Fully-dynamic Approximation of Betweenness Centrality

Elisabetta Bergamini and Henning Meyerhenke

Institute of Theoretical Informatics
Karlsruhe Institute of Technology (KIT), Germany
Email: {elisabetta.bergamini, meyerhenke}@kit.edu

Abstract. Betweenness is a well-known centrality measure that ranks the nodes of a network according to their participation in shortest paths. Since an exact computation is 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 evolving networks. In previous work we proposed the first semi-dynamic algorithms that recompute an *approximation* of betweenness in connected graphs after batches of edge insertions.
In this paper we propose the first fully-dynamic approximation algorithms (for weighted and unweighted graphs that need not to be connected) with a provable guarantee on the maximum approximation error. The transfer to fully-dynamic and disconnected graphs implies additional algorithmic problems that can be of independent interest. In particular, we propose a new upper bound on the vertex diameter for weighted undirected graphs. For both weighted and unweighted graphs, we also propose the first fully-dynamic algorithms that keep track of such upper bound. In addition, we extend our former algorithm for semi-dynamic BFS to batches of both edge insertions and deletions.
Using approximation, our algorithms are the first to make in-memory computation of betweenness in fully-dynamic networks with millions of edges feasible. Our experiments show that they can achieve substantial speedups compared to recomputation, up to several orders of magnitude.
Keywords: betweenness centrality, algorithmic network analysis, fully-dynamic graph algorithms, approximation algorithms, shortest paths

1 Introduction

The identification of the most central nodes of a network is a fundamental problem in network analysis. *Betweenness centrality* (BC) is an index that ranks the importance of nodes according to their participation in *shortest paths*. Intuitively, a node has high BC when it lies on many shortest paths between pairs of other nodes. Hence, BC is an interesting measure whenever some flow traverses the network along shortest paths. This can mean, for example, the identification of important intersections in street networks, influential people in social networks or key infrastructure nodes in the internet. Formally, BC of a node v is defined as $c_B(v) = \frac{1}{n(n-1)} \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where n is the number of nodes, σ_{st} is the number

of shortest paths between two nodes s and t and $\sigma_{st}(v)$ is the number of these paths that go through node v. Since it depends on *all* shortest paths, the exact computation of BC is expensive: the best known algorithm [4] is quadratic in the number of nodes for sparse networks and cubic for dense networks, 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. For this reason, approximation algorithms [5,8,1] based on random sampling of shortest paths must be used in practice. In addition, many large graphs of interest evolve continuously, making the efficient recomputation of BC a necessity. In a previous work, we proposed the first two approximation algorithms [3] (IA for unweighted and IAW for weighted graphs) that can efficiently recompute the approximate BC scores after batches of edge insertions or weight decreases. IA and IAW are the only semi-dynamic algorithms that can actually be applied to large networks. The algorithms build on RK [17], a static algorithm with a theoretical guarantee on the quality of the approximation, and inherit this guarantee from RK. However, IA and IAW target a relatively restricted configuration: only connected graphs and edge insertions/weight decreases.

Our contributions. In this paper we present the first fully-dynamic algorithms (handling edge insertions, deletions and arbitrary weight updates) for BC approximation in weighted and unweighted undirected graphs. Our algorithms extend the semi-dynamic ones we presented in [3], while keeping the theoretical guarantee on the maximum approximation error. The transfer to fully-dynamic and disconnected graphs implies several additional problems compared to the restricted case we considered previously [3]. Consequently, we present the following intermediate results, all of which can be of independent interest. (i) We propose a new upper bound on the vertex diameter *VD* (i. e. number of nodes in the shortest path(s) with the maximum number of nodes) for weighted undirected graphs. This can improve significantly the one used in the RK algorithm [17] if the network's weights vary in relatively small ranges (from the size of the largest connected component to at most twice the vertex diameter times the ratio between the maximum and the minimum edge weights). (ii) For both weighted and unweighted graphs, we present the first fully-dynamic algorithm for updating an approximation of *VD*, which is equivalent to the diameter in unweighted graphs. (iii) We extend our previous semi-dynamic BFS algorithm [3] to batches of both edge insertions and deletions. In our experiments, we compare our algorithms to recomputation with RK on both synthetic and real dynamic networks. Our results show that our algorithms can achieve substantial speedups, often several orders of magnitude on single-edge updates and are always faster than recomputation on batches of more than 1000 edges.

2 Related work

2.1 Overview of algorithms for computing BC

The best static exact algorithm for BC (BA) is due to Brandes [4] and requires $\Theta(nm)$ operations for unweighted graphs and $\Theta(nm + n^2 \log n)$ for graphs with

positive edge weights. The algorithm computes a single-source shortest path (SSSP) search from every node s in the graph and adds to the BC score of each node $v \neq s$ the fraction of shortest paths that go through v. Several static approximation algorithms have been proposed that compute an SSSP search from a set of randomly chosen nodes and extrapolate the BC scores of the other nodes [5,8,1]. The static approximation algorithm by Riondato and Kornaropoulos (RK) [17] samples a set of shortest paths and adds a contribution to each node in the sampled paths. This approach allows a theoretical guarantee on the quality of the approximation and will be descried in Section 2.2. Recent years have seen the publication of a few dynamic exact algorithms [10,9,12,11,15]. Most of them store the previously calculated BC values and additional information, like the distance of each node from every source, and try to limit the recomputation to the nodes whose BC has actually been affected. All the dynamic algorithms perform better than recomputation on certain inputs. Yet, none of them is in general better than BA. In fact, they all require updating an all-pairs shortest paths (APSP) search, for which no algorithm has an improved worst-case complexity compared to the best static algorithm [18]. Also, the scalability of the dynamic exact BC algorithms is strongly compromised by their memory requirement of $\Omega(n^2)$. To overcome these problems, we presented two algorithms that efficiently recompute an approximation of the BC scores instead of their exact values [3]. The algorithms have shown significantly high speedups compared to recomputation with RK and a good scalability, but they are limited to connected graphs and batches of edge insertions/weight decreases (see Section 2.3).

