'. The levels replace the priorities and, in this case, also represent the candidate distances for the nodes. Algorithm 4 describes the pseudocode for unweighted graphs. As in Algorithm 3, we first scan the batch (Lines 3–5) and insert the nodes in the queues. Then (Lines 6–44), we scan the queues in order of increasing distance from s, in a fashion similar to that of a priority queue. In order not to insert a node in the queues multiple times, we use colors: initially we set all the nodes to white and then we set a node w to black only when we find the final distance of w (i.e., when we set $d(w)$ to k) (Line 15). Black nodes extracted from a queue are then skipped (Line 10). At the end we reset all nodes to white. The replacement of the priority queue with the list of queues decreases the complexity of the SSSP update algorithms for unweighted graphs, which we call updateApprVD-U and updateSSSP-U, in analogy with those for weighted graphs.

Lemma 5.4. *The time required by* updateApprVD-U *(*updateSSSP-U*) to update the distances and $\tilde{V}D$ (the number of shortest paths) is $O(|\beta| + ||A|| + d_{\max})$, where $d_{\max}$ is the maximum distance from s reached during the update.*

Proof. The complexity of the initialization (Lines 3–5) of Algorithm 4 is $O(|\beta|)$, because we have to scan the batch. In the main loop (Lines 6–44), we scan the list of all queues, whose final size is $d_{\max}$. Every time we extract a node w whose color is not black, we scan all the incident edges, therefore, this operation is linear in the number of neighbors of w. If the first time we extract w (say at level k) $con(w)$ is equal to k, then w will be set to black and will not be scanned again. If the first time we extract w, $con(w)$ is, instead, greater than k, w will be inserted into the queue at level $con(w)$ (if $con(w) < \infty$). Also, other inconsistent neighbors of w might insert w in one of the queues. However, after the first time w is extracted, its distance is set to ∞, so its neighbors will not be scanned unless $con(w) = k$, in which case they will be scanned again, but for the last time, since w will be set to black. To summarize, each affected node and its neighbors can be scanned at most twice. The complexity of the algorithm is, therefore, $O(|\beta| + ||A|| + d_{\max})$. $\qquad\square$

5.5. Fully Dynamic VD Approximation

The algorithm keeps track of a VD approximation for an undirected graph G, i.e., for each connected component of G. It is composed of two phases. In the initialization, we compute an SSSP from a source node s_i for each connected component C_i. During the SSSP search from s_i, we also compute a VD approximation $\tilde{V}D_i$ for C_i, as described in Sections 3 and 4.2. In the update, we recompute the SSSPs and the VD approximations with updateApprVD-W (or updateApprVD-U). Because components might split or

---

Algorithm 4: SSSP update for unweighted graphs (updateSSSP-U)

Input : Graph $G = (V, E)$, vector d of distances from s, vector σ of number of shortest
        paths from s, batch β

Output: New values of $d(v)$ and $\sigma(v)$ $\forall v \in V$

1 **Assumption:** $color(w) = white \quad \forall w \in V$;

2 $Q[] \leftarrow$ array of empty queues;

3 **foreach** $e = \{u, v\} \in \beta, d(u) < d(v)$ **do**

4 | $k \leftarrow d(v) + 1$; enqueue $v \to Q[k]$;

5 **end**

6 $k \leftarrow 1$;

7 **while** *there are nodes in $Q[j]$, $j \geq k$* **do**

8 | **while** $Q[k] \neq \emptyset$ **do**

9 | | dequeue $w \leftarrow Q[k]$;

10 | | **if** $color(w) = black$ **then** continue;

11 | | $con(w) \leftarrow \min_{z:(z,w) \in E} d(z) + 1$;

12 | | **if** $con(w) = k$ **then**

13 | | | update d' and d'';

14 | | | **if** $d(w) = \infty$ **then** $vis(w) \leftarrow vis(w) + 1$;

15 | | | $d(w) \leftarrow k$; $\sigma(w) \leftarrow 0$; $color(w) \leftarrow black$;

16 | | | **foreach** *incident edge (z, w)* **do**

17 | | | | **if** $d(w) = d(z) + 1$ **then**

18 | | | | | $\sigma(w) \leftarrow \sigma(w) + \sigma(z)$;

19 | | | | **end**

20 | | | | **if** $d(z) > k$ **then**

21 | | | | | enqueue $z \to Q[k + 1]$;

22 | | | | **end**

23 | | | **end**

24 | | **end**

25 | | **else**

26 | | | **if** $d(w) \neq \infty$ **then**

27 | | | | $d(w) \leftarrow \infty$;

28 | | | | $vis(w) \leftarrow vis(w) - 1$;

29 | | | | **if** $vis(w)=0$ **then**

30 | | | | | insert w into U;

31 | | | | **end**

32 | | | | **if** $con(w) \neq \infty$ **then**

33 | | | | | enqueue $w \to Q[con(w)]$;

34 | | | | | **foreach** *incident edge (z, w)* **do**

35 | | | | | | **if** $d(z) > k$ **then**

36 | | | | | | | enqueue $z \to Q[k + 1]$;

37 | | | | | | **end**

38 | | | | | **end**

39 | | | | **end**

40 | | | **end**

41 | | **end**

42 | **end**

43 | $k \leftarrow k + 1$;

44 **end**

45 Set to white all the nodes that have been in Q;

---

merge, we might need to compute new approximations, in addition to updating the old ones. To do this, for each node, we keep track of the number $vis(v)$ of times it has been visited. This way we discard source nodes that have already been visited and compute a new approximation for components that have become unvisited. Algorithm 5 describes the initialization. Initially, we put all the nodes in a queue and compute an SSSP from the nodes we extract (Line 18). During the SSSP search, we increase the number of visits $vis(v)$ for all the nodes we traverse (Line 24). When extracting the nodes, we skip those that have already been visited (Line 8): this avoids computing multiple approximations for the same component.

   In the update (Algorithm 6), we recompute the SSSPs and the VD approximations with updateApprVD-W (or updateApprVD-U) (Line 7). Since components might split, we might need to add VD approximations for some new subcomponents, in addition to recomputing the old ones. Also, if components merge, we can discard the superfluous approximations. To do this, we update $vis(v)$ for each node within updateApprVD-W (or updateApprVD-U). Before the update, all the nodes are visited exactly once. While updating an SSSP from s_i, we increase (decrease) by one $vis(v)$ of the nodes v that become reachable (unreachable) from s_i. This way, we can skip the update of the SSSPs from nodes that have already been visited (Line 8). After the update, for all nodes v that have become unvisited ($vis(v) = 0$), we compute a new VD approximation from scratch (Lines 11–18). The complexity of the update of the VD approximation derives from the $\tilde{VD}$ update in the single components, using updateApprVD-W and updateApprVD-U.

