Efficient Algorithms for Updating Betweenness Centrality in Fully Dynamic Graphs

Min-Joong Lee

Department of Computer Science, 291 Daehak-ro, Yuseong-gu, KAIST, Daejeon, Korea

Sunghee Choi

Department of Computer Science, 291 Daehak-ro, Yuseong-gu, KAIST, Daejeon, Korea

Chin-Wan Chung*

Division of Web Science and Technology & Department of Computer Science, 291 Daehak-ro, Yuseong-gu, KAIST, Daejeon, Korea

Abstract

Betweenness centrality of a vertex (edge) in a graph is a measure for the relative participation of the vertex (edge) in the shortest paths in the graph. Betweenness centrality is widely used in various areas such as biology, transportation, and social networks. In this paper, we study the update problem of betweenness centrality in fully dynamic graphs. The proposed update algorithm substantially reduces the number of shortest paths which should be re-computed when a graph is changed. In addition, we adapt a community detection algorithm using the proposed algorithm to show how much benefit can be obtained from the proposed algorithm in a practical application. Experimental results on real graphs show that the proposed algorithm efficiently update betweenness centrality and detect communities in a graph.

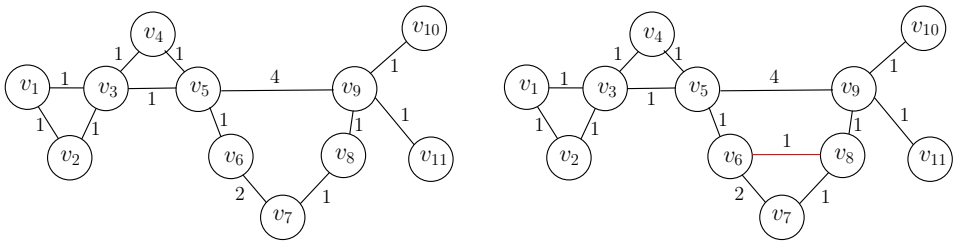
Keywords: Betweenness centrality, Update algorithm, Biconnected component, Dynamic graph, Community detection

1. Introduction

Centralities are one of the essential concepts for the analysis of networks, and betweenness centrality [22] is one of the most prominent measures among several centrality measures. Betweenness centrality of a vertex (edge) in a graph is a measure for the participation of the vertex (edge) in the shortest paths in

*Corresponding author

Email addresses: mjlee@islab.kaist.ac.kr (Min-Joong Lee), sunghee@kaist.edu (Sunghee Choi), chungcw@kaist.edu (Chin-Wan Chung)



(a) G (b) G^U

Figure 1: Graph Update Example (Numbers are edge weights)

the graph. It represents the relative importance of a vertex (edge) in the graph, and allows an understanding of the extent to which a vertex (edge) contributes in the flow of information.

Motivations and Applications. Betweenness centrality is widely used in diverse applications across many different disciplines. It is used to find the most prominent vertices in a complex network, whether they are individuals in a social network [15], elements in a biological network [19], intersections or junctions in a transportation network [20], physical elements in a computer network [18], or documents in a hyper-link network [50]. For example, in a social network, an individual with a higher centrality can be viewed as a more influential individual than an individual with a lower centrality. The importance and applications of betweenness centrality and its several variants are well explained in [13].

Although betweenness centrality problem has been extensively studied in the literature, most of existing studies did not address the problem of updating betweenness centrality. Currently, many real graphs such as social network graphs change over time [39]. In addition, although a graph is static itself, some algorithms iteratively alter the graph to achieve their objectives. For example, a community detection algorithm using betweenness centrality calculates betweenness centralities of edges in a graph iteratively, and removes the edge with the highest centrality until the graph is disconnected. Therefore, the need for updating betweenness centrality is evident. The difficulty on updating betweenness centrality was addressed by several researchers such as Brandes [13], Koschützki et al. [34] and Newman et al. [44]. However, none of them proposed a solution for fully dynamic graphs [1].

Solution Overview. Consider an edge insertion example in Figure 1. We want to figure out which shortest paths are changed due to the insertion of edge $e(v_6, v_8)$. Let G be the original graph and G^U be the updated graph of G. It is trivial to see that the shortest path between v_6 and v_8 is changed. In addition,

[1] A graph is fully dynamic if there are no limits on graph updates, i.e., all insertions and deletions of edges and vertices, and incremental and decremental edge weight changes are possible. In contrast, a graph is partially dynamic if only some of graph updates are possible.

the shortest paths, which include the original shortest path between v_6 and v_8, are changed (e.g., the shortest path between v_5 and v_8). Let us assume that we have an index structure which stores all the shortest paths in the graph. Then we may easily identify such shortest paths. However, some shortest paths in the graph are changed although they did not include the original shortest path between v_6 and v_8. The shortest path between v_1 and v_{11} in G did not include the former shortest path between v_6 and v_8, but it is changed to include the inserted edge $e(v_6, v_8)$. Such shortest paths cannot be easily identified even if we store all the shortest paths in a graph in an index.

However, we observe that there exist vertices (edges) whose betweenness centralities are not changed although the shortest paths, which include the vertices (edges), are changed. In the above example, betweenness centrality of v_3 remains the same although the shortest path between v_1 and v_{11}, which includes v_3, is changed. This is because, only a part (v_5, v_9) of the shortest path not including v_3 is changed due to the update. In Figure 1(b), betweenness centralities of v_5, v_6, v_7, v_8 and v_9 are changed, while those of the other vertices are not changed.

Based on the above observation, we propose a new algorithm for updating betweenness centrality. The key idea of the proposed update algorithm is to perform betweenness centrality computation on a subgraph with vertices and edges whose betweenness centralities should be updated. We first find a subgraph of vertices and edges of which betweenness centralities can be changed due to the graph update. In Figure 1(b), betweenness centralities of vertices and edges in a subgraph induced by a set of vertices $\{v_5,v_6,v_7,v_8,v_9\}$ can be changed while those of other vertices and edges are not changed. Such a subgraph is called the *re-calculation subgraph*. However, computing the new betweenness centralities of vertices and edges of a *re-calculation subgraph* using the *re-calculation subgraph* only is insufficient since the following shortest paths are not yet considered; 1) The shortest paths of each of which the source or the target is not in the subgraph, and 2) the shortest paths which go through the subgraph. We propose a novel approach to identify a *re-calculation subgraph* and to compute amounts of increase in betweenness centralities of vertices and edges of a *re-calculation subgraph* due to the shortest paths 1) and 2) above.

In case of a vertex insertion, although it affects all the vertices and edges in the graph, we can effectively identify the amounts of betweenness centrality changes of all the vertices as follows. First, we separate the vertex to be inserted and its incident edges into a unit vertex, which has only one incident edge, and remaining edges. Second, insert the unit vertex to the graph. The amounts of betweenness centrality changes for inserting the unit vertex can be calculated by priority first traversing the graph from the unit vertex. This is because, all and only the newly created shortest paths in the graph are the single source shortest paths from the unit vertex. Then, insert remaining edges to the graph. The vertex deletion can be handled in a similar way.

Contributions. The contributions of this paper are as follows.

- We devise a novel algorithm for updating betweenness centrality in fully dynamic graphs. To the best of our knowledge, the proposed algorithm is the first work which deals with fully dynamic graphs.
- Our proposed algorithm efficiently updates betweenness centralities without re-computing all-pairs shortest paths in the entire graph. Moreover, this is the only work which does not require any structure to be maintained or pre-processed for updating betweenness centrality.
- Based on the proposed update algorithm, we also devise the highest betweenness centrality edge finding algorithm and a community detection algorithm.
- We conduct experiments on several real graphs. The experimental results show that the proposed algorithm outperforms existing algorithms for all update operations.

Organization. The rest of the paper is organized as follows. In Section 2, related works on betweenness centrality are reviewed. Section 3 consists of three subsections. Formal definitions of betweenness centrality and basic concepts are discussed in Section 3.1. Main ideas for efficiently updating betweenness centralities with theoretical justifications are discussed in Sections 3.2, and efficient community detection algorithm based on the proposed update algorithm is discussed in Section 3.3. We show experimental results in Section 4, and conclude the paper in Section 5.

2. Related Work

The earliest work to define the measure which quantifies the idea of betweenness centrality is introduced by Anthonisse et al. [6] and Freeman [22]. 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. [33] propose a group betweenness measure which can be applied to groups and classes as well as individuals. Freeman et al. [23] extend Freeman's work [22] to introduce a new measure of centrality based on the concept of network flows, which considers both shortest and certain non-shortest paths. Newman [43] proposes a measure of betweenness centrality based on random walks. Brandes [13] 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.

Traditionally, betweenness centrality was determined by computing the number of shortest paths between all pairs, and then summing up pair-dependencies

of all pairs [22]. Brandes [12] points out that the weakness in this approach [22] is computing more information than needed, and he presents a faster algorithm based on aggregating path counts from different source vertices in the network. This is the fastest known algorithm to compute exact betweenness centralities for all the vertices and it requires $O(|V||E|)$ and $O(|V||E| + |V|^2 log|V|)$ time on weighted and unweighted graphs, respectively.

Although the big improvement was made over the very initial betweenness centrality computation algorithm, many researchers argue that the Brandes algorithm is still too costly for large graphs. In order to overcome such limitation, researchers propose approximation algorithms [14, 9, 8, 24, 31]. Brandes et al. [14] propose a heuristic estimation method for betweenness centrality computation. Bader et al. [9] present a parallel approximation algorithm, optimized for scale-free sparse graphs. They also suggest an algorithm [8] to compute betweenness centrality of a single vertex faster than computing the betweenness of all vertices. Geisberger et al. [24] suggest a bisection scaling algorithm for approximating a variant of betweenness centrality. Kang et al. [31] propose a scalable algorithm for MapReduce by using approximation and line graph decomposition. However, those algorithms cannot be used for computing or updating the exact betweenness centrality.

Betweenness centrality is used in diverse applications across many different disciplines. Leydesdorff [40] demonstrates in his research how betweenness centrality is shown to be an indicator of the interdisciplinary of scientific journals. Backes et al. [7] study a problem which approximates a contour to a simple polygon using betweenness centralities of vertices in the graph constructed based on the contour. Jin et al. [30] demonstrate an application of parallel betweenness centrality to detect potentially harmful nodes in an electrical grid, which is an interconnected network for delivering electricity from suppliers to consumers. Holme [28] studies the relationship between betweenness centrality and the density of a traffic model, and Lammer et al. [36] use betweenness centrality in approximating the importance of a road or a junction and investigate the scaling laws associated with urban road networks in Germany. Betweenness centrality is also used in the community detection. Newman et al. [44] propose a divisive community detection technique which iteratively removes edges with the highest betweenness centrality value from the network. Pinney et al. [47] suggest an alternative community detection algorithm in which the network decomposition is based on vertex betweenness centrality instead of edge betweenness centrality. Newman et al. [44] discuss a weakness in the existing algorithms which incur a high computation cost associated with iterative recalculations of all-pair shortest paths when edges are deleted.