2.2 RK algorithm

The static approximation algorithm RK [17] is the foundation for the incremental approach we presented in [3] and our new fully-dynamic approach. RK samples a set $S = \{p_{(1)}, ..., p_{(r)}\}$ of r shortest paths between randomly-chosen source-target pairs (s, t). Then, RK computes the approximated betweenness $\tilde{c}_B(v)$ of a node v as the fraction of sampled paths $p_{(k)} \in S$ that go through v, by adding $\frac{1}{r}$ to v's score for each of these paths. In each of the r iterations, the probability of a shortest path p_{st} to be sampled is $\pi_G(p_{st}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma_{st}}$. The number r of samples required to approximate the BC scores with the given error guarantee is $r = \frac{c}{\epsilon^2}\left(\lfloor \log_2(VD - 2) \rfloor + 1 + \ln \frac{1}{\delta}\right)$, where ϵ and δ are constants in $(0, 1)$ and $c \approx 0.5$. Then, if r shortest paths are sampled according to π_G, with probability at least $1 - \delta$ the approximations $\tilde{c}_B(v)$ 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 source-target node pair (s, t) uniformly at random and performs an SSSP search (Dijkstra or BFS) from the source s, keeping also track of the number σ_{sv} of shortest paths from s to v and of the list of predecessors $P_s(v)$ (i. e. the nodes that immediately precede v in the shortest paths from s to 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.

Approximating the vertex diameter. RK uses two upper bounds of *VD* that can be both computed in $O(n+m)$, instead of solving an APSP problem. 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 weighted graphs, RK approximates *VD* with the size of the largest connected component, which can be a significant overestimation for complex networks, possibly of orders of magnitude. In this paper, we present a new approximation for weighted graphs, described in Section 3.

2.3 IA and IAW algorithms

IA and IAW are the incremental approximation algorithms (for unweighted and weighted graphs, respectively) that we presented previously [3]. The algorithms are based on the observation that if only edge insertions are allowed and the graph is connected, *VD* cannot increase, and therefore also the number r of samples required by RK for the theoretical guarantee. Instead of recomputing r new shortest paths after a batch of edge insertions, IA and IAW *replace* each old shortest path $p_{s,t}$ with a new shortest path between the same node pair (s,t). In IAW the paths are recomputed with a slightly-modified T-SWSF [2], whereas IA uses a new semi-dynamic BFS algorithm. The BC scores are updated by subtracting $1/r$ to the BC of the nodes in the old path and adding $1/r$ to the BC of nodes in the new shortest path.

2.4 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 [16,7,2] and compared in experimental studies [2,6]. The experiments show that the tuned algorithm T-SWSF presented in [2] performs well on many types of graphs and edge updates. For batches of only edge insertions in unweighted graphs, we developed an algorithm asymptotically faster than T-SWSF [3]. The algorithm is in principle similar to T-SWSF, but has an improved complexity thanks to different data structures.

3 New *VD* approximation for 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^T_{xy} 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 x and y in G, and analogously for $|p^T_{xy}|$ and $d^T(x,y)$. Let $\overline{\omega}$ and $\underline{\omega}$ be the maximum and minimum

edge weights, 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 $\tilde{VD} := 1 + \frac{d(s,u)+d(s,v)}{\underline{\omega}}$. Then:

Proposition 1. *$VD \leq \tilde{VD} < 2 \cdot \frac{\overline{\omega}}{\underline{\omega}} VD$. (Proof in Section B.1, Appendix)*

To obtain the upper bound $\tilde{VD}$, 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 $\tilde{VD}$ extends the upper bound proposed for RK [17] for unweighted graphs: When the graph is unweighted and thus $\underline{\omega} = \overline{\omega}$, $\tilde{VD}$ becomes equal to the approximation used by RK. Complex networks are often characterized by a small diameter and in networks like coauthorship, friendship, communication networks, VD and $\frac{\overline{\omega}}{\underline{\omega}}$ can be several order of magnitude smaller than the size of the largest component. This translates into a substantially improved VD approximation.

