Real Time Closeness and Betweenness Centrality Calculations on Streaming Network Data

Wei Wei
School of Computer Science
Carnegie Mellon University, Pittsburgh, PA, U.S.A
weiwei@cs.cmu.edu

Kathleen M. Carley
School of Computer Science
Carnegie Mellon University, Pittsburgh, PA, U.S.A
kathleen.carley@cs.cmu.edu

Abstract

Closeness and betweenness are among the most important metrics in social network analysis. They are essential to the evaluation of various research problems such as viral marketing, network stability and network traffic predictions, which play an important role in social media research. However, both of these metrics are expensive to compute. We propose an efficient online algorithm framework to handle both closeness and betweenness in the situation where network structure changes frequently. Whenever a link change is received as the input, the algorithm utilizes existing facts about the calculation to update centrality values with minimal effort. Experimental results on data sets collected from online social media websites show that our approach is 4-7 orders of magnitude faster for closeness and 2-4 orders of magnitude faster for betweenness calculations over baseline methods. We also show how those two metrics share some common calculations so that the running time can be dramatically reduced when calculated together. To the best of our knowledge, this is the first work to improve the running time when those two algorithms are calculated at the same time on streaming network data.

Keywords: betweenness; closeness; streaming network data; social network; real-time network analysis; centrality; fast metrics

1. Introductions

Social networks consist of agents and their connections to each other. In social network analysis, the assessment of node position plays an important role in understanding various research questions such as information diffusion [1], network dynamics [2], behavior analysis [3] and community detection [4]. Multiple metrics to assess node level performance have been proposed [5-7]. Among all these assessments, two particular network metrics are particularly interesting because of their relationships to the shortest path problem, which usually represents the optimal way to reach an objective in a network. The first one, closeness [5], evaluates the difficulties for a node to reaching other nodes through shortest path in the network. The second metric, betweenness [5] measures the importance of a node based on the number of shortest paths pass through it in the network. These metrics have seen abundant applications in various research topics [8-10].

Despite the usefulness of closeness and betweenness, the practice of applying those two important metrics on streaming network data faces great challenges. In streaming network data, information that modifies the network (e.g. by adding a link, deleting a link, or by modifying link weights) is organized into data streams. Unfortunately, state of the art algorithms are designed to work on static networks and perform poorly on streaming data. Take betweenness for example, it takes $O(VE + V^2 \, logV)$ time for the most widely used algorithm[11] to calculate, where V and E are the number of vertices and the number of links in the network, respectively. Given the fact that social networks usually have millions of users and connections, it's impossible for these algorithms to respond to streaming network data in real time.

In this paper, we build an efficient online algorithm framework to handle the calculations of both closeness and betweenness on streaming network data that contains link changes organized in a streaming fashion. The algorithm has the merits of responding to streaming data efficiently by avoiding unnecessary calculations that have already been done in the previous time steps. We divide the calculation of both metrics into a unified two-step process: Convergence and Aggregation. In the convergence step, a calculation process will be repeated until the shortest path is converged. In the aggregation step, those shortest path calculations will be aggregated into closeness and betweenness centralities. Both of the steps will be updated incrementally and are optimized for streaming network data. We also show that the most part of the convergence steps of both the betweenness and closeness algorithms can be shared and only the aggregation steps need to be done separately. This further decreases the running time if those two metrics are calculated together. To the best of our knowledge, this is the first paper to improve the running time of the two metrics when they are calculated together on streaming data.

The rest of the paper is be organized as follows: Section 2 will introduce background information of the paper. Section 3 will detail the methodologies. Section 4 is the experimental section where we compare the running time of our algorithms. Section 5 will conclude our paper.

2. Preliminary

2.1 Definitions

In a network with V nodes and E edges, the closeness of specific node k, noted as $C_C(k)$ is defined in Equation (1) to be the inverse of the sum of shortest paths $d_{k,t}$ from node k to some other node t. Closeness measures the average cost to reach other nodes in a network through shortest path. The higher this measure is, the less it costs for a node to reach the rest of the network.

$$C_C(k) = \frac{1}{\sum_{t=1}^{|V|} d_{k,t}} \quad (1)$$

Betweenness utilizes shortest path information in a different way. It is defined in Equation (2) to be the sum of the ratio of the number of shortest paths passing through node k. Here $\sigma_{s,t}$ is the number of shortest path from *s* to *t* and $\sigma_{s,t}(k)$ is the number of shortest paths from *s* to *t* that go through *k*. Nodes with high betweenness are typically the hubs of the network due to their central positions. Removing those nodes will result in changing a significant amount of shortest paths in the network.

$$C_B(k) = \sum_{s=1,s\neq k}^{|V|} \sum_{t=1,t\neq s,t\neq k}^{|V|} \frac{\sigma_{s,t}(k)}{\sigma_{s,t}} \quad (2)$$

2.2 Algorithms for Static Networks

The use of closeness and betweenness is significantly limited by the complexity of the calculations. To calculate closeness for a static network, the best algorithm requires at least $O(VlogV + E)$ time to compute [12] [13] by utilizing the all pairs shortest path algorithms. The calculation of betweenness is more complicated since it requires not only the correct computation of shortest paths but also correct number of paths that pass through each node. To calculate betweenness for a static network, the best algorithm in general case is Brandes' approach [14], which avoids many unnecessary calculations by using a special sequence. Brandes introduced a quantity $\delta_{s,*}(k)$ which is defined in equation (3).

$$\delta_{s,*}(k) = \sum_{t=1,s\neq t,t\neq k}^{|V|} \frac{\sigma_{s,t}(k)}{\sigma_{s,t}} \quad (3)$$

This quantity can be calculated in a recursive fashion in equation (4). This is done by fixing each *s* and applying equation (4) to calculate $\delta_{s,*}(k)$ according to the non-increasing order of the shortest path distance from *s* to *k*.

$$\delta_{s,*}(k) = \begin{cases} 0 & if\ s = w \\ \sum_{t:\, k \in \pi_{s,t}} \frac{\sigma_{s,k}}{\sigma_{s,t}}\left(1 + \delta_{s,*}(t)\right) & otherwise \end{cases} \quad (4)$$

Here $\pi_{s,t}$ is the set of all in neighbors of *s* that are on the shortest paths from *s* to *t*. After $\delta_{s,*}(k)$ is calculated, the betweenness centrality can be acquired by summing all the $\delta_{s,*}(k)$ together according to equation (5). This can be easily validated by taking equation (3) into equation (5) and compare the results with equation (2). Brandes argued that the betweenness can be calculated in $O(VE + V^2logV)$.

$$C_B(k) = \sum_{s=1,s\neq k}^{|V|} \delta_{s,*}(k) \quad (5)$$

Some efforts have been made to further improve the computational time of both metrics. Some use approximations[15, 16], which will not always generate exactly the same metric values as the definitions. Others use distributed[17, 18] and parallel techniques [19, 20].