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 fully dynamic network environment exists at present.

A straight-forward solution for the update problem is adapting all-pairs shortest paths update algorithms. There were several researches [16, 49] which

aim to maintain all-pairs shortest paths. However, this straight-forward adaptation is infeasible for updating betweenness centrality for two reasons. First, they require enormous space ($O(z^2 \cdot |V||E|)$) to maintain their structure while our algorithm does not maintain any structure. Second, they assume the uniqueness of shortest paths. This assumption is not true for the most of real graphs. Moreover, definition of betweenness centrality is based on the number of shortest paths between two vertices which assumes multiple shortest paths between two vertices.

Recently, following our preliminary work [38] of this paper, a few works [27, 32, 42, 25] address an updating problem in partially dynamic graphs. Green et al. [27] maintain BFS trees each of which is rooted at each vertex in the graph. For each BFS tree, they compute a new partial value of betweenness centrality from the root of the BFS tree. Then they replace the previous partial value of betweenness centrality from the root to the new partial value. Kas et al. [32] adapt the all-pairs shortest paths update algorithm proposed by Reps et al. [48]. They maintain a directed acyclic subgraph for each vertex of the graph containing all the edges that belong to at least one shortest path from the vertex. Then, shortest paths are updated by running a Dijkstra-like procedure. However, Demetrescu et al. [17] report that Reps's algorithm is even worse than the static Dijkstra algorithm for some inputs. Nasre et al. [42] similarly maintain directed acyclic graphs for each vertex, but they provide a new complexity analysis using the maximum number of edges that lie on the shortest paths from a single vertex. However, algorithms in [27, 32, 42] require a huge space to store and an addition cost to maintain BFS tress or directed acyclic graphs while our algorithm does not require any structure to maintain. Moreover, the algorithms presented in [27, 32, 42] are limited to the insertion of an edge or a vertex, and cannot handle the deletion of an edge or a vertex. In our preliminary work [38], which is called QUBE, we propose an efficient algorithm for updating betweenness centrality based on the following idea. While the insertion or deletion of an edge affects many shortest paths in the graph, betweenness centralities of some vertices remain the same although the vertices are included in the affected shortest paths. Based on QUBE, Goel et al. [25] propose an algorithm for updating betweenness centralities when a vertex is deleted or inserted. However, the algorithm in [25] cannot handle the deletion/insertion of a vertex which increases/decreases the number of connected components in the graph.

Improvements over Preliminary Work. The proposed algorithm can handle fully dynamic graphs while the graph update in QUBE is limited to the insertion and deletion of a non-bridge edge. A bridge edge is the edge such that the removal of the edge increases the number of connected components in a graph. Moreover, even for the non-bridge edge update, the proposed algorithm achieves up to 22 times better performance compared to QUBE by reducing

[2] z is the upper-bound of the numbers of historical paths between two vertices.

the number of the shortest paths to be re-computed. Also, this paper discusses betweenness centralities of an edge and a vertex while QUBE only discusses betweenness centrality of a vertex. Computing betweenness centralities of edges has a clear advantage over computing those of vertices (Details are discussed at the end of Section 3.2.2.1.). Based on the proposed update algorithm, we also devise a highest betweenness centrality edge finding algorithm and a community detection algorithm. The experiments are also extended to use real road networks, to compare to a new work [25] following QUBE, and to include new results for finding the highest centrality edge and for finding communities.

3. Algorithm

3.1. Preliminary

Here, we introduce the formal definition of betweenness centrality, and other related concepts.

Definition 1 (Vertex Betweenness Centrality)**.** Betweenness centrality of a vertex $v_j \in V$ is:

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

where $v_i, v_j, v_k \in V$, $v_i \neq v_j, v_j \neq v_k, v_i \neq v_k$, $\sigma_{v_i,v_k}(v_j)$ 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.

Definition 2 (Edge Betweenness Centrality)**.** Betweenness centrality of an edge $e_j \in E$ is:

$$c_e(e_j) = \sum_{v_i, v_k} \frac{\sigma_{v_i,v_k}(e_j)}{\sigma_{v_i,v_k}} \tag{2}$$

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

Definition 3 (Biconnected Component)**.** A graph is biconnected, if the graph remains connected after removing any one vertex. A biconnected component

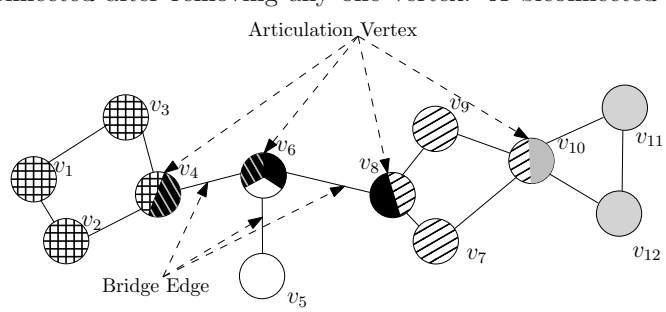


Figure 2: Biconnected Components Example

of a graph is a maximal biconnected subgraph of the graph. Six different fill patterns correspond to biconnected components in Figure 2. A vertex and an edge whose removal increases the number of connected components is called the *articulation vertex* and the *bridge edge*, respectively. For computing biconnected components, a linear time algorithm based on a depth-first search is presented by Hopcroft and Tarjan [29].

Table 1: Summary of Notation

Symbol	Description
$G = (V, E)$	the original graph before the update
$G^U = (V^U, E^U)$	the updated graph after the update
$G_{v_i}^{G^T} = (V_{v_i}, E_{v_i})$	the subgraph connected to $G^T = (V^T, E^T)$ through an articulation vertex $v_i \in G^T$ in G; where $G^T, G_{v_i}^{G^T} \subseteq G$, and $V^T \cap V_{v_i} = \emptyset$
$\sigma_{v_i,v_k}(e_j)$	the number of shortest paths between v_i and v_k that include e_j
σ_{v_i,v_k}	the number of shortest paths between v_i and v_k
$c_e(e_i)/c_v(v_i)$	edge/vertex betweenness centrality of edge e_i/vertex v_i
$c_e^L(e_i)$	local betweenness centrality of edge e_i
$c_e^{T_1}(e_i)$	the total amount of increase in betweenness centrality for e_i due to Type 1 shortest paths
$c_e^{T_2}(e_i)$	the total amount of increase in betweenness centrality for e_i due to Type 2 shortest paths
$sp_G(v_s, v_t)$	the set of shortest paths from v_s to v_t in graph G
$\widehat{c_e}(G)$	upper-bound of betweenness centralities of edges in graph G

3.2. Betweenness Centrality Update

Recomputing betweenness centralities of the vertices and edges in a graph whenever an update occurs is too expensive. In this section, we will discuss an efficient way for updating betweenness centrality. First, we discuss the possible graph update operations in a fully dynamic graph in Section 3.2.1. Then, we provide efficient update solutions for the update operation in Section 3.2.2 and 3.2.3.

3.2.1. Graph Update Operation

The update operations on fully dynamic graphs can be categorized as follows.

Fundamental Operation. The following two fundamental update operations are basic building blocks of all graph update operations.

- F1. Non-bridge edge update
  The non-bridge edge update is the most challenging operation and it changes betweenness centralities of vertices and edges on the shortest paths which include the updated edge, either before or after the update.

We present a novel approach for updating betweenness centralities without performing all-pairs shortest paths computation. This operation will be discussed in Section 3.2.2.

- F2. Unit vertex update
  Let us refer a vertex with only one incident edge as a unit vertex. An unit vertex update can be handled in a relatively straight-forward way, although it changes betweenness centralities of all vertices and edges on the graph. This is because, the shortest paths sourced from the updated unit vertex are the only shortest paths to be generated (removed) due to the unit vertex insertion (deletion). This operation will be discussed in Section 3.2.3.

Applied Operation. The following applied update operations can be handled by combining fundamental update operations.

- A1. Bridge edge update
  This operation can be handled similarly to a unit vertex update operation. Let us assume that the bridge edge $e(v_6, v_8)$ is deleted from the graph in Figure 2. Then, for a vertex or an edge in the disconnected component, which consisting of $v_7, v_8, v_9, v_{10}, v_{11}$ and v_{12}, the deletion can be seen as being the same as the deletion of six unit vertices from v_8; since the number of vertices in the other disconnected component is six. The insertion can be handled in a similar way.

- A2. Several edges update
  First, each bridge edge among the updated edges is handled. Then, non-bridge edges are handled. Note that several non-bridge edges can be handled at once.

- A3. Vertex with multiple incident edges update
  This can be handled in two steps. We construct the unit vertex by selecting arbitrarily one edge among the incident edges of the vertex to be inserted, and insert the constructed unit vertex. Then, we update remaining edges. The process is reversed in the case of deletion. Edges excluding any arbitrary edge are deleted first and then remaining unit vertex is deleted.

There are three types of graph updates which occur in a fully dynamic graph; 1) insertion and deletion of an edge, 2) insertion and deletion of a vertex, and 3) decrease and increase of an edge weight. 1) and 2) can be directly handled by F1 and A1, and F2 and A3, respectively. 3) is the same as the insertion of the edge with new weight by rewriting the new edge weight and considering the edge as the new edge. Thus 3) can be handled by F1 and A1. And applied operations A1, A2, and A3 can be handled by combining fundamental operations F1 and F2 as mentioned above. Therefore, all possible graph updates can be handled by adapting two fundamental update operations, and we will mainly discuss two

fundamental update operations in the rest of this paper. From now on, we simply refer a non-bridge edge to an edge and a unit vertex to a vertex.

3.2.2. Non-bridge Edge Update

The overall flow of updating betweenness centralities for a non-bridge edge update is as follows. First, we identify a *re-calculation subgraph* which includes all vertices and edges of which betweenness centralities are changed due to the update (Section 3.2.2.1). Then, we perform a normal betweenness centrality computation using only the *re-calculation subgraph* to compute local betweenness centralities[3]. Finally, we calculate the amount of increase in betweenness centralities with respect to the shortest paths which are not yet considered (e.g., a shortest path from and to vertices which are not in the *re-calculation subgraph*), and add the amount of increase to local betweenness centrality to compute global betweenness centrality[4] (Section 3.2.2.2).

3.2.2.1 Re-calculation Subgraph

To efficiently calculate the new betweenness centralities due to the edge update, we first find a subgraph of the entire graph called the *re-calculation subgraph*. The *re-calculation subgraph* must satisfy the following two conditions.

1. The *re-calculation subgraph* should include all the vertices and edges of which betweenness centralities are changed due to the update.
2. All shortest paths between any two vertices in the *re-calculation subgraph* should not include vertices and edges that are not in the *re-calculation subgraph*.

In the following subsections, we will discuss a tight *re-calculation subgraph* which is the smallest subgraph which satisfies the conditions above. However, the computation of such a tight *re-calculation subgraph* is computationally difficult. Therefore, we also present a bit loose *re-calculation subgraph* which can be easily computed.