4 New fully-dynamic algorithms

Overview. We propose two fully-dynamic algorithms, one for unweighted (DA, dynamic approximation) and one for weighted (DAW, dynamic approximation weighted) graphs. Similarly to IA and IAW, our new fully-dynamic algorithms keep track of the old shortest paths and substitute them only when necessary. However, if G is not connected or edge deletions occur, VD can grow and a simple substitution of the paths is not sufficient anymore. Although many real-world networks exhibit a shrinking-diameter behavior [14], to ensure our theoretical guarantee, we need to keep track of $\tilde{VD}$ over time and sample new paths in case $\tilde{VD}$ increases. The need for an efficient update of $\tilde{VD}$ augments significantly the difficulty of the fully-dynamic problem, as well as the necessity to recompute the SSSPs after batches of both edge insertions and deletions. The building block for the BC update are basically two: a fully-dynamic algorithm that updates distances and number of shortest paths from a certain source node (SSSP update) and an algorithm that keeps track of a VD approximation for each connected component of G. The following paragraphs give an overview of such building blocks, which can be of independent interest. The last paragraph outlines the dynamic BC approximation algorithm. **Due to space constraints, a detailed description of the algorithms as well as the pseudocodes and the omitted proofs can be found in the Appendix**.

SSSP update in weighted graphs. Our SSSP update is based on T-SWSF [2], which recomputes distances from a source node s after a batch β of weight updates (or edge insertions/deletions). For our BC algorithm, we need two extensions of T-SWSF: an algorithm that also recomputes the number of shortest paths from s to the other nodes (updateSSSP-W) and one that also updates a VD approximation for the connected component of s (updateApprVD-W). The VD approximation is computed as described in Section 3. 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* the 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 re-inserted into Q with a higher candidate distance. In both cases, the affected neighbors of w are inserted into Q. In updateApprVD-W, d' and d'' are recomputed while updating the distances and $\underline{\omega}$ is updated while scanning β. In updateSSSP-W, the number $\sigma(w)$ of shortest paths of w is recomputed as the sum of the $\sigma(z)$ of the new predecessors z of w.

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

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, as the one we used in [3] for the incremental BFS. Each queue represents a *level* from 0 (which only the source belongs to) to the maximum distance d'. The levels replace the priorities and also in this case represent the candidate distances for the nodes. In order not to visit a node multiple times, we use colors to distinguish the unvisited nodes from the visited ones. The replacement of the priority queue with the list of queues decreases the complexity of the SSSP update algorithms for unweighted graphs, that we call updateApprVD-U and updateSSSP-U, in analogy with the ones for weighted graphs.

Lemma 2. *The time required by updateApprVD-U (updateSSSP-U) to update the distances and $\tilde{VD}$ (the number of shortest paths) is $O(|\beta| + ||A|| + d_{\max})$, where $d_{\max}$ is the maximum distance from s reached during the update.*

Fully-dynamic VD approximation. The algorithm keeps track of a *VD* approximation for the whole 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 $\tilde{VD}_i$ for C_i, as described in Sections 2.2 and 3. In the update, we recompute the SSSPs and the *VD* approximations with updateApprVD-W (or updateApprVD-U). Since components might split or merge, we might need to compute new approximations, in addition to update the old ones. To do this, for each node, we keep track of the number 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. 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 1. *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.*

Dynamic BC approximation. Let G be an undirected graph with n_c connected components. Now that we have defined our building blocks, we can outline a fully-dynamic BC algorithm: we use the fully dynamic *VD* approximation to recompute $\tilde{VD}$ after a batch, we update the r sampled paths with updateSSSP and, if $\tilde{VD}$ (and therefore r) increases, we 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 such 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 r sampled paths (i. e. 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'. The high-level description of the update after a batch β is shown as Al-

Algorithm 1: BC update after a batch β of edge updates

```
1  applyBatch(G, β);
2  for i ← 1 to r do
3  |   V̂D_i ← updateSSSPVD(s_i, β);
4  |   replacePath(s_i, t_i) ;                    /* update of BC scores */
5  end
6  foreach C_i ∈ R' do
7  |   V̂D_i ← updateApprVD(C_i, β);
8  end
9  foreach unvisited C_j do
10 |   add C_j to R';
11 |   V̂D_j ← initApprVD(C_j);
12 end
13 V̂D ← max_{C_i ∈ R ∪ R'} V̂D_i;
14 r_new ← (c/ε²)(⌊log₂(V̂D − 2)⌋ + ln(1/δ));
15 if r_new > r then
16 |   sampleNewPaths() ;                        /* update of BC scores */
17 |   foreach v ∈ V do
18 |   |   c̃_B(v) ← c̃_B(v) · r/r_new ;    /* renormalization of BC scores */
19 |   end
20 |   r ← r_new;
21 end
22 return {(v, c̃_B(v)) : v ∈ V}
```

gorithm 1. After changing the graph according to β (Line 1), we recompute the previous r samples and the *VD* approximations for their components (Lines 2 - 5). Then, similarly to IA and IAW, we update the BC scores of the nodes in the old and in the new shortest paths. Thus, we update a *VD* approximation for the components in R' (Lines 6 - 8) and compute a new approximation for new components that have formed applying the batch (Lines 9 - 12). Then, we use the results to update the number of samples (Lines 13 - 14). If necessary, we sample additional paths and normalize the BC scores (Lines 18 - 21). 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. Differently from RK and our previous algorithms IA and IAW, in DA and DAW we scan the neighbors every time we need the predecessors instead of storing them. This allows us to use $O(|V|)$ memory per sample instead of $O(|E|)$, while the experimental results show that the running time is hardly influenced.

Theorem 2. *Algorithm 7 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,* $\Pr(\exists v \in V \text{ s.t. } |c'_B(v) - \tilde{c}'_B(v)| > \epsilon) < \delta$.

