Space Efficient Incremental Betweenness Algorithm for Directed Graphs

Reynaldo Gil-Pons(✉)

CERPAMID, Santiago de Cuba, Cuba
rey@cerpamid.co.cu
https://www.cerpamid.co.cu/

Abstract. Betweenness is one of the most popular centrality measures in the analysis of social networks. Its computation has a high computational cost making it implausible for relatively large networks. The dynamic nature of many social networks opens up the possibility of developing faster algorithms for the dynamic version of the problem. In this work we propose a new incremental algorithm to compute the betweenness centrality of all nodes in directed graphs extracted from social networks. The algorithm uses linear space, making it suitable for large scale applications. Our experimental evaluation on a variety of real-world networks have shown our algorithm is faster than recalculation from scratch and competitive with recent approaches.

Keywords: Social network analysis · Betweenness centrality · Dynamic algorithms · Dynamic graphs

1 Introduction

Centrality is one of the most important concepts in the analysis of social networks. Among centrality measures, one of the most popular is betweenness centrality [1,6]. The betweenness of a node is a measure of the control this node has on the communication paths in the network. Therefore, it can be used to rank nodes according to their relative importance in a graph. Betweenness has been used effectively in a variety of applications, such as: design and control of communications networks [15], traffic monitoring [13], identifying key actors in terrorist networks [11], finding essential proteins [8], and many others.

Computing the betweenness of all nodes in a network has a high computational cost, so efficiency is the target of much related research. Nowadays, most graphs are inherently dynamic. When a graph suffers small changes, recomputing betweenness from scratch would be very inefficient. Therefore, dynamic algorithms capable of computing betweenness faster by using previous computations have been proposed [10,12]. None of these is better than Brandes [3] (**brandes**) in the worst case, and there is evidence that this is likely very hard to overcome [16]. Despite that, good speedups in typical instances have been achieved [2,7].

R. Vera-Rodriguez et al. (Eds.): CIARP 2018, LNCS 11401, pp. 262–270, 2019.
https://doi.org/10.1007/978-3-030-13469-3_31

In this work, we focus on the exact computation of betweenness centrality in incremental graphs. While not allowing edges to be deleted, incremental graphs cover some important applications, as has been pointed by several authors before [2,9,12]. Two recently proposed algorithms deal with the same problem, obtaining better performance than previous work, so we compare with them:

1. **icentral** [7] works on undirected connected graphs, and allows edges to be deleted and inserted. It only stores the betweenness of all nodes of the graph, so memory requirement is linear. First, it decomposes the graph into biconnected components, and then updates betweenness of nodes in the component affected by the update. In the article it's proven that for undirected graphs, the betweenness can change only for nodes in the affected component. Its time complexity is highly dependent on the size of the affected biconnected component.
2. **ibet** [2] works on directed graphs, and allows edges to be inserted. It stores all distances between pairs of nodes, so memory requirement is quadratic. First, it identifies efficiently all pairs of nodes which distance or number of shortest paths are affected by the update. Then it applies an optimized procedure to calculate changes in betweenness for nodes affected by the update. Experiments showed it outperforms previous approaches requiring quadratic memory.

In this paper we present a space efficient algorithm to compute the betweenness centrality of all nodes in a directed incremental network. Its space complexity is linear in the size of the input graph and its time complexity is similar to that of **icentral**. In the worst case, it's equivalent to recalculating betweenness in the biconnected component where the added edge resides, plus some linear overhead. Up to the authors knowledge it's the first algorithm calculating betweenness centrality in incremental directed graphs, showing better performance than recalculation, and at the same time, having less than quadratic space complexity. On the other hand, it works with disconnected graphs, detail usually left out by previous approaches, but important in real world applications.

In the next section we define betweenness, biconnected component and incremental algorithms. In Sect. 3 we present the proposed algorithm, prove its correctness, and determine space and time complexity. In Sect. 4 we show the experimental validation of our algorithm. At the end, the conclusions and references.

2 Preliminaries

For simplicity, we will refer to directed, simple and unweighted graphs. In the following we will refer to a graph $G = (V, E)$ with n nodes and m edges.