Affected Subgraph. Before we introduce the *re-calculation subgraph*, we first introduce an *affected subgraph* denoted by G^A; which consists of all the edges and vertices that lie on the shortest paths including the updated edge, either before or after the update. The *affected subgraph*, G^A; with respect to the updated edge $e_u(v_i, v_{i+1})$, can be computed as follows.

1. Let $G = (V, E)$ be a graph and the updated edge denoted by e_u is not a bridge edge, and G^+ G^-, and G^U are defined as follows.

[3]Local betweenness centrality is computed by considering only the shortest paths in the *re-calculation subgraph*

[4]Global betweenness centrality counts all the shortest paths in the entire graph.

$$\text{If } e_u \text{ is inserted,} \quad \begin{cases} G^U = & G^+ = (V^+, E^+) = (V, E \cup \{e_u\}) \\ & G^- = (V^-, E^-) = (V, E) \end{cases}$$

$$\text{If } e_u \text{ is deleted,} \quad \begin{cases} & G^+ = (V^+, E^+) = (V, E) \\ G^U = & G^- = (V^-, E^-) = (V, E \backslash \{e_u\}) \end{cases}$$

2. Let VP^A be the set of vertex pairs such that each pair consists of a source vertex and an end vertex of a shortest path that includes the updated edge $e_u(v_i, v_{i+1})$ on graph G^+. Formally, $VP^A = \{(x, y) \in V^+ \times V^+ | \exists P \in sp_{G^+}(x, y), \langle v_i, v_{i+1} \rangle \subseteq$ [5] $P\}$
3. Let V^A be the set of all the vertices in changed shortest paths and original shortest paths of the changed shortest paths due to the update edge e_u. Formally, $V^A = \{P^+ \cup$ [6] $P^- | P^+ \in sp_{G^+}(x, y), P^- \in sp_{G^-}(x, y), (x, y) \in VP^A\}$
4. Let $G^A = (V^A, E^A)$ be a subgraph of G induced by V^A. Then, vertices and edges only in G^A can be affected by the update.

Tight Subgraph. A tight *re-calculation subgraph* G^T can be computed based on the *affected subgraph* G^A as follows.

1. Let $G^A = (V^A, E^A)$ be an *affected subgraph* with respect to the updated edge $e_u(v_i, v_{i+1})$.
2. Let $P_{e_u}(x, y) = \langle x = v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{n-1}, y = v_n \rangle$ be a shortest path from x to y which includes the updated edge $e_u(v_i, v_{i+1})$ where $x, y \in V^A$. Two partial paths of $P_{e_u}(x, y)$, $P^{\leftarrow}_{e_u}(x, y)$ and $P^{\rightarrow}_{e_u}(x, y)$, are defined as follows.
   (a) Let A be the set of articulation vertices whose deletion make $P^{\leftarrow}_{e_u}(x, y)$ disconnected.
   (b) If $\exists v_j \in A$ such that $1 < z \leq j \leq i$ for all $v_z \in A$, $P^{\leftarrow}_{e_u}(x, y) = \langle v_1, \ldots, v_j \rangle$, otherwise $P^{\leftarrow}_{e_u}(x, y) = \emptyset$.
   (c) If $\exists v_k \in A$ such that $i + 1 \leq k \leq z < n$ for all $v_z \in A$, $P^{\rightarrow}_{e_u}(x, y) = \langle v_k, \ldots, v_n \rangle$, otherwise $P^{\rightarrow}_{e_u}(x, y) = \emptyset$.
3. Set $E^{\leftarrow} = \bigcup_{x,y \in V^A} \{e | e(s, t) \in E^A, \langle s, t \rangle \subseteq P^{\leftarrow}_{e_u}(x, y)\}$ and $E^{\rightarrow} = \bigcup_{x,y \in V^A} \{e | e(s, t) \in E^A, \langle s, t \rangle \subseteq P^{\rightarrow}_{e_u}(x, y)\}$. Then, betweenness centralities of edges in $E^{\leftarrow} \cup E^{\rightarrow}$ are not changed.
4. Let $G^T = (V^T, E^T)$ be a subgraph of G^A induced by $E^T = E^A \backslash (E^{\leftarrow} \cup E^{\rightarrow})$.

[5] We say $P_1 \subseteq P_2$, if P_1 is a partial path of P_2.
[6] Without loss of generality, we can use a sequence of vertices as the set of vertices.

Claim 1. Betweenness centralities of vertices and edges in G^T are only changed by the updated edge e_u.

Proof. For any shortest path $\langle v_1, ..., v_j, ..., v_i, v_{i+1}, ..., v_k, ..., v_n \rangle$ where v_j and v_k are articulation vertices which make the path disconnected, even if $e_u(v_i, v_{i+1})$ is deleted from the path; 1) There always exists a new shortest path since $e_u(v_i, v_{i+1})$ is not a bridge edge. 2) The new shortest path still includes v_j and v_k since v_j and v_k are articulation vertices which make the path disconnected. Therefore, partial paths from v_1 to v_j ($E^{\leftarrow}$) and from v_k to v_n ($E^{\rightarrow}$) remain the same after the edge update, and betweenness centralities of only vertices and edges in G^T are changed by the deletion of edge e_u. The insertion case is similar to the deletion case. □

However, to construct G^A from VP^A and G^T from G^A, all-pairs shortest path computation is required on G^+ to find VP^A. This requires the same cost as computing betweenness centralities from the scratch on G^U. Therefore, constructing the tight *re-calculation subgraph* G^T is computationally meaningless.

A Bit Loose Subgraph. We propose a new *re-calculation subgraph* $G^{T'}$ which is a bit larger than the tight *re-calculation subgraph* G^T presented above but easily obtainable. The key idea is using the reachability condition instead of the shortest path condition when we compute *Affected Subgraph*. The *re-calculation subgraph* $G^{T'}$ is computed as follows.

1. G, G^+, and G^- are defined the same as before. Let $e_u(v_i, v_{i+1})$ be the updated non-bridge edge.
2. Construct $V^{A'}$ as follows by relaxing the shortest path condition to the reachability condition.

$$V^{A'} = \{v \in V^+ | v \rightsquigarrow {}^7 v_i\} \cup \{v \in V^+ | v_{i+1} \rightsquigarrow v\}$$

3. Let $G^{A'} = (V^{A'}, E^{A'})$ be a subgraph of G induced by $V^{A'}$.
4. $G^{T'}$ is constructed similar to G^T, except for the following differences.
   (a) $G^{T'}$ is constructed based on $G^{A'}$ instead of G^A.
   (b) Let $P'_{e_u}(x, y) = \langle x = v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{n-1}, y = v_n \rangle$ be any path from x to y which includes $e_u(v_i, v_{i+1})$ where $x, y \in V^{A'}$.
5. $P'^{\leftarrow}_{e_u}(x, y)$ and $P'^{\rightarrow}_{e_u}(x, y)$ are similarly defined using $P'_{e_u}(x, y)$. $E'^{\leftarrow}$ and $E'^{\rightarrow}$ are also similarly defined using $P'^{\leftarrow}_{e_u}(x, y)$ and $P'^{\rightarrow}_{e_u}(x, y)$, respectively.
6. Let $G^{T'} = (V^{T'}, E^{T'})$ be a subgraph of $G^{A'}$ induced by $E^{T'} = E^{A'} \backslash (E'^{\leftarrow} \cup E'^{\rightarrow})$.

[7] $s \rightsquigarrow t$ is true *iff* t is reachable from s

Claim 2. $G^{T'}$ includes all the vertices and edges of which betweenness centralities should be updated.

Proof. Let us show $G^T = (V^T, E^T)$ is a subgraph of $G^{T'} = (V^{T'}, E^{T'})$. It is sufficient to show that $E^{T'} \supseteq E^T$ since $G^{T'}$ and G^T are induced subgraphs by $E^{T'}$ and E^T, respectively. Let $E^{N'} = E'^{\leftarrow} \cup E'^{\rightarrow}$ and $E^N = E^{\leftarrow} \cup E^{\rightarrow}$.

$$
\begin{aligned}
\text{By definition,} \quad & E^A \cap E^{N'} = E^N \\
\Rightarrow \quad & E^A \setminus E^{N'} = E^A \setminus E^N \\
\Leftrightarrow \quad & E^A \setminus E^{N'} = E^T \Leftrightarrow E^A \cap E^{N'^c} = E^T \\
\Leftrightarrow \quad & \left(E^{A'} \cap E^A\right) \cap \left(E^{N'^c} \cap E^{N^c}\right) = E^T; \\
& \text{since } E^A \subseteq E^{A'}, E^N \subseteq E^{N'} \Leftrightarrow E^{N'^c} \subseteq E^{N^c} \\
\Leftrightarrow \quad & \left(E^{A'} \cap E^{N'^c}\right) \cap \left(E^A \cap E^{N^c}\right) = E^T \\
\Leftrightarrow \quad & \left(E^{A'} \setminus E^{N'}\right) \cap \left(E^A \setminus E^N\right) = E^T \\
\Leftrightarrow \quad & E^{T'} \cap E^T = E^T \Leftrightarrow E^{T'} \supseteq E^T
\end{aligned}
$$

□

Claim 3. All shortest paths between any two vertices in $V^{T'}$ only include vertices and edges in $G^{T'} = (V^{T'}, E^{T'})$.

Proof. This can be proved by contradiction.

Assume $\exists P \in sp_{G^U}(s,t)$ such that $v \in P$ and $v \notin V^{T'}$. By definition, $E^{T'}$ has all edges such that the edges are *edge reachable* to and from the updated edge without crossing any articulation vertex which makes the path disconnected. Thus, from $s \in V^{T'}$, we should cross an articulation vertex which makes the path disconnected to include $v \notin V^{T'}$ and we should cross the same articulation vertex again to reach $t \in V^{T'}$. Then, P includes at least one articulation vertex twice, and this makes a cycle in P. Therefore, P is not a shortest path and this contradicts the assumption. □

Surprisingly, the *re-calculation subgraph* $G^{T'}$; with respect to the updated edge e_u, is exactly the same as the maximal biconnected graph which includes e_u in G^+. Therefore, we adapt a method [29] for finding biconnected components in a graph to find a maximal biconnected subgraph which includes e_u only.

Edge and Vertex Betweenness Centrality. Betweenness centrality of a vertex can be computed from betweenness centralities of edges that are incident to the vertex as follows.

$$
c_v(v) = \sum_{e \in \Gamma(v)} \frac{c_e(e)}{2} - (|V| - 1) \tag{3}
$$

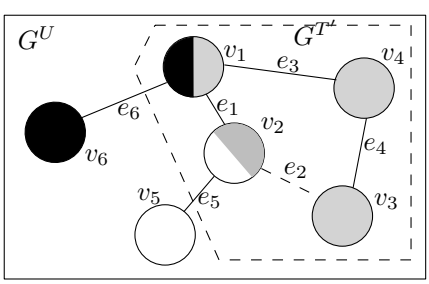
where $\Gamma(v)$ is a set of edges that incident to v. By Definition 1, the source and target vertices of a shortest path are considered as not participating in the shortest path. This is why we subtract $|V| - 1$ in Equation 3.