Theorem 3. *Let Δr be the difference between the value of r before and after the batch and let $||A^{(i)}||$ be the sum of affected nodes and their incident edges in the i-th SSSP. The time required for the BC update in unweighted graphs is $O((r+r')|\beta| + \sum_{i=1}^{r+r'}(||A^{(i)}|| + d^{(i)}_{\max}) + \Delta r(|V|+|E|))$. In weighted graphs, it is $O((r+r')|\beta|\log|\beta| + \sum_{i=1}^{r+r'}||A^{(i)}||\log||A^{(i)}|| + \Delta r(|V|\log|V| + |E|))$.*

Notice that, if $\tilde{VD}$ does not increase, $\Delta r = 0$ and the complexities are the same as the only-incremental algorithms IA and IAW we proposed in [3]. Also, notice that in the worst case the complexity can be as bad as recomputing from scratch. However, no dynamic SSSP (and therefore no BC) algorithm exists that is faster than recomputation.

5 Experiments

Implementation and settings. We implement our two dynamic approaches DA and DAW in C++, building on the open-source *NetworKit* framework [19], which also contains the static approximation RK. In all experiments we fix δ to 0.1 and ϵ to 0.05, as a good tradeoff between running time and accuracy [3]. This means that, with a probability of at least 90%, the computed BC values deviate at most 0.05 from the exact ones. In our previous experimental study [3], we showed that for such values of ϵ and δ, the ranking error (how much the ranking computed by the approximation algorithm differs from the rank of the exact algorithm) is low, in particular for nodes with high betweenness (the ones most applications are interested in). Since our algorithms simply update the approximation of RK, our accuracy in terms or ranking error does not differ from that of RK (see [3] for an experimental evaluation of the accuracy for different values of ϵ). The machine used has 2 x 8 Intel(R) Xeon(R) E5-2680 cores at 2.7 GHz, of which we use only one core, and 256 GB RAM.

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
arXivCitations	coauthorship	28,093	3,148,447	Unweighted
englishWikipedia	hyperlink	1,870,709	36,532,531	Unweighted

Table 1: Overview of real dynamic graphs used in the experiments.

Data sets and experiments. We use both real-world dynamic and synthetic networks. The real-world networks are taken from The Koblenz Network Collection (KONECT)[1] [13] and are summarized in Table 1. All the edges of the KONECT graphs are characterized by a time of arrival. In 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 $1/|E_{st}|$. 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. Since the networks do not include edge deletions, we implement additional simulated dynamics. In particular, we consider the following experiments. (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 $(0, 2)$.

For synthetic graphs we use a generator based on a unit-disk graph model in hyperbolic geometry [20], 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, like small diameter and power-law degree distribution (see [20] and the references therein). We generate seven networks, with $|E|$ ranging from about $2 \cdot 10^4$ to about $2 \cdot 10^7$ and $|V|$ approximately equal to $|E|/10$.

Speedups. Figure 1 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 sample new paths and therefore increase 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 3 of the Appendix for random dynamics. Table 2 summarizes

[1] http://http://konect.uni-koblenz.de/

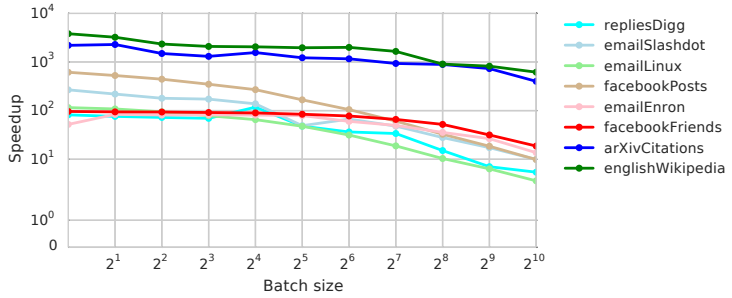

Fig. 1: Speedups of DA on RK in real unweighted networks using real dynamics.

	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`	0.078	1.028	76.11	5.42	0.008	0.832	94.00	4.76
`emailSlashdot`	0.043	1.055	219.02	9.91	0.038	1.151	263.89	28.81
`emailLinux`	0.049	1.412	108.28	3.59	0.051	2.144	72.73	1.33
`facebookPosts`	0.023	1.416	527.04	9.86	0.015	1.520	745.86	8.21
`emailEnron`	0.368	1.279	83.59	13.66	0.203	1.640	99.45	9.39
`facebookFriends`	0.447	1.946	94.23	18.70	0.448	2.184	95.91	18.24
`arXivCitations`	0.038	0.186	2287.84	400.45	0.025	1.520	2188.70	28.81
`englishWikipedia`	1.078	6.735	3226.11	617.47	0.877	5.937	2833.57	703.18

Table 2: Times and speedups of DA on RK in unweighted real graphs under real dynamics and random updates, for batch sizes of 1 and 1024.

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 (`arXivCitations` 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 3 in Section C in the Appendix. 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 2), the speedups of DA on RK increase with the size of the graph. Table 4 in the Appendix contains the 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. The results show that DA and DAW 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 fraction 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 limit the recomputation to the nodes that are actually affected by the batch. Also, processing the edges in batches, we avoid to update multiple times nodes that are affected by several edges of the batch.