2.1 Betweenness

Betweenness centrality of a node is formally defined by the following formula:

$$C_B(v) = \sum_{\substack{s \neq v, t \neq v \\ s,t \in V}} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

where $\sigma_{st}(v)$ is the number of shortest paths from s to t passing through v and σ_{st} is the number of shortest paths from s to t. A naive algorithm using this formula has $\mathcal{O}(n^3)$ complexity.

In [3] Brandes showed a more efficient way to calculate betweenness values:

$$C_B(v) = \sum_{s \neq v, s \in V} \delta_{s\cdot}(v) \tag{2}$$

where $\delta_{s\cdot}(v) = \sum_{s \neq v, t \neq v, t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Using this formula betweenness values can be computed in time $\mathcal{O}(n \cdot m)$, by running a BFS (Breath First Search [4]) on each node and computing the required values (distances, σ, δ). For a complete explanation see [3].

2.2 Biconnected Components

Biconnected components were first proposed as a good heuristic for speeding up betweenness computations in [14], and more recently in the context of dynamic graphs in [7]. We will make use of the following definitions:

Definition 1. *Let G be an undirected graph. A biconnected component is a connected induced subgraph A of G, such that the removal of any node doesn't disconnect A, and is maximal.*

Definition 2. *Any node belonging to more than one biconnected component is called articulation point.*

2.3 Incremental Graphs

We call a dynamic graph incremental if edges can be inserted, but not deleted. As previously mentioned in this work we focus on incremental graphs. Computing betweenness in such context is usually done in two steps. In the first step some preprocessing is done and initial betweenness is computed. Next, after each edge insertion, betweenness is updated. The two steps could have different time complexities, so both define the time complexity of an incremental algorithm. All algorithms mentioned here have the same complexity in the first step (the same as **brandes**), so in comparisons we will only take into account the update step.

3 Algorithm

The proposed algorithm is a generalization of **icentral** to deal with directed graphs.

Definition 3. *Let G be a graph, and let G^* be the graph G after inserting a new edge (u, v). We define affected component as the biconnected component of the undirected version of G^* to which the newly inserted edge (u, v) belongs.*

The main obstacle in generalizing **icentral** is that, in directed graphs, when an edge is inserted, betweenness values of nodes outside the affected component can change as well. In the next theorem we prove a formula allowing to compute those changes efficiently.

Theorem 1. *Let x be a node outside the affected component A, and let s be the articulation point inside the component such that its removal disconnects x from A. Then after the update, the betweenness of x changes by*

$$\delta_s(x) \cdot (reach^*(s) - reach(s)) + \delta_s^r(x) \cdot (reach^{*r}(s) - reach^r(s)) \tag{3}$$

where $reach(s)$ equals the number of nodes z such that there exists a shortest path from z to x passing through A, superscript r means the function is applied to the reversed graph, and superscript $*$ indicates the function is applied to the updated graph.

Proof. For the sake of clearness, lets rename variables in the definition of betweenness 1:

$$C_B(x) = \sum_{\substack{a \neq x, b \neq x \\ a,b \in V}} \frac{\sigma_{ab}(x)}{\sigma_{ab}} \tag{4}$$

In the sum on the right the only terms that can change after an update are such that a and b are in different biconnected components, and such that all shortest paths from a to b pass through A. Therefore, all these paths must pass through s. Then, two cases may occur, according to the relative orders of s and x in the paths from a to b that go through x:

1. $a, s, x, b \implies \frac{\sigma_{ab}(x)}{\sigma_{ab}} = \frac{\sigma_{as}\sigma_{sx}\sigma_{xb}}{\sigma_{as}\sigma_{sb}} = \frac{\sigma_{sx}\sigma_{xb}}{\sigma_{sb}} = \frac{\sigma_{sb}(x)}{\sigma_{sb}}$
2. $a, x, s, b \implies \frac{\sigma_{ab}(x)}{\sigma_{ab}} = \frac{\sigma_{ax}\sigma_{xs}\sigma_{sb}}{\sigma_{as}\sigma_{sb}} = \frac{\sigma_{ax}\sigma_{xs}}{\sigma_{as}} = \frac{\sigma_{as}(x)}{\sigma_{as}}$