On the other hand, edge betweenness centrality cannot be computed from vertex betweenness centrality in a straight forward way. Maintaining and updating edge betweenness centrality rather than maintaining and updating vertex betweenness centrality has the following strong advantage. Let us explain with Figure 3. $G^{T'}$ is the *re-calculation subgraph* with respect to the e_u edge insertion. Betweenness centrality of the articulation vertex v_3 in $G^{T'}$ is associated with only one of two shortest paths whose sources and targets are both not in $G^{T'}$ such as the shortest paths from v_5 to v_6. Therefore, we should extend $G^{T'}$ to the dashed line area, depicted as $\hat{G}$. Alternatively, for each articulation vertex, we should separately store each amount of betweenness centrality with respect to each biconnected component including the articulation vertex. The former approach degrades the performance due to a bigger *re-calculation subgraph*. The latter approach requires a lot of counterintuitive reasonings to update betweenness centrality, and a complex structure to maintain betweenness centralities of articulation vertices. Therefore, from now on, we focus on updating edge betweenness centrality.

3.2.2.2 Recomputation Using Re-calculation Subgraph

In the previous subsection, we discussed a way to construct the *re-calculation subgraph* $G^{T'}$. We guarantee that betweenness centralities of vertices and edges not in $G^{T'}$ are not changed. Yet, calculating betweenness centrality of edge $e \in G^{T'}$ using only the vertices and edges in $G^{T'}$ is insufficient. This is because, some shortest paths are not yet considered, although those paths pass edges in $G^{T'}$.

The following shortest paths are not yet considered in local betweenness centrality of the edge e_1 in Figure 3. 1) The shortest paths of which source or target is not in $G^{T'}$ such as $\langle v_5, v_2, v_1, v_4 \rangle$; 2) The shortest paths that pass though $G^{T'}$ such that both source and target vertices of the shortest paths are not in $G^{T'}$ such as $\langle v_5, v_2, v_1, v_6 \rangle$. We refer the shortest paths in 1) and 2) to Type 1 and Type 2 shortest paths, respectively.



i	$c_v(v_i)$ (global)	$c_v^L(v_i)$ (local)	$c_e(e_i)$ (global)	$c_e^L(e_i)$ (local)
1	10	1	12	4
2	10	1	8	4
3	2	1	8	4
4	2	1	6	4
5	0	X	10	X
6	0	X	10	X

Figure 3: Local and Global Betweenness Centralities

Now, we explain how to restore the exact global betweenness centrality from

local betweenness centrality without performing the expensive all-pairs shortest path computation in G^U. Let $c_e^L(e_i)$ ($c_v^L(v_i)$) be local betweenness centrality of e_i (v_i) which only considers the *re-calculation subgraph* $G^{T'}$, and $c_e(e_i)$ ($c_v(v_i)$) be global betweenness centrality of e_i (v_i) which considers the entire updated graph G^U. There can be several articulation vertices in $G^{T'}$ each of which connects one or more biconnected components. The subgraph connected to $G^{T'}$ through an articulation vertex v_j is referred to as $G_{v_j}^{G^{T'}} = (V_{v_j}, E_{v_j})$. That is, $G_{v_j}^{G^{T'}}$ will be disconnected from $G^{T'}$, if we remove v_j.

Lemma 1. Let $v_j \in V^{T'}$ be an articulation vertex which connects $G_{v_j}^{G^{T'}}$ to $G^{T'}$ in G^U. Then each shortest path $P' \in sp_{G^{T'}}(v_j, v_t)$ is a partial path of some shortest path $P \in sp_{G^U}(v_s, v_t)$ where $v_s \in V_{v_j}$, and $v_t \in V^{T'}$.

Proof: All shortest paths from $v_s \in V_{v_j}$ to $v_t \in V^{T'}$ go through an articulation vertex v_j. A partial path of a shortest path is also a shortest path. Therefore, each shortest path $P' \in sp_{G^{T'}}(v_j, v_t)$ is a partial path of some shortest path $P \in sp_{G_U}(v_s, v_t)$. □

Lemma 1 allows us to calculate the amount of increase in betweenness centrality due to Type 1 shortest paths each of which either the source or the target is in V_{v_j}. Such the increase in betweenness centrality for e_i is denoted by $c_e^{T_1^{V_{v_j}}}(e_i)$, and calculated as follows.

$$c_e^{T_1^{V_{v_j}}}(e_i) = \sum_{v_t \in V^{T'}} \frac{|V_{v_j}| \cdot \sigma_{v_j, v_t}(e_i)}{\sigma_{v_j, v_t}} \tag{4}$$

where $v_j \in V^{T'}$ is an articulation vertex which connects $G_{v_j}^{G^{T'}}$ to $G^{T'}$ in G^U.

The total amount of increase in betweenness centrality for e_i due to Type 1 shortest paths, denoted by $c_e^{T_1}(e_i)$, and can be computed as follows.

$$c_e^{T_1}(e_i) = \sum_{v_j \in V^{T'}} c_e^{T_1^{V_{v_j}}}(e_i) \tag{5}$$

where $v_j \in V^{T'}$ and v_j is an articulation vertex.

Lemma 2. Let $v_j, v_k \in V^{T'}$ be articulation vertices which connect $G_{v_j}^{G^{T'}}$ and $G_{v_k}^{G^{T'}}$ to $G^{T'}$ in G^U, respectively. Then each shortest path $P' \in sp_{G^{T'}}(v_j, v_k)$ is a partial path of some shortest path $P \in sp_{G^U}(v_s, v_t)$ where $v_s \in V_{v_j}$, $v_t \in V_{v_k}$.

Proof: All shortest paths from $v_s \in V_{v_j}$ to $v_t \in V_{v_k}$ go through articulation vertices v_j and v_k. A partial path from v_j to v_k of the shortest path from v_s to v_t is also a shortest path. Therefore, each shortest path $P' \in sp_{G^{T'}}(v_j, v_k)$ is a partial path of some shortest path $P \in sp_{G^U}(v_s, v_t)$. □

Lemma 2 allows us to calculate the amount of increase in betweenness centrality due to Type 2 shortest paths of which source and target are in V_{v_j} and

V_{v_k}, respectively. Such the increase in betweenness centrality for e_i is denoted by $c_e^{T_2^{V_{v_j},V_{v_k}}}(e_i)$ and calculated as follows.

$$c_e^{T_2^{V_{v_j},V_{v_k}}}(e_i) = \frac{|V_{v_j}| \cdot |V_{v_k}| \cdot \sigma_{v_j,v_k}(e_i)}{\sigma_{v_j,v_k}} \tag{6}$$

where $v_j, v_k \in V^{T'}$, v_j, v_k are articulation vertices, and $v_j \neq v_k$.

The total amount of increase in betweenness centrality for e_i due to Type 2 shortest paths, denoted by $c_e^{T_2}(e_i)$, and can be computed as follows.

$$c_e^{T_2}(e_i) = \sum_{v_j,v_k \in V^{T'}} c_e^{T_2^{V_{v_j},V_{v_k}}}(e_i) \tag{7}$$

where $v_j, v_k \in V^{T'}$, v_j, v_k are articulation vertices, and $v_j \neq v_k$

Corollary 1. *(Betweenness Centrality Update Corollary on Edge Update)* By Lemma 1 and Lemma 2, we can compute global betweenness centrality of edge e_i, $c_e(e_i)$ as follows.

$$c_e(e_i) = c_e^L(e_i) + c_e^{T_1}(e_i) + c_e^{T_2}(e_i) \tag{8}$$

where $c_e^L(e_i)$ is local betweenness centrality of e_i, and $e_e^{T_1}(e_i)$ and $c_e^{T_2}(e_i)$ are the amounts of increases in betweenness centrality of e_i due to Type 1 shortest paths and Type 2 shortest paths, respectively.

By Corollary 1, we can compute global betweenness centrality using local betweenness centrality and the number of vertices in subgraphs, each of which is connected to $G^{T'}$ through an articulation vertex in $G^{T'}$, without performing all-pairs shortest paths computation on all the vertices in a graph G^U.

Example 1. Table 2 shows local betweenness centralities, the amounts of increases of betweenness centralities for Type 1 and Type 2 shortest paths, and global betweenness centralities for edges in $G^{T'}$ depicted in Figure 3. Due to space limitations, we do not differentiate a path from v_s to v_t and a path from v_t to v_s. For example, $c_e^{T_2^{V_{v_1},V_{v_3}}}(e_1)$ due to the shortest path $\langle v_1, v_3 \rangle$ is 30 and due to the shortest path $\langle v_3, v_1 \rangle$ is 30. However, we denote the shortest path $\langle v_1, v_3 \rangle$ only, and doubled the value (60).

For e_2, local betweenness centrality, $c_e^L(e_2)$, is three. The amount of increase due to Type 1 shortest paths of which source or target is in G_{v_1}, is 15. This amount is calculated as follows. 1) All the shortest paths, whose sources are in V_{v_1} and targets are v_4, include a partial path $\langle v_1, v_4 \rangle$; and 2) a half of the shortest paths, whose sources are in V_{v_1} and targets are v_2, include a partial path $\langle v_1, v_4, v_2 \rangle$. Therefore, the amount for 1) is $5 \cdot 2$, and for 2) is $5 \cdot 2/2$ since $|V_{v_1}| = 5$. In total, $c_e^{T_1^{V_{v_1}}}(e_2) = 15$. Similarly, $c_e^{T_1^{V_{v_2}}}(e_2) = 4$.

