Scalable Online Betweenness Centrality in Evolving Graphs

Nicolas Kourtellis
Telefonica I+D, Barcelona, Spain
nicolas.kourtellis@telefonica.com

Gianmarco De Francisci Morales
Aalto University, Helsinki, Finland
gdfm@acm.org

Francesco Bonchi
The ISI Foundation, Torino, Italy
francesco.bonchi@isi.it

I. Introduction

Betweenness centrality measures the importance of an element of a graph, either a vertex or an edge, by the fraction of shortest paths that pass through it [1]. This measure is notoriously expensive to compute, and the best known algorithm, proposed by Brandes [2], runs in $\mathcal{O}(nm)$ time. The problems of efficiency and scalability are exacerbated in a dynamic setting, where the input is an evolving graph seen edge by edge, and the goal is to keep the betweenness centrality up to date. In this paper [8] we propose the first truly scalable and practical framework for computing vertex and edge betweenness centrality of large evolving graphs, incrementally and online. Our proposal represents an advancement over the state of the art in four main aspects as summarized in Table I.

Consider a $G = (V, E)$ with $|V| = n$ and $|E| = m$. Let $P_s(t)$ denote the *set of predecessors* of a vertex t on shortest paths from s to t in G. Let $\sigma(s,t)$ denote the *total number of shortest paths* from s to t in G and, for any $v \in V$, let $\sigma(s, t \mid v)$ denote the number of shortest paths from s to t in G that *go through* v. Note that $\sigma(s,s) = 1$, and $\sigma(s, t \mid v) = 0$ if $v \in \{s, t\}$ or if v does not lie on any shortest path from s to t. Similarly, for any edge $e \in E$, let $\sigma(s, t \mid e)$ denote the number of shortest paths from s to t that *go through* e. The betweenness centrality of a vertex $v \in V$ is the sum over all pairs of vertices of the fractional count of shortest paths going through v: $VBC(v) = \sum_{s,t \in V, s \neq t} \frac{\sigma(s,t|v)}{\sigma(s,t)}$. Similarly, the betweenness centrality $EBC(e)$ of an edge e is the sum over all pairs of end vertices of the fractional count of shortest paths going through e.

Brandes' algorithm [2] leverages the notion of *dependency score* of a source vertex s on another vertex v, defined as $\delta_s(v) = \sum_{t \neq s,v} \frac{\sigma(s,t|v)}{\sigma(s,t)}$. The betweenness centrality $VBC(v)$ of any vertex v can be expressed in terms of dependency scores as $VBC(v) = \sum_{s \neq v} \delta_s(v)$. The following recursive relation on $\delta_s(v)$ is the key to Brandes' algorithm:

$$\delta_s(v) = \sum_{w : v \in P_s(w)} \frac{\sigma(s,v)}{\sigma(s,w)} (1 + \delta_s(w)) \qquad (1)$$

The algorithm runs in two phases. During the first phase, it performs a search on the whole graph to discover shortest paths, starting from every source vertex s. When the search ends, it performs a *dependency accumulation* step by backtracking along the shortest paths discovered. During these two phases, the algorithm maintains four data structures for each vertex found on the way: a predecessors list $P_s[v]$, the distance $d_s[v]$ from the source, the number of shortest paths from the source $\sigma_s[v]$, and the dependency $\delta_s[v]$ accumulated when backtracking at the end of the search.

Method	Year	Space	C_V	C_E	+	−	‖	$\lvert V\rvert$	$\lvert E\rvert$
Lee et al. [7]	2012	$\mathcal{O}(n^2{+}m)$	✓	✗	✓	✓	✗	12k	65k
Green et al. [3]	2012	$\mathcal{O}(n^2{+}nm)$	✓	✗	✓	✗	✗	23k	94k
Kas et al. [6]	2013	$\mathcal{O}(n^2{+}nm)$	✓	✗	✓	✗	✗	8k	19k
Nasre et al. [5]	2014	$\mathcal{O}(n^2)$	✓	✗	✓	✗	✗	-	-
This work	2014	$\mathcal{O}(n^2)$	✓	✓	✓	✓	✓	2.2M	5.7M

TABLE I: *Comparison with previous studies: vertex (C_V) and edge centrality (C_E), edge addition (+) and removal (-), parallel and streaming computation (‖), size of the largest graph used in the experiments ($|V|$, $|E|$).*

On unweighted graphs, Brandes' algorithm uses a breadth first search (BFS) to discover shortest paths: its running time is $\mathcal{O}(nm)$ and its space complexity is $\mathcal{O}(m + n)$.