Therefore, the terms that can change equal:

$$\sum_{a,b \text{ in case } 1} \frac{\sigma_{sb}(x)}{\sigma_{sb}} + \sum_{a,b \text{ in case } 2} \frac{\sigma_{as}(x)}{\sigma_{as}} = \text{reach}(s) \cdot \delta_s(x) + \text{reach}^r(s) \cdot \delta_s^r(x) \tag{5}$$

and the theorem easily follows.

Following Theorem 1, pseudocode for the function updating betweenness outside A is shown in Algorithm 3. Then, it remains to update betweenness inside the component; this can be done as in **icentral**, and is shown in Algorithm 2. The Brandes-like function in lines 8 and 9 computes delta values in the affected component, as in **icentral**, using reach_o^r values to add the contribution of nodes outside the affected component. r and $*$ have the same meaning as in Theorem 1. The pseudocode of the proposed algorithm is shown in Algorithm 1.

Algorithm 1. Update Betweenness

procedure UPDATE-BETWEENNESS(G, (u, v), C_B)
  $A \leftarrow$ Affected-Biconnected-Component$(G, (u, v))$
  $A^* \leftarrow A$ with edge (u, v) inserted
  update-betweenness-inside$(G, A, (u, v), A^*, C_B)$
  update-betweenness-outside(G, A, A^*, C_B)
  return C_B

Algorithm 2. Update Betweenness Inside Affected Component

1: **procedure** UPDATE-BETWEENNESS-INSIDE(G, A, (u, v), A^*, C_B)
2: $Sr \leftarrow$ Affected-Sources$(A, (u, v))$
3: **for all** $s \in$ articulation-points(A) **do**
4: reach$_o(s) \leftarrow$ number of nodes directly reaching s outside A
5: reach$_o^r(s) \leftarrow$ number of nodes directly reachable from s outside A
6: **for all** $s \in Sr$ **do**
7: $\delta_s \leftarrow$ Brandes-like(A, s)
8: $\delta_s^* \leftarrow$ Brandes-like(A^*, s)
9: **for all** $x \in A$ **do**
10: $C_B(x) \leftarrow C_B(x) + \delta_s^*(x) - \delta_s(x)$
11: **if** s is articulation point **then**
12: **for all** $x \in A$ **do**
13: $C_B(x) \leftarrow C_B(x) + (\delta_s^*(x) - \delta_s(x))\cdot$reach$_o(s)$

3.1 Complexity

The overall space complexity is linear (in the size of the graph) as the algorithm only uses a constant number of arrays with linear size (C_B, the different variants of reach, A, A^*, and the different variants of δ). Only C_B and the graph itself persist across updates.

Time complexity of the proposed algorithm (Algorithm 1) equals the complexity of finding biconnected components (linear), plus the complexity of Algorithm 2, plus the one of Algorithm 3. Let n_A and m_A be the number of nodes and edges respectively in the affected component. Algorithm 2 has exactly the same complexity as **icentral**, which is $\mathcal{O}(n + m + |Sr| * (n_A + m_A))$, where Sr is the set of affected sources (as defined in [7]).

In Algorithm 3, for a given s all variants of reach (lines 3 and 7) can be computed using BFS in time $\mathcal{O}(n_A + m_A)$, as there is no need to do any computation outside A at this point. As any node outside A will have at most one corresponding articulation point s, in lines 4, 5, 6, 8, 9, and 10 each node and edge of the graph appears at most once, and so the total complexity of these is $\mathcal{O}(n + m)$. Summing up, the complexity of Algorithm 3 is $\mathcal{O}(n + m + |\text{articulation-points}(A)| * (n_A + m_A))$.

Overall, using that there are at most n_A articulation points in A, and also at most n_A affected sources, complexity of the proposed algorithm is proven to be $\mathcal{O}(n + m + n_A * (n_A + m_A))$, matching that of **icentral**.