Theorem 5.5. *The time required to update the VD approximation is $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ in weighted graphs and $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d_{max}^{(i)})$ in unweighted graphs, where n_c is the number of components in G before the update and $A^{(i)}$ is the sum of affected nodes in C_i and their incident edges.*

Proof. In the first part (Lines 2–9 of Algorithm 6), we update an SSSP with updateApprVD-W or updateApprVD-U for each source node s_i such that $vis(s_i)$ is not greater than 1. Therefore, the complexity of the first part is $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ in weighted graphs and $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d_{max}^{(i)})$ in unweighted, by Lemmas 5.3 and 5.4. Only some of the affected nodes (those whose distance from a source node becomes ∞) are inserted into the queue U. Therefore, the cost of scanning U in Lines 11–18 is $O(\sum_{i=1}^{n_c} ||A^{(i)}||)$. New SSSP searches are computed for new components that are not covered by the existing source nodes any longer. However, such searches involve only the affected nodes, and each affected node (and its incident edges) is scanned at most once during the search. Therefore, the total cost is $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ for weighted graphs and $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d_{max}^{(i)})$ for unweighted graphs. □

Lemma 5.6. *At the end of Algorithm 5, $vis(v) = 1$ for all $v \in V$, and exactly one VD approximation is computed for each connected component of G.*

Proof. Let v be any node. Then v must be scanned by *at least* one source node s_i in the while loop (Lines 6–13): In fact, either v is visited by some s_i before v is extracted from U, or $vis(v) = 0$ at the moment of the extraction and v becomes a source node itself. This implies that $vis(v) \geq 1$, $\forall v \in V$. However, $vis(v)$ cannot be greater than 1. In fact, let us assume by contradiction that $vis(v) > 1$. This means that there are at least two source

nodes s_i and s_j ($i < j$, w.l.o.g.) that are in the same connected component as v. Then, s_i and s_j are also in the same connected component, and s_j is visited during the SSSP search from s_i. Then, $vis(s_j) = 1$ before s_j is extracted from U and s_j cannot be a source node. Therefore, $vis(v)$ is exactly equal to 1 for each $v \in V$, which means that exactly one VD approximation is computed for each connected component of G.

□

Proposition 5.7. *Let $\mathcal{C}' = \{C'_1, \ldots, C'_{n'_c}\}$ be the set of connected components of G after the update. Algorithm 6 updates or computes exactly one VD approximation for each $C'_i \in \mathcal{C}'$.*

Proof. Let $\mathcal{C} = \{C_1, \ldots, C_{n_c}\}$ be the set of connected components before the update. Let us consider three basic cases (then it is straightforward to see that the proof holds also for combinations of these cases): (i) $C_i \in \mathcal{C}$ is also a component of $\mathcal{C}'$, (ii) $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$ merge into one component C'_k of $\mathcal{C}'$, (iii) $C_i \in \mathcal{C}$ splits into two components C'_j and C'_k of $\mathcal{C}'$. In case (i), the VD approximation of C_i is updated exactly once in the for loop (Lines 2–9). In case (ii), (assuming $i < j$, w.l.o.g.) the VD approximation of C'_k is updated in the for loop from the source node $s_i \in C_i$. In its SSSP search, s_i also visits $s_j \in C_j$, increasing $vis(s_j)$. Therefore, s_j is skipped and exactly one VD approximation is computed for C'_k. In case (iii), the source node $s_i \in C_i$ belongs to one of the components (say C'_j) after the update. During the for loop, the VD approximation is computed for C'_j via s_i. Also, for all the nodes v in C'_k, $vis(v)$ is set to 0 and v is inserted into U. Then, some source node $s'_k \in C'_k$ must be extracted from U in Line 12 and a VD approximation is computed for C'_k. Because all the nodes in C'_k are set to be visited during the search, no other VD approximations are computed for C'_k.

□

5.6. Combined Dynamic BC Approximation

Let G be an undirected graph with n_c connected components. In Section 5.2, we described an algorithm to update the betweenness approximations in fully dynamic graphs. If the graph is undirected, we can use the fully dynamic VD approximation to recompute $\tilde{VD}$ after a batch, instead of recomputing it from scratch. Then, we could update the r sampled paths with updateSSSP and, if $\tilde{VD}$ (and therefore r) increases, we could sample new paths. However, since updateSSSP and updateApprVD share most of the operations, we can "merge" them and update at the same time the shortest paths from a source node s and the VD approximation for the component of s. We call this hybrid function updateSSSPVD. Instead of storing and updating n_c SSSPs for the VD approximation and r SSSPs for the BC scores, we recompute a VD approximation for each of the r samples while recomputing the shortest paths with updateSSSPVD. This way, we do not need to compute an additional SSSP for the components covered by the r sampled paths (i.e., the components in which the paths lie), saving time and memory. Only for components that are not covered by any of them (if they exist), we compute and store a separate VD approximation. We refer to such components as R' (and to $|R'|$ as r').

In the initialization (Algorithm 7), we first compute the r SSSP, like in RK (Lines 4–18). However, we also check which nodes have been visited, as in Algorithm 5. While we compute the r SSSPs, in addition to the distances and number of shortest paths, we also compute a VD approximation for each of the r source nodes and increase $vis(v)$ of all the nodes we visit during the sources with initApprVD (Line 8). Because it is possible that the r shortest paths do not cover all the components of G, we compute an additional VD

approximation for nodes in the unvisited components, such as in Algorithm 5 (Lines 21–28). Basically, we can divide the SSSPs into two sets: the set R of SSSPs used to compute the r shortest paths and the set R' of SSSPs used for a VD approximation in the components that were not scanned by the initial R SSSPs. We call r' the number of the SSSPs in R'.

Algorithm 5: Dynamic VD approximation (initialization)

    Input : Graph $G = (V, E)$
    Output: Upper bound on VD

1   $U \leftarrow []$;
2   **foreach** *node* $v \in V$ **do**
3 | $vis(v) \leftarrow 0$; insert v into U;
4   **end**
5   $i \leftarrow 1$;
6   **while** $U \neq \emptyset$ **do**
7 | extract s from U;
8 | **if** $vis(s) = 0$ **then**
9 | $s_i \leftarrow s$;
10 | $\check{\mathsf{VD}}_i \leftarrow \texttt{initApprVD}(G, s_i)$;
11 | $i \leftarrow i + 1$;
12 | **end**
13   **end**
14   $n_C \leftarrow i - 1$;
15   $\tilde{VD} \leftarrow \max_{i=1,\dots,n_C} \tilde{VD}_i$;
16   **return** $\tilde{VD}$;

    // initApprVD computes $\check{VD}$ and adds 1 to $vis(v)$ of the nodes it
        visits

17   **Function** $\texttt{initApprVD}(G, s)$
18 | $\texttt{SSSP}(G, s)$;
19 | $d' \leftarrow \max\{d(s, u) | u \in V, d(u, v) \neq \infty\}$;
20 | $d'' \leftarrow \max\{d(s, v) | v \in V, v \neq u, d(s, v) \neq \infty\}$;
21 | $\underline{\omega} \leftarrow \min\{\omega(x, y) | (x, y) \in E\}$;
22 | $\tilde{VD} \leftarrow 1 + \frac{d' + d''}{\underline{\omega}}$;
23 | **foreach** *node* $w \in V$ *s.t.* $d(s, w) \neq \infty$ **do**
24 | $vis(w) \leftarrow vis(w) + 1$;
25 | **end**
26 | **return** $\tilde{VD}$;