2.3 Algorithms for Streaming Network Data

The algorithm illustrated in Section 2.2 is designed to handle static network data and will be too slow to be applied on streaming network data. In order to better understand algorithms for streaming data, we need to first clarify two important concepts: An *aggregated network* at time t, denoted as AN_t consists of all the edges in the network at time t. If an edge $e(s,t)$ exists in the network AN_t (i.e. $(s,t) \in AN_t.E$), $AN_t.\omega_{s,t}$ represents its corresponding edge weight. On the other hand, a *delta network* at time *t*, denoted as DN_t consists of only the changes made to the network between time t-1 and time t. Link changes can come in one of the following forms: *link addition*, *link deletion* or *link weight modification*. For simplicity, we will only consider link addition and link deletion in this paper since a change of link weight can be considered as a link deletion followed by a link addition. An edge $e(s,t) \in DN_t.E$ will either has a finite link weight $DN_t.\omega_{s,t} < \infty$ if this is a link addition on time step *t* or $DN_t.\omega_{s,t} = \infty$ if this is a link deletion on this time step. At time 1, AN_1=DN_1. At time *t*>1, AN_t is an aggregation result of all the delta network from the time 1 to time *t* $\{DN_1, DN_2, \ldots, DN_t\}$.

An algorithm that is capable of handling streaming data should take a delta network instead of aggregated network as input. The calculations should thus take advantage of the delta network so that unnecessary calculations can be avoided or minimalized. Some recent work tried to address this issue, but efforts were limited to implementing either betweenness *or* closeness. On the side of closeness, [21] has a good solution in unweighted networks. [22] use similar approaches to further speed up the algorithm in a parallel fashion. On the side of betweenness, [23-25] looks for solutions on binary network data. [26] managed to produce an algorithm on weighted network data. However, the techniques used by these authors cannot extend to closeness. In contrasts, the efforts in the present work extend to both algorithms and additionally are able to handle weighted data.

3. Methodology

We divide the calculation of closeness and betweenness into two steps: 1) the *convergence step*, where shortest path information is updated, and 2) the *aggregation step*, where metric values are updated based on the results from convergence step. Convergence step and aggregation step will needed to be executed each time new data is received. To help readers to better understand the paper, we prepared Table 1 that contains a summary of all the major notations used in the paper.

Table 1 Summary of major notations. clo=closeness. bet=betweenness

$d_{s,t}$	Shortest path length from *s* to *t*. (clo & bet)
$\Delta d_{s,t}$	Change of shortest path from *s* to *t*. (clo)
$\psi_{s,t}$	One of *s*' out-neighbors on shortest path from *s* to *t*. (clo)
$\sigma_{s,t}$	Num. shortest path from *s* to *t*. (bet)
$\Delta\sigma_{s,t}$	Change in Num. shortest path from *s* to *t*. (bet)
$\phi_{s,t}$	*s*' out-neighbors on shortest path from *s* to *t*. (bet)

$\pi_{s,t}$	*t*' in-neighbors on shortest path from *s* to *t*. (bet)
$\delta_s(t)$	See equation (4). (bet)
$\varepsilon_s(t)$	See equation (7). (bet)

3.1 Closeness

3.1.1 Initialization for Closeness

Algorithm 1.1 is the initialization for closeness. In this algorithm, a complex global G is created, initialized and returned. This variable G contains information that will not be released as long as there are new streaming data coming in. The first member of G, $d_{s,t}$ is the length of shortest path from *s* to *t*. It is initialized to be 0 when *s*=*t* based on the fact that there is always a shortest path of length 0 from a node to itself. When *s*≠*t*, $d_{s,t}$ is initialized to be ∞ to indicate that no shortest path is available at this moment. The variable $\psi_{s,t}$ is defined to be one of the out-neighbors of *s* that lies on the shortest path from *s* to *t*. $\psi_{s,t}$ is initialized to be *t* if *s*=*t* and *null* otherwise. $C_C(s)$ is the actual closeness centrality value and is initialized to be 0. AN is an aggregated network that is used to keep track of the complete network.

Algorithm 1.1 Initialize_Clo()

```
1:  for $s \in V$ do
2:     for $t \in V$ do
3:        if $s = t$ then
4:           $G.d_{s,t} \leftarrow 0$; $G.\psi_{s,t} \leftarrow t$;
5:        else
6:           $G.d_{s,t} \leftarrow \infty$; $G.\psi_{s,t} \leftarrow null$;
7:        end if
8:     end for
9:  $G.C_C(s) \leftarrow 0$;
10: end for
11: $G.AN \leftarrow \phi$;
12: return $G$;
```

3.1.2 Convergence Step for Closeness

The goal of converge step is to update shortest path information incrementally and efficiently. The main idea behind the convergence step is to propagate changes of convergence variables based on routes that are currently *Active*.

Definition 1 Active Route

A route $\alpha = (s,t)$ is active if the shortest path from s to t, $d_{s,t}$ is changed during the convergence step.

Active routes are results of route updates. There are two types of updates: Direct Link Update (DLU), which is triggered by a direct link change, and *Remote Link Update (RLU)*, which is triggered by a remote link change. Fig. 1 illustrates these two cases. On the left, a new link is added from node *s* to node *u*. A DLU will be triggered to update the shortest path from *s* to *t* since *u* might provide a route to *t* with better path weight. On the right, a shortest path change from *s* to *t* just happened. It will trigger a RLU to update route from *s'* to *t*.. This differs from the previous case in that this update is not triggered by a direct link change.

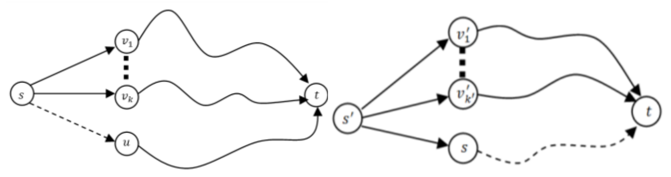


Fig. 1. Illustration of DLU (left) and RLU (right). Solid links represent existing path while dotted links represent new path just been found. Straight links represent actual link in the network while curved ones represent paths that consist of multiple links.

Once a route $\alpha = (s,t)$ is active, it will propagate route updates to the in-neighbors of s, namely *s'*. The routes from every such in-neighbor to t will be updated and potentially become new active routes, in turn propagating updates to their in-neighbors. A route will become inactive after it has propagated its updates to all the in-neighbors of s. It might become active again if its path length gets changed again. If there are no active routes being generated, the system will reach a status of convergence.

Algorithm 2 displays the algorithm for convergence. There are four input of the algorithm: G, which is the Global Variable returned by the Algorithm 1.1, DN, which is the delta network containing streaming network changes, and two function objects Init and Update defined in Algorithm 1.2 and Algorithm 1.3 respectively. Algorithm 2 returns a variable C as output, which stands for Convergence Variable. C is initialized by algorithm 1.2.

The first thing Algorithm 2 does is to apply all the changes from DN into AN. Links that are marked as deletions (i.e. those have ∞ link weight)in DN will delete the corresponding links in AN and those links marked as inserts in DN will be added into AN. The algorithm then iterates all the link changes in *DN* and tries initializing DLU updates. If the update actually changes the shortest path, this route will become active in the next convergence step and will be recorded in $FutureActive$. In the next convergence step, similar route updating procedures for RLU will happen and active routes will be added to the convergence step after that in $FutureActive$. This process will repeat until no more active route is found which indicates the algorithm reaches the convergence.