II. Overview of Framework

We assume new edges are added to the graph or existing edges are removed from the graph, and these changes are seen as a stream of updates, i.e., one by one. Even though we focus on undirected graphs, our method can be easily applied to directed graphs as well.

The proposed framework is composed of two basic steps shown in Figure 1. It accepts as input a graph $G(V, E)$ and a stream of edges E_S to be added/removed, and outputs, for an updated graph $G'(V', E')$, the new betweenness centrality of vertices ($VBC'(v), v \in V'$) and edges ($EBC'(e), e \in E'$). The framework uses Brandes' algorithm as a building block in step 1 (executed once, offline, before any update).

Input: Graph $G(V, E)$ and edge update stream E_S
Output: $VBC'[V']$ and $EBC'[E']$ for updated $G'(V', E')$
Step 1: Execute Brandes' alg. on G to create & store data structures for incremental betweenness.
Step 2: **For each** update $e \in E_S$, execute algorithm 1 in [8].
Step 2.1 Update vertex and edge betweenness.
Step 2.2 Update data structures in memory or disk for next edge addition or removal.

Fig. 1: *The proposed algorithmic framework.*

Edge betweenness. Our method maintains *both* vertex and edge betweenness centrality up-to-date for the same overall computational cost, while the previously proposed methods are only tailored for vertex betweenness. To compute simultaneously both betweenness scores, the algorithm stores intermediate dependencies (Eq. 1) independently for each vertex.

Graph updates. We handle *both additions and removals* of edges in a unified approach, while the previously proposed methods, besides QUBE [7], can handle only addition of edges. To allow for this incremental computation, we need to maintain some additional data. In particular, we need a compact representation of the *directed acyclic graph of shortest paths rooted in the source vertex* (which we refer to as SPDAG), and the accumulated dependency values. Thus, for each source vertex s we maintain an additional data structure $BD[s]$ that stores its *betweenness data*. For each other vertex t, $BD[s]$ stores the distance of t from source s, the number of shortest paths starting from source s and ending at the given vertex t, and the dependency accumulated on the vertex t in the backtracking to source s. The data structure is initialized in step 1, and populated at the end of the dependency accumulation phase. Then, it is used in step 2.1 and updated in step 2.2.

Memory optimization. Brandes' algorithm builds a list of predecessors during the search phase to speed up the backtracking phase. Differently from the other data structures, the size of this list is variable and can grow considerably. To reduce the space complexity of the algorithm, we remove the predecessors lists. When backtracking in the dependency accumulation phase, the algorithm checks all the neighbors of the current vertex, and uses the level of the vertex in the SPDAG (i.e., the distance from the source) to pick the next vertices to visit. For each source, we need to maintain an SPDAG to all other vertices in the graph ($\mathcal{O}(n)$), along with the edges to their predecessors ($\mathcal{O}(m)$). Therefore, each SPDAG takes $\mathcal{O}(n+m)$ space. In total, the space complexity of the original algorithm is $\mathcal{O}(n(n+m))$ with the predecessors lists. By removing the predecessors lists, we reduce the space complexity to $\mathcal{O}(n^2)$ [4]. Furthermore, the time complexity remains unchanged and is reduced in practice, since we avoid the overhead of building and parsing these variable-length lists during the traversal. This simplification allows us to use very efficient out-of-core techniques to manage $BD[\cdot]$ when its size outgrows the available main memory.