The BC update after a batch is described in Algorithm 8. First (Lines 2–19), we recompute the shortest paths like in Algorithm 2: we update the SSSPs from each source node s in R and we replace the old shortest path with a new one (subtracting $1/r$ from the nodes in the old shortest path and adding $1/r$ to those in the new shortest path). To update the SSSPs, we use the fully-dynamic **updateSSSPVD** that updates also the VD approximation and keeps track of the nodes that become unvisited. Then (Lines 24–31), we add a new SSSP to R' for each component that has become unvisited (by both R and R'). After this, we have at least a VD approximation for each component of G. We take the maximum over all these approximations and recompute the number of samples r (Lines 32–33). If r has increased, we need to sample new paths and, therefore, new SSSPs to add to R. Finally, we normalize the BC scores, i.e., we multiply them by the old value of r divided by the new value of r (Line 37). We refer to the algorithm for unweighted graphs as DA and the one for weighted as DAW. The difference between DA and DAW is the way

the SSSPs and the VD approximation are updated: in DA we use updateApprVD-U and in DAW updateApprVD-W.

Theorem 5.8. *Algorithm 8 preserves the guarantee on the maximum absolute error, i.e., naming $c'_B(v)$ and $\tilde{c}'_B(v)$ the new exact and approximated BC values, respectively,* $\Pr(\exists v \in V \; s.t. \; |c'_B(v) - \tilde{c}'_B(v)| > \epsilon) < \delta.$

Proof. Let G be the old graph and G' be the modified graph after the batch of edge updates. Let p'_{xy} be a shortest path of G' between nodes x and y. To prove the theoretical guarantee, we need to prove that the probability of any sampled path $p'_{(i)}$ is equal to p'_{xy} (i.e., that the algorithms adds $1/r'$ to the nodes in p'_{xy}) is $\frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$. Algorithm 8 replaces the first r shortest paths with other shortest paths $p'_{(1)}, \ldots, p'_{(r)}$ between the same node pairs (Lines 12–18), using Algorithm 2, for which we already proved that $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$ (Lemma 5.1). The additional Δr shortest paths (Line 35) are recomputed from scratch with RK, therefore, also in this case $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$ by Lemma 7 of [25]. □

Algorithm 6: Dynamic VD approximation (updateApprVD)

> **Input** : Graph $G = (V, E)$, vector vis
> **Output:** New VD approximation

1 $U \leftarrow []$;
2 **foreach** s_i **do**
3   **if** $vis(s_i) > 1$ **then**
4   remove s_i and $\tilde{VD}_i$; decrease n_C;
5   **end**
6   **else**
        // updateApprVD updates vis, inserts all v for which $vis(v) = 0$
            into U and recomputes a VD approximation $\tilde{VD}_i$
7          $\tilde{VD}_i \leftarrow$ updateApprVD(G, s_i) ;
8   **end**
9 **end**
10 $i \leftarrow n_C$;
11 **while** $U \neq \emptyset$ **do**
12   extract s' from U;
13   **if** $vis(s') = 0$ **then**
14         $s'_i \leftarrow s'$;
15         $\tilde{VD}_i \leftarrow$ initApprVD(G, s'_i);
16         $i \leftarrow i + 1; n_C \leftarrow n_C + 1$;
17   **end**
18 **end**
19 reset $vis(v)$ to 1 for nodes v such that $vis(v) > 1$;
20 $\tilde{VD} \leftarrow \max_{i=1,\ldots,n_C} \tilde{VD}_i$;
21 **return** $\tilde{VD}$

5.7. Complexity of the Dynamic BC Algorithms

In this section we presented different algorithms for updating BC approximations after batches of edge updates. Algorithm 2 can be used on graphs for which we can be sure that the vertex diameter cannot increase after a batch of edge updates. This includes, for example,

unweighted connected graphs on which only edge insertions are allowed. We refer to the unweighted version of this algorithm as incremental approximation (IA) and to the weighted version as incremental approximation weighted (IAW). On general directed graphs, we can use the algorithms described in Section 5.2. We name the unweighted version dynamic approximation directed (DAD) and the weighted one DADW. Finally, for undirected graphs, we can use the optimized algorithms presented in Section 5.6 (Algorithm 8), to which we refer as DA and DAW. Theorem 5.9 presents the complexities of all the BC update algorithms. In the following, we name $||A^{(i)}||$ the sum of affected nodes and their incident edges in the ith sampled SSSP. We also name r the number of samples. In case we need to sample new additional paths after the update (in DAD, DADW, DA and DAW), we refer to the difference between the value of r before and after the batch as Δr. In DA and DAW, we call r' the number of additional samples necessary for the VD approximation.

Theorem 5.9. *Given a graph $G = (V, E)$ with n nodes and m edges, the times required by the different algorithms to update the BC approximations after a batch β are as follows:*

(i)   IA: $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})$
(ii) IAW: $O(r \cdot |\beta| \log |\beta| + \sum_{i=1}^{r} ||A^{(i)}|| \log ||A^{(i)}||)$
(iii) DAD: $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)}) + (\Delta r + 1)(n + m))$
(iv) DADW: $O((r \cdot |\beta| \log |\beta| + \sum_{i=1}^{r} ||A^{(i)}|| \log ||A^{(i)}|| + (\Delta r + 1)(n \log n + m))$
(v)   DA: $O((r + r')|\beta| + \sum_{i=1}^{r+r'}(||A^{(i)}|| + d_{\max}^{(i)}) + \Delta r(n + m))$
(vi) DAW: $O((r + r')|\beta| \log |\beta| + \sum_{i=1}^{r+r'} ||A^{(i)}|| \log ||A^{(i)}|| + \Delta r(n \log n + m))$

Proof. We prove each case separately.

(i)   IA updates each sampled path with updateSSSP-U. Therefore, the total complexity is the sum of the times required to update each of the r paths, i.e., $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)}))$.

(ii) Same as (i), with the only difference, that we use updateSSSP-W for weighted graphs.

(iii) In DAD, we need to update the existing r samples, exactly as in IA. In addition to that, we might need to sample new Δr additional paths using a BFS, whose complexity is $O(n + m)$. Also, we need to recompute the upper bound on VD, whose complexity is also $O(n + m)$ (see Section 4.1). Therefore, in this case, we have to add an additional $O((\Delta r + 1)(n + m))$ term to the complexity of IA.

(iv) Similarly to (iii), we need to sample Δr additional paths, but in weighted graphs the cost of an SSSP is $O(n \log n + m)$. Also, the VD approximation described in Section 4.3 requires $O(n \log n + m)$ time.