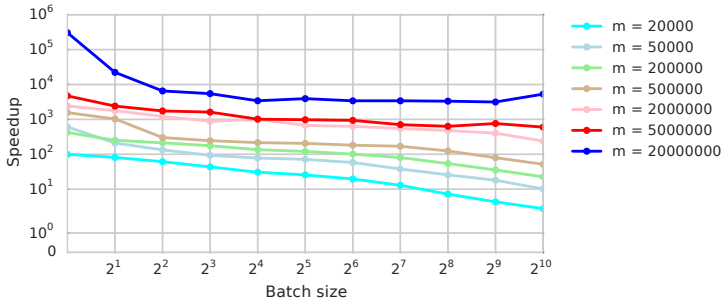

Fig. 2: Speedups of DA on RK in hyperbolic unit-disk graphs.

6 Conclusions

Betweenness is a widely used centrality measure, yet expensive if computed exactly. In this paper we have presented the first fully-dynamic algorithms for betweenness approximation (for weighted and for unweighted graphs). The consideration of edge deletions and disconnected graphs is made possible by the efficient solution of several algorithmic subproblems (some of which may be of independent interest). Now BC can be approximated with an error guarantee for a much wider set of dynamic real graphs compared to previous work.

Our experiments show significant speedups over the static algorithm RK. In this context it is interesting to remark that dynamic algorithms require to store additional memory and that this can be a limit to the size of the graphs they can be applied to. By not storing the predecessors in the shortest paths, we reduce the memory requirement from $O(|E|)$ per sampled path to $O(|V|)$ – and are still often more than 100 times faster than RK despite rebuilding the paths.

Future work may include the transfer of our concepts to approximating other centrality measures in a fully-dynamic manner, e. g. closeness. Moreover, making the betweenness code run in parallel will further accelerate the computations in practice. Our implementation will be made available as part of a future release of the network analysis tool suite *NetworKit* [19].

Acknowledgements. This work is partially supported by DFG grant FINCA (ME-3619/3-1) within the SPP 1736 *Algorithms for Big Data*. We thank Moritz von Looz for providing the synthetic dynamic networks and the numerous contributors to the *NetworKit* project.

References

1. David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *5th Workshop on Algorithms and Models for the Web-Graph (WAW '07)*, volume 4863 of *Lecture Notes in Computer Science*, pages 124–137. Springer, 2007.

2. Reinhard Bauer and Dorothea Wagner. Batch dynamic single-source shortest-path algorithms: An experimental study. In *8th Int. Symp. on Experimental Algorithms (SEA '09)*, volume 5526 of *LNCS*, pages 51–62. Springer, 2009.
3. Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. Approximating betweenness centrality in large evolving networks. In *17th Workshop on Algorithm Engineering and Experiments, ALENEX 2015*, pages 133–146. SIAM, 2015.
4. Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.
5. Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7):2303–2318, 2007.
6. Annalisa D'Andrea, Mattia D'Emidio, Daniele Frigioni, Stefano Leucci, and Guido Proietti. Experimental evaluation of dynamic shortest path tree algorithms on homogeneous batches. In *13th Int. Symp. on Experimental Algorithms (SEA '14)*, volume 8504 of *LNCS*, pages 283–294. Springer, 2014.
7. Daniele Frigioni, Alberto Marchetti-spaccamela, and Umberto Nanni. Semi-dynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22:250–274, 2008.
8. Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *10th Workshop on Algorithm Engineering and Experiments (ALENEX '08)*, pages 90–100. SIAM, 2008.
9. Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *SocialCom/PASSAT*, pages 11–20. IEEE, 2012.
10. Min joong Lee, Ryan H. Choi, Jungmin Lee, Chin wan Chung, and Jaimie Y. Park. Qube: a quick algorithm for updating betweenness centrality, 2012.
11. Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Advances in Social Networks Analysis and Mining 2013 (ASONAM '13)*, pages 33–40. ACM, 2013.
12. Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. *CoRR*, abs/1401.6981, 2014.
13. Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd Int. World Wide Web Conf., WWW '13*, pages 1343–1350, 2013.
14. Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *11th Int. Conf. on Knowledge Discovery and Data Mining*, pages 177–187. ACM, 2005.
15. Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality - incremental and faster. *CoRR*, abs/1311.2147, 2013.
16. G. Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1992.
17. Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *7th ACM Int. Conf. on Web Search and Data Mining (WSDM '14)*, pages 413–422. ACM, 2014.
18. Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.
19. Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. `http://arxiv.org/abs/1403.3005`, 2014.
20. Moritz von Looz, Christian L. Staudt, Henning Meyerhenke, and Roman Prutkin. Fast generation of complex networks with underlying hyperbolic geometry. `http://arxiv.org/abs/1501.03545v2`, 2015.

A Description of the fully-dynamic algorithms

A.1 Dynamic *VD* approximation

Algorithm 2 describes the initialization. Initially, we put all the nodes in a queue and compute an SSSP from the nodes we extract. During the SSSP search, we mark as visited all the nodes we scan. When extracting the nodes, we skip those that have already been visited: This avoids us to compute multiple approximations for the same component. In the update (Algorithm 3), we recompute the SSSPs and the *VD* approximations with updateApprVD-W (or updateApprVD-U). Since components might split, we might need to add *VD* approximations for some new subcomponents, in addition to recompute the old ones. Also, if components merge, we can discard the superfluous approximations. To do this, we keep track, for each node, of the number of times it has been visited. Let $vis(v)$ denote this number for node v. 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. After the update, for all nodes v that have become unvisited ($vis(v) = 0$), we compute a new *VD* approximation from scratch.