Algorithm 2 Convergence ($G, DN, Init, Update$)

```
1:  $C \leftarrow$ Init ();
2:  $FutureActive \leftarrow \emptyset$;
3:  $G.AN.Apply\_Changes(DN)$;
4:  for $e(s,t) \in DN_t$ do
5:     for $u \in G.AN.V$ do
6:        if Update($s,t,v,DLU,G,C,DN$) = CHG then
7:           $FutureActive \leftarrow FutureActive \cup \{(s,t)\}$;
8:        end if
9:     end for
10: end for
11: While $FutureActive \neq \emptyset$ do
12:    $CurrentActive \leftarrow FutureActive$;
13:    $FutureActive \leftarrow \emptyset$;
```

```
14: for $(s,t) \in CurrentActive$ do
15:   for $s' \in AN.InNeghbours(s)$ do
16:     if Update($s', t, s, RLU, G, C, DN$) = CHG then
17:       $FutureActive \leftarrow FutureActive \cup \{(s',t)\}$;
18:     end if
19:   end for
20: end for
21: end do
22: return $C$;
```

Algorithm 1.2 is the initialization of the convergence step for closeness. The only variable that needs to be initialized here is Δd, which is a map to record changes of shortest path weight $d_{s,t}$. Δd is used in Algorithm 1.3 and the aggregation step.

Algorithm 1.2 Init_Clo ()

```
1 : $C.\Delta d \leftarrow empty\ map$;
2:  return $C$;
```

Algorithm 1.3 is the update function for closeness. The algorithm takes several inputs: the key points of the route which consists of the identities of three nodes (i.e. *s*,*t*,*v*), the type of update (DLU or RLU), the global variable (*G*), the convergence variable (*C*) and the delta network (*DN*). The algorithm will update shortest path information from node *s* to node *t* via node *v*, which must be one of the out-neighbors of *s*. The key thing here is to compare d', which is the new path length from *s* to *v* via *t*, and the old value $d_{s,t}$. If this is a DLU update $d' = AN.\omega_{s,v} + DN.\omega_{v,t}$. If it is a link deletion, $DN.\omega_{v,t}$ will be ∞ and d' will be ∞ as well. If otherwise it is a RLU update, $d_{s,t}$=$AN.\omega_{s,v} + d_{v,t}$.

There are three cases for the update algorithm. The first case is when the length of the new path *d'* turns out to be better than that of the existing path $d_{s,t}$. In that case, a change occurs and we need to update $\phi_{s,t}$ and $\Delta d_{s,t}$. CHG will be returned to signal a change has occurred. In the second case where the new path (i.e. *d'*) is not better (i.e. d'>$d_{s,t}$) but $\phi_{s,t} = v$, the *previous* considered optimal shortest path no longer valid and needs to be deleted. This usually happens when there is a link deletion, which causes the distance to increase. In this case, we need to search for an alternative route. To begin search, we first set $d_{s,t}$ to be ∞ and $\psi_{s,t}$ to be null and then iterate all the out-neighbors of *s* and update the route. Once the search is finished, we will return CHG. In the third case, the new route is not better and it is not the route we have found before, no change will be made and UNCHG will be returned.

Algorithm 1.3 Update_Clo ($s, t, v, type, G, C, DN$)

```
1:  if $type = DLU$ then
2:      $d' = G.AN.\omega_{s,v} + DN.\omega_{v,t}$;
3:  else
4:      $d' = G.AN.\omega_{s,v} + G.d_{v,t}$;
5:  end if
6:  if $d' < G.d_{s,t}$ then
7:      $G.\psi_{s,t} \leftarrow v$;
8:      $C.\Delta d_{s,t} \leftarrow C.\Delta d_{s,t} + d' - G.d_{s,t}$;
9:      $G.d_{s,t} \leftarrow d'$;
10:     Return CHG;
11: else if $v = G.\psi_{s,t}$ and $d' > G.d_{s,t}$ then
12:     $G.d_{s,t} \leftarrow \infty$;
13:     $G.\psi_{s,t} = null$;
14:     for $v' \in G.AN.OutNeighbors(s)$ do
15:         Update_Clo ($s, t, v', RLU, G, C, DN$);
16:     end for
17:     Return CHG;
18: end if
19: Return UNCHG;
```

3.1.3 Aggregation Step for Closeness

The goal of the aggregation step is to update closeness based on the results from convergence step. Recall equation (1) that the closeness of node *s* can be calculated by first summing all the $d_{s,t}$ over *t* then make an inverse of the sum. Also recall that Δd keeps track of all the changes being made to $d_{s,t}$ during the convergence step. The aggregation algorithm of closeness would thus be to update the *sum* based on the old value *sum'* and the Δd using equation (6).

$$\begin{aligned} sum_s &= \sum_t d_{s,t} \\ &= \sum_t (d'_{s,t} + \Delta d_{s,t}) \\ &= sum'_s + \textstyle\sum_t \Delta d_{s,t} \end{aligned} \qquad (6)$$

Algorithm 1.4 displays the aggregation step for closeness. It works by first iterating all the changes in Δd and gradually add the changes to *sum*. Closeness will be returned by inversing the sum.

Algorithm 1.4 Aggregation_Clo(*C*)

```
1: for $(s,t) \in C.\Delta d.keys$ do
2:   if $C_c(s) = \infty$ then
3:       sum= $C.\Delta d_{s,t}$;
4:   else
5:       sum=$\frac{1}{C_c(s)} + C.\Delta d_{s,t}$;
6:   end if
7:   $C_c(s) \leftarrow \frac{1}{sum}$;
8: end for
9: return $C_c$;
```

3.1.4 Procedures to Calculate Closeness

To put everything together, we have Algorithm 1.5 to illustrate the general workflow of the closeness calculation. The initialization only needs to be executed once to initialize *global variable G*. In the following step, convergence and aggregation needs to be executed in pair to respond to network change.