(v)   Let $\Delta r'$ be the difference between the values of r' before and after the batch. After processing β, we might need to sample new paths for the betweenness approximation ($\Delta r > 0$) and/or sample paths in new components that are not covered by any of the sampled paths ($\Delta r' > 0$). Then, the complexity for the betweenness approximation update is $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})) + O(\Delta r(n + m))$. The VD update requires $O(r' \cdot |\beta| + \sum_{i=1}^{r'}(||A^{(i)}|| + d_{\max}^{(i)}))$ to update the VD approximation in the already covered components and $\sum_{i=1}^{\Delta r}(|V_i| + |E_i|)$ for the new ones, where V_i and E_i are nodes and edges of the ith component, respectively. From this derives the total complexity.

(vi) Same as (v), using updateSSSP-W, approxVD-W.

$\square$

Algorithm 7: BC initialization

Input : Graph $G = (V, E)$, source node s, number of iterations r, batch β
Output: Approximated BC values $\forall v \in V$

1 foreach *node* $v \in V$ **do**
2 $\quad$ $\tilde{c}_B(v) \leftarrow 0$; $vis(v) \leftarrow 0$;
3 end
4 $\check{VD} \leftarrow$ getApproxVertexDiameter(G);
5 $r \leftarrow (c/\epsilon^2)(\lfloor \log_2(\check{VD} - 2) \rfloor + \ln(1/\delta))$;
6 for $i \leftarrow 1$ **to** r **do**
7 $\quad$ $(s_i, t_i) \leftarrow$ sampleUniformNodePair(V);
8 $\quad$ $\check{VD}_i \leftarrow$ initApprVD$(G, s_i,)$;
9 $\quad$ $v \leftarrow t_i$;
10 $\quad$ $p_{(i)} \leftarrow$ empty list;
11 $\quad$ $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
12 $\quad$ **while** $P_{s_i}(v) \neq \{s_i\}$ **do**
13 $\quad\quad$ sample $z \in P_{s_i}(v)$ with probability $\sigma_{s_i}(z)/\sigma_{s_i}(v)$;
14 $\quad\quad$ $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
15 $\quad\quad$ add $z \rightarrow p_{(i)}$; $v \leftarrow z$;
16 $\quad\quad$ $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
17 $\quad$ **end**
18 end
19 $U \leftarrow V$;
20 $i \leftarrow r + 1$;
21 while $U \neq \emptyset$ **do**
22 $\quad$ extract s' from U;
23 $\quad$ **if** $vis(s') = 0$ **then**
24 $\quad\quad$ $s_i' \leftarrow s'$;
25 $\quad\quad$ $\check{VD}_i \leftarrow$ initApprVD(G, s_i');
26 $\quad\quad$ $i \leftarrow i + 1$;
27 $\quad$ **end**
28 end
29 $r' \leftarrow r - i - 1$;
30 return $\{(v, \tilde{c}_B(v)) : v \in V\}$

Notice that, if $\check{VD}$ does not increase, $\Delta r = 0$ and the complexities of DA and DAD (DAW and DADW, respectively) are the same as the only incremental algorithm IA (IAW, respectively). This case includes, for example, connected graphs subject to a batch of only edge insertions, or any batch that neither splits the graph into more components nor increases VD. Also, notice that in the worst case the complexity can be as bad as recomputing from scratch, or even slightly worse. Indeed, $||A^{(i)}||$ can be as large as m, for $i = 1, \ldots, r$, and $d_{max}^{(i)}$ can be as large as n. Assuming $\beta = \Theta(m)$ as a worst-case batch size, the running times of IA and IAW are then $O(r \cdot (m + n))$ and $O(r \cdot (m \log m))$, respectively. Analogously, the complexities of DAD and DADW are $O((r + \Delta r) \cdot (n + m))$ and $O((r + \Delta r) \cdot (m \log m))$, where we recall that $(r + \Delta r)$ is the number of samples required after the batch. In the optimized versions DA and DAW, the worst-case running time is even longer: $O((r + \Delta r + n_c) \cdot (n + m))$ and $O((r + \Delta r + n_c) \cdot (m \log m))$, where n_c is the number of connected components of the graph. However, these worst-case running times are not observed in our experiments. Indeed, in the next section, we will show that our dynamic algorithms perform very well in practice. Also, notice that no dynamic SSSP (and so, probably, also no BC approximation) algorithm exists that is asymptotically faster than recomputation on all graphs.

Algorithm 8: Dynamic update of BC approximation (DA)

Input : Graph $G = (V, E)$, source node s, number of iterations r, batch β
Output: New approximated BC values $\forall v \in V$

1 $U \leftarrow []$;
2 **for** $i \leftarrow 1$ **to** r **do**
3       $d_i^{old} \leftarrow d_{s_i}(t_i)$;
4       $\sigma_i^{old} \leftarrow \sigma_{s_i}(t_i)$;
     // updateSSSPVD updates vis, inserts all $v : vis(v) = 0$ into U and
         updates the VD approximation
5       $\tilde{VD}_i \leftarrow$ updateSSSPVD(G, s_i, β);
     // we replace the shortest path between s_i and t_i
6   **foreach** $w \in p_{(i)}$ **do**
7           $\tilde{c}_B(w) \leftarrow \tilde{c}_B(w) - 1/r$;
8   **end**
9       $v \leftarrow t_i$;
10       $p_{(i)} \leftarrow$ empty list;
11       $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
12   **while** $P_{s_i}(v) \neq \{s_i\}$ **do**
13   sample $z \in P_{s_i}(v)$ with probability $= \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
14           $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
15   add z to $p_{(i)}$;
16           $v \leftarrow z$;
17           $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
18   **end**
19 **end**
20 **for** $i \leftarrow r + 1$ **to** $r + r'$ **do**
21       $\tilde{VD}_i \leftarrow$ updateApprVD(G, s_i, β);
22 **end**
23 $i \leftarrow r + r' + 1$;
24 **while** $U \neq \emptyset$ **do**
25   extract s' from U;
26   **if** $vis(s') = 0$ **then**
27           $s_i' \leftarrow s'$;
28           $\tilde{VD}_i \leftarrow$ initApprVD(G, s_i');
29           $i \leftarrow i + 1$; $r' \leftarrow r' + 1$;
30   **end**
31 **end**
// compute the maximum over all the VD_i computed by updateApprVD
32 $\tilde{VD} \leftarrow \max_{i=1,...,r+r'} \tilde{VD}_i$;
33 $r_{new} \leftarrow (c/\epsilon^2)(\lfloor \log_2(\tilde{VD} - 2) \rfloor + \ln(1/\delta))$;
34 **if** $r_{new} > r$ **then**
35   sample new paths;
36   **foreach** $v \in V$ **do**
37           $\tilde{c}_B(v) \leftarrow \tilde{c}_B(v) \cdot r/r_{new}$
38   **end**
39       $r \leftarrow r_{new}$;
40 **end**
41 **return** $\{(v, \tilde{c}_B(v)) : v \in V\}$

Graph	Type	Nodes	Edges	Type
`ca-GrQc`	coauthorship	5 242	14 496	Unweighted, Undirected
`p2p-Gnutella09`	file sharing	8 114	26 013	Unweighted, Directed
`ca-HepTh`	coauthorship	9 877	25 998	Unweighted, Undirected
`PGPgiantcompo`	social / web of trust	10 680	24 316	Unweighted, Undirected
`as-22july06`	internet	22 963	48 436	Unweighted, Undirected

Table I Overview of small real-world networks used in the experiments.