Table 2: Example of Updating Betweenness Centralities of Graph in Figure 3

	$sp_{G^{T'}}(v_s, v_t)$	Path	e_1	e_2	e_3	e_4	e_u
$c_e^L(e_i)$			3	3	3	3	2
$c_e^{T_1^{V_{v_1}}}(e_i)$	$sp_{G^{T'}}(v_1, v_3)$	$\langle v_1, v_3 \rangle$	10				
	$sp_{G^{T'}}(v_1, v_4)$	$\langle v_1, v_4 \rangle$		10			
	$sp_{G^{T'}}(v_1, v_2)$	$\langle v_1, v_3, v_2 \rangle$	10/2		10/2		
		$\langle v_1, v_4, v_2 \rangle$		10/2		10/2	
$c_e^{T_1^{V_{v_2}}}(e_i)$	$sp_{G^{T'}}(v_2, v_3)$	$\langle v_2, v_3 \rangle$			8		
	$sp_{G^{T'}}(v_2, v_4)$	$\langle v_2, v_4 \rangle$				8	
	$sp_{G^{T'}}(v_2, v_1)$	$\langle v_2, v_3, v_1 \rangle$	8/2		8/2		
		$\langle v_2, v_4, v_1 \rangle$		8/2		8/2	
$c_e^{T_1^{V_{v_3}}}(e_i)$	$sp_{G^{T'}}(v_3, v_1)$	$\langle v_3, v_1 \rangle$	12				
	$sp_{G^{T'}}(v_3, v_2)$	$\langle v_3, v_2 \rangle$			12		
	$sp_{G^{T'}}(v_3, v_4)$	$\langle v_3, v_4 \rangle$					12
$c_e^{T_2^{V_{v_1}, V_{v_2}}}(e_i)$	$sp_{G^{T'}}(v_1, v_2)$	$\langle v_1, v_3, v_2 \rangle$	40/2		40/2		
		$\langle v_1, v_4, v_2 \rangle$		40/2		40/2	
$c_e^{T_2^{V_{v_2}, V_{v_3}}}(e_i)$	$sp_{G^{T'}}(v_2, v_3)$	$\langle v_2, v_3 \rangle$			48		
$c_e^{T_2^{V_{v_1}, V_{v_3}}}(e_i)$	$sp_{G^{T'}}(v_1, v_3)$	$\langle v_1, v_3 \rangle$	60				
$c_e(e_i)$			114	42	100	40	14

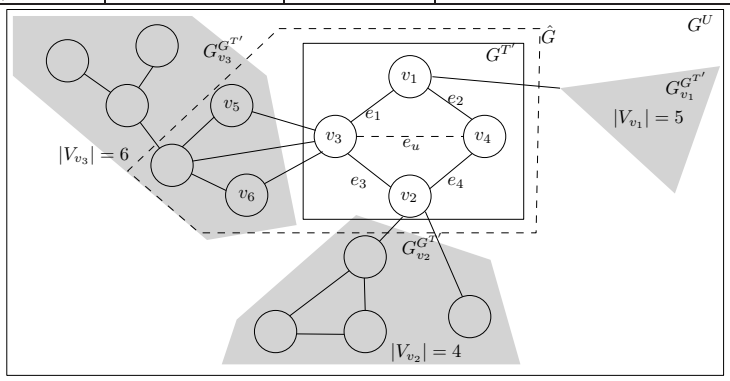


Table 3: Running Example of Betweenness Centrality Update (vertices and edges in $G_{v_1}^{G^{T'}}$ are omitted.)

There are also Type 2 shortest paths which pass e_2. Type 2 shortest paths, whose sources are in V_{v_1} and targets are in V_{v_2}, include shortest paths from v_1 to v_2. However, only a half of the shortest paths from v_1 to v_2 include e_2. Therefore, the amount of increase is $(4 \cdot 5) \cdot 2/2$ since $|V_{v_1}| = 5$ and $|V_{v_2}| = 4$. Finally, global betweenness centrality is calculated as $3 + 15 + 4 + 20 = 42$.

As we mentioned in Section 3.2.2.1, betweenness centrality of a vertex is

easily obtainable from betweenness centralities of the incident edges of the vertex (Equation 3). For example, in Figure 3, $c_v(v_4)$ is $\frac{42+40+14}{2} - 18 = 30$. Betweenness centralities of v_1, v_2, and v_3 are 130, 118, and 174, respectively.

Time and Space Complexity.

$O(|V| + |E|)$ time is required to identify the biconnected component which includes the updated edge, and compute the number of vertices in the subgraphs that are connected to the biconnected component. If we use the Brandes algorithm for local betweenness centrality computation (which is the dominant computation), our algorithm requires $O(|V^{T'}||E^{T'}|)$ and $O(|V^{T'}||E^{T'}| + |V^{T'}|^2 log|V^{T'}|)$ time on weighted and unweighted graphs for the local betweenness centrality computation, respectively. In total, our update algorithm requires $O(|V^{T'}||E^{T'}| + |V| + |E|)$ and $O(|V^{T'}||E^{T'}| + |V^{T'}|^2 log|V^{T'}| + |V| + |E|)$ time on weighted and unweighted graphs for updating betweenness centrality, respectively. Also, the space complexity of our algorithm is $O(|V^{T'}||E^{T'}|)$. These are easily obtainable by replacing $G = (V, E)$ with $G^{T'} = (V^{T'}, E^{T'})$ from the time and space complexities of the Brandes algorithm [12]. Note that, $|V^{T'}| \leq |V|$ and $|E^{T'}| \leq |E|$. The graphs used in the experiments have 20 to 3.5 times smaller $|V^{T'}|$ than $|V|$.

3.2.3. Unit Vertex Update

Let us denote *one-sided pair dependency* of vertex v_s on *an intermediary edge* e_i is the ratio of shortest paths from v_s to other vertices in a graph that e_i lies on. The *one-sided pair dependencies* of a vertex on other *vertices* are defined by Brandes et al. [12][8]. We adapt it to calculate the *one-sided pair dependencies* of a vertex on *edges*. Let $G^U = (V^U, E^U)$ be the updated graph. The *one-sided pair dependency* of the updated unit vertex $v_u \in V^U$ on edge $e_i(v_1, v_2) \in E^U$ is computed as follows.

1. Let us denote the pair dependency of a vertex pair (v_u, v_t) on an intermediary vertex v_i as
$$\delta_{v_u,v_t}(v_i) = \frac{\sigma_{v_u,v_t}(v_i)}{\sigma_{v_u,v_t}}$$
2. Then, the *one-sided pair dependency* of a vertex v_u on a vertex v_i can be defined as
$$\delta_{v_u,\bullet}(v_i) = \sum_{v_t \in V^U} \delta_{v_u,v_t}(v_i) \tag{9}$$
3. Similarly, the pair dependency of a vertex pair (v_u, v_t) on an intermediary edge $e_i(v_1, v_2)$ can be defined as follows.
$$\delta_{v_u,v_t}(e_i(v_1, v_2)) = \begin{cases} \frac{\sigma_{v_u,v_1}}{\sigma_{v_u,v_2}}, & \text{if } v_t = v_2 \\ \frac{\sigma_{v_u,v_1}}{\sigma_{v_u,v_2}} \cdot \frac{\sigma_{v_u,v_t}(v_2)}{\sigma_{v_u,v_t}}, & \text{if } v_t \neq v_2 \end{cases} \tag{10}$$

[8] In [12], *one-sided pair dependency* is simply called *dependency*.

4. Therefore, the *one-sided pair dependency* of a vertex v_u on an edge $e_i(v_1, v_2)$ is denoted by $\delta_{v_u,\bullet}(e_i(v_1, v_2))$ and computed as follows.

$$\begin{aligned}
\delta_{v_u,\bullet}(e_i(v_1, v_2)) = & \sum_{v_t \in V^U} \delta_{v_u,v_t}(e_i(v_1, v_2)) \\
= & \frac{\sigma_{v_s,v_1}}{\sigma_{v_s,v_2}} + \sum_{v_t \in V^U, v_t \neq v_2} \frac{\sigma_{v_s,v_1}}{\sigma_{v_s,v_2}} \cdot \frac{\sigma_{v_s,v_t}(v_2)}{\sigma_{v_s,v_t}} \quad by\ (10) \\
= & \sum_{v_t \in V^U, v_t \neq v_2} \frac{\sigma_{v_s,v_1}}{\sigma_{v_s,v_2}} \cdot (1 + \delta_{v_u,\bullet}(v_2)) \quad by\ (9) \quad (11)
\end{aligned}$$

Lemma 3. When vertex v_u is inserted/deleted, we can calculate the updated betweenness centrality of edge $e_i \in E^U$ by adding/subtracting the *one-sided pair dependency* of the inserted/deleted vertex $v_u \in V^U$ on edge e_i.

Proof : This is obvious because all the shortest paths to be generated/deleted due to the inserted/deleted unit vertex v_u are from v_u. The *one-sided pair dependency* of vertex v_u on an intermediary edge e_i is the ratio of shortest paths from v_u to other vertices in a graph that e_i lies on. Therefore, we can calculate the updated betweenness centralities of edge e_i by adding/subtracting the *one-sided pair dependency* of the inserted/deleted vertex $v_u \in V^U$ on edge e_i.

Corollary 2. *(Betweenness Centrality Update Corollary on Vertex Update)* By Lemma 3 and Equation 11, we can calculate a updated betweenness centrality of e_i, $c_e(e_i)$, from the former betweenness centrality of e_i, denoted by $c_e^f(e_i)$, due to the inserted/deleted vertex v_u as follows.

$$c_e(e_i) = \begin{cases} c_e^f(e_i) + \delta_{v_u,\bullet}(e_i), \text{ if } v_u \text{ is inserted.} \\ c_e^f(e_i) - \delta_{v_u,\bullet}(e_i), \text{ if } v_u \text{ is deleted.} \end{cases}$$

Time and Space Complexity.

The *one-sided pair dependencies* of a vertex on all edges in a graph can be computed by performing BFS for unweighted graph in time $O(|E|)$, and performing the Dijkstra algorithm for weighted graph in time $O(|E| + |V|log|V|)$ using Fibonacci heap [21]. The space complexity is $O(|V| + |E|)$, and is also directly inherited from the space complexities of BFS and the Dijkstra algorithm.

3.2.4. Betweenness Centrality Update Algorithm

Algorithm 1 shows a complete procedure to update betweenness centrality when a graph is updated. Algorithm 1 gets the original graph G, the original betweenness centrality array $c_e[]$, and sets of edges and vertices to be updated as inputs. Algorithm 1 outputs the updated betweenness centrality array $c_e[]$. For the simplicity, a vertex to be updated with multiple incident edges is already

separated into a unit vertex and remaining edges before the process (refer Section 3.2.1). Through Line 4 to Line 7 the amount of change due to the updated vertices is calculated. $\delta^G_{v,\bullet}(e_i)$ is the *one-sided pair dependency* of a vertex v on an edge e_i in the graph G. As mentioned in Section 3.2.1, the insertion of a bridge edge can be seen as being the same as an insertion of multiple unit vertices. Therefore, for edges and vertices in G_s (G_t), the insertion can be seen as being the same as an insertion of $|V_t|$ ($|V_s|$) number of unit vertices to v_s (v_t). Through Line 8 to Line 11 (Line 13 to Line 16), the amount of increase (decrease) due to the inserted (deleted) bridge edges is updated. Betweenness centrality of the inserted bridge itself is computed in Line 12.

Non-bridge edges are handled through Line 17 to Line 29. Several edges belong to a *re-calculation subgraph* are handled simultaneously. First, we find a set of *re-calculation subgraphs*, each of which is a maximal biconnected subgraph including at least one updated edge through Line 17 to Line 21. Then, for each *re-calculation subgraph*, local betweenness centralities are computed in Line 23, and the amounts of increases of betweenness centralities due to Type 1 and Type 2 shortest paths are computed through Line 25 to Line 29. Note that we can use any existing betweenness centrality computation algorithm for computing local betweenness centralities in Line 23. Finally, the algorithm outputs the updated betweenness centrality array c_e.