Algorithm 1.5 Closeness(*DN*)

```
1:  $G \leftarrow$ Initialize_Clo();
2:  $t \leftarrow 0$;
```

```
3: While $t < DN.length$ do
4:     $C \leftarrow$ Convergence($G, DN_t$, Init_Clo, Update_Clo );
5:     $C_c \leftarrow$ Aggregation_Clo($C$);
6:     Output $C_c$;
7: end do
```

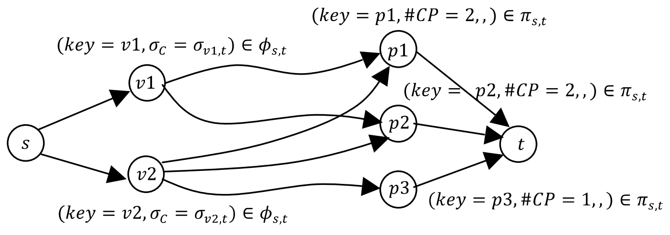
3.2 Betweenness

3.2.1 Initialization for Betweenness

To understand the initialization step for betweenness, we need to first understand two important quantities $\phi_{s,t}$ and $\pi_{s,t}$. Fig. 2 illustrates these two quantities. In this scenario, a node *s* has multiple shortest paths to node *t*. The value $\phi_{s,t}$ is defined to be a map with each key *v* to be the out-neighbors of *s* that are on the shortest paths from *s* to *t* and values to be the count of such shortest paths, which is $\sigma_{s,t}(v)$. Since *v* is the out-neighbor of *s*, there will be exactly one path from *s* to *v*. Thus we have $\sigma_{s,t}(v) = \sigma_{v,t}$. In Fig. 2, v_1, v_2 are two such out-neighbors. The number of shortest paths that go through each of them are $\sigma_{v1,t}$ and $\sigma_{v2,t}$. They will be recorded along with the keys in $\phi_{s,t}$. The variable $\pi_{s,t}$ is a map that contains the in-neighbors of *t* that are on the shortest paths. In the example of Fig. 2, there are three such in-neighbors of *t*. Each key has two values: #CP and ΔCP, which stands for the number of composite paths and the changes in the number of composite paths. Definition 2 defines composite path.

Definition 2 Composite Path (CP)

A composite path on the shortest path from *s* to *t* and ends with node p, $CP_p^{s,t} \in \pi_{s,t}$ is defined to be a collection of paths on the shortest paths that start from one of *s'* out-neighbors and end in one of the in-neighbors of *t* if $s \neq t$. If *s*=*t*, it is defined to be a single path from *s* to itself. A composite path has to overlap with one of the shortest paths from *s* to *t*.



$(key = p1, \#CP = 2,,) \in \pi_{s,t}$
$(key = v1, \sigma_C = \sigma_{v1,t}) \in \phi_{s,t}$
$p1$
$v1$
$(key = p2, \#CP = 2,,) \in \pi_{s,t}$
$p2$
s
t
$v2$
$p3$
$(key = p3, \#CP = 1,,) \in \pi_{s,t}$
$(key = v2, \sigma_C = \sigma_{v2,t}) \in \phi_{s,t}$

Fig. 2. Shortest paths from s to t are being illustrated to demonstrate $\phi_{s,t}$ and $\pi_{s,t}$

In Fig. 2, there are two different composite paths ending with p_1, which is v1->p1 and v2->p1. The number of composite paths, #CP will be 2 for p_1. Similarly, there are #CP=2 for p_2 and #CP=1 for p_3.

Algorithm 3.1 is the initialization process for betweenness. Similar to that of closeness, $d_{s,t}$, AN and the centrality value C_B needs to be initialized. Unlike closeness, betweenness needs to create four other variables. The variable $\sigma_{s,t}$ is the number of shortest paths from *s* to *t*. It is initialized to 1 when *s*=*t* and 0 otherwise. The variables $\phi_{s,t}$ and $\pi_{s,t}$ are the two maps previously discussed . Another variable that we need to initialize is $\varepsilon_s(t)$, which is defined in equation (7). We will defer the explanation of this quantity to later sections.

$$\varepsilon_s(t) = \frac{1 + \delta_{s,*}(t)}{\sigma_{s,t}} \tag{7}$$

Algorithm 3.1 Initialize_Bet()

```
1: for $s \in V$ do
2:   for $t \in V$ do
3:     if $s = t$ then
4:       $G.d_{s,t} \leftarrow 0$; $G.\sigma_{s,t} \leftarrow 1$;
5:       $G.\phi_{s,t} \leftarrow \{(key = t, \tilde{\sigma} = 0)\}$;
6:       $G.\pi_{s,t} \leftarrow \{(key = s, \#CP = 1, \Delta CP = 0)\}$;
7:     else
8:       $G.d_{s,t} \leftarrow \infty$; $G.\sigma_{s,t} \leftarrow 0$;
9:       $G.\phi_{s,t} \leftarrow empty$; $G.\pi_{s,t} \leftarrow empty$;
10:    end if
11:    $G.\delta_s(t) \leftarrow 0$;
12:    $G.\varepsilon_s(t) \leftarrow 0$;
13:  end for
14: $G.C_B(s) \leftarrow 0$;
15: end for
16: $G.AN \leftarrow \phi$;
17: return $G$;
```

3.2.2 Convergence Algorithm for Betweenness

The convergence algorithm for betweenness utilizes the same convergence function as the one for closeness defined in algorithm 2. To make the convergence function work, we need to supply algorithm 3.2 and algorithm 3.3 to serve as function objects to pass into algorithm 2.

Algorithm 3.2 Init_Bet()

```
1: ∀s $C.Changed_s^+ \leftarrow \phi$, $C.Changed_s^- \leftarrow \phi$;
2: ∀s, t $C.\Delta\sigma_{s,t} \leftarrow 0$;
3: return C;
```

Algorithm 3.2 is the initialization function for betweenness. Two maps named $Changed_s^+$ and $Changed_s^-$ are being initialized. The keys of the maps are nodes and values are distances from *s* to that particular node represented by the key. These maps are used to keep track of positive changes (when a shortest path from *s* to a specific node is being added or changed) and negative changes (when a shortest path from *s* is being deleted or changed) made during the convergence step. $\Delta\sigma_{s,t}$ is used to keep track of the change of $\sigma_{s,t}$ throughout the convergence step.

Algorithm 3.3 Update_Bet($s, t, v, type, G, C, DN$)

```
1: if $type = DLU$ then
2:     $d' = G.AN.\omega_{s,v} + DN.\omega_{t,v}$;
3: else
4:     $d' = G.AN.\omega_{s,v} + G.d_{v,t}$;
5: end if
6: if $d' < G.d_{s,t}$ then
7:   $G.d_{s,t} \leftarrow d'$;
8:   $G.\phi_{s,t} \leftarrow \{(key = v, \sigma = G.\sigma_{v,t})\}$;
9:   $C.\Delta\sigma_{s,t} \leftarrow C.\Delta\sigma_{s,t} + G.\sigma_{v,t} - G.\sigma_{s,t}$;
10:  $G.\sigma_{s,t} \leftarrow G.\sigma_{v,t}$;
11:  UpdatePath (s,t,v,G,C,CLEAR);
```

```
12:  UpdatePath (s,t,v,G,C,INSERT);
14:  Return CHG;
15: else if $v \in \phi_{s,t}.keys$ and $d' > G.d_{s,t}$ then
16:  $G.\sigma_{s,t} \leftarrow G.\sigma_{s,t} - G.\phi_{s,t}[v].\sigma$;
17:  $C.\Delta\sigma_{s,t} \leftarrow C.\Delta\sigma_{s,t} - G.\phi_{s,t}[v].\sigma$;
18:  $G.\phi_{s,t} \leftarrow G.\phi_{s,t}\backslash\{v\}$;
19:  UpdatePath (s,t,v,G,C,DELETE);
20:    if $G.\phi_{s,t}.empty = True$ then
21:      $G.d_{s,t} \leftarrow \infty$;
22:      for $v' \in G.AN.OutNeighbors(s)$ do
23:        Update($s,t,v',RLU,G,C,DN$);
24:      end for
25:      Return CHG;
26:    end if
27: else if $v \notin G.\phi_{s,t}.keys$ and $d' = G.d_{s,t}$ then
28:  $C.\Delta\sigma_{s,t} \leftarrow C.\Delta\sigma_{s,t} + G.\sigma_{v,t} - G.\sigma_{s,t}$;
29:  $G.\sigma_{s,t} \leftarrow G.\sigma_{s,t} + G.\sigma_{v,t}$;
30:  $G.\phi_{s,t} \leftarrow G.\phi_{s,t} \cup \{(key = v, \sigma = G.\sigma_{v,t})\}$;
31:  UpdatePath (s,t,v,G,C,INSERT);
32:  return CHG;
33: end if
34: Return UNCHG;
```