6. EXPERIMENTS

Implementation and settings. For an experimental comparison, we implemented our six approaches IA, IAW, DAD, DADW, DA, DAW, as well as the static approximation RK [25]. In addition to that, we implemented the static exact BA and the dynamic exact algorithms GMB [16] and KDB [18], which were shown to have the best speedups on unweighted graphs. For a comparison on weighted graphs, we also implemented the algorithm by [22] (NPR). In the implementation of RK, we used the optimization proposed in [25], stopping all the SSSP searches once the target node had been found. Also, we computed the number of samples, using our new bounds on VD the presented in Section 4. We implemented all algorithms in C++, building on the open-source *NetworKit* framework [27]. In all experiments, we fix δ to 0.1 while the error bound ϵ varies. The machine we employ is used for its 256 GB RAM–the comparison to exact approaches requires a substantial amount of memory. Of the machine's 2 x 8 Intel(R) Xeon(R) E5-2680 cores running at 2.7 GHz, we use only one; all computations are sequential to make the comparison to previous work more meaningful.

Datasets. We use both real-world and synthetic networks. For our experiments on the accuracy and for comparison with the exact algorithms, we use relatively small networks, on which the nonscalable algorithms can also be executed. These networks are summarized in Table I and are publicly available from the collection compiled for the 10th DIMACS Challenge[1] and from the SNAP[2] collection. Due to a shortage of actual dynamic networks in this size range, we simulate dynamics by removing a small fraction of random edges and adding them back in batches. We also use synthetic networks obtained with the Dorogovtsev–Mendes generator, a simple model for networks with power-law degree distribution [11].

To compare the running times of the scalable algorithms (RK and our dynamic algorithms), we use real dynamic networks, taken from The Koblenz Network Collection (KONECT) [19] and summarized in Table II. All the edges of the KONECT graphs are characterized by a time of arrival. In the case of multiple edges between two nodes, we extract two versions of the graph: one unweighted, where we ignore additional edges, and one weighted, where we replace the set E_{st} of edges between two nodes with an edge of weight $1/|E_{st}|$ (more tightly coupled nodes receive a smaller distance). In our experiments, we let the batch size vary from 1 to 1024 and for each batch size, we average the running times over 10 runs. Because the networks do not include edge deletions, we implement additional simulated dynamics. In particular, we consider the following experiments.

[1]`http://www.cc.gatech.edu/dimacs10/downloads.shtml` [2].
[2]`http://snap.stanford.edu`

Graph	Type	Nodes	Edges	Type
repliesDigg	communication	30,398	85,155	Weighted
emailSlashdot	communication	51,083	116,573	Weighted
emailLinux	communication	63,399	159,996	Weighted
facebookPosts	communication	46,952	183,412	Weighted
emailEnron	communication	87,273	297,456	Weighted
facebookFriends	friendship	63,731	817,035	Unweighted
arXivCoauthors	coauthorship	28,093	3,148,447	Unweighted
englishWikipedia	hyperlink	1,870,709	36,532,531	Unweighted

Table II Overview of real dynamic graphs used in the experiments.[3]

(i) *Real dynamics.* We remove the x edges with the highest timestamp from the network and we insert them back in batches, in the order of timestamps. (ii) *Random insertions and deletions.* We remove x edges from the graph, chosen uniformly at random. To create batches of both edge insertions and deletions, we add back the deleted edges with probability $1/2$ and delete other random edges with probability $1/2$. (iii) *Random weight changes.* In weighted networks, we choose x edges uniformly at random and we multiply their weight by a random value in the interval $(0, 2)$.

To study the scalability of the methods, we also use synthetic graphs obtained with a generator based on a unit-disk graph model in hyperbolic geometry [28], where edge insertions and deletions are obtained by moving the nodes in the hyperbolic plane. The networks produced by the model were shown to have many properties of real complex networks, such as small diameter and power-law degree distribution (for details and references the interested reader is referred to [28]). We generate seven networks, with $|E|$ ranging from about $2 \cdot 10^4$ to about $2 \cdot 10^7$ and $|V|$ approximately equal to $|E|/10$.

We also compare our new upper bound on VD for directed graphs presented in Section 4.1 with the one used in RK. For this, we use directed real-world graphs of different sizes taken from the SNAP collection.

6.1. Accuracy

We consider the accuracy of the approximated centrality scores both in terms of absolute error and, more importantly, the preservation of the ranking order of nodes. Because we only replace the samples without changing their number, our dynamic algorithm has exactly the same accuracy as RK. The authors of [25] also study the behavior of RK experimentally, considering the average and maximum estimation error on a small set of real graphs. We study the experimental errors on additional graphs. For our tests we use the networks summarized in Table I and Dorogovtsev–Mendes graphs of several sizes. Our results confirm those of [25] in the sense that the measured absolute errors are *always* below the guaranteed maximum error ϵ and the measured average error is often orders of magnitude smaller than ϵ. Table III shows the measured errors for the real networks. We also study the *relative rank error* introduced by [14] (i.e., $\max\{\rho, 1/\rho\}$, denoting ρ the ratio between the estimated rank and the true rank), which we consider the most relevant measure of the quality of the approximations. Figure 4 shows the results for `PGPgiantcompo`, a

---

[3]Taken from http://konect.uni-koblenz.de/.

Graph	ca-GrQc	ca-HepTh	PGP giantcompo	as-22july06	p2p-Gnutella09
max. error ($\epsilon = 0.1$)	1.70e-02	1.69e-02	3.10e-02	3.22e-02	1.56e-02
max. error ($\epsilon = 0.05$)	9.12e-03	7.62e-03	1.38e-02	1.60e-02	6.55e-03
max. error ($\epsilon = 0.01$)	1.67e-03	1.41e-03	2.99e-03	3.45e-03	1.23e-03
avg. error ($\epsilon = 0.1$)	4.55e-04	3.87e-04	4.56e-04	8.55e-05	5.92e-04
avg. error ($\epsilon = 0.05$)	2.42e-04	2.10e-04	2.54e-04	5.35e-05	3.15e-04
avg. error ($\epsilon = 0.01$)	4.63e-05	4.29e-05	5.10e-05	1.33e-05	6.55e-05

Table III Maximum and average absolute errors on real networks for different values of ϵ ($\delta = 0.1$). The values are averaged over 10 runs.

similar trend can be observed on our other test instances as well. On the left, we see the errors for the whole set of nodes (ordered by exact rank) and, on the right, we focus on the top 100 nodes. The straight lines in the plot on the left correspond to nodes with betweenness 0, which are therefore undistinguishable. The plots show that for a small value of ϵ (0.01), the ranking is very well preserved over all the positions. With higher values of ϵ, the rank error of the nodes with low betweenness increases, because they are more difficult to approximate. However, the error remains small for the nodes with highest betweenness, the most important ones for many applications.