Using Brandes Algorithm for Computing Local Betweenness Centrality. This subsection explains an efficient implementation of our proposed algorithm using the Brandes algorithm [12], which is known to be the fastest algorithm so far, for computing local betweenness centrality. Since the Brandes algorithm aims to compute vertex betweenness centrality, we slightly modified the Brandes algorithm to calculate edge betweenness centrality. Instead of computing local betweenness centralities (Line 23 in Algorithm 1) and update the amounts of increases of betweenness centralities for Type 1 and 2 shortest paths (Line 25 to Line 29 in Algorithm 1), we can directly consider the amounts of increases for Type 1 and 2 shortest paths during local betweenness centrality computation. The detailed explanation for Algorithm 2 is as follow.

Since Type 2 shortest paths include local shortest paths[9] with specific source and target articulation vertices, we first accumulate the number of Type 2 shortest paths which include v_n (Line 22) and then, add the amount of increase using the accumulated number (Line 28) during the *one-sided dependency* computation. $\sigma_t[v_i]$ is an array to store the number of Type 2 shortest paths which pass v_i. In contrast to Type 2 shortest paths, Type 1 shortest paths include a local shortest paths with a specific source articulation vertex v_s only. Therefore, the amount of increase due to Type 1 shortest paths can be accumulated and added during the computation of the *one-sided dependencies* of v_s (Line 39 and Line 30). The additional lines apart from the original Brandes algorithm are underlined.

[9]The shortest paths in the *re-calculation subgraph*.

Algorithm 1: UpdateBC

input : $G = (V, E)$ - Original graph,
E^{Ins}, E^{Del} - Set of edges to be updated
V^{Ins}, V^{Del} - Set of unit vertices to be updated,
$c_e[]$ - Original betweenness centrality array
output : $c_e[]$ - Updated betweenness centrality array

begin
  $V^U = (V \backslash V^{Del}) \cup V^{Ins}$, $E^U = (E \backslash E^{Del}) \cup E^{Ins}$;
  $G^U = (V^U, E^U)$;
  for *each* $v \in V^{Ins}$ **do** `/* vertex insertion */`
    For all $e_i \in E^U$, $c_e[e_i] = c_e[e_i] + \delta^{G^U}_{v,\bullet}(e_i)$;
  for *each* $v \in V^{Del}$ **do** `/* vertex deletion */`
    For all $e_i \in E^U$, $c_e[e_i] = c_e[e_i] - \delta^{G}_{v,\bullet}(e_i)$;
  for *each* $e(v_s, v_t) \in E^{Ins}$ *such that e is a bridge edge in* G^U **do** `/* bridge edge insertion */`
    Let $G_s = (V_s, E_s)$ and $G_t = (V_t, E_t)$ be subgraphs connected by $e(v_s, v_t)$ where $v_s \in V_s, v_t \in V_t$. ;
    For all $e_i \in E_s$, $c_e[e_i] = c_e[e_i] + |V_t| \cdot \delta^{G^U}_{v_s,\bullet}(e_i)$;
    For all $e_i \in E_t$, $c_e[e_i] = c_e[e_i] + |V_s| \cdot \delta^{G^U}_{v_t,\bullet}(e_i)$;
    $c_e[e] = 2 \cdot |V_s| \cdot |V_t|$;
  for *each* $e(v_s, v_t) \in E^{Del}$ *such that e is a bridge edge in G* **do** `/* bridge edges deletion */`
    Let $G_s = (V_s, E_s)$ and $G_t = (V_t, E_t)$ be subgraphs disconnected by $e(v_s, v_t)$ where $v_s \in V_s, v_t \in V_t$. ;
    For all $e_i \in E_s$, $c_e[e_i] = c_e[e_i] - |V_t| \cdot \delta^{G}_{v_s,\bullet}(e_i)$;
    For all $e_i \in E_t$, $c_e[e_i] = c_e[e_i] - |V_s| \cdot \delta^{G}_{v_t,\bullet}(e_i)$;
  `/* Find` ***`re-calculation subgraphs`*** `*/`
  $\mathcal{G} = \emptyset$;
  for *each* $e \in E^{Ins}$ *such that e is not a bridge edge in* G^U **do**
    $\mathcal{G} = \mathcal{G} \cup \{biConnected(e, G^U)\}$;
  for *each* $e \in E^{Del}$ *such that e is not a bridge edge in G* **do**
    $\mathcal{G} = \mathcal{G} \cup \{biConnected(e, G) \backslash \{e\}\}$;
  for *each* $G^{T'} \in \mathcal{G}$ **do** `/* non-bridge edges */`
    $c_e[] = \text{Betweenness}(G^{T'})$;
    for *each edge* e_i *in* $G^{T'}$ **do**
      for *each articulation vertex* $v_j \in V^{T'}$ **do** `/* Type 1 */`
        $c_e[e_i] = c_e[e_i] + c_e^{T_1^{V_{v_j}}}(e_i)$;
      for *each articulation vertex pair* v_j, v_k *such that* $v_j, v_k \in V^{T'}$ *and* $v_j \neq v_k$ **do** `/* Type 2 */`
        $c_e[e_i] = c_e[e_i] + c_e^{T_2^{V_{v_j}, V_{v_k}}}(e_i)$;
  return $c_e[]$

Algorithm 2: UPDATE-BRANDES

```
input  : G^{T'} - Re-calculation subgraph
         c_e[] - Betweenness centrality array
output : c_e[] - Updated betweenness centrality array
begin
    for v_s ∈ G^{T'} do
        S ← empty stack; Q ← empty queue ;
        P[v_i] ← empty list, for all v_i ∈ G^{T'} ;
        σ[v_i] := 0, for all v_i ∈ G^{T'}; σ[v_s] := 1 ;
        d[v_i] := -1, for all v_i ∈ G^{T'}; d[v_s] := 0 ;
        /* Store number of Type 2 SPs each v_i lies on */
        σ_t[v_i] := 0 for all v_i ∈ G^{T'} ;
        enqueue v_s → Q ;
        while Q not empty do
            dequeue v_i ← Q; push v_i → S ;
            for each neighbor v_n of v_i do
                if d[v_n] < 0 then
                    enqueue v_n → Q ;
                    d[v_n] := d[v_i] + 1 ;
                if d[v_n] = d[v_i] + 1 then
                    σ[v_n] := σ[v_n] + σ[v_i] ;
                    append v_i → P[v_n] ;
        δ[v_i] := 0, for all v_i ∈ G^{T'} ;
        while S not empty do
            pop v_n ← S ;
            if v_s, v_n are articulation vertices and v_n ≠ v_s then /* Accumulate
            increase for Type 2 SPs */
                σ_t[v_n] := σ_t[v_n] + |V_{v_s}| · |V_{v_n}| ;
            for v_p in P[v_n] do
                δ[v_p] := δ[v_p] + σ[v_p]/σ[v_n] · (1 + δ[v_n]) ;
                c_e[(v_p, v_n)] := c_e[(v_p, v_n)] σ[v_p]/σ[v_n] · (1 + δ[v_n]) ;
                if v_s is articulation vertex then
                    /* Calculate increase for Type 2 SPs */
                    σ_t[v_p] := σ_t[v_p] + σ_t[v_n]·σ[v_p]/σ[v_n] ;
                    /* Add increase for Type 2 SPs */
                    c_e[(v_p, v_n)] := c_e[(v_p, v_n)] + σ_t[v_n]·σ[v_p]/σ[v_n] ;
                    /* Calculate & add increase for Type 1 SPs */
                    c_e[(v_p, v_n)] := c_e[(v_p, v_n)] + |V_{v_s}|·σ[v_p]/σ[v_n] · (1 + δ[v_n]);
                    c_e[(v_n, v_p)] := c_e[(v_n, v_p)] + |V_{v_s}|·σ[v_p]/σ[v_n] · (1 + δ[v_n]);
    return c_e[]
```

Using Approximation Algorithm for Computing Local Betweenness Centrality. We can also use any approximation betweenness centrality computation algorithm to calculate local betweenness centrality. If we use an approximation betweenness centrality computation algorithm to calculate local betweenness centrality, our algorithm becomes an approximate update algorithm. Interestingly, this approximate update algorithm results in a better approximation with the faster running time than the approximation betweenness centrality computation algorithm since the approximate update algorithm only approximates betweenness centralities of edges in the re-calculation subgraph (i.e., it only approximates local betweenness centrality).

3.3. Community Detection

As we mentioned earlier, the community detection algorithm using betweenness centrality calculates betweenness centralities of edges in a graph iteratively, and removes the edge with the highest centrality until the graph is disconnected. Each of the connected components composes a community. The most well-known community detection algorithm using betweenness centrality is proposed by Newman et al [44]. The algorithm detects k communities as follows.

1. Calculate betweenness centralities for all edges in the graph.
2. Find the edge with the highest betweenness centrality and remove it from the graph.
3. Re-calculate betweenness centralities for all the remaining edges.
4. Repeat Step 2 and 3 until the number of disconnected components is equal to k.

Detecting communities using the existing betweenness centrality algorithm is impractical even for relatively small graphs due to the iterative betweenness centrality computation. Newman et al. [44] also discussed this weakness of the existing betweenness centrality algorithms. However, the proposed algorithm can efficiently update betweenness centralities of remaining edges in Step 3, without performing betweenness centrality computation from the scratch.

A few works [37, 10, 35, 46] disscuss the d-highest centrality problem. However, Olsen et al. [46] focuses on closeness centrality, and Bhowmick et al. [10] and Kourtellis et al. [35] discuss approximation solutions for the d-highest centrality problem. The work [37] done by us is the only exact algorithm for finding the d-highest betweenness centrality vertices. Therefore, we modify our algorithm [37] for finding the d-highest betweenness centrality vertices to an algorithm for finding the highest betweenness centrality edge. The modified algorithm for Step 1 and 2 is as below.

Highest Betweenness Centrality Edge.

The above community detection algorithm is only interested in the highest betweenness centrality edge. Also, many applications such as finding the most influencer in a social network and locating the bottlenecked junction are only

interested in the edge with the highest betweenness centrality rather than all betweenness centralities of all edges in a graph.

There are two challenges for efficiently finding the highest betweenness centrality edge. The first challenge is that we should compute betweenness centralities of a small group of edges without computing all pairs shortest paths in the graph. Corollary 1 and Equation 8 in Section 3.2.2.2 provide a solution to this challenge. The second challenge is that even if we can efficiently compute betweenness centralities of a small group of edges, we should know which group(s) include the highest betweenness centrality edge. The second challenge is addressed by computing the upper-bound of betweenness centralities of edges in each group.

The overall process for finding the highest betweenness centrality edge is as follows.

1. Decompose G into biconnected components. Let $\mathcal{B}$ be a set of the biconnected components.
2. Calculate $\widehat{c_e}(G_b)$, the upper-bound of betweenness centralities of edges in G_b, for each biconnected component $G_b \in \mathcal{B}$.
3. Let G_h be G_b with the highest $\widehat{c_e}(G_b)$ among $G_b \in \mathcal{B}$. Calculate the exact betweenness centralities of edges in G_h.
4. Update $\mathcal{B}$ to $\mathcal{B} \setminus \{G_h\}$.
5. Repeat 3-4 until the known highest betweenness centrality is higher than the new $\widehat{c_e}(G_h)$.