Algorithm 3.3 is organized in a similar fashion to the closeness algorithm. First *d'* is calculated based on whether it is a DLU or RLU update. Then we will consider four different cases instead of three in the closeness algorithm.

The first case is when $d' < G.d_{s,t}$. In this case, we need to update $d_{s,t}$, $\phi_{s,t}$, $\sigma_{s,t}$ and $\Delta\sigma_{s,t}$. It is obvious that when a new path with shorter length is present, all the old information needs to be discarded. Thus there will be only one element in $\phi_{s,t}$ and $\sigma_{s,t}$ will be set to $\sigma_{v,t}$. The value $\Delta\sigma_{s,t}$ is updated by adding it to the change of $\sigma_{s,t}$, which is $\sigma_{v,t} - \sigma_{s,t}$ in this case. The value $\pi_{s,t}$ is updated by calling the function *UpdatePath***,** which is defined in Algorithm 3.4. Calling *UpdatePath* with parameter CLEAR will mark all the paths in $\pi_{s,t}$ to be invalid (i.e. making $\Delta CP = -\#CP$). Keys found in $\pi_{s,t}$ will record a negative change to *s* in $Changed_s^-$. A second function call to *UpdatePath* with parameter INSERT will insert all the composite paths p' found in $\pi_{v,t}$into $\pi_{s,t}$. In algorithm 3.4, cases for INSERT are divided into two branches: on the first branch, if p' is already in $\pi_{s,t}$, then we update ΔCP field of key *p'*. Both $Changed_s^+[p']$ and $Changed_s^-[p']$ will be recorded in this case because this is considered to be a change to the composite path. If the second branch is taken, we will create a new entry for p' with $\#CP = 0$ and ΔCP to be the increments. This case will be considered to be an addition and only $Changed_s^+[p']$ will be recorded. CHG will be returned.

In the second case, $d' > G.d_{s,t}$ but *v* is recorded in $\phi_{s,t}$ (i.e $v \in \phi_{s,t}.keys$). This means the shortest path information that was previously recorded in the system is no longer valid. This happens usually when a link on the shortest path is being deleted. In that case, we need to take *v* out from all variables. We first update $\sigma_{s,t}$, $\Delta\sigma_{s,t}$ and $\phi_{s,t}$. Then we update $\pi_{s,t}$ by calling *UpdatePath* using a DELETE parameter. Here, for all p' found in $\pi_{v,t}$ we subtract the values in $\pi_{s,t}$. This is still considered to be a change to composite path. There might still be some other composite paths after the deletion and hence both $Changed_s^+[p']$ and $Changed_s^-[p']$ also needs to be updated. After calling the *UpdatePath*, we will see whether or not it is necessary to perform a search for an alternative shortest path if we found $\phi_{s,t}$ to be empty. If so, we will search for an alternative shortest path by calling *Update*. In either case, CHG will be returned.

In the third case, if we found a node *v* with equal shortest path length as the one recorded but itself not recorded(i.e. $v \notin \phi_{s,t}$), we will update $\sigma_{s,t}$, $\Delta\sigma_{s,t}$, $\phi_{s,t}$ and call *UpdatePath* to update $\pi_{s,t}$. CHG will be returned.

In the last case, there will be no change at all and UNCHG will be returned.

Algorithm 3.4 UpdatePath(*s,t,v,G,C,Mode***)**

```
1:  if $Mode = INSERT$ then
2:    for $p' \in G.\pi_{v,t}.keys$ do
3:      if $p' \in G.\pi_{s,t}.keys$ then
4:        $G.\pi_{s,t}[p'].\Delta CP \leftarrow G.\pi_{s,t}[p'].\Delta CP +$
              $G.\pi_{v,t}[p'].\#CP + G.\pi_{v,t}[p'].\Delta CP$;
5:        $C.Changed_s^+[p'] \leftarrow G.d_{s,t}$;
6:        $C.Changed_s^-[p'] \leftarrow G.d_{s,t}$;
7:      else
8:        $G.\pi_{s,t} \leftarrow G.\pi_{s,t} \cup \{(key = p', \#CP = 0;$
              $\Delta CP = G.\pi_{v,t}[p'].\#CP + G.\pi_{v,t}[p'].\Delta CP, \delta_C = 0)\}$;
9:        $C.Changed_s^+[p'] \leftarrow G.d_{s,t}$;
10:     end if
11:   end for
12: else if $Mode$ =CLEAR then
13:   for $p \in G.\pi_{s,t}.keys$ do
14:     $G.\pi_{s,t}[p].\Delta CP \leftarrow -1 \cdot G.\pi_{s,t}[p].\#CP$;
15:     $C.Changed_s^-[p] \leftarrow G.d_{s,t}$;
16:   end for
17: else if $Mode = DELETE$ then
18:   for $p' \in G.\pi_{v,t}.keys$ do
19:     $G.\pi_{s,t}[p'].\Delta CP \leftarrow G.\pi_{s,t}[p'].\Delta CP - G.\pi_{v,t}[p'].\#CP -$
            $G.\pi_{v,t}[p'].\Delta CP$;
20:     $C.Changed_s^+[p'] \leftarrow G.d_{s,t}$;
21:     $C.Changed_s^-[p'] \leftarrow G.d_{s,t}$;
22:   end for
23: end if
```

3.2.3 Aggregation Step for Betweenness

The aggregation step for betweenness relies on the idea of Brandes' approach. Remember betweenness can be calculated by aggregating $\delta_{s,*}(p)$ according to equation (5). $\delta_{s,*}(p)$ can be calculated by iterating all p in the nonincreasing order of their distance to *s* using equation (4). We follow the same idea to build the aggregation function for betweenness.

Algorithm 3.5 illustrates the procedure in detail. The aggregation step will process two kinds of updates: negative updates and positive updates. Negative updates consist of link deletions and link changes, which are stored in $Changed_s^-$ while positive updates consists of link additions and link changes, which are stored in $Changed_s^+$. Negative updates will always come first. The *Mode* variable will first be NEGATIVE and then POSITIVE to reflect this procedure. For each node s in the network, we then build a queue by putting all the elements from either $Changed_s^-$ or $Changed_s^+$ depending on the value of *Mode* parameter.