6.2. New Upper Bound on *VD* for Directed Graphs

We compute the new upper bound on VD for directed graphs presented in Section 4.1 and compare it with the upper bound used in RK [25], i.e., the size of the largest weakly connected component. All the networks used in the experiment are real directed graphs. Since finding VD exactly would be expensive in most of the graphs we used, we also compute a lower bound on VD by sampling nodes in the graph, computing their eccentricity (i.e., the maximum distance reachable from the node), and adding 1 to it. In Table IV, we report this lower bound ($VD^{\star}$), our new upper bound ($\tilde{VD}$), and the one used in RK ($\tilde{VD}_{RK}$). The results show that $\tilde{VD}$ is always several orders of magnitude smaller than $\tilde{VD}_{RK}$ and never more than a factor 4 from the lower bound $VD^{\star}$ (and, therefore, also from VD). This difference is mitigated by the logarithm in the number of samples required for the approximation (3.1). However, Table IV shows that $\tilde{VD}$ is almost always more than
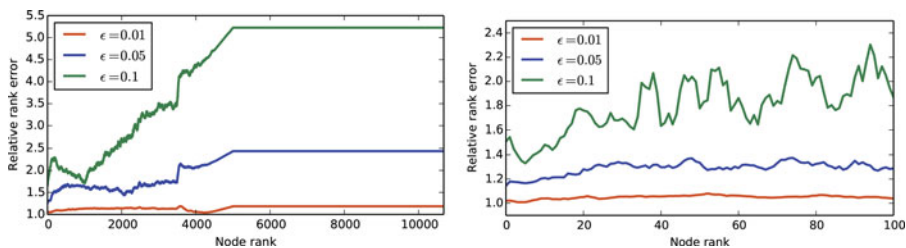

Figure 4 Relative rank error on `PGPgiantcompo` for nodes ordered by rank. Left: relative errors of all nodes. Right: relative errors of the 100 nodes with highest betweenness.

Graph	Nodes	Edges	$VD^\star$	$\tilde{VD}$	$\tilde{VD}_{RK}$
p2p-Gnutella24	26 518	65 369	20	47	26 498
soc-Epinions1	75 879	508 837	13	41	75 877
slashdot081106	77 356	516 575	14	39	77 349
twitter-comb	81 306	1 768 149	9	34	81 306
slashdot090216	81 870	545 671	14	40	81 866
amazon0302	262 111	1 234 877	71	183	262 111
email-EuAll	265 214	420 045	9	23	224 832

Table IV Lower bound on VD ($VD^\star$) and upper bounds (our new bound $\tilde{VD}$ and the one proposed in RK, $\tilde{VD}_{RK}$) on real-world networks.

2^{10} times smaller than $\tilde{VD}_{RK}$, resulting in at least 10 times fewer samples required for the theoretical guarantee.

6.3. Running Times

In this section, we discuss the running times of all the algorithms, the speedups of the exact incremental approaches (GMB, KDB) on BA, and the speedups of our algorithms on RK. (Note that the term *speedup* is used in this study to compare different algorithms, not sequential vs. parallel execution.)

In all of our tests on relatively small graphs (Table I), KDB performs worse than GMB, therefore, we report only the results of GMB. Figure 5 shows the behavior of the four algorithms on PGPgiantcompo. Because the graph is connected and we are considering batches of edge insertions, IA and DA are identical in this case (therefore, in Figure 5 we refer to our algorithm as IA). On the left, the figure shows the running times of the four algorithms, whereas the plot on the right reports the speedups of GMB on BA and of IA on RK. GMB can process edges one by one only, therefore, its running time increases linearly with the batch size, becoming slower than the static algorithm already with a batch size of 64. Our algorithm shows much better speedups and proves to be significantly faster than recomputation even with a batch of size 1024. The reasons for our high speedup are mainly two: First, we process the updates in a batch, processing only once the nodes affected by multiple edge insertions. Second, our algorithm does not need to recompute the dependencies, in contrast to all dynamic algorithms based on BA (i.e., all existing dynamic
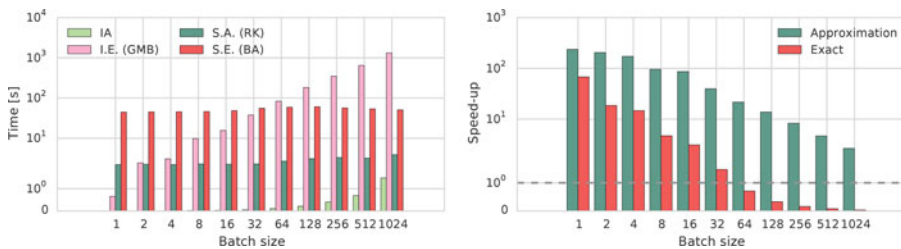

Figure 5 Running times and speedups on PGPgiantcompo, with $\epsilon = 0.05$ and with batches of different sizes. Left: Running times of static exact (BA), static approximation (RK), incremental exact (GMB) and our incremental approximation IA. Right: Speedups of GMB on BA (exact) and of IA on RK (approximation).
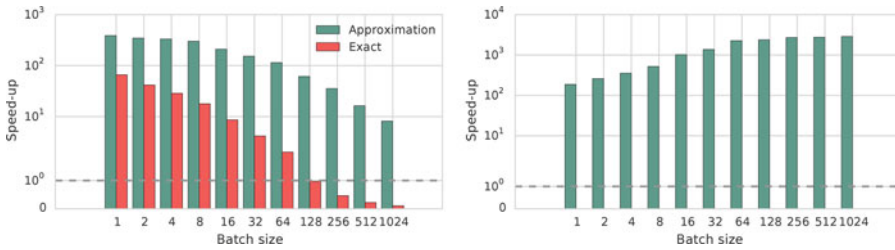
Figure 6 Speedups on Dorogovtsev-Mendes synthetic graphs ($m = 40k$), with $\epsilon = 0.05$ with batches of different sizes. Left: comparison of the speedups of GMB on BA (exact) and of IA on RK (approximation). Right: speedups of IA on GMB.
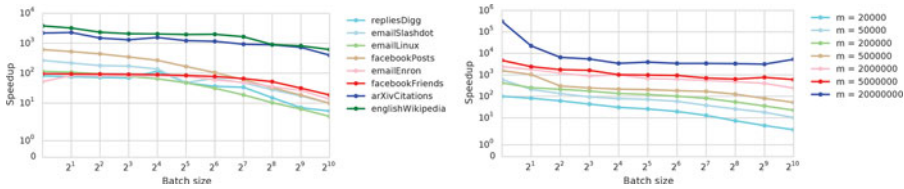

Figure 7 Speedups of DA on RK, with $\epsilon = 0.05$ and with batches of different sizes. Left: real unweighted networks using real dynamics. Right: hyperbolic unit-disk graphs of different sizes.

exact algorithms). For each SSSP, the dependencies need to be recomputed not only for nodes whose distance or number of shortest paths from the source have changed after the edge insertion(s), but also for all the intermediate nodes in the old shortest paths, even if their distance and number of shortest paths are not affected. This number is significantly higher, because for every node that changes its distance or increases its number of shortest paths, the dependencies of *all* the nodes in *all* the old shortest paths are affected.