In Step 1, a biconnected component corresponds to a group. Step 2 is related to the second challenge, and Step 3 is related to the first challenge.

Challenge 1. Compute betweenness centralities of a small group of edges without computing all pairs shortest paths in the graph.

As mentioned, the solution for this challenge is from Corollary 1 and Equation 8 in Section 3.2.2.2. Assume that $G = (V, E)$ is the entire graph, and we want to compute betweenness centralities of edges in biconnected component $G_b = (V_b, E_b)$. Let $\mathcal{G}$ be the graph induced by the set of vertices $V \setminus V_b$, and refer $G_{v_i} \subseteq \mathcal{G}$ to a subgraph which is connected to G_b through an articulation vertex $v_i \in V_b$ in G. Let $\mathbb{G}$ be a set of subgraphs G_{v_i} for all articulation vertex $v_i \in V_b$. Betweenness centrality of an edge e in G_b can be efficiently computed as follow.

$$c_e(e) = \quad c_e^{G_b}(e) + \sum_{v_i \in V_b, G_{v_j} \in \mathbb{G}} \frac{|V_{v_j}| \cdot \sigma_{v_i,v_j}(e)}{\sigma_{v_i,v_j}} \quad + \sum_{G_{v_i}, G_{v_j} \in \mathbb{G}} \frac{|V_{v_i}| \cdot |V_{v_j}| \cdot \sigma_{v_i,v_j}(e)}{\sigma_{v_i,v_j}}$$

where V_{v_j} is a vertex set of G_{v_j}, and $v_i \neq v_j$.

The first term $c_e^{G_b}(e)$ represents a portion of betweenness centrality of e with respect to the shortest paths between vertices in G_b, and it is conceptually the same as local betweenness centrality of e in the update problem. The second

term represents a portion of betweenness centrality with respect to the shortest paths between a vertex in V_b and a vertex not in V_b, and the third term represents a portion of betweenness centrality with respect to the shortest paths between vertices not in V_b. Note that the above equation requires only the shortest paths between pairs of vertices v_i, v_j in $V_b \subseteq V$.

Challenge 2. Find the group of edges which includes the highest betweenness centrality edge.

1. Betweenness centrality of any edge in G_b cannot exceed $\lceil |V_b|/2 \rceil^2$. It is used to exclude biconnected components with small vertices[10].
2. The number of the shortest paths between a vertex in V_b and a vertex not in V_b is $(|V| - |V_b|) \cdot (|V_b| - 1)$. The reason that we subtract 1 from $|V_b|$ is the same as the reason that we subtract 1 in Equation 3.
3. Among the shortest paths between vertices not in V_b, $\sum_{G_{v_i}, G_{v_j} \in \mathbb{G}} |V_{v_i} - 1| \cdot |V_{v_j} - 1|$ shortest paths include at least one vertex in V_b. It is used to exclude outlier biconnected components. Here, we subtract 1 from numbers of vertices in the subgraphs in $\mathbb{G}$ since the articulation vertex between each subgraph and G_b is already counted.

Using the above numbers, the upper-bound of betweenness centralities of edges in G_b, denoted as $\widehat{c_e}(G_b)$, can be computed as

$$\lceil |V_b|/2 \rceil^2 + (|V| - |V_b|) \cdot (|V_b| - 1) + \sum_{G_{v_i}, G_{v_j} \in \mathbb{G}} |V_{v_i} - 1| \cdot |V_{v_j} - 1|$$

In the majority of cases, this upper-bound is an overestimate compared to the actual centralities. Despite the overestimate, it is enough to give a concrete theoretical reason for pruning small and outlier components.

Therefore, using the above highest betweenness centrality edge finding algorithm and the proposed update algorithm, we can efficiently find the k communities from a graph. Note that, we can optimize the process by removing d-highest centrality edges at once instead of removing the highest centrality edge for the efficiency. This simple optimization yields a significant speed-up with a small accuracy loss (i.e., four times speed-up with only 10% error). Moon et al. [41] reports that if we delete four highest betweenness centrality edges together for deleting 40 edges, we get 10% error compared to deleting the highest betweenness centrality edge one by one in terms of the deleted edges.

4. Experimental Results

[10] This is different from [37]. The difference comes from the difference between vertex and edge betweenness centrality.

In this section, we first compare the proposed update algorithm with three comparison algorithms in Section 4.2, and then we compare two community detection algorithms using the proposed update algorithm and using the Brandes algorithm in Section 4.3 to show how much benefit can be obtained from the proposed algorithm in a practical application.

4.1. Environment Setting

We select various real graphs which are prone to frequent changes. In cases of directed real graphs, we convert directed edges into undirected edges. If a graph consists of several connected components, the Brandes algorithm can compute and update betweenness centralities of vertices and edges in each connected component separately with a simple arrangement. Therefore, for the fair comparison, we use the maximally connected component of each real graph.

For an edge update, we randomly remove an existing edge e in the graph $G = \{V, E\}$ to construct a new graph $G' = \{V, E/\{e\}\}$ and insert back e to G' to simulate the real edge insertion. For a vertex update, we randomly select a vertex then add a new vertex connected to the selected vertex for the insertion, and delete the selected vertex with all incident edges for the deletion. For the community detection, we vary k, which is the number of communities want to detect, from 2 to 20. We implement all algorithms in Java, and conduct all experiments on a PC equipped with Intel Xeon 2.53GHz CPU and 128GB main memory. Each value in the results is an average over 30 repeated executions.

4.2. Betweenness Centrality Update

Table 4: Edge Update Speed-up on Real Graphs

Graph	Type	$\|V\|$	$\|E\|$	*Avg. ratio* (%)		Speed-up(times)	
				QUBE	Proposed	vs Brandes	vs QUBE
disease[26, 3]	Ownership	516	2376	30.59	11.08	423	21.85
eva[45]	Ownership	4475	4562	3.38	2.21	6476.3	12.2
erdos972[3, 5]	Collab.	4680	7030	8.4	8.38	1064.46	1.04
geom[3]	Collab.	3621	8276	55.95	26.3	68.54	8.34
CAGrQc[39, 4]	Social	4158	13422	62.35	44.68	10.15	2.74
power [3]	Power Net.	4941	13188	41.27	35.69	13.65	1.23
pgp[11]	Social	10680	24340	10.07	6.19	2032.23	3.21
CAhep[39, 4]	Collab.	8638	24298	69.46	49.17	7.43	3.67
contact[2]	Social	11604	88806	36.31	35.88	12.42	1.13
Colorado[a][1]	Road Net.	16539	21587	50.24	48.76	8.32	1.16
NY[b][1]	Road Net.	9263	15256	47.39	47.32	7.43	1.05
					Average	920.36	5.24

[a]Colorado Springs area
[b]Manhattan area

Edge Update. Table 4 shows the speed-up achieved by the proposed algorithm for the edge update and the overall statistics of each real graph. For the edge update, the proposed algorithm and QUBE utilize a subgraph containing

of vertices and edges whose betweenness centralities should be changed as a result of the edge update. *Ratio* is the percentage of the vertices that are in the subgraph with respect to the edge update to those in the entire graph. It is the major factor affecting the performance of the proposed algorithm and QUBE. *Average ratio* of a graph is the average of *ratio*s with respect to a certain number of edge updates on the graph. We use 100 edge updates for computing *average ratio*. However, we do not differentiate between the edge insertion and deletion since each algorithm used in edge update experiments requires the same time for the edge insertion and deletion.

Ratio with respect to the edge update for the proposed algorithm is always smaller than that for QUBE. Speed-up in Table 4 shows how fast the proposed algorithm is compared to the Brandes algorithm and QUBE. Table 4 clearly shows that the proposed algorithm outperforms QUBE and the Brandes algorithm, and the performance of the proposed algorithm increases as *ratio* decreases. Compared to the Brandes algorithm and QUBE, the proposed algorithm performs 920 times and 22 times faster on the average, respectively. The proposed algorithm is always faster than QUBE since *ratio* of the proposed algorithm is always smaller than that of QUBE.

Average ratio. *Average ratio* is highly correlated with the size of the largest biconnected component, but *Average ratio* is always smaller than the percentage of the vertices that are in the largest biconnected component to those in the entire graph. This is because, a *re-calculation subgraph* with respect to an edge update is not always the largest biconnected component.

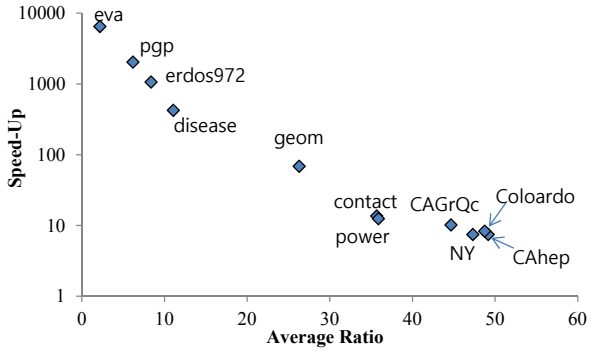


Figure 4: Edge Update Performance vs. Average Ratio

To clearly show the relation between the performance and the size of a *re-calculation subgraph* with respect to an edge update. As *average ratio* decreases, the performance gap compared to the Brandes algorithm increases.

Re-calculation Subgraph Construction Time. Not only the size of the *re-calculation subgraph* in the proposed algorithm is smaller than that in QUBE, but also the construction time of the *re-calculation subgraph* in the proposed algorithm is much faster than that in QUBE. The results are listed in Table 5.

Table 5: Re-calculation Subgraph Construction

Graph	Type	$\|V\|$	$\|E\|$	Time (ms)		Speed-Up vs QUBE
				QUBE	Proposed	
disease[26, 3]	Ownership	516	2376	4.26	0.54	7.8
eva[45]	Ownership	4475	4562	193.93	17.13	11.3
erdos972[3, 5]	Collab.	4680	7030	298.66	21.79	13.7
geom[3]	Collab.	3621	8276	239.4	15.1	15.8
CAGrQc[39, 4]	Social	4158	13422	334.27	22.07	15.1
power [3]	Power Net.	4941	13188	385.43	27.66	13.9
pgp[11]	Social	10680	24340	1895.18	104.23	18.2
CAhep[39, 4]	Collab.	8638	24298	1947.61	90.2	21.6
contact[2]	Social	11604	88806	7719.15	163.37	47.2
Colorado[1]	Road Net.	16539	21587	5363.3	453.2	11.83
NY[1]	Road Net.	9263	15256	1142.23	161.2	7.09
					Average	16.7

On the average, the proposed algorithm is about 17 times faster than QUBE for constructing the *re-calculation subgraph*. Because of such an improvement, the *re-calculation subgraph* with respect to the edge update can be constructed on the fly in the proposed algorithm. On the other hand, in QUBE, all *Minimum Union Cycles* are precomputed, and maintained with respect to graph updates.