Scalability & online updates. Our framework is truly scalable and amenable to real-world deployment [8]. The framework is carefully engineered to use *out-of-core* techniques to store its data structures on disk in a compact binary format. Data structures are read sequentially by employing *columnar storage*, and memory structures are *mapped directly on disk* to minimize memory copies. Furthermore, the method can be parallelized and deployed on top of modern parallel data processing engines that run on clusters of commodity hardware, such as Storm or Hadoop. This scalability also allows the algorithm to keep the betweenness centrality up-to-date *online*, i.e., the time to update the measure is always smaller than the inter-arrival time between two consecutive updates. An open-source implementation of our method is available on GitHub.[1]

III. Evaluation

We evaluate three versions of our framework (in memory with/without predecessor lists, and on disk without predecessor lists), using real and synthetic datasets on a large scale parallel distributed engine. We assess the speedup achieved over Brandes' algorithm and show how this speedup outperforms past

[1] http://github.com/nicolas-kourtellis/StreamingBetweenness

Dataset	$\|V\|$	$\|E\|$	Addition			Removal		
			Min	Med	Max	Min	Med	Max
1k	1k	6k	3	12	23	2	10	19
10k	10k	59k	16	34	62	2	35	155
100k	100k	588k	21	49	96	4	45	134
1000k	1000k	5897k	5	10	20	1	12	78
wikielections	7k	101k	9	47	95	1	45	92
slashdot	51k	117k	15	25	121	8	24	127
facebook	63k	817k	10	66	462	1	102	243
epinions	119k	705k	24	56	138	2	45	90
dblp	1105k	4835k	3	8	15	3	8	429
amazon	2146k	5743k	2	4	15	2	3	5

TABLE II: *Summary of key speedup results over Brandes'.*

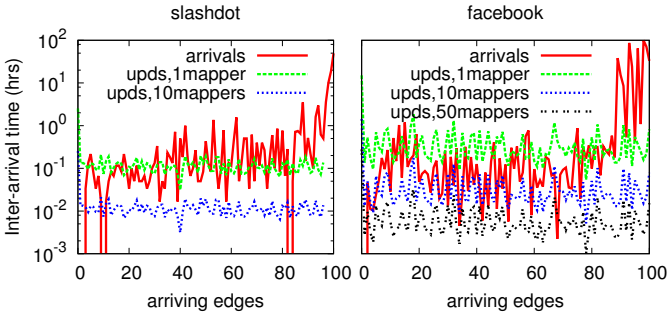

Fig. 2: *Inter-arrival time of facebook and slashdot edges and update times for betweenness centrality.*

works in many of the graphs tested. Furthermore, we demonstrate the scalability properties of our framework to handle larger graphs just by adding more computation resources. Due to space constraints, we summarize only key speedup results in Table II, for more results refer to the main paper [8].

In addition, we investigate how the scalability of the framework enables the *online* computation of betweenness on two real graphs. Figure 2 shows the inter-arrival time of new edges, and the time needed by the framework to produce updated betweenness values. For slashdot, the framework manages to produce online updates for 98.91% of edges with 10 machines. For facebook instead, the arrival rate is higher and 10 machines are not enough. However, the scalability of our algorithm allows to simply use more machines to decrease the response time of the system. Thus, with 100 machines, the system can produce online updates for 98.99% of edges.

References

[1] J. Anthonisse. The rush in a directed graph. Technical Report BN9/71, Stichting Mathematisch Centrum, Amsterdam, Netherlands, 1971.

[2] U. Brandes. A faster algorithm for betweenness centrality. *J. of Mathematical Sociology*, 25(2):163–177, 2001.

[3] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *SOCIALCOM-PASSAT*, 2012.

[4] O. Green, and D. A. Bader. Faster betweenness centrality based on data structure experimentation. In *International Conference on Computational Science*, 2013.

[5] M. Nasre, M. Pontecorvi, and V. Ramachandran. Betweenness Centrality Incremental and Faster. In *Mathematical Foundations of Computer Science*, 577–588, LNCS(8635), 2014.

[6] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *ASONAM*, 2013.

[7] M.-J. Lee, J. Lee, J. Park, R. Choi, and C.-W. Chung. A quick algorithm for updating betweenness centrality. In *WWW*, 2012.

[8] N. Kourtellis, G. De Francisci Morales and F. Bonchi. Scalable Online Betweenness Centrality in Evolving Graphs. In *TKDE, 27(9):2494–2506 (2015).*