Results on the other small graphs and on synthetic Dorogovtsev–Mendes graphs are analogous to those shown in Figure 5. Figure 6 (left) shows the speedups of IA on RK and of GMB on BA, averaged over 10 synthetic graphs. We see in Figure 6 (right) that our algorithm is significantly faster than GMB, and the speedup of IA on GMB clearly increases with the batch size. We also compared NPR with IAW on small weighted graphs. Since NPR performs very poorly on all tested instances, we do not report the results here, however, they can be found in our conference article [5]. GMB and NPR also have very high memory requirements ($\Theta(n^2 + mn)$), which makes the algorithms unusable on networks with more than a few thousand edges. The memory requirement is the same, also, for
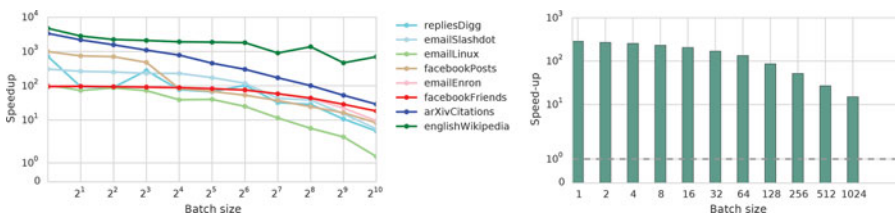

Figure 8 Left: speedups of DA on RK in real unweighted graphs under random updates. Right: Speedups of DAD on RK on the `facebookPosts` directed graph, using real dynamics.

	Real				Random			
	Time [s]		Speedups		Time [s]		Speedups	
Graph	$|\beta| = 1$	$|\beta| = 1024$	$|\beta| = 1$	$|\beta| = 1024$	$|\beta| = 1$	$|\beta| = 1024$	$|\beta| = 1$	$|\beta| = 1024$
repliesDigg	0.053	3.032	605.18	14.24	0.049	3.046	658.19	14.17
emailSlashdot	0.790	5.387	50.81	16.12	0.716	5.866	56.00	14.81
emailLinux	0.324	24.816	5780.49	75.40	0.344	24.857	5454.10	75.28
facebookPosts	0.029	6.672	2863.83	11.42	0.029	6.534	2910.33	11.66
emailEnron	0.050	9.926	3486.99	24.91	0.046	50.425	3762.09	4.90

Table VI Times and speedups of DAW on RK in weighted real graphs under real dynamics and random updates, for batch sizes of 1 and 1024.

| | Hyperbolic | | | |
| | Time [s] | | Speedups | |
Number of edges	$\|\beta\| = 1$	$\|\beta\| = 1024$	$\|\beta\| = 1$	$\|\beta\| = 1024$
$m = 20000$	0.005	0.195	99.83	2.79
$m = 50000$	0.002	0.152	611.17	10.21
$m = 200000$	0.015	0.288	422.81	22.64
$m = 500000$	0.012	0.339	1565.12	51.97
$m = 2000000$	0.049	0.498	2419.81	241.17
$m = 5000000$	0.083	0.660	4716.84	601.85
$m = 20000000$	0.006	0.401	304338.86	5296.78

Table VII Times and speedups of DA on RK in hyperbolic unit-disk graphs, for batch sizes of 1 and 1024.

all other existing dynamic algorithms, with the exception of KDB, which requires $\Theta(n^2)$ memory, still impractical for large networks.

Because the exact algorithms are not scalable, for the comparison on larger networks (Table II) we used only RK and our algorithms. Figure 7 (left) reports the speedups of DA on RK in real graphs using real dynamics. Although some fluctuations can be noticed, the speedups tend to decrease as the batch size increases. We can attribute fluctuations to two main factors: First, different batches can affect areas of G of varying sizes, influencing also the time required to update the SSSPs. Second, changes in the VD approximation can require to sampling new paths and therefore increasing the running time of DA (and DAW). Nevertheless, DA is significantly faster than recomputation on all networks and for every tested batch size. Analogous results are reported in Figure 8 (left) for random dynamics. Table V summarizes the running times of DA and its speedups on RK with batches of size 1 and 1024 in unweighted graphs, under both real and random dynamics. Even on the larger graphs (`arXivCoauthors` and `englishWikipedia`) and on large batches, DA requires at most a few seconds to recompute the BC scores, whereas RK requires about one hour for `englishWikipedia`. The results on weighted graphs are shown in Table VI. In both real dynamics and random updates, the speedups vary between ≈ 50 and $\approx 6 \cdot 10^3$ for single-edge updates and between ≈ 5 and ≈ 75 for batches of size 1024.

On hyperbolic graphs (Figure 7, right), the speedups of DA on RK increase with the size of the graph. Table VII contains the precise running times and speedups on batches of 1 and 1024 edges. The speedups vary between ≈ 100 and $\approx 3 \cdot 10^5$ for single-edge updates and between ≈ 3 and $\approx 5 \cdot 10^3$ for batches of 1024 edges.

Some graphs of Table II can also be interpreted as directed (i. e., `repliesDigg`, `emailSlashdot`, `emailLinux`, `facebookPosts`, `emailEnron`, `arXivCoauthors`). Therefore, we test DAD on the directed version of the networks, using real dynamics. Also on directed networks, using DAD is always faster than recomputation with RK, by orders of magnitude on small batches. Figure 8 (right) shows the speedups of DAD on RK on the `facebookPosts` graph.

To summarize, our results show that our dynamic algorithms are faster than recomputation with RK in all the tested instances, even when large batches of 1024 edges are applied to the graph. With small batches, the algorithms are always orders of magnitude faster than RK, often with running times of fractions of seconds or seconds compared to minutes or hours. Such high speedups are made possible by the efficient update of the sampled shortest

paths, which limits the recomputation to the nodes that are actually affected by the batch. Also, processing the edges in batches, we avoid updating multiple times those nodes that are affected by several edges of the batch.

7. CONCLUSIONS

Because betweenness centrality considers all shortest paths between pairs of nodes, its exact computation is out of reach for the large complex networks that come up in many applications today. However, approximate scores obtained by sampling paths are often sufficient to identify the most important nodes and rank nodes in an order that is very similar to exact BC. Since many applications are concerned with rapidly evolving networks, we have explored whether a dynamic approach–which updates paths when a batch of edges arrives or is deleted from the network–is more efficient than recomputing BC scores from scratch. We introduce the first dynamic betweenness approximation algorithms with provable error guarantee. They work for weighted, unweighted, directed, and undirected graphs. Our theoretical results are confirmed by experiments on a diverse set of real-world networks with both real and simulated dynamics, which show the effectiveness of our approach.