To understand how we are going to update the centrality, consider Equation (4) and its rewritten format in equation (8). The variable $\delta_{s,*}(p)$ can be written as the product of $\sigma_{s,p}$ and the sum of $\varepsilon_s(t)$ over *t*. The aggregation algorithm will 1) Check if $\sigma_{s,p}$ has changed and if so then update $\delta_{s,*}(p)$ and $C_B(p)$ accordingly (implemented by function *ScaleQuantity* in algorithm 3.7) and 2) check whether any $\varepsilon_s(t)$ has changed. If so, it will then either call $DecreaseQuantity$ (algorithm 3.8) or *IncreaseQuantity* (algorithm 3.9) to update both $\delta_{s,*}(p)$ and $C_B(p)$.

$$\begin{aligned}\delta_{s,*}(p) &= \sum_{t:\, p\, \in\, \pi_{s,t}} \frac{\sigma_{s,p}}{\sigma_{s,t}} \cdot (1+\delta_{s,*}(t)) \\ &= \sigma_{s,p} \sum_{t:\, p\, \in\, \pi_{s,t}} \varepsilon_s(t)\end{aligned} \tag{8}$$

The algorithm will then work by retrieving the elements from the queue by the descending order of the values in $Changed_s^-$ or $Changed_s^+$, which is the distance $d_{s,t}$. We first save the old $\varepsilon_s(t)$ to be ε_{old}. We will need to deduct this old value from betweenness and add the new one to it. We update $\varepsilon_s(t)$ to be the newest value according to Equation (8). We then iterate all the composite paths and update quantities. If $\Delta\sigma_{s,p} \neq 0$, that means there is some changes for $\sigma_{s,p}$ during the convergence step. We will first call *ScaleQuantity* to update $\sigma_{s,p}$. Since $\Delta\sigma_{s,p}$ is the change of $\sigma_{s,p}$. $\sigma'_{s,p} = G.\sigma_{s,p} - \mathrm{C}.\Delta\sigma_{s,p}$ is the old value of $\sigma_{s,p}$ before the convergence step and the new value of $\delta_s(p)$ will be $\delta_s(p) \cdot \frac{\sigma_{s,p}}{\sigma'_{s,p}} = \delta_s(p) \frac{\sigma_{s,p}}{G.\sigma_{s,p} - \mathrm{C}.\Delta\sigma_{s,p}}$. We will also update $C_B(s)$ by minus the old value of $\delta_s(p)$ and add the new value of $\delta_s(p)$, which is $(\frac{G.\sigma_{s,p}}{G.\sigma_{s,p} - \mathrm{C}.\Delta\sigma_{s,p}} - 1) \cdot G.\delta_s(p)$.

The next step is to determine whether or not we need to update $\varepsilon_s(t)$. Before doing that, we will generate a Status based on $\pi_{s,t}[\mathrm{p}].\#CP$ and $\pi_{s,t}[\mathrm{p}].\Delta CP$ using function defined in algorithm 3.6. When #CP=0 while $\Delta CP > 0$, a whole new path from s to p needs to be added and the status will be INSERT. When #CP=0 but $\Delta CP = 0$, the path from s to p is first added and later deleted. The corresponding status will be DUMMY. When #CP>0 and #CP+$\Delta CP = 0$, a previously established path from s to p is being completely deleted. The algorithm will return a status DELETE. In the final case where #CP>0 and # CP+$\Delta CP > 0$, a change occurried to the path from s to p and will return CHANGE. The update function will then update the centrality based on the status. If it is CHANGE then we will first decrease the quantity by dropping the corresponding $\varepsilon_s(t)$ when *Mode* is NEGATIVE and increase the quantity by adding the path when *Mode* is positive. The decrease function is defined in Algorithm 3.8 which essentially decreases the $\varepsilon_{old} \cdot G.\sigma_{s,t}$ for both $\delta_s(p)$ and $C_B(p)$. Note that $\sigma_{s,t}$ here is the up to date value of $\sigma_{s,t}$ not the old value $\sigma'_{s,t}$. The reason to use the up to date value is that we have updated $\sigma_{s,t}$ in both $\delta_s(p)$ and $C_B(p)$ when calling *ScaleQuantity* before. *IncreaseQuantity* is implemented in algorithm 3.9 by adding the most current value of $\varepsilon_s(t) \cdot G.\sigma_{s,p}$ back to $\delta_s(p)$ and $C_B(p)$. When the status is DELETE or DUMMY, we will only call $DecreaseQuantity$. While the status is INSERT we will only call ***IncreaseQuantity*.** We will also always add ΔCP to $\#CP$ and clear ΔCP after each operation.

Algorithm 3.5 Aggregate_Betweenness(G, C)

```
1: for Mode ∈ {NEGATIVE, POSITIVE} do
2:   for $s \in G.AN.V$ do
3:     $queue \leftarrow empty\ priority\ queue$;
4:     if Mode=NEGATIVE then
5:       $Changed \leftarrow C.Changed_s^-$;
6:     else
7:       $Changed \leftarrow C.Changed_s^+$;
8:     end if
9:   for $t \in Changed$.keys do
10:    $queue.enqueue(t, Changed[t])$;
11:  end for
12:  while $queue.empty = False$ do
13:    $t \leftarrow queue.dequeue()$;
14:      $\varepsilon_{old} \leftarrow \varepsilon_s(t)$;
15:      $G.\varepsilon_s(t) \leftarrow \frac{1}{G.\sigma_{s,t}} \cdot (1 + G.\delta_s(t))$;
16:    for $p \in G.\pi_{s,t}.keys$ do
17:      if $\mathrm{C}.\Delta\sigma_{s,p} \neq 0$ then
18:        ScaleQuantity $(s, t, p, G, C)$;
19:      end if
20:      St= GetST($G.\pi_{s,t}$[p].$\#CP$, $G.\pi_{s,t}$[p].$\Delta CP$);
21:      if St= CHANGE then
22:        if Mode=NEGATIVE then
23:          DecreaseQuantity (s,t,p, $\varepsilon_{old}$,G);
24:        else
25:          IncreaseQuantity $(s, t, p, G)$;
26:          $G.\pi_{s,t}$[p].$\#CP \leftarrow G.\pi_{s,t}$[p].$\#CP + G.\pi_{s,t}$[p].$\Delta CP$;
27:          $G.\pi_{s,t}$[p].$\Delta CP \leftarrow 0$;
28:        end if
29:      else if St =DELETE or St = DUMMY then
30:        if Mode=NEGATIVE then
31:          DecreaseQuantity (s,t,p,G);
32:          $G.\pi_{s,t} \leftarrow G.\pi_{s,t} \backslash \{p\}$;
33:        end if
34:      end if
35:      $queue.enqueue(p, d_{s,p})$;
36:    end for
37: end do
38: return $G.C_B(s)$;
```