Algorithm 2: Dynamic *VD* approximation (initialization)

```
1  U ← [];
2  foreach node v ∈ V do
3  |  vis(v) ← 0; insert v into U;
4  end
5  i ← 1;
6  while U ≠ ∅ do
7  |  extract s from U;
8  |  if vis(s) = 0 then
9  |  |  s_i ← s;
   |  |  // initApprVD adds 1 to vis(v) of the nodes it visits
10 |  |  ṼD_i ← initApprVD(G, s_i);
11 |  |  i ← i + 1;
12 |  end
13 end
14 n_C ← i − 1;
15 ṼD ← max_{i=1,...,n_C} ṼD_i;
16 return ṼD
```

A.2 Dynamic SSSP update for weighted graphs

Algorithm 4 describes the SSSP update for weighted graphs. The pseudocode updates both the *VD* approximation for the connected component of s and the

Algorithm 3: Dynamic VD approximation (update)

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

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), we assume there is no shortest path from s to w anymore, setting $d(w)$ 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 $d(w)$. When updating $d(w)$, we also increase $vis(w)$ in case 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).

Algorithm 4: SSSP update for weighted graphs

```
1  Q ← empty priority queue;
2  foreach e = {u, v} ∈ β, d(u) < d(v) do
3  |  Q ← insertOrDecreaseKey(v, p(v) = min{d(u) + ω({u, v}), d(w)});
4  end
5  ω ← min{ω, ω(e) : e ∈ β};
6  while there are nodes in Q do
7  |  {w, p(w)} ← extractMin(Q);
8  |  con(w) ← min_{z:(z,w)∈E} d(z) + ω(z, w);
9  |  if con(w) = p(w) then
10 |  |  update d' and d'';
11 |  |  if d(w) = ∞ then
12 |  |  |  vis(w) ← vis(w) + 1;
13 |  |  end
14 |  |  d(w) ← p(w); σ(w) ← 0;
15 |  |  foreach incident edge (z, w) do
16 |  |  |  if d(w) = d(z) + ω(z, w) then
17 |  |  |  |  σ(w) ← σ(w) + σ(z);
18 |  |  |  end
19 |  |  |  if d(z) ≥ d(w) + ω(z, w) then
20 |  |  |  |  Q ← insertOrDecreaseKey(z, p(z) = d(w) + ω(z, w));
21 |  |  |  end
22 |  |  end
23 |  end
24 |  else
25 |  |  if d(w) ≠ ∞ then
26 |  |  |  vis(w) ← vis(w) − 1;
27 |  |  |  if vis(w)=0 then
28 |  |  |  |  insert w into U;
29 |  |  |  end
30 |  |  |  if con(w) ≠ ∞ then
31 |  |  |  |  Q ←insertOrDecreaseKey(w, p(w) = con(w));
32 |  |  |  |  foreach incident edge (z, w) do
33 |  |  |  |  |  if d(z) = d(w) + ω(w, z) then
34 |  |  |  |  |  |  Q ←insertOrDecreaseKey(z, p(z) = d(w) + ω(z, w));
35 |  |  |  |  |  end
36 |  |  |  |  end
37 |  |  |  |  d(w) ← ∞;
38 |  |  |  end
39 |  |  end
40 |  end
41 end
```

Algorithm 5: SSSP update for unweighted graphs

```
1  Assumption: color(w) = white   ∀w ∈ V;
2  Q[] ← array of empty queues;
3  foreach e = {u, v} ∈ β, d(u) < d(v) do
4  |  k ← d(v) + 1; enqueue v → Q[k];
5  end
6  k ← 1;
7  while there are nodes in Q[j], j ≥ k do
8  |  while Q[k] ≠ ∅ do
9  |  |  dequeue w ← Q[k];
10 |  |  if color(w) = black then continue;
11 |  |  con(w) ← min_{z:(z,w)∈E} d(z) + 1;
12 |  |  if con(w) = k then
13 |  |  |  update d' and d'';
14 |  |  |  if d(w) = ∞ then vis(w) ← vis(w) + 1;
15 |  |  |  d(w) ← k; σ(w) ← 0; color(w) ← black;
16 |  |  |  foreach incident edge (z, w) do
17 |  |  |  |  if d(w) = d(z) + 1 then
18 |  |  |  |  |  σ(w) ← σ(w) + σ(z);
19 |  |  |  |  end
20 |  |  |  |  if d(z) > k then
21 |  |  |  |  |  enqueue z → Q[k + 1];
22 |  |  |  |  end
23 |  |  |  end
24 |  |  end
25 |  |  else
26 |  |  |  if d(w) ≠ ∞ then
27 |  |  |  |  d(w) ← ∞;
28 |  |  |  |  vis(w) ← vis(w) − 1;
29 |  |  |  |  if vis(w)=0 then
30 |  |  |  |  |  insert w into U;
31 |  |  |  |  end
32 |  |  |  |  if con(w) ≠ ∞ then
33 |  |  |  |  |  enqueue w → Q[con(w)];
34 |  |  |  |  |  foreach incident edge (z, w) do
35 |  |  |  |  |  |  if d(z) > k then
36 |  |  |  |  |  |  |  enqueue z → Q[k + 1];
37 |  |  |  |  |  |  end
38 |  |  |  |  |  end
39 |  |  |  |  end
40 |  |  |  end
41 |  |  end
42 |  end
43 |  k ← k + 1;
44 end
45 Set to white all the nodes that have been in Q;
```

A.3 Dynamic SSSP update for unweighted graphs

Algorithm 5 shows the pseudocode. As in Algorithm 4, 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.