Algorithm 3. Update Betweenness Outside Affected Component

```
1: procedure UPDATE-BETWEENNESS-OUTSIDE(G, A, A*, C_B)
2:     for all s ∈ articulation-points(A) do
3:         Compute reach(s), reach*(s) using BFS (defined in Theorem 1)
4:         δ_s ← δ values of nodes reachable from s outside A
5:         for all x ∈ nodes directly reachable from s outside A do
6:             C_B(x) ← C_B(x) + δ_s(x) · (reach*(s)−reach(s))
7:         Compute reach^r(s), reach^{*r}(s) using BFS (defined in Theorem 1)
8:         δ^r_s ← δ values of nodes that reach s outside A
9:         for all x ∈ nodes directly reaching s outside A do
10:            C_B(x) ← C_B(x) + δ^r_s(x) · (reach^{*r}(s)−reach^r(s))
```

3.2 Notes

It's possible to modify slightly the proposed algorithm to work with graphs with arbitrary positive weights, by using Dijkstra algorithm [5] instead of BFS. In graphs with multiples edges, parallel edges can be substituted with the edge with smallest weight, and then obtain a simple graph with the same betweenness.

On the other hand, it's straightforward to parallelize the most time consuming part of the algorithm, the computation of the betweenness changes inside the affected component. As the δ values respect to affected sources are computed independently, this computations could be done by different nodes in a parallel environment. In this environment, good speedups are expected, similar to those in [7].

4 Experiments

We experimentally evaluate the proposed algorithm by measuring time and memory, and comparing it with **icentral**, **ibet** and **brandes**. All algorithms were implemented in pure python, and graphs were stored and manipulated using the python library NetworkX[1]. Algorithms were run on a GNU/Linux 64 bit machine, processor Intel(R) Core(TM) i3-4160 CPU @ 3.60 GHz, with 5 GBytes of main memory.

The datasets used for experimentation were taken from online sources, some of them being already referenced in [2] or [7]; p2p-Gnutella08, Wiki-Vote, and CollegeMsg, were taken from SNAP graphs collection. The description of the data is shown in Table 1.

For each graph, we randomly selected 100 edges that were not already contained in the graph, and measured the average time and maximum memory used by each algorithm to update the betweenness of all nodes when each edge is inserted. In the case of algorithms that work with directed graphs, when testing on an undirected one, each edge was transformed into two edges, one for each possible direction. Results are shown in Table 2. Note it's not possible to test **icentral** on directed graphs.

[1] http://networkx.github.io/.

Table 1. Statistics of graph datasets (lbc refers to largest biconnected component)

Graph dataset	# nodes	# edges	Diameter	lbc	Edge type
CollegeMsg	1893	20292	8	1498	Directed
Cagr	4158	13428	16	2651	Undirected
Epa	4253	8897	10	2163	Undirected
Eva	4475	4654	17	234	Undirected
p2p-Gnutella08	6299	20776	9	4535	Directed
Wiki-Vote	7066	103663	7	4786	Directed

Table 2. Results, time given in seconds and memory in MBytes.

Graph dataset	**brandes**		**ibet**		**icentral**		Ours	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
CollegeMsg	2.7	**131**	**0.7**	380	-	-	2.2	154
Cagr	31.0	**148**	**4.9**	1643	15.3	145	12.3	175
Epa	26.4	**146**	**4.2**	1851	9.0	143	8.2	167
Eva	17.1	142	6.8	1989	**0.8**	**135**	1.2	159
p2p-Gnutella08	18.1	**148**	**3.0**	3249	-	-	13.3	179
Wiki-Vote	27.6	**198**	**2.9**	3781	-	-	10.4	331

As expected, both our algorithm and **icentral** perform very similar, both in time and memory, and are consistently faster than **brandes**. This speedup is highly dependent on the size of the affected component. Best performance respect to **brandes** was obtained in dataset Eva, where the number of nodes in the largest biconnected component is relatively small. On average, our algorithm is between 2 and 3 times faster than **brandes**.