Algorithm 3.6 GetST $(\#CP, \Delta CP)$

```
1: if #CP=0 then
2:   if $\Delta CP > 0$ then
3:     return INSERT;
4:   else
5:     return DUMMY;
```

```
6:      end if
7:  else   //#CP>0
8:      if #CP+$\Delta CP = 0$ then
9:          return DELETE;
10:     else   //# CP+$\Delta CP > 0$
11:         return CHANGE;
12:     end if
13: end if
```

Algorithm 3.7 ScaleQuantity (s, t, p, G, C)

```
1:  $G.C_B(s) \leftarrow G.C_B(s) + (\frac{G.\sigma_{s,p}}{G.\sigma_{s,p} - C.\Delta\sigma_{s,p}} - 1) \cdot G.\delta_s(p)$;
2:  $G.\delta_s(p) \leftarrow G.\delta_s(p) \cdot \frac{G.\sigma_{s,p}}{G.\sigma_{s,p} - C.\Delta\sigma_{s,p}}$;
3:  $C.\Delta\sigma_{s,p} \leftarrow 0$;
```

Algorithm 3.8 DecreaseQuantity $(s, t, p, \varepsilon_{old}\ G)$

```
1: $G.\delta_s(p) \leftarrow G.\delta_s(p) - \varepsilon_{old} \cdot G.\sigma_{s,t}$;
2: $G.C_B(p) \leftarrow G.C_B(p) - \varepsilon_{old} \cdot G.\sigma_{s,t}$;
```

Algorithm 3.9 IncreaseQuantity (s, t, p, G)

```
1: $G.\delta_s(p) \leftarrow G.\delta_s(p) + \varepsilon_s(t) \cdot G.\sigma_{s,p}$;
2: $G.C_B(p) \leftarrow G.C_B(p) + \varepsilon_s(t) \cdot G.\sigma_{s,p}$;
```

3.2.4 Procedures to Calculate Betweenness

Algorithm 3.10 to illustrated the procedures to calculate betweenness.

Algorithm 3.10 Betweenness (DN)

```
1: $G$ ← Initialize_Bet();
2: $t \leftarrow 0$;
3: While $t < DN.length$ do
4:     $C$ ← Convergence($G, DN$, Init_Bet, Update_Bet);
5:     $C_B$ ← Aggregation_Bet($C$);
6:     Output $C_B$;
7: end do
```

3.3 Calculating Betweenness and Closeness Together

Since the convergence variable of closeness is a subset of that of betweenness, maintaining a single copy of the convergence variable of betweennes will avoid unnecessary calculations in the situation when closeness and betweenness are both being calculated. During the aggregation stage however, closeness and betweenness need to call their own aggregation functions. Since the majority of the calculations concentrate on the convergence step, calculating those two metrics together will significantly reduce the running time as compared to calculate them separately. Due to space limitations, we will omit the algorithm.

4. Experimental Results

4.1 Implementations and Experimental Settings

We implemented both algorithms and compared them against baseline methods. The baseline algorithms are *Dijkstra* and *Johnson* for closeness and *Brandes* for betweenness. Experiments are run on a 64-bit machine with 4 Intel Xeron 7550 CPUs. The compiler is MSVC 2012 64 bit.

4.2 Data Sets

We tested our algorithms on two data sets. Table 2 shows the summary of the two data sets.

WikiVote Data Set

The wiki vote data set consists of the vote logs among Wikipedia administrators when choosing the new administrators. When a user casts a vote for another user, a link will be recorded with a timestamp. The link weight is the total number of critiques of the user who made the vote. The data is processed into a delta network stream by day. Each day is a time step and there are 1267 time steps.

Foursquare Travel Sequence Data Set

The second data set we use in the experiment is collected from public location API of foursquare in New York City and Pittsburgh area. The source node of the link represent the first location a specific user has visited and the target of the link is the next one with link weight being the difference of those two visits in milliseconds. The network data is again processed into a delta network by day. Each day is a time step and there are 578 time steps.

Table 2 Summary of the data sets

	Num.Nodes	Num.Links	Time Steps
WikiVote	26,773	26,773	1267
Foursquare	84,722	67,404	578

4.3 Results

Experimental results show that our algorithms significantly reduced running time on both data sets. Fig. 3 shows the speed up of closeness compared to the baseline methods (in speedups). Using our approach, the calculations are sped up by roughly 5 to 7 orders of magnitudes. In the foursquare data, the speed up tends to decrease over time. This is because there are more link changes at later time steps than earlier time steps. Fig. 4 shows the running time comparisons of betweenness. The speed up is not as significant as the closeness because of the increased complexity brought by additional calculations. However, the speed up is still 2 to 4 orders of magnitudes on both data sets.

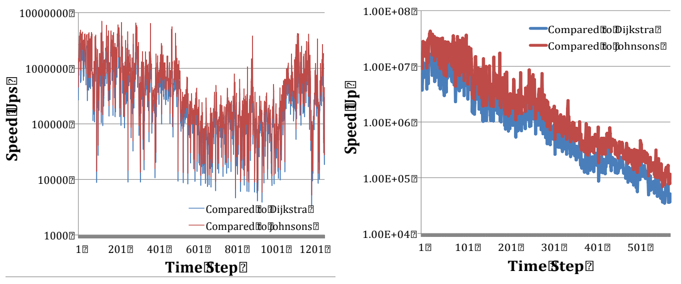


Fig. 3. Running time of closeness compared to baseline algorithms on WikiVote (left) and Foursquare (right).

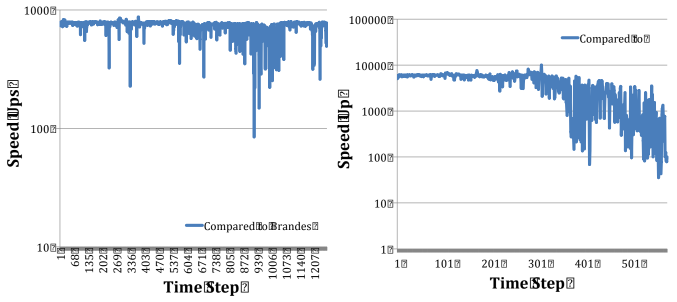


Fig. 4. Running time of betweenness compared to baseline algorithm on WikiVote (left) and Foursquare (right).

Fig. 5 and Fig. 6 illustrate the relationship between the number of link changes and the running time of closeness and betweenness. We see that the running time of the algorithm follows a fairly linear relationship to the number of link change. For closeness on foursquare data, the relationship is almost a perfect line. This suggests that the actual running time of the algorithm depends strongly on the number of link changes rather than the size of the networks.

Another way to look at the efficacy is to investigate how many steps those algorithms take in the convergence step. Fig. 7 shows the average steps to converge for both algorithms at each time step. It is illustrated that both of the algorithms have a fairly similar number of convergence steps. This is not surprising because we use a unified convergence algorithm for both of them. It is also noticed that the number of steps to converge increased dramatically for foursquare data. This is consistent with the observation in the previous analysis that there are more link changes being made in the later part of that data set.