Table 6: Vertex Update Speed-up on Real Graphs (ms)

Graph	Type	$\|V\|$	$\|E\|$	Unit Vertex Insertion Speed-Up vs. Brandes	Vertex Deletion Speed-Up vs. Goel et al.[25]
disease[26, 3]	Ownership	516	2376	178	17.23
eva[45]	Ownership	4475	4562	4024	10.63
erdos972[3, 5]	Collab.	4680	7030	4954	1.02
geom[3]	Collab.	3621	8276	3493	14.12
CAGrQc[39, 4]	Social	4158	13422	3789	4.86
power [3]	Power Net.	4941	13188	4789	2.34
pgp[11]	Social	10680	24340	10018	2.25
CAhep[39, 4]	Collab.	8638	24298	8986	3.23
contact[2]	Social	11604	88806	11619	1.12
Colorado[1]	Road Net.	16539	21587	20390	1.05
NY[1]	Road Net.	9263	15256	9297	1.03
			Average	7412.5	5.35

Vertex Update. Table 6 shows speed-ups achieved by the proposed algorithm for the vertex update on real graphs compared to the Brande algorithm and Goel et al's algorithm [25]. Since Goel et al. address the vertex deletion only[11], we compare the proposed algorithm with Brandes for the vertex insertion. For the unit vertex insertion, the proposed algorithm is tremendously faster than the Brandes algorithm. The proposed algorithm is 178 times faster than the Brandes algorithm for *disease*, which is the smallest among all graphs, and 20390 times faster for *Colorado*, which is the largest. For the vertex dele-

[11] Although Goel et al.'s algorithm could be extend to the insertion case, this requires some modifications that are not covered in their paper as they already mentioned in the paper.

tion, the proposed algorithm is about 5 times faster than Goel et al's work. Note that the deletion of an vertex invokes deletions of incident edges of the vertex.

4.3. Community Detection

We also conduct experiments for the community detection algorithm in Section 3.3. Before we discuss the results for the community detection algorithm, we present the results for finding the highest centrality edge since the community detection algorithm utilizes the highest betweenness centrality edge finding algorithm internally.

Table 7: Speed-up for Finding the Highest Betweenness Centrality Edge

Graph	Type	$\|V\|$	$\|E\|$	Time (ms)		Speed-Up
				Brandes	Proposed	vs Brandes
disease[26, 3]	Ownership	516	2376	535	29.4	18.22
eva[45]	Ownership	4475	4562	257558	60.8	4235.83
erdos972[3, 5]	Collab.	4680	7030	150202	173.9	863.56
geom[3]	Collab.	3621	8276	212231	6583	32.24
CAGrQc[39, 4]	Social	4158	13422	349608	73447	4.76
power [3]	Power Net.	4941	13188	3596501	489987	7.34
pgp[11]	Social	10680	24340	1941034	1036	1872.96
CAhep[39, 4]	Collab.	8638	24298	4613027	728756	6.33
contact[2]	Social	11604	88806	15720935	1800794	8.73
Colorado[1]	Road Net.	16539	21587	2259284	522982	4.32
NY[1]	Road Net.	9263	15256	2677067	551973	4.85
					Average	641.74

Finding the Highest Betweenness Centrality Edge.

Since an exact algorithm for finding the highest betweenness centrality edge does not exist, we compare our algorithm to the Brandes algorithm which is the fastest known algorithm for the exact betweenness centrality computation. Note that all existing algorithms require all-pairs shortest paths even for computing betweenness centrality of one edge in a graph. We show the experimental results with the summary of graphs in Table 7. The speed-up shows how much improvement is achieved by our algorithm compared to the Brandes algorithm.

Finding k Communities.

Figure 5 shows the community detection times of the community detection algorithms using the proposed algorithm, and using the Brandes algorithm. Each line shows the time taken to detect k communities. Each vertical bar shows the number of deleted edges to detect k communities. In case of *eva*, the community detection algorithm using the proposed algorithm finds the first two communities 2370 times faster than that using the Brandes algorithm. For detecting 20 communities, the community detection algorithm using the proposed algorithm performs 1628 times faster. Note that more than 500 edges are removed from the graph to detect 20 communities in *erdos972*. In *CAGrQc*, the community detection algorithm removes 441 edges to detect one more community after detecting 11 communities. This explains the steep time increase

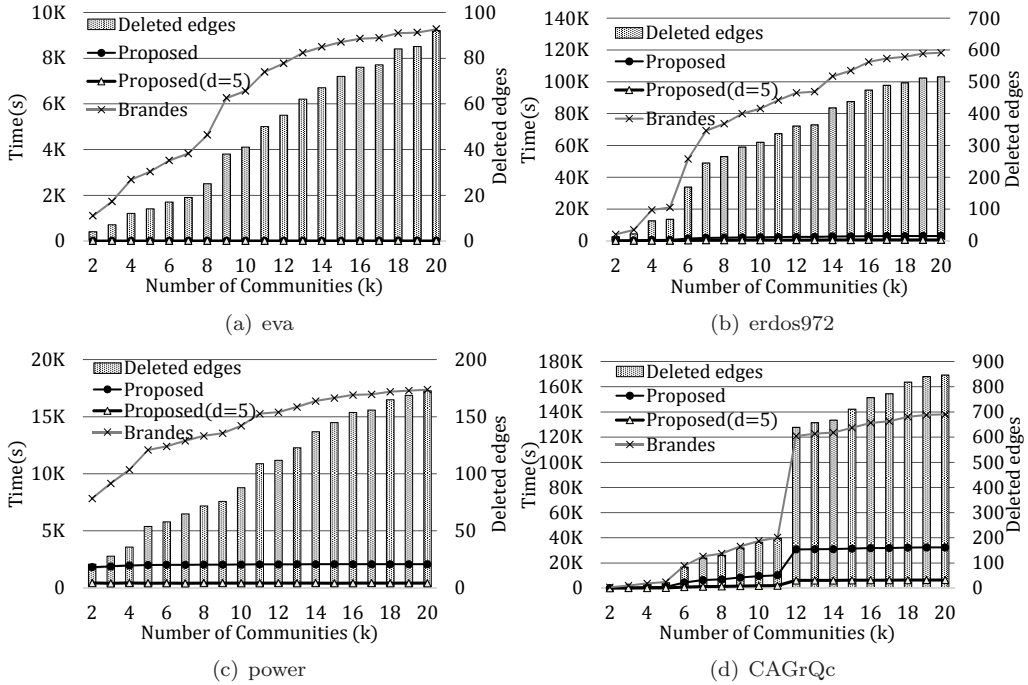


Figure 5: Community Detection on Real Graphs

for detecting 12 communities in both the proposed and the Brandes based algorithms. Due to the space limitations, we only show the results for four graphs but the results for other graphs are similar.

As mentioned, we can optimize the process by removing d-highest centrality edges together instead of removing the highest centrality edge one by one for the efficiency. This simple optimization will yield a significant speed-up with a small accuracy loss. Generally, we can obtain d times speed-up by removing d-highest centrality edges instead of the highest centrality edge. The results when d is 5 are also depicted in Figure 5 (labeled as Proposed(d=5)).

5. Conclusions

In this paper, we propose an efficient betweenness centrality update algorithm for fully dynamic graphs. To the best of our knowledge, this is the first work which deals with the betweenness centrality update problem in fully dynamic graphs ,and the only work which does not requires any structure to maintin for the edge update. When an edge is updated, the algorithm identifies a subgraph called *re-calculation subgraph* containing of vertices and edges whose betweenness centralities should be changed, and updates betweenness centralities of edges in the *re-calculation subgraph* without computing all-pairs shortest paths. When a vertex is updated, the algorithm computes the amounts of increases or decreases in betweenness centralities of edges in a graph by performing

a single-source shortest path computation from the updated vertex. Furthermore, as a practical application for using betweenness centrality, we also adapt a community detection algorithm using the proposed update algorithm and the highest betweenness centrality edge finding algorithm. The experimental results on 11 real graphs show that the proposed algorithm outperforms our preliminarily work, QUBE, and the Brandes algorithm for all update operations.

[1] 9th dimacs implementation challenge. `http://www.dis.uniroma1.it/challenge9/download`.

[2] Metafilter infodump. `http://stuff.metafilter.com/infodump`.

[3] Pajek datasets. `http://pajek.imfm.si/doku.php?id=data:urls:index`.

[4] Stanford large network dataset collection. `http://snap.stanford.edu/data/`.

[5] The university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[6] J.M. Anthonisse and Stichting Mathematisch Centrum/ 1Amsterdam. Afdeling Mathematische besliskunde. The rush in a directed graph. Technical report, 1971.

[7] André Ricardo Backes and Odemir Martinez Bruno. Polygonal approximation of digital planar curves through vertex betweenness. *Inf. Sci.*, 222:795–804, 2013.

[8] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena 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.

[9] David A. Bader and Kamesh 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.

[10] Sanjukta Bhowmick, Vyacheslav V. Rykov, and Vladimir Ufimtsev. Acm src poster: a scalable group testing based algorithm for finding d-highest betweenness centrality vertices in large scale networks. In *SC Companion*, pages 121–122, 2011.

[11] 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, November 2004.

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

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

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

[15] Daniel J Brass. Being in the right place: A structural analysis of individual influence in an organization. *Administrative science quarterly*, pages 518–539, 1984.

[16] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, November 2004.

[17] Camil Demetrescu and Giuseppe F Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms (TALG)*, 2(4):578–601, 2006.

[18] Shlomi Dolev, Yuval Elovici, and Rami Puzis. Routing betweenness centrality. *J. ACM*, 57(4):25:1–25:27, May 2010.

[19] Ruth Dunn, Frank Dudbridge, and Christopher M Sanderson. The use of edge-betweenness clustering to investigate biological function in protein interaction networks. *BMC bioinformatics*, 6(1):39, 2005.

[20] Douglas K Fleming and Yehuda Hayuth. Spatial characteristics of transportation hubs: centrality and intermediacy. *Journal of Transport Geography*, 2(1):3–18, 1994.

[21] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

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

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

[24] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In J. Ian Munro and Dorothea Wagner, editors, *ALENEX*, pages 90–100. SIAM, 2008.

[25] Keshav Goel, Rishi Ranjan Singh, Sudarshan Iyengar, and Sukrit. A faster algorithm to update betweenness centrality after node alteration. In *WAW*, pages 170–184, 2013.

[26] Kwang-Il Goh, Michael E Cusick, David Valle, Barton Childs, Marc Vidal, and Albert-László Barabási. The human disease network. *Proceedings of the National Academy of Sciences*, 104(21):8685–8690, 2007.

[27] Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *SocialCom/PASSAT*, pages 11–20, 2012.

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

[29] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[30] Shuangshuang Jin, Zhenyu Huang, Yousu Chen, Daniel G. Chavarria-Miranda, John Feo, and Pak Chung Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, pages 1–7. IEEE, 2010.

[31] U. Kang, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong. Centralities in