On the other hand, **ibet** is the fastest of all on most datasets, but it's memory usage is very high, making it very expensive for graphs of tens of thousands of nodes. Also note, that for datasets like Eva, **ibet** is outperformed by algorithms **icentral** and our proposal, stressing the relevance of algorithms using the biconnected components decomposition.

5 Conclusions

In this work an algorithm for computing betweenness in incremental directed graphs has been proposed. Its memory usage is linear allowing it to scale to large graphs. It's time complexity is similar to that of algorithm proposed in [7], despite of handling the more general case of directed graphs. Experiments have proven it can be a practical replacement of **brandes** for directed and undirected graphs, mostly when quadratic memory usage is not feasible due to large input.

As future work we plan to conduct experiments with a distributed and parallel implementation of the proposed algorithm. Also, we will extend the proposed algorithm to work with edge deletions. Moreover, it seems possible to apply some of the optimizations proposed in [2] to update betweenness values inside the affected biconnected component.

References

1. Anthonisse, J.M.: The Rush in a Directed Graph. Stichting Mathematisch Centrum, Mathematische Besliskunde (1971)
2. Bergamini, E., Meyerhenke, H., Ortmann, M., Slobbe, A.: Faster betweenness centrality updates in evolving networks. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 75, pp. 1–16 (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.23
3. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Soc. **25**(2), 163–177 (2001). https://doi.org/10.1080/0022250X.2001.9990249
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
5. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959). https://doi.org/10.1007/BF01386390
6. Freeman, L.C.: A set of measures of centrality based on betweenness (1977). https://doi.org/10.2307/3033543
7. Jamour, F., Skiadopoulos, S., Kalnis, P.: Parallel algorithm for incremental betweenness centrality on large graphs. IEEE Trans. Parallel Distrib. Syst. (2017). https://doi.org/10.1109/TPDS.2017.2763951
8. Joy, M.P., Brock, A., Ingber, D.E., Huang, S.: High-betweenness proteins in the yeast protein interaction network. J. Biomed. Biotechnol. **2005**(2), 96–103 (2005). https://doi.org/10.1155/JBB.2005.96
9. Kas, M., Wachs, M., Carley, K.M., Carley, L.R.: Incremental algorithm for updating betweenness centrality in dynamically growing networks. In: Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining - ASONAM 2013, pp. 33–40 (2013). https://doi.org/10.1145/2492517.2492533
10. Kourtellis, N., Morales, G.D.F., Bonchi, F.: Scalable online betweenness centrality in evolving graphs. IEEE Trans. Know. Data Eng. **27**(9), 2494–2506 (2015). https://doi.org/10.1109/TKDE.2015.2419666
11. Krebs, V.E.: Mapping networks of terrorist cells. Connections **24**(3), 43–52 (2002)
12. Nasre, M., Pontecorvi, M., Ramachandran, V.: Betweenness centrality – incremental and faster. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) MFCS 2014. LNCS, vol. 8635, pp. 577–588. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44465-8_49
13. Puzis, R., Altshuler, Y., Elovici, Y., Bekhor, S., Shiftan, Y., Pentland, A.S.: Augmented betweenness centrality for environmentally aware traffic monitoring in transportation networks. J. Intell. Transp. Syst. **17**(1), 91–105 (2013). https://doi.org/10.1080/15472450.2012.716663

14. Puzis, R., Zilberman, P., Elovici, Y., Dolev, S., Brandes, U.: Heuristics for speeding up betweenness centrality computation. In: Proceedings - 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust and 2012 ASE/IEEE International Conference on Social Computing, SocialCom/PASSAT 2012, pp. 302–311 (2012). https://doi.org/10.1109/SocialCom-PASSAT.2012.66
15. Tizghadam, A., Leon-Garcia, A.: Betweenness centrality and resistance distance in communication networks. IEEE Netw. **24**(6), 10–16 (2010). https://doi.org/10.1109/MNET.2010.5634437
16. Williams, V.V.: On some fine-grained questions in algorithms and complexity. In: Proceedings of the ICM (2018)