A.4 Fully-dynamic BC approximation

Similarly to IA and IAW, we replace the r sampled paths between vertex pairs (s, t) with new shortest paths between the same vertex pairs. However, here we also check whether $\tilde{VD}$ (and consequently the number r of samples) has increased after the batch of edge updates. If so, we sample additional paths (computing new SSSPs from scratch) according to the new value of r. Instead of updating $\tilde{VD}$ and then the paths in two successive steps, we use the SSSPs from the r source nodes s to compute and update also $\tilde{VD}$, computing new SSSPs only for the components that are not covered by any of the source nodes. In the initialization (Algorithm 6), we first compute the r SSSP, like in RK (Lines 4 - 18). However, we also check which nodes have been visited, as in Algorithm 2. 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 initSSSPVD (Line 8). Since 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, like in Algorithm 2 (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'. The BC update after a batch is described in Algorithm 7. First (Lines 2 - 21), we recompute the shortest paths like in our incremental algorithms IA and IAW [3]: we update the SSSPs from each source node s in R and, in case the distance or the number of shortest paths from s to t has changed (Line 6), we replace the old shortest path with a new one (subtracting $1/r$ to the nodes in the old shortest path and adding $1/r$ to those in the new shortest path). Notice that here we do not store the predecessors so we need to recompute them (Lines 12 and 18). Instead of using an incremental SSSP algorithm like in IA-IAW, here we use the fully-dynamic updateSSSPVD that updates also the VD approximation and updates and keeps track of the nodes that become unvisited. Then (Lines 26 - 33), 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 34

- 35). 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 39).

Algorithm 6: BC initialization

```
1  foreach node v ∈ V do
2  |  c̃_B (v) ← 0; vis(v) ← 0;
3  end
4  ṼD ←getApproxVertexDiameter(G);
5  r ← (c/ε²)(⌊log₂(ṼD − 2)⌋ + ln(1/δ));
6  for i ← 1 to r do
7  |  (s_i, t_i) ← sampleUniformNodePair(V);
8  |  ṼD_i ← initSSSPVD(G, s_i,);
9  |  v ← t_i;
10 |  p_(i) ← empty list;
11 |  P_{s_i}(v) ← {z : {z, v} ∈ E ∩ d_{s_i}(v) = d_{s_i}(z) + ω({z, v})};
12 |  while P_{s_i}(v) ≠ {s_i} do
13 |  |  sample z ∈ P_{s_i}(v) with probability σ_{s_i}(z)/σ_{s_i}(v);
14 |  |  c̃_B(z) ← c̃_B(z) + 1/r;
15 |  |  add z → p_(i); v ← z;
16 |  |  P_{s_i}(v) ← {z : {z, v} ∈ E ∩ d_{s_i}(v) = d_{s_i}(z) + ω({z, v})};
17 |  end
18 end
19 U ← V;
20 i ← r + 1;
21 while U ≠ ∅ do
22 |  extract s′ from U;
23 |  if vis(s′) = 0 then
24 |  |  s′_i ← s′;
25 |  |  ṼD_i ← initApprVD(G, s′_i);
26 |  |  i ← i + 1;
27 |  end
28 end
29 r′ ← r − i − 1;
30 return {(v, c̃_B(v)) : v ∈ V}
```

Algorithm 7: BC update after a batch

$U \leftarrow []$;
for $i \leftarrow 1$ **to** r **do**
  $d_i^{old} \leftarrow d_{s_i}(t_i)$;
  $\sigma_i^{old} \leftarrow \sigma_{s_i}(t_i)$;
  // `updateSSSPVD updates` vis, `inserts all` $v : vis(v) = 0$ `into` U `and updates the` VD `approximation`
  $\tilde{VD}_i \leftarrow$ `updateSSSPVD`(G, s_i, β);
  if $d_{s_i}(t_i) < d_i^{old}$ ***or*** $\sigma_{s_i}(t_i) \neq \sigma_i^{old}$ **then**
    foreach $w \in p_{(i)}$ **do**
      $\tilde{c}_B\ (w) \leftarrow \tilde{c}_B(w) - 1/r$;
    end
    $v \leftarrow t_i$;
    $p_{(i)} \leftarrow$ empty list;
    $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    while $P_{s_i}(v) \neq \{s_i\}$ **do**
      sample $z \in P_{s_i}(v)$ with probability $= \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
      $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
      add z to $p_{(i)}$;
      $v \leftarrow z$;
      $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    end
  end
end
for $i \leftarrow r + 1$ **to** $r + r'$ **do**
  $\tilde{VD}_i \leftarrow$ `updateApprVD`(G, s_i, β);
end
$i \leftarrow r + r' + 1$;
while $U \neq \emptyset$ **do**
  extract s' from U;
  if $vis(s') = 0$ **then**
    $s'_i \leftarrow s'$;
    $\tilde{VD}_i \leftarrow$ `initApprVD`(G, s'_i);
    $i \leftarrow i + 1$; $r' \leftarrow r' + 1$;
  end
end
// `compute the maximum over all the` VD_i `computed by updateApprVD`
$\tilde{VD} \leftarrow \max_{i=1,\ldots,r+r'} \tilde{VD}_i$;
$r_{\text{new}} \leftarrow (c/\epsilon^2)(\lfloor \log_2(\tilde{VD} - 2) \rfloor + \ln(1/\delta))$;
if $r_{new} > r$ **then**
  sample new paths;
  foreach $v \in V$ **do**
    $\tilde{c}_B(v) \leftarrow \tilde{c}_B(v) \cdot r/r_{\text{new}}$
  end
  $r \leftarrow r_{\text{new}}$;
end
return $\{(v, \tilde{c}_B(v)) : v \in V\}$