The dynamic updating of paths implies a higher memory footprint, but also enables significant speedups compared to recomputation (e.g., factor 100 for a batch of 1024 edge insertions). The scalability of our algorithms is primarily limited by the available memory. Each sampled path requires $O(n)$ memory and the number of required samples grows quadratically as the error bound ϵ is tightened. This leaves the user with a trade-off between running time and memory consumption on the one hand and BC score error on the other hand. However, our experiments indicate that even a relatively high error bound (e.g., $\epsilon = 0.1$) for the BC scores preserves the ranking for the more important nodes reasonably well.

We studied sequential implementations for simplicity and comparability with related work, but parallelization is possible, part of future work, and can yield further speedups in practice. Our implementations are based on *NetworKit*[4], the open-source framework for large-scale network analysis. Most of the source code used for this article is already available in *NetworKit*; the remaining code will follow in upcoming releases of the package.

An interesting open problem that remains is the update of BC approximations in scenarios where nodes can also be inserted and deleted. The presence of a new node (or the remotion of an existing one) would modify the probability distribution that regulates the path sampling, introducing the necessity of new techniques.

[4]`http://networkit.iti.kit.edu`

FUNDING

REFERENCES

[1] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. "Approximating Betweenness Centrality." In *5th Workshop on Algorithms and Models for the Web-Graph (WAW '07)*, pp. 124–137, LNCS 4863. Berlin, Heidelberg: Springer, 2007.

[2] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. "Benchmarking for Graph Clustering and Partitioning." In *Encyclopedia of Social Network Analysis and Mining*, edited by R. Alhajj and J. Rokne, pp. 73–82. New York, NY: Springer, 2014.

[3] R. Bauer and D. Wagner. "Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study." In *8th Int. Symp. on Experimental Algorithms (SEA '09)*, pp. 51–62, LNCS 5526. Berlin, Heidelberg: Springer, 2009.

[4] E. Bergamini and H. Meyerhenke. "Fully-Dynamic Approximation of Betweenness Centrality." In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Proceedings*, edited by N. Bansal and I. Finocchi, pp. 155–166. Berlin, Heidelberg: Springer, 2015.

[5] E. Bergamini, H. Meyerhenke, and C. Staudt. "Approximating Betweenness Centrality in Large Evolving Networks." In *17th Workshop on Algorithm Engineering and Experiments, ALENEX 2015*, pp. 133–146. SIAM, 2015.

[6] M. Borassi, P. Crescenzi, and A. Marino. "Fast and Simple Computation of Top-k Closeness Centralities." Available online (`http://arxiv.org/abs/1507.01490`), 2015.

[7] U. Brandes. "A Faster Algorithm for Betweenness Centrality." *Journal of Mathematical Sociology* 25 (2001), 163–177.

[8] U. Brandes and C. Pich. "Centrality Estimation in Large Networks." *I. J. Bifurcation and Chaos* 17:7 (2007), 2303–2318.

[9] M. H. Chehreghani. "An Efficient Algorithm for Approximate Betweenness Centrality Computation." *Comput. J.* 57:9 (2014), 1371–1382.

[10] A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti. "Experimental Evaluation of Dynamic Shortest Path Tree Algorithms on Homogeneous Batches." In *13th Int. Symp. on Experimental Algorithms (SEA '14)*, pp. 283–294, LNCS 8504. Berlin, Heidelberg: Springer, 2014.

[11] S. N. Dorogovtsev and J. F. Mendes. *Evolution of Networks: From Biological Nets to the Internet and WWW*. Oxford, UK: Oxford University Press, 2003.

[12] L. C. Freeman. "A Set of Measures of Centrality Based on Betweenness." *Sociometry* 40:1 (1977), 35–41.

[13] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. "Semi-Dynamic Algorithms for Maintaining Single-Source Shortest Path Trees." *Algorithmica* 22 (2008), 250–274.

[14] R. Geisberger, P. Sanders, and D. Schultes. "Better Approximation of Betweenness Centrality." In *10th Workshop on Algorithm Engineering and Experiments (ALENEX '08)*, pp. 90–100. SIAM, 2008.

[15] K. Goel, R. R. Singh, S. Iyengar, and S. Gupta. "A Faster Algorithm to Update Betweenness Centrality after Node Alteration." *Internet Mathematics* 11:4-5 (2015), 403–420.

[16] O. Green, R. McColl, and D. A. Bader. "A Fast Algorithm for Streaming Betweenness Centrality." In *SocialCom/PASSAT*, pp. 11–20. IEEE, 2012.

[17] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. "Incremental Algorithm for Updating Betweenness Centrality in Dynamically Growing Networks." In *Advances in Social Networks Analysis and Mining 2013 (ASONAM '13)*, pp. 33–40. New York, NY: ACM, 2013.

[18] N. Kourtellis, G. D. F. Morales, and F. Bonchi. "Scalable Online Betweenness Centrality in Evolving Graphs." *IEEE Trans. Knowl. Data Eng.* 27:9 (2015), 2494–2506.

[19] J. Kunegis. "KONECT: The Koblenz Network Collection." In *22nd Int. World Wide Web Conf., WWW '13*, pp. 1343–1350. WWW, 2013.

[20] M. Lee, J. Lee, J. Y. Park, R. H. Choi, and C. Chung. "QUBE: A Quick Algorithm for Updating Betweenness Centrality." In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012*, edited by A. Mille, F. L. Gandon, J. Misselis, M. Rabinovich, and S. Staab, editors, pp. 351–360. New York, NY: ACM, 2012.

[21] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. "Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations." In *11th Int. Conf. on Knowledge Discovery and Data Mining*, pp 177–187. New York, NY: ACM, 2005.

[22] M. Nasre, M. Pontecorvi, and V. Ramachandran. "Betweenness Centrality - Incremental and Faster." In *Mathematical Foundations of Computer Science 2014 - 39th Int. Symp., MFCS 2014*, pp. 577–588, LNCS 8635. Berlin, Heidelberg: Springer, 2014.

[23] M. Pontecorvi and V. Ramachandran. "A Faster Algorithm for Fully Dynamic Betweenness Centrality." *CoRR*, abs/1506.05783, 2015.

[24] G. Ramalingam and T. Reps. "An Incremental Algorithm for a Generalization of the Shortest-Path Problem." *Journal of Algorithms* 21 (1992), 267–305.

[25] M. Riondato and E. M. Kornaropoulos. "Fast Approximation of Betweenness Centrality Through Sampling." *Data Mining and Knowledge Discovery* 30:2 (2016), 438–475.

[26] L. Roditty and U. Zwick. "On Dynamic Shortest Paths Problems." *Algorithmica* 61:2 (2011), 389–401.

[27] C. Staudt, A. Sazonovs, and H. Meyerhenke. "NetworKit: An Interactive Tool Suite for High-Performance Network Analysis." (`http://arxiv.org/abs/1403.3005`), 2014.

[28] M. von Looz, H. Meyerhenke, and R. Prutkin. "Generating Random Hyperbolic Graphs in Subquadratic Time." In *Proc. 26th Int'l Symp. on Algorithms and Computation (ISAAC 2015)*, LNCS. Berlin, Heidelberg: Springer, 2015. To appear.