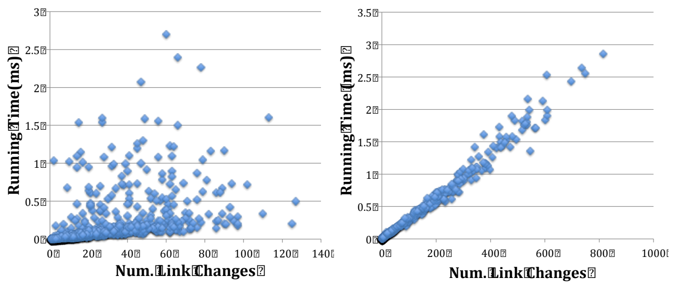


Fig. 5. Num. link changes V.S. running time for closeness on WikiVote (left) and Foursquare (right)

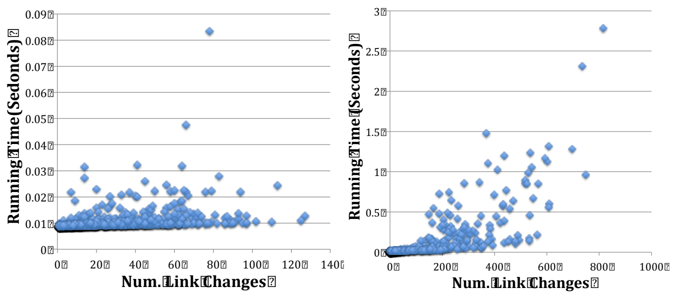


Fig. 6. Num. link changes V.S. running time for betweenness on WikiVote (left) and Foursquare (right)

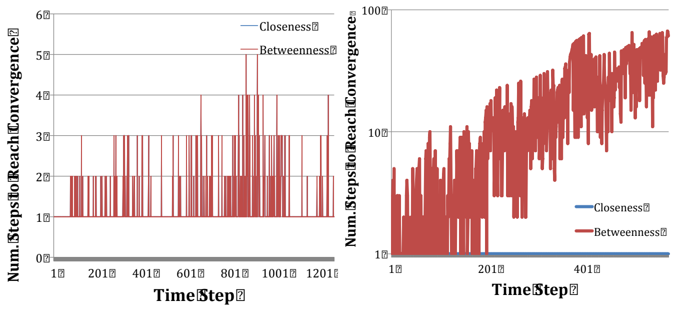


Fig. 7. Number of steps to converge for both metrics on WikiVote (left) and foursquare (right).

5. Conclusions and Discussions

In this work, we proposed a unified framework to handle closeness and betweenness on streaming network data. The algorithms utilize existing calculation results and update the centrality values incrementally. Experimental results show several magnitudes of speed up compared to baseline algorithms. The speed ups make the application of these two metrics on large and frequently changed social network data feasible. Various social media research that relies on these metrics such as real time network stability analysis, real time link prediction, network clustering algorithm and dynamic information diffusion models will benefit from this research.

In addition, we showed that the standardized algorithm framework could be divided into convergence step and aggregation step. The convergence step for both betweenness and closeness are similar except that betweenenss requires more quantities to be calculated. In the situation where both metrics are required, maintaining a convergence variable of betweenness while using different aggregation functions will significantly reduce the running time.

There are several ways to extend the work in that paper. First, the algorithm proposed in that paper has the potential to be paralleled or converted into a distributed algorithm. This will enable the algorithm to handle larger data sets at a higher link change rate. Another way to improve upon the current work would be to decrease the additional complexity of betweenness calculation and make it close to that of the closeness calculation.

Acknowledgement

This work is part of the dynamics networks project at the center for Computational Analysis of Social and Organizational Systems (CASOS) of the School of Computer Science (SCS) at Carnegie Mellon University (CMU). Support was provided, in part, by AFOSR (FA9550-11-1-0179), DTRA (HDTRA11010102) and ONR MURI (N000140811186). The views and proposal contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, the National Science Foundation, or the U.S. government.

References

1. Chen, W., et al. *Influence maximization in social networks when negative opinions may emerge and propagate*.

2. Bird, C., et al. Structure and Dynamics of Research Collaboration in Computer Science. in SDM. 2009. SIAM.
3. Srivastava, J., et al. Data mining based social network analysis from online behavior. in Tutorial at the 8th SIAM International Conference on Data Mining (SDM'08). 2008.
4. Chen, J., O.R. Zaïane, and R. Goebel. Detecting Communities in Social Networks Using Max-Min Modularity. in SDM. 2009.
5. Freeman, L.C., Centrality in Social Networks Conceptual Clarification. Social Networks, 1979. 1(3): p. 215-239.
6. Bonacich, P., Power and Centrality: A Family of Measures. American Journal of Sociology, 1987. 92(5): p. 1170-1182.
7. Latora, V. and M. Marchiori, A measure of centrality based on the network efficiency. 2004.
8. Jeong, H., et al., Lethality and centrality in protein networks. Nature, 2001. 411(6833): p. 41-42.
9. Eppstein, D. and J. Wang. Fast approximation of centrality. in Symposium on Discrete Algorithms. 2001.
10. Voloshin, S.A. and A.M. Poskanzer, The physics of the centrality dependence of elliptic flow. 1999.
11. Brandes, U., A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology, 2001. 25: p. 163-177.
12. Dijkstra, E.W., A note on two problems in connexion with graphs. Numerische Math, 1959: p. 269-271.
13. Johnson, D.B., Efficient algorithms for shortest paths in sparse networks. Journal of the ACM 24, 1977. 1(1-13).
14. Brandes, U., A faster algorithm for betweenness centrality. 2001.
15. Brandes, U. and C. Pich, Centrality estimation in large networks. International Journal of Bifurcation and Chaos, 2007. 17(07): p. 2303-2318.
16. Geisberger, R., P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. in ALENEX. 2008.
17. Lichtenwalter, R. and N.V. Chawla. DisNet: A framework for distributed graph computation. in Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on. 2011. IEEE.
18. Edmonds, N., T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. in High Performance Computing (HiPC), 2010 International Conference on. 2010. IEEE.
19. Madduri, K., et al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. in Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. 2009. IEEE.
20. Baglioni, M., et al. Fast exact computation of betweenness centrality in social networks. in Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012). 2012. IEEE Computer Society.
21. Sarıyüce, A.E., et al., Incremental Algorithms for Closeness Centrality.
22. Sariyuce, A.E., et al. STREAMER: A distributed framework for incremental closeness centrality computation. in Cluster Computing (CLUSTER), 2013 IEEE International Conference on. 2013. IEEE.
23. Green, O., R. McColl, and D.A. Bader. A fast algorithm for streaming betweenness centrality. in Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom). 2012. IEEE.
24. Lee, M.-J., et al. QUBE: a Quick algorithm for Updating BEtweenness centrality. in Proceedings of the 21st international conference on World Wide Web. 2012. ACM.
25. Kourtellis, N., G.D.F. Morales, and F. Bonchi, Scalable Online Betweenness Centrality in Evolving Graphs. arXiv preprint arXiv:1401.6981, 2014.
26. Kas, M., et al. 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*. 2013. ACM.