B Omitted proofs

B.1 Proof of Proposition 1

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 $\tilde{VD}$ 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, $\tilde{VD} \leq 1+(|p_{su}|+|p_{sv}|-2)\cdot\frac{\overline{\omega}}{\underline{\omega}} < 2 \cdot |p_{su}| \cdot \frac{\overline{\omega}}{\underline{\omega}}$, supposing that $|p_{su}| \geq |p_{sv}|$ without loss of generality. By definition of VD, $|p_{su}| \leq VD$. Therefore, $\tilde{VD} < 2 \cdot VD \cdot \frac{\overline{\omega}}{\underline{\omega}}$. □

B.2 Proof of Lemma 1

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, this happens 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)), w can be inserted into the queue again. However, his distance is set to ∞ 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 $O(|\beta| \log |\beta| + ||A|| \log ||A||)$. □

B.3 Proof of Lemma 2

Proof. The complexity of the initialization (Lines 3 - 5) of Algorithm 5 is $O(|\beta|)$, as we have to scan the batch. In the main loop (Lines 6 - 44), we scan all the list of 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 anymore. 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 $O(|\beta| + \|A\| + d_{\max})$. □

B.4 Correctness of Algorithm 2 and Algorithm 3

Lemma 3. *At the end of Algorithm 2, $vis(v) = 1,\ \forall v \in V$ 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 $vis(v) \geq 1,\ \forall v \in V$. On the other hand, $vis(v)$ 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 also s_i and s_j are 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 the connected component of each v, i. e. exactly one *VD* approximation is computed for each connected component of G. □

Lemma 4. *Let $C' = \{C'_1, ..., C'_{n'_c}\}$ be the set of connected components of G after the update. Algorithm 3 updates or computes exactly one VD approximation for each $C'_i \in C'$.*

Proof. Let $C = \{C_1, ..., C_{n_c}\}$ 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) $C_i \in C$ is also a component of C', (ii) $C_i \in C$ and $C_j \in C$ merge into one component C'_k of C', (iii) $C_i \in C$ splits into two components C'_j and C'_k of 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 visits also $s_j \in C_j$, 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. Since all the nodes in C'_k are set to visited during the search, no other *VD* approximations are computed for C'_k. □

B.5 Proof of Theorem 1

Proof. In the first part (Lines 2 - 9 of Algorithm 3), 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, for Lemmas 1 and 2. Only some of the affected nodes (those whose distance from a source node becomes equal to ∞) 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 anymore. However, also 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. □

B.6 Proof of Theorem 2

Proof. Let G be the old graph and G' 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 7 replaces the first r shortest paths with other shortest paths $p'_{(1)}, ..., p'_{(r)}$ between the same node pairs (Lines 13 - 19) using Algorithm 4.1 of [3], for which it was already proven that $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)} \frac{1}{\sigma'_x(y)}$ [3, Theorem 4.1]. The additional Δr shortest paths (Line 37) 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 [17]. □

B.7 Proof of Theorem 3

Proof. Let $\Delta r'$ be the difference between the values of r' before and after the batch. Let us start from the simplest case: the graph G is such that there is (before and after the update) one sample in each component and *VD* does not increase after the update. 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*. In this case, $\Delta r = 0$ and $\Delta r' = 0$ and we only need to update the r old shortest paths. Then, the total complexity is $O(r \cdot |\beta| + \sum_{i=1}^{r} (||A^{(i)}|| + d_{\max}^{(i)}))$, where $A^{(i)}$ is the set of nodes affected in the ith SSSP, and $d_{\max}^{(i)}$ is the maximum distance in the ith SSSP. In general graphs, 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 $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})) + O(\Delta r(|V| + |E|))$. 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. □

C Additional Experimental Results

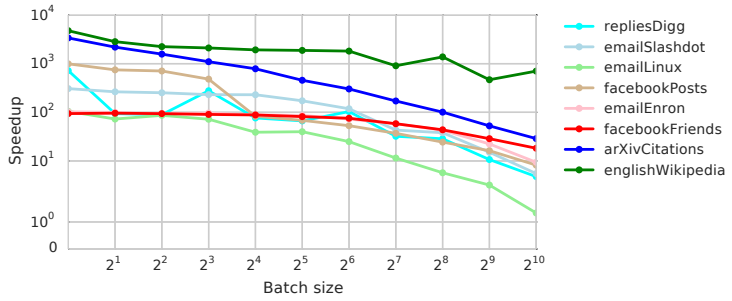

Fig. 3: Speedups on RK in real unweighted graphs under random updates.

	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 3: 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 4: Times and speedups of DA on RK in hyperbolic unit-disk graphs, for batch sizes of 1 and 1024.