QUBE: a Quick algorithm for Updating BEtweenness centrality

Min-Joong Lee
Department of Computer Science, KAIST
291 Daehak-ro, Yuseong-gu
Daejeon, Korea
mjlee@islab.kaist.ac.kr

Jungmin Lee
Department of Computer Science, KAIST
291 Daehak-ro, Yuseong-gu
Daejeon, Korea
jungmin@islab.kaist.ac.kr

Jaimie Y. Park
Division of Web Science and Technology, KAIST
291 Daehak-ro, Yuseong-gu
Daejeon, Korea
yjpark@islab.kaist.ac.kr

Ryan H. Choi
Department of Computer Science, KAIST
291 Daehak-ro, Yuseong-gu
Daejeon, Korea
rchoi@islab.kaist.ac.kr

Chin-Wan Chung
Division of Web Science and Technology & Department of Computer Science, KAIST
291 Daehak-ro, Yuseong-gu
Daejeon, Korea
chungcw@kaist.edu

ABSTRACT

The betweenness centrality of a vertex in a graph is a measure for the participation of the vertex in the shortest paths in the graph. The Betweenness centrality is widely used in network analyses. Especially in a social network, the recursive computation of the betweenness centralities of vertices is performed for the community detection and finding the influential user in the network. Since a social network graph is frequently updated, it is necessary to update the betweenness centrality efficiently. When a graph is changed, the betweenness centralities of all the vertices should be recomputed from scratch using all the vertices in the graph. To the best of our knowledge, this is the first work that proposes an efficient algorithm which handles the update of the betweenness centralities of vertices in a graph. In this paper, we propose a method that efficiently reduces the search space by finding a candidate set of vertices whose betweenness centralities can be updated and computes their betweenness centeralities using candidate vertices only. As the cost of calculating the betweenness centrality mainly depends on the number of vertices to be considered, the proposed algorithm significantly reduces the cost of calculation. The proposed algorithm allows the transformation of an existing algorithm which does not consider the graph update. Experimental results on large real datasets show that the proposed algorithm speeds up the existing algorithm 2 to 2418 times depending on the dataset.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms, Path and circuit problems*; E.1 [**Data**]: Data structures—*Graphs and networks*

WWW 2012, April 16˜20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

General Terms

Algorithms, Theory

Keywords

Betweenness centrality, Update algorithm

1. INTRODUCTION

The betweenness centrality is a measure that computes the relative importance of a vertex in a graph, and it is widely used in network analyses such as a social network analysis, biological graph analysis, and road network analysis. For example, in the social network analysis, a vertex with higher centrality can be viewed as a more important vertex than a vertex with lower centrality. The betweenness centrality of a vertex in a graph is a measure for the participation of the vertex in the shortest paths in the graph.

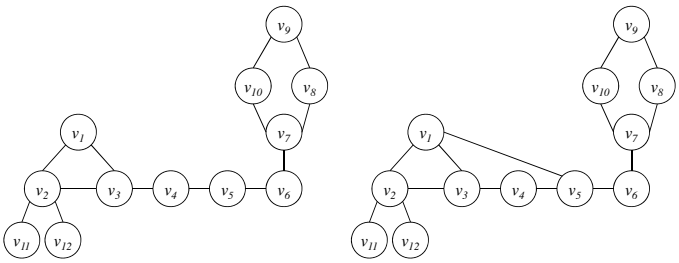
There are many previous works on the betweenness centrality problem. The concept of the betweenness centrality is proposed in [1], but the definition proposed in [10] is more widely used. Recently, many variants of the definition are proposed in [6]. [5] improves the computation time of the betweenness centrality based on a modified breadth-first search algorithm and the dependency of a vertex, and it is the fastest known algorithm that computes the exact betweenness centralities of all the vertices in a graph. As the computation of shortest paths between all pairs of vertices are time consuming, [22] proposes another definition of betweenness centrality, which is based on a random walk. In [22], each vertex has a probability of visiting its neighbor vertices. Also, [7], [2] and [12] propose approximation algorithms for computing the betweenness centrality. [23] and [25] adopt the betweenness centrality for detecting communities in a social network.

Although many works on calculating the betweenness centrality exist and the betweenness centrality is one of the major measures used in analyzing social network graphs, none of the works for computing the betweenness centrality address the problem of updating betweenness centrality.

Applying the previous algorithms to find influential users or detect communities over frequently updated graphs such as a social network graph is inefficient. This is because, calculating the betweenness centralities of all users in the graph involves computing the shortest paths between all pairs of users in the graph. In all previous works, the recomputation for all the vertices is inevitable whenever a new edge is inserted to the graph. This recomputation is clearly time-consuming. As the number of edges in the social network graph increases over time [19], the need for updating the betweenness centrality is evident.

It is difficult to update the betweenness centrality, because even a single edge insertion or a single edge deletion leads to the changes in many shortest paths in the graph. This change causes the updates of the betweenness centralities of many vertices in the graph. It is trivial to see that when an edge $(v_i,\ v_j)$ is inserted to a graph, the shortest path between v_i and v_j is changed. Also, the shortest paths that include the original shortest path from v_i to v_j are changed. For example, in Figure 1, let G_1 be a graph and G_1' be an updated graph of G_1. When an edge $(v_1,\ v_5)$ is inserted, the shortest path between v_1 and v_5 is changed. Also, there are more shortest paths that are changed e.g., the shortest path between v_{12} and v_5 and the shortest path between v_{10} and v_{11}.

However, we observe that there exist vertices whose betweenness centralities do not change even when the graph is updated. In Figure 1(b), the betweenness centralities of $v_1,\ v_3,\ v_4$ and v_5 change, while the betweenness centralities of the other vertices do not change. The betweenness centralities of $v_2,\ v_6,\ v_7,\ v_8,\ v_9,\ v_{10},\ v_{11}$ and v_{12} do not change, because the source-target pairs of original shortest paths that go through $v_2,\ v_6,\ v_7,\ v_8,\ v_9,\ v_{10},\ v_{11}$ or v_{12} do not change even when G_1 is updated.



(a) G_1 (b) G_1'

Figure 1: An example of a graph update

Based on the above observation, we proposed a Quick algorithm for Updating BEtweenness centrality (QUBE). The key idea of the proposed algorithm is to perform the betweenness centrality computation on a reduced set of vertices. We first find the set of vertices whose betweenness centralities can be changed and the set(s) of vertices whose betweenness centralities do not change. In Figure 1(b), $\{v_1,v_2,v_3,v_4,v_5\}$ is the set of vertices whose betweenness centralities can be changed, and $\{v_6,v_7,v_8,v_9,v_{10}\}$, $\{v_{11}\}$, $\{v_{12}\}$ are the sets of vertices whose betweenness centralities do not change. The method of finding these sets is explained in Section 4, and it is easy to see that the latter three sets correspond to connected components after removing the first set from G_1'.

We compute the betweenness centrality only on the first set of vertices. In the previous works, all pair shortest paths recomputation is necessary to compute the betweenness centrality and the number of shortest paths that need to be recomputed on G_1' would be $12 \cdot 11/2 = 66$. On the other hand, in our approach, only $5 \cdot 4/2 = 10$ shortest paths need to be recomputed. Clearly, the smaller the cardinality of the first set, the shorter the amount of time it would take in computing the necessary shortest paths.

In order to recompute the betweenness centrality of a vertex in the reduced set, in addition to the betweenness centrality in the reduced set, the number of shortest paths that satisfy the following conditions needs to be considered.

1. The shortest path goes through the vertex in the reduced set.
2. The shortest path's source or target or both are not in the reduced set.

The number can be obtained based on the cardinality of the set(s) of vertices whose betweenness centralities do not change without actually computing the shortest paths. For example, the shortest paths from v_{12} to v_6 always go through vertices in the reduced set. Similarly, the shortest paths from $v_i \in \{v_{12}\}$ to $v_j \in \{v_6,\ v_7,\ v_8,\ v_9,\ v_{10}\}$ always go through vertices in the reduced set. Therefore, the number of shortest paths from v_i to v_j is a product of the cardinalities of the two sets, which is 5.

The contributions of this paper are as follows.

1. We propose a method that identifies a set of vertices whose betweenness centralities can be updated and sets of vertices whose betweenness centralities do not change, based on the comprehensive analysis of changes in the betweenness centrality when a graph is updated.
2. We devise a *Betweenness Centrality Update Theorem*. The theorem enables an efficient update of betweenness centrality without traversing the entire graph. Based on the proposed theorem, we propose an efficient algorithm for updating betweenness centrality.
3. We conduct experiments on various synthetic datasets as well as large real datasets. The experimental results show that the incorporation of our algorithm outperforms an existing algorithm, in updating the betweenness centrality. In cases where the size of the reduced set of vertices is 1/10 of the number of vertices in the synthetic graphs, the proposed algorithm speeds up the existing algorithm 577 times on the average. For real datasets, the proposed algorithm speeds up the exist algorithm 2 to 2418 times depending on the dataset.

The rest of the paper is organized as follows. In Section 2, related works on betweenness centrality are reviewed. In Section 3, we formally define betweenness centrality and explain basic concepts. In Section 4, we devise a method which finds the reduced set of vertices whose betweenness centralities can be updated. Section 5 explains how to efficiently update the betweenness centralities of vertices in the reduced set. In Section 6, we show experimental results, and we conclude the paper in Section 7.

2. RELATED WORK

Computation of betweenness centrality has been gaining much importance in social network analyses, and is widely

used in many applications. The earliest work to define the measure which quantifies this idea of betweenness centrality is introduced by Anthonisse et al. [1] and Freeman [10]. Freeman's original method of finding betweenness centrality is based on counting geodesic paths for all pairs of vertices on a graph. Following Freeman's work, variations of centrality measures are proposed. Everette et al. [17] propose a group betweenness measure which can be applied to groups and classes as well as individuals. Freeman et al. [11] extend Freeman's work [10] to introduce a new measure of centrality based on the concept of network flows, which considers both shortest and certain non-shortest paths. Newman [22] proposes a measure of betweenness centrality based on random walks of any length instead of shortest paths. Brandes [6] reviews a number of variants of betweenness centrality based on shortest paths including bounded-distance betweenness, distance-scaled betweenness, edge betweenness, and group betweenness, and discusses algorithms to compute each variant efficiently. As part of the discussion, Brandes points out that the efficient recomputation of betweenness centrality in dynamically changing networks on the algorithmic side is a remaining challenge.

Currently, the fastest known algorithm to compute exact betweenness centralities for all the vertices [5] requires $O(|V||E|)$ and $O(|V||E| + |V|^2 log|V|)$ time on weighted and unweighted graphs, respectively. Traditionally, betweenness centrality was determined by first computing the lengths and number of shortest paths between all pairs, and then summing up pair-dependencies of all pairs [10]. Pair-dependency of a pair $s, t \in V$ on an intermediary vertex $v \in V$ is defined as the ratio of shortest paths between s and t that v lies on to all shortest paths between s and t. Brandes [5] points out the weakness in this approach arguing it is computing more information than needed. The faster algorithm is presented by Brandes [5], based on aggregating path counts from different source vertices in the network.

Although big improvement was made over the very initial betweenness centrality computation algorithm, many researchers argued that the Brandes' algorithm is still too costly for large graphs. In order to overcome such limitation, researchers propose approximation algorithms to compute the estimated betweenness centrality, claiming that good approximation would be an acceptable alternative to exact score as long as fast computation is possible. Brandes et al. [7] propose a heuristic estimation method for betweenness centrality computation and conduct experiments with various selection strategies of the source vertices to assess the quality of the estimation. Bader et al. [3] present a parallel algorithm for computing betweenness centrality, optimized for scale-free sparse graphs. They [2] also suggest an algorithm to compute the betweenness centrality of a single vertex in time faster than computing the betweenness of all vertices. Geisberger et al. [12] suggest a bisection scaling algorithm for approximating a variant of betweenness centrality. Makarychev [21] suggests a linear time approximation algorithm to find the ordering of the vertices that maximizes the number of satisfied betweenness constraints.

Betweenness centrality is used in diverse applications across many different disciplines. Betweenness centrality allows an understanding of the extent to which a vertex contributes in the flow of information. It is mainly used in finding the most prominent vertices in complex networks, whether they are individuals in social networks, elements in biological networks, intersections or junctions in transportation networks, physical elements in computer networks, or documents in World Wide Web. For example, Leydesdorff [20] demonstrates in his research how betweenness centrality is shown to be an indicator of the interdisciplinarity of scientific journals, and del Sol et al. [8] use the betweenness centrality in identifying the most central residues in protein-protein complex structures. Jin et al. [15] demonstrate an application of parallel betweenness centrality to detect potentially harmful nodes in an electrical grid. The electrical grid is an interconnected network for delivering electricity from suppliers to consumers. Holme [13] studies the relationship between betweenness centrality and the density of a traffic model, and Lammer et al. [18] use betweenness centrality in approximating the importance of a road or a junction and investigated the scaling laws associated with urban road networks in Germany. In many applications, the network structures are typically not static. As the network evolves, the network graphs constantly change over time, which implies that there is a strong need for an efficient algorithm to update betweenness centrality.

Betweenness centrality is also used in community detection. Newman et al. [23] propose a divisive community detection technique which iteratively removes edges with the highest betweenness centrality value from the network. Pinney et al. [25] suggest an alternative community detection algorithm in which the network decomposition is based on vertex betweenness instead of edge betweenness. Newman et al. [23] discuss a weakness in the existing algorithms which is a high computation cost associated with iterative recalculation of all-pair shortest paths when the edges are removed.

As observed in many applications, dynamic nature of many real-life networks is a clear evidence that efficiently updating betweenness centrality is an important issue. Yet no literature dealing with the problem of efficiently updating betweenness centrality in a dynamic network environment exists at present.

3. PRELIMINARY

In this section, we introduce the formal definition of betweenness centrality of a vertex, and explain the basic concept of a minimum cycle basis. Also we briefly introduce the overall process of the proposed algorithm.

3.1 Betweenness Centrality

Betweenness centrality is a measure that computes the relative importance of a vertex in a graph. The formal definition is presented below.

A graph is represented by $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of edges. A path in a graph is represented by a sequence of vertices, $(v_1, ..., v_n)$ where $v_i, v_j \in V$ for $1 \leq i, j \leq n$, $i \neq j$ except possible $1 = n$.

Definition 1 (Betweenness Centrality)**.** The betweenness centrality of a vertex $v_j \in G$ is:

$$c(v_j) = \sum_{i,k} \frac{\sigma_{v_i,v_k}(v_j)}{\sigma_{v_i,v_k}} \tag{1}$$

where $v_i, v_j, v_k \in V$, $i \neq j \neq k$, $\sigma_{v_i,v_k}(v_j)$ is the number of shortest paths between v_i and v_k that include v_j, and σ_{v_i,v_k} is the number of shortest paths between v_i and v_k.

The betweenness centrality can be computed as follows:

1. For each pair of vertices (v_s and v_t), compute the shortest paths between the two vertices.
2. For each pair of vertices, compute the ratio of the participation of each vertex in the shortest path(s). The ratio is the number of shortest paths between v_s and v_t that go through v_j divided by the number of shortest paths between v_s and v_t.
3. Accumulate the ratio for all pairs of vertices.

Let us consider updating the betweenness centrality caused by a graph update. Even a simple update, for example inserting an edge to a graph, could change existing shortest paths for many pairs of vertices in the graph. One of the biggest drawbacks in updating the betweenness centrality using the previous algorithms is that the shortest paths for all pairs of vertices are recomputed whenever an update occurs in a graph.

3.2 Minimum Cycle Basis

Definition 2 (Cycle Basis)**.** Let a graph $G = (V, E)$ be an undirected graph. A cycle C is a subset of edges such that every vertex of V is incident to an even number of edges in C. Each cycle C can be represented by an edge incidence vector in $\{0, 1\}^{|E|}$ where a component is equal to 1 precisely when $e \in C$. A maximal set of linearly independent cycles is called a cycle basis.

Definition 3 (Minimum Cycle Basis (MCB))**.** Let a graph $G = (V, E)$ be an undirected connected graph with a non-negative weight w_e assigned to each edge $e \in E$. Minimum Cycle Basis is a cycle basis C of minimum total weight, i.e., which minimizes $w(C) = \sum_{i=1}^{v} w(C_i)$, where $w(C_i) = \sum_{e \in C_i} w_e$.

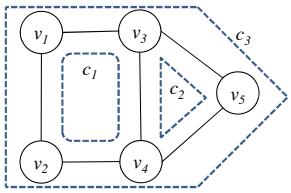


Figure 2: An example of a cycle basis and the minimum cycle basis

The example depicted in Figure 2 has three cycle basis sets $\{C_1, C_2\}$, $\{C_1, C_3\}$, and $\{C_2, C_3\}$. If every edge in the graph has the same weight (i.e., 1 for all edges), MCB is $\{C_1, C_2\}$. The detailed definitions of cycle basis and minimum cycle basis can be found in [16].

3.3 Overall Process

The overall flow of the proposed betweenness centrality update algorithm is as follows. First, we identify the set of vertices whose betweenness centralities can be changed, and the set(s) of vertices whose betweenness centralities do not change. Through the analysis of possible changes in the betweenness centrality that can occur as a result of graph updates, we discovered the characteristics of the sets of vertices in which the changes in the betweenness centralities do and do not occur. Observed pattern is applicable for any type of connected graphs. Theoretical evidence on the generalization is presented in Section 5.1.

Next, we perform the betweenness centrality computation on the identified sets of vertices whose betweenness centralities can be changed. We refer to the computed values as the local betweenness centrality. On top of the local betweenness centrality, we perform additional calculations on the vertices whose shortest paths are not yet considered. Details are presented in Section 5. Through simple additional calculations, the exact betweenness centrality can be restored without performing an expensive computation on all the vertices on a graph, such as the calculation of all pair shortest paths.

4. MINIMUM UNION CYCLE

In this section we introduce the concept of the minimum union cycle (MUC) upon which our update algorithm is built. As explained in the previous section, a set of vertices whose betweenness centralities can be changed is distinguished from the set(s) of vertices whose betweenness centralities do not change. Such sets are identified by using MUCs obtained during the preprocessing time. The initial set of MUCs is found and stored during the preprocessing time. As changes occur in a graph, stored MUCs also need to be changed. Changes in MUCs are managed during the runtime. In Sections 4.2 and 4.3, we explain how to find MUCs and how to update MUCs.

4.1 Definition of MUC

Definition 4 (Minimum Union Cycle (MUC))**.** Given a minimum cycle basis C and minimum cycles $C_i \in C$, let V_{C_i} be the set of vertices in C_i. Recursively union two V_{C_i}s together if they share at least one common vertex. Then each final set of vertices forms a MUC.

Each vertex appears in only one MUC since MUCs are disjoint sets. We denote $MUC(v)$ as MUC which contains vertex v.

Definition 5 (Connection vertex)**.** Vertex $v \in MUC$ is a connection vertex, if v is an articulation vertex[1] and v has an edge to a vertex $w \notin MUC(v)$.

In Figure 5, let us assume that an edge (v_3, v_4) is inserted. $MUC(v_3)$ is $\{v_1, v_2, v_3, v_4\}$, and the connection vertices of $MUC(v_3)$ are v_1, v_2 and v_3.

The deletion of a connection vertex makes the graph disconnected since the connection vertex is also an articulation vertex. We denote a graph that is disconnected from $MUC(v_i)$ as a result of the deletion of a connection vertex v_i as a disconnected subgraph G_i. In Figure 5, G_1, G_2, and G_3 are disconnected subgraphs generated from the deletion of connection vertex v_1, v_2, and v_3, respectively.

4.2 Finding MUCs

In this subsection, we present how to generate MUCs, a set of connection vertices for each MUC and disconnected subgraphs derived from the deletion of connection vertices.

[1]A vertex a is called an articulation vertex if the deletion of a with its incident edges from G makes the graph disconnected. Equivalently, there must exist two vertices v and w such that every path from v to w goes through an articulation vertex a.

Algorithm 1: FindMUC(C)

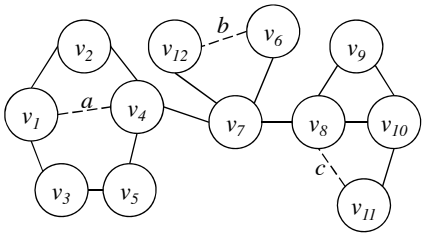
```
   input  : C - minimum cycle basis
1  begin
2      MUCSet := A minimum cycle basis C;
3      while ∃ c_i, c_j ∈ MUCSet, where c_i and c_j share at
       least one common vertex do
4          c_i := c_i union c_j ;
5          Remove c_j from MUCSet ;
6      for each MUC ∈ MUCSet do
7          Conn(MUC) := a set of connection vertices in MUC
           ;
8          for each connection vertex v_i ∈ Conn(MUC) do
9              G_i := disconnected subgraphs originated from
               the deletion of a connection vertex v_i ;
```

Algorithm 1 uses a minimum cycle basis C as an input, and it finds a set of MUCs ($MUCSet$), and a set of connection vertices with corresponding subgraphs.

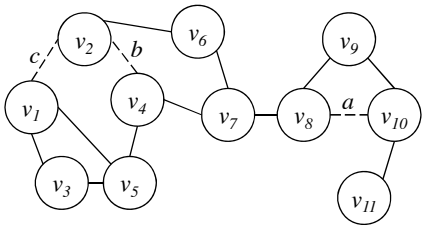
The calculation of a minimum cycle basis is well studied in the field of graph theory, and many efficient algorithms, such as Horton's algorithms [14] and Kavitha's algorithm [16], exist. In line 2, we calculate a minimum cycle basis using an existing algorithm. In line 3-line 5, the algorithm finds a set of MUCs ($MUCSet$) by unioning the cycles in a minimum cycle basis until the unioned cycles are disjoint from each other. A set of connection vertices for each MUC and disconnected subgraphs derived from the deletion of connection vertices are found in line 7-line 9.

Note that Algorithm 1 is performed during the preprocessing time. However, the MUC updating algorithm (Algorithm 2) needs to be processed during the runtime. MUC updating algorithm for an insertion and deletion of an edge is presented in the following subsection.

4.3 Updating MUCs



(a) *Insertion*



(b) *Deletion*

Figure 3: An example of updating MUC

We now present our technique on maintaining a set of MUCs, a set of connection vertices for each MUC and disconnected subgraphs derived from the deletion of a connection vertex. We explain each case of updating MUCs according to the insertion or deletion of an edge as follow (Initial MUCs in Figure 3(a) are $\{v_1, v_2, v_3, v_4, v_5\}$ and $\{v_8, v_9, v_{10}\}$. Initial MUCs in Figure 3(b) are $\{v_1,\ v_2, v_3,\ v_4, v_5\ , v_6, v_7\}$ and $\{v_8, v_9, v_{10}\}$.):

Algorithm 2: UpdateMUC($v_i, v_j, MUCSet, G$)

```
   input  : v_i - a vertex in inserted/deleted edge
            v_j - a vertex in inserted/deleted edge
            MUCSet - a set of MUCs
            G - an original graph
1  begin
       // if v_i is not contained any MUC, MUC(v_i)
          returns v_i only
2      if Insertion Operation then
3          if MUC(v_i) = MUC(v_j) then
4              // Do Nothing
5          else
6              NewMUC := Let be an empty set;
7              for each vertices v in ςρ(v_i, v_j) in G do
8                  NewMUC := NewMUC ∪ MUC(v) ;
9                  remove MUC(v) from MUCSet ;
10             add NewMUC to MUCSet ;
11         add edge (v_i, v_j) to graph G;
12     else
13         delete edge (v_i, v_j) from graph G;
14         if 1 = |Path(v_i, v_j)| in G then
15             remove MUC(v_i) from MUCSet ;
16         else
17             if ∃v in all Path(v_i, v_j) then
18                 split MUC(v) into MUCs;
19             else
20                 // Do Nothing
```

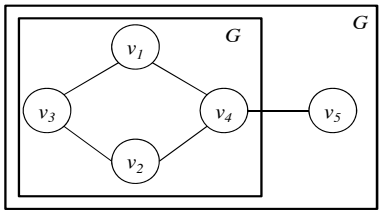
1. When an edge is inserted
   (a) No change, if the new edge connects two vertices in one MUC. In Figure 3(a), the insertion of $edge(a)$ does not affect any MUCs (Line 4 in Algorithm 2).
   (b) A new MUC is created, if vertices in an existing shortest path between two vertices in the new edge are not included in any MUC. In Figure 3(a), the insertion of $edge(b)$ induces a creation of a new MUC consisting of $\{v_6, v_7, v_{12}\}$ (Line 6-Line 10 in Algorithm 2. $\varsigma\rho(v_i, v_j)$ is the set of vertices in shortest paths between v_i and v_j).
   (c) MUC is merged with the vertices and other MUCs to create a new MUC, if vertices in existing shortest paths between two vertices of the new edge are included in some MUCs. In Figure 3(a), the insertion of $edge(c)$ induces $MUC(v_{10})$ to merge with v_{11} (Line 6-Line 10 in Algorithm 2).
2. When an edge is deleted
   (a) MUC is destroyed, if there exists only one path between two vertices in the deleted edge as a result of the deletion. In Figure 3(b), the deletion of $edge(a)$ causes the destruction of $MUC(v_{10})$ (Line 14-Line 15 in Algorithm 2. $Path(v_i, v_j)$ is the set of paths between v_i and v_j.).

(b) No change, after the deletion, if there still exists more than one path between the two vertices and does not exist a vertex appearing in all the paths between the two vertices. In Figure 3(b), the deletion of $edge(b)$ does not affect any MUCs (Line 20 in Algorithm 2).

(c) An existing MUC is split into MUC(s) and vertex(s), after the deletion, if there still exists more than one path between two vertices and exists a vertex appearing in all paths between the two vertices. In Figure 3(b), the deletion of $edge(c)$ induces the separation of MUC into two MUCs (Line 18 in Algorithm 2).

5. UPDATING BETWEENNESS CENTRALITY

In this section, we describe how to compute the betweenness centrality values. As mentioned in Section 4, after an insertion or deletion of the edge $e(v_i, v_j)$, we guarantee that the betweenness centralities of vertices in $MUC(v_i)$ can be changed. Therefore, after we find the reduced set of vertices, which we refer to as MUC, we need to efficiently calculate and update the betweenness centralities of the vertices in the MUC to which the updated vertices belong. From now on, we simply denote such MUC as MUC_U.



	G	G
$c(v_1)$	1	0.5
$c(v_2)$	1	0.5
$c(v_3)$	0.5	0.5
$c(v_4)$	3.5	0.5
$c(v_5)$	0	

Figure 4: An example of the dependency of the betweenness centrality

Recomputing betweenness centralities of the vertices in a graph every time an update occurs is expensive, because in general, the recomputation involves computation of all pair shortest paths in the graph. In the previous section, we proposed a way to find a set of vertices whose betweenness centralities can be changed. This set of vertices is referred to as MUC_U. Yet, calculating the betweenness centrality using only the vertices in MUC_U is insufficient. In fact, the betweenness centralities calculated using only the vertices in MUC_U are always smaller than the betweenness centralities calculated using all the vertices in a graph. This is because, (1) the shortest paths whose source or target is not in MUC_U, and (2) the shortest paths that pass though MUC_U and both the source and the target of the shortest paths are not in MUC_U, are not yet considered. For example, Figure 4 shows the betweenness centralities of the vertices in G, and a subgraph of G, G'. The betweenness centralities of vertices calculated using only the vertices in G' are smaller than the values calculated using the vertices in G. This is because, the paths from v_5 to each vertex in G' are missing.

Based on this idea, we now explain how to restore the exact betweenness centrality by considering the vertices in MUC_U only. Let us refer to the betweenness centrality calculated using only the vertices in MUC_U as the local betweenness centrality and the betweenness centrality calculated using the entire vertices in the graph as the global betweenness centrality.

5.1 Betweenness Centrality Update Theorem

Before we introduce our technique, we define some terminologies for a better understanding. $c_{_{MUC}}(v_i)$ denotes the local betweenness centrality of a vertex v_i calculated using the vertices in MUC_U only. c_i represents a connection vertex. $\varsigma\rho(v_i, v_j)$ is the set of vertices in the shortest paths between v_i and v_j, and $SP(v_i, v_j)$ is the set of shortest paths between v_i and v_j. Therefore, $|SP(v_i, v_j)|$ is the number of the shortest paths between v_i and v_j. For example, in Figure 4, $|SP(v_1, v_5)| = 1$, $|SP(v_1, v_2)| = 2$, and $\varsigma\rho(v_1, v_2)$ is $\{v_1, v_2, v_3, v_4\}$.

G_j represents a disconnected subgraph originated from a deletion of the connection vertex, c_j. G_j^l represents the lth connected component of G_j. V_{G_j} is the set of vertices of G_j. In Figure 5, G_1, G_2 and G_3 represent disconnected subgraphs originated from the deletions of connection vertices, v_1, v_2, and v_3, respectively. G_2^1 and G_2^2 are connected components of G_2. If the dotted edge is inserted, MUC_U is $\{v_1, v_2, v_3, v_4\}$ and connection vertices of MUC_U to G_1, G_2, and G_3 are v_1, v_2, and v_3, respectively.

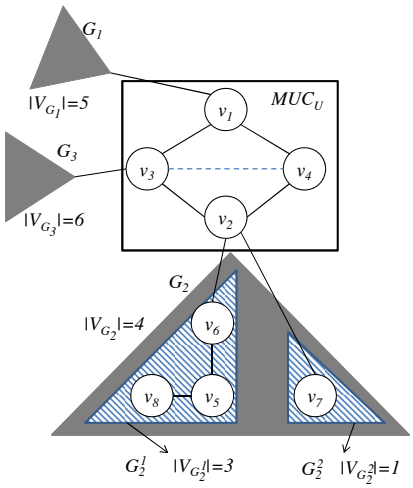


Figure 5: An example of updating the betweenness centrality (vertices in G_1 and G_3 are omitted.)

Lemma 1. Let $v_s \in V_{G_j}$, $v_t \in MUC_U$ and c_j be a connection vertex which connects MUC_U with G_j. Then each vertex in $\varsigma\rho(c_j, v_t)$ must be included in a $\varsigma\rho(v_s, v_t)$.

Proof: Since a connection vertex in MUC_U is also an articulation vertex, all paths from $v_s \in V_{G_j}$ to $v_t \in MUC_U$ go through a connection vertex c_j. Therefore $\varsigma\rho(v_s, v_t)$ always includes $\varsigma\rho(c_j, v_t)$.□

Lemma 1 allows us to calculate the increase of the betweenness centrality due to the shortest paths whose source or target is not in MUC_U (the shortest paths *between* the vertices in MUC_U and the vertices not in MUC_U). Such increase of the betweenness centrality for v_i is denoted as $c_{b_j}(v_i)$.

$$c_{b_j}(v_i) = \begin{cases} \frac{|V_{G_j}|}{|SP(v_s, v_t)|} & if\ v_i\ in\ \varsigma\rho(c_j, v_t) - \{v_t\} \\ 0 & otherwise \end{cases} \quad (2)$$

where v_s in V_{G_j}, $v_t \in MUC_U$, c_j is a connection vertex to G_j.

Lemma 2. Let $v_s \in V_{G_j}$, $v_t \in V_{G_k}$, and c_j and c_k be connection vertices which connect MUC_U with G_j, and MUC_U with G_k, respectively. Then each vertex in $\varsigma\rho(c_j, c_k)$ must be included in a $\varsigma\rho(v_s, v_t)$.

Proof: Since c_j and c_k are articulation vertices, all paths from $v_s \in V_{G_j}$ to $v_t \in V_{G_k}$ go through connection vertices c_j and c_k. Therefore $\varsigma\rho(v_s, v_t)$ always includes $\varsigma\rho(c_j, c_k)$.□

Lemma 2 allows us to calculate the increase of the betweenness centrality due to the shortest paths that *pass through* MUC_U and whose source and target are both not in MUC_U. Such increase of the betweenness centrality for v_i is denoted as $c_{t_j^k}(v_i)$.

$$c_{t_j^k}(v_i) = \begin{cases} \frac{|V_{G_j}| \cdot |V_{G_k}|}{|SP(v_s, v_t)|} & if\ v_i\ in\ \varsigma\rho(c_j, c_k) \\ 0 & otherwise \end{cases} \quad (3)$$

where $v_s \in V_{G_j}$, $v_t \in V_{G_k}$, c_j and c_k are connection vertices to G_j and G_k, respectively.

In the case where G_i is disconnected, all shortest paths between the two vertices from different connected components of G_i always *pass through* v_i. For example, in Figure 5, a shortest path from $v_s \in G_2^1$ to $v_t \in G_2^2$ must *pass through* v_2. Such an increase of the betweenness centrality for v_i is denoted as $c_{t_i}(v_i)$ and calculated as follows:

$$c_{t_i}(v_i) = \begin{cases} |V_{G_i}|^2 - \sum_{l=1}^{n}(|V_{G_i^l}|^2) & if\ G_i\ is\ disconnected \\ 0 & otherwise \end{cases} \quad (4)$$

where G_j^l is the lth connected component of G_i, n is the number of connected components in G_i, and v_i is the connection vertex to G_i.

Theorem 1. *(Betweenness Centrality Update Theorem)* By Lemma 1 and Lemma 2, we can compute the betweenness centrality of a vertex v_i, $c(v_i)$.

$$c(v_i) = c_{MUC}(v_i) + \sum_{G_j \subset G} c_{b_j}(v_i) + \sum_{G_j, G_k \subset G, j \neq k} c_{t_j^k}(v_i) + \sum_{G_i \subset G} c_{t_i}(v_i) \quad (5)$$

where $c_{b_i}(v_i)$ is from Equation 2 (Lemma 1) and $c_{t_j^k}(v_i), c_{t_i}(v_i)$ are from Equation 3 and 4 (Lemma 2).

By Theorem 1, we can compute the global betweenness centrality using the local betweenness centrality and the number of vertices in each disconnected subgraph G_i without performing all pair shortest paths computation on the all the vertices in a graph.

5.2 Betweenness Centrality Update Algorithm (QUBE)

Algorithm 3 shows how to update betweenness centrality only using vertices in MUC_U that updated vertices belong to. Algorithm 3 uses MUC_U as an input and calculates the updated betweenness centrality ($C[v_i]$) as an output. The set of all pair shortest paths in MUC_U and the local betweenness centralities of vertices in MUC_U are calculated using the existing betweenness centrality algorithms (Line 4)[2]. Then for each shortest path between the vertices in MUC_U (Line 5), add the increase of betweenness centrality values due to the shortest paths between the vertices in MUC_U and the vertices in other subgraphs (Line 9), as well as the shortest paths between the vertices in two other subgraphs, which pass through MUC_U (Line 13) and the shortest paths between the two vertices from different connected components of a subgraph (Line 15). Note that it does not require additional costs to obtain SP, the set of all pair shortest paths in MUC_U, since all pair shortest paths are already calculated when we compute the local betweenness centrality and can be easily obtained.

Algorithm 3: QUBE(MUC_U)

```
   input  : MUC_U - Minimum Union Cycle that updated
            vertices belong to
   output : C[v_i] - Updated Betweenness Centrality Array
1  begin
2      Let SP be the set of all pair shortest paths in MUC_U ;
3      Let C[v_i] be an empty array, v_i ∈ MUC_U ;
4      SP, C[v_i] ← Betweenness() ;
5      for each shortest path <v_a, ..., v_b> in SP do
6          if v_a is a connecting vertex then
7              G_a := Subgraph connected by a connection
                      vertex v_a ;
8              for each v_i ∈ <v_a, ..., v_b> - {v_b} do
9                  C[v_i] := C[v_i] + |V_Ga| / |SP(v_a,v_b)| ;
10                 if v_b is also a connecting vertex then
11                     G_b := Subgraph connected by a
                              connection vertex v_b ;
12                     for each v_i ∈ < v_a, ..., v_b > do
13                         C[v_i] := C[v_i] + |V_Ga|·|V_Gb| / |SP(v_a,v_b)| ;
14             if G_a is disconnected then
15                 C[v_a] := C[v_a] + |V_Ga|^2 - Σ_{l=1}^{n}(|V_{G_a^l}|^2)
```

Example 1. Table 5.2 shows the values computed using Equation 2, Equation 3, and Equation 4 for the vertices in MUC_U depicted in Figure 5. Due to the space limitation, we do not differentiate a path from v_s to v_t and a path from v_t to v_s in this example. Therefore, the actual betweenness centralities are twice as big as the values shown in this example. For vertex v_2, the local betweenness centrality, $c_{MUC}(v_2)$, is 0. And there are four shortest paths (v_2-v_3-v_1, v_2-v_4-v_1, v_2-v_3, v_2-v_4) which start from v_2. Therefore, we add '$|V_{G_1}| = 4$' 2 times for the paths v_2-v_3 and v_2-v_4. Since $|SP(v_2, v_1)| = 2$, we add '$|V_{G_1}|/2 = 2$' 2 times to v_2's betweenness centrality for the paths v_2-v_3-v_1 and v_2-v_4-v_1.

Also, v_1-v_3-v_2, v_1-v_4-v_2 are shortest paths which include v_2 and connect G_1 and G_2 via v_1 and v_2. Therefore, we add a half of the product of the numbers of vertices in G_1 and G_2, which is '$|V_{G_1}| \cdot |V_{G_2}|/2 = 5 \cdot 4/2$' for each path. The path v_2-v_3 connects G_2 and G_3 via v_2 and v_3. Similar to the above case, we add '$|V_{G_2}| \cdot |V_{G_3}| = 4 \cdot 6$' to v_2's betweenness centrality. G_2 is a disconnected graph and v_2 is a connection vertex to G_2. Therefore, we add the product of the numbers of vertices in G_2^1 and G_2^2, which is $3 \cdot 1$. Finally, we get 59

[2]Algorithm 3 can use any existing betweenness centrality algorithm. For the implementation of Algorithm 3, we use the Brandes' algorithm which is the fastest known algorithm so far. Our implementation is explained in Section 6.1

Table 1: An example of updating the betweenness centrality

	$SP(c_j, \bullet)$	*Paths*	v_1	v_2	v_3	v_4
$c_{MUC}(v_i)$			0	0	0.5	0.5
$c_{b_1}(v_i)$	$SP(c_1, v_2)$	v_1-v_3-v_2	5/2		5/2	
		v_1-v_4-v_2	5/2			5/2
	$SP(c_1, v_3)$	v_1-v_3	5			
	$SP(c_1, v_4)$	v_1-v_4	5			
$c_{b_2}(v_i)$	$SP(c_2, v_1)$	v_2-v_3-v_1		4/2	4/2	
		v_2-v_4-v_1		4/2		4/2
	$SP(c_2, v_3)$	v_2-v_3		4		
	$SP(c_2, v_4)$	v_2-v_4		4		
$c_{b_3}(v_i)$	$SP(c_3, v_1)$	v_3-v_1			6	
	$SP(c_3, v_2)$	v_3-v_2			6	
	$SP(c_3, v_4)$	v_3-v_4			6	
$c_{t_1^2}(v_i)$	$SP(c_1, c_2)$	v_1-v_3-v_2	5·4/2	5·4/2	5·4/2	
		v_1-v_4-v_2	5·4/2	5·4/2		5·4/2
$c_{t_1^3}(v_i)$	$SP(c_1, c_3)$	v_1-v_3	5·6		5·6	
$c_{t_2^3}(v_i)$	$SP(c_2, c_3)$	v_2-v_3		4·6	4·6	
$c_{t_2}(v_i)$		v_2		3·1		
$c(v_i)$			65	59	87	15
Actual value			130	118	174	30

as the global betweenness centrality which is the same value resulting from the calculation of the betweenness centrality of v_2 using all the vertices in the original graph. The betweenness centrality values for the other vertices v_1, v_3 and v_4 can also be calculated in the same way as the case of v_2.

6. EXPERIMENTS

We explain how we implement an updatable version of the Brandes' algorithm using QUBE based on the Brandes' algorithm in Section 6.1. We compare this updatable version of the Brandes' algorithm with the original Brandes' algorithm and we show how much improvement is achieved with the help of QUBE in Section 6.2. Recall that all previous betweenness centrality calculation algorithms which do not consider the graph update, such as the Brandes' algorithm, inevitably require the computation of the betweenness centrality from scratch whenever a graph changes. We conduct experiments on Intel Xeon CPU with 2.53GHz and 20GB main memory.

6.1 Implementation

As we explained in Algorithm 3, QUBE can be applied to any betweenness centrality calculation algorithm since it reduces the search space by identifying the candidate vertices and restores the global betweenness centrality using the local betweenness centrality. Since the Brandes' algorithm known to be the fastest algorithm so far for computing the exact betweenness centrality, we implement QUBE based on the Brandes' algorithm. The Brandes' algorithm computes one-sided pair dependencies of all the vertices in a graph for a given source vertex by solving a single source shortest path problem. The one-sided pair dependency of a vertex v_t is the number of all shortest paths that go through v_t, where the start vertex of the paths is fixed to a specific vertex v_s. A detailed description of the Brandes' algorithm can be found in [5].

As we mentioned in Section 5, in addition to the local betweenness centrality, we need to calculate (1) the increase of the betweenness centrality (Equation 2) due to the shortest paths whose source or target is a connection vertex, and (2) the increase of the betweenness centrality (Equation 3 and Equation 4) due to the shortest paths whose source and target are both connection vertices. Since the Brandes' algorithm does not explicitly calculate the all pair shortest paths, the necessary values for the calculation of (1) and (2) are obtained during the computation of the one-sided pair dependencies on the Brandes' algorithm. The detailed implementation of the updatable version of the Brandes' algorithm using QUBE is shown in Algorithm 4. The additional increases in the betweenness centrality explained through Equation 2, Equation 3, and Equation 4 are computed in (Line 34), (Line 23-Line 26 and Line 29-Line 31), and (Line 38), respectively. The additional lines apart from the original Brandes' algorithm are underlined.

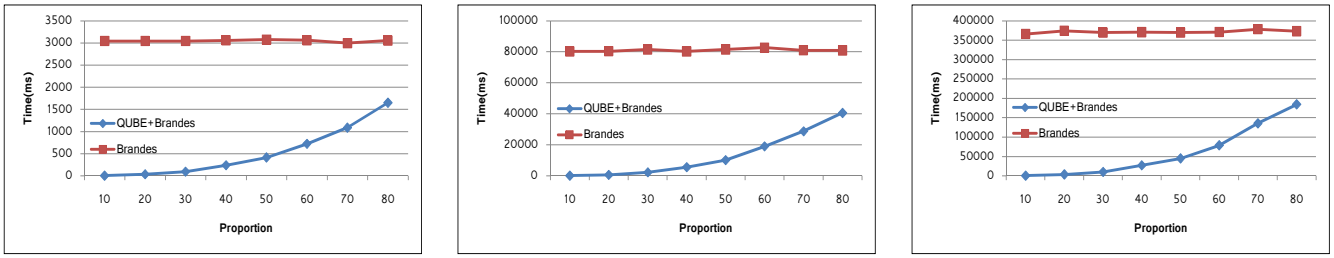
6.2 Experiment Results

To evaluate the proposed algorithm, we measure the betweenness centrality update time using synthetic datasets and real datasets. We synthetically generate connected, undirected, and unweighted graphs of varying numbers of vertices and edges in order to observe the performance with respect to the graph size and proportions. The proportion is computed as $(|MUC_U|/|V|)\cdot 100$, and it indicates the percentage of vertices whose betweenness centralities should be recalculated due to the update of the graph. Therefore, it mainly affects the performance of QUBE. The Erdös-Rényi model [9], the most widely used random graph model, is used to generate synthetic graphs. Each edge in the graph is generated independently of existing edges, with an equal probability of being generated.

Figure 6 shows the running time for updating the betweenness centrality on synthetic graphs of size 1000, 3000, and 5000, respectively. For each graph, we randomly insert 100 edges and take the average value. QUBE significantly reduces the betweenness centrality update time as the proportion decreases. In Figure 6(c), when the proportions are 80, 40, and 10, QUBE enables the original Brandes' algorithm to perform about 2, 13, and 623 times faster, respectively. These results provide a clear evidence that finding MUC_U dramatically improves the performance of updating the betweenness centrality. Regardless of the size of a graph, QUBE makes the original Brandes' algorithm perform much faster.

Besides the update time of the betweenness centrality, we measure the update time of MUCs. Since it is negligible compared to the overall processing time, we do not explicitly present the update time of MUCs.

In order to estimate how QUBE performs in real world graphs, we select various real datasets which are prone to frequent changes. For each real graph, we extract the maximally connected subgraph. In cases of directed real graphs, we convert directed edges into undirected edges. We compare the betweenness centrality update time over 8 different real datasets. The results are shown in Table 2 and Figure 7. Table 2 shows the speed-up achieved by QUBE and the overall statistics of each real dataset. Recall that the proportion is the percentage of vertices in MUC. The low proportion means that there exists a small number of vertices whose betweenness centralities can be changed. Speed-



(a) |V|=1000 (b) |V|=3000 (c) |V|=5000

Figure 6: The betweenness centrality update time on the synthetic data

Algorithm 4: QUBE-BRANDES(MUC_U)

input : MUC_U - Minimum Union Cycle that an updated vertices belong to
SGs - A set of disconnected subgraphs connected by each connection vertices in MUC_U
output : $C[v_i]$ - Updated Betweenness Centrality Array

begin
  for $v_s \in MUC_U$ **do**
    $S \leftarrow$ empty stack ;
    $P[v_i] \leftarrow$ empty list, for all $v_i \in MUC_U$;
    $\sigma[v_i] := 0$, for all $v_i \in MUC_U$; $\sigma[v_s] := 1$;
    $\sigma_t[v_i] := 0$ for all $v_i \in MUC_U$; $d[v_s] := 0$;
    $d[v_i] :=$ -1, for all $v_i \in MUC_U$; $d[v_s] := 0$;
    $Q \leftarrow$ empty queue ;
    enqueue $v_s \rightarrow Q$;
    while Q *not empty* **do**
      dequeue $v_i \leftarrow Q$;
      push $v_i \rightarrow S$;
      for *each neighbor* v_n *of* v_i **do**
        if $d[v_n] < 0$ **then**
          enqueue $v_n \rightarrow Q$;
          $d[v_n] := d[v_i] + 1$;
        if $d[v_n] = d[v_i] + 1$ **then**
          $\sigma[v_n] := \sigma[v_n] + \sigma[v_i]$;
          append $v_i \rightarrow P[v_n]$;
    $\delta[v_i] := 0$, for all $v_i \in MUC_U$;
    while S *not empty* **do**
      pop $v_n \leftarrow S$;
      if v_s, v_n *are connection vertices and* $v_n \neq v_s$ **then**
        $c_t := |V_{G_s}| \cdot |V_{G_n}|$;
        $\sigma_t[v_n] := \sigma_t[v_n] + c_t$;
        $C[v_n] := C[v_n] + c_t$;
      for v_p *in* $P[v_n]$ **do**
        $\delta[v_p] := \delta[v_p] + \frac{\sigma[v_p]}{\sigma[v_n]} \cdot (1 + \delta[v_n])$;
        if v_s *is connection vertex* **then**
          $\sigma_t[v_p] := \sigma_t[v_p] + \sigma_t[v_n] \cdot \frac{\sigma[v_p]}{\sigma[v_n]}$;
          $C[v_p] := C[v_p] + \sigma_t[v_n] \cdot \frac{\sigma[v_p]}{\sigma[v_n]}$;
      if $v_n \neq v_s$ **then**
        $C[v_n] := C[v_n] + \delta[v_n]$;
      if v_s *is connection vertex* **then**
        $C[v_n] := C[v_n] + \delta[v_n] \cdot |V_{G_s}| \cdot 2$;
  for $G_i \in SGs$ **do**
    if G_i *is disconnected* **then**
      $C[v_i] := C[v_i] + |V_{G_i}|^2 - \sum_{l=1}^{n} (|V_{G_i^l}|^2)$;

Table 2: The speed-up on real data

Name	Type	|V|	|E|	Avg. Prop.	Speed-up
Eva[24]	Ownership	4457	4562	6.41	2418.17
Erdos02[a]	Collaboration	5534	8472	28.66	39.57
Erdos972[a]	Collaboration	4680	7030	30.00	34.39
Pgp[4]	Social	4680	24316	42.14	13.09
Epa[b]	Web link	4253	8897	52.25	6.67
Contact [c]	Social	11604	65441	62.60	4.00
Wikivote[19]	Trust	7066	100736	67.73	3.00
CAGrQc[19]	Collaboration	4158	13422	77.92	2.06

[a]http://vlado.fmf.uni-lj.si/pub/networks/data
[b]http://www.cs.cornell.edu/courses/cs685/2002fa/
[c]http://stuff.metafilter.com/infodump/

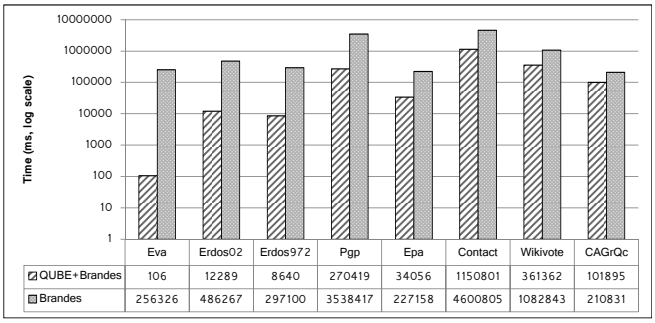


Figure 7: The betweenness centrality update time on real data

up in Table 2 shows how fast the updatable version of the Brandes' algorithm is compared to the original Brandes' algorithm. Table 2 clearly shows that the performance of the updatable version of the Brandes' algorithm increases as the proportion decreases. Figure 7 shows the average betweenness centrality update times measured on real graphs. Note that we use a log scale for the y-axis in Figure 7. To represent the precise update time, a table is included in Figure 7. QUBE makes the original Brandes' algorithm perform about 2 times faster on 'CAGrQc' dataset whose proportion is about 77 and perform about 2418 times faster on 'Eva' dataset whose proportion is about 6. When the proportion is about 30, QUBE makes the original Brandes' algorithm perform about 37 times faster and when the proportion is about 70, QUBE makes the original Brandes' algorithm perform about 3 times faster.

7. CONCLUSIONS

In this paper, we devise a betweenness centrality update theorem and propose an efficient algorithm (QUBE) based on this theorem. QUBE identifies a set of vertices whose betweenness centralities can be changed. QUBE efficiently updates the betweenness centralities based on the betweenness centrality calculated using the vertices in the set only and the number of vertices not in the set. Any existing betweenness centrality algorithm which does not consider the graph update can be changed to an efficient updatable betweenness centrality algorithm with an adoption of QUBE. We implement an updatable version of the Brandes' algorithm by adopting QUBE. For the synthetic graphs whose proportions are 10, the Brandes' algorithm with QUBE is about 557 times faster compared to the original Brandes' algorithm. For the real graphs whose proportions are about 30, the Brandes' algorithm with QUBE performs about 37 times faster than the original Brandes' algorithm. The performance improvement becomes even larger when the proportion decreases.

8. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers. This work was supported in part by WCU (World Class University) program under the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology of Korea (No. R31-30007), and in part by the National Research Foundation of Korea grant funded by the Korea government (MEST) (No. 2011-0000377).

9. REFERENCES

[1] J. Anthonisse and S. M. C. A. A. M. besliskunde. The rush in a directed graph. Technical report, 1971.

[2] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 124–137, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 539–550, Washington, DC, USA, 2006. IEEE Computer Society.

[4] M. Boguñá, R. Pastor-Satorras, A. Diaz-Guilera, and A. Arenas. Models of social networks based on social distance attachment. *Phys. Rev. E*, 70(5):056122, Nov. 2004.

[5] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(1994):163–177, 2001.

[6] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, 2008.

[7] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal Of Bifurcation And Chaos*, 17(7):2303, 2007.

[8] A. del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein-protein complex structures. *Bioinformatics*, 21(8):1311–1315, Apr. 2005.

[9] P. Erdös and A. Rényi. On random graphs, I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[10] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[11] L. C. Freeman, S. P. Borgatti, and D. R. White. Centrality in valued graphs: A measure of betweenness based on network flow. *Social Networks*, 13(2):141 – 154, 1991.

[12] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In J. I. Munro and D. Wagner, editors, *ALENEX*, pages 90–100. SIAM, 2008.

[13] P. Holme. Congestion and centrality in traffic flow on complex networks. *Advances in Complex Systems*, 6(2):163–176, Jan. 2003.

[14] J. D. Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM J. Comput.*, 16:358–366, April 1987.

[15] S. Jin, Z. Huang, Y. Chen, D. G. Chavarria-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, pages 1–7. IEEE, 2010.

[16] T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch. A faster algorithm for minimum cycle basis of graphs. In *ICALP*, pages 846–857, 2004.

[17] E. D. Kolaczyk, D. B. Chua, and M. Barthélemy. Group betweenness and co-betweenness: Inter-related notions of coalition centrality. *Social Networks*, 31(3):190–203, July 2009.

[18] S. Lammer, B. Gehlsen, and D. Helbing. Scaling laws in the spatial structure of urban road networks. *Physica A: Statistical Mechanics and its Applications*, 363(1):89–95, Apr. 2006.

[19] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[20] L. Leydesdorff. Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. *Journal of the American Society for Information Science and Technology*, 58(9):1303–1309, 2009.

[21] Y. Makarychev. Simple linear time approximation algorithm for betweenness. Technical report, 2009.

[22] M. E. J. Newman. A measure of betweenness centrality based on random walks. *Social Networks*, 27(1):39–54, 2005.

[23] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):26113, 2004.

[24] K. Norlen, G. Lucas, M. Gebbie, and J. Chuang. EVA: Extraction, Visualization and Analysis of the Telecommunications and Media Ownership Network, Aug. 2002.

[25] J. W. Pinney and D. R. Westhead. Betweenness-based decomposition methods for social and biological networks. In *Interdisciplinary Statistics and Bioinformatics*, pages 87–90. Leeds University Press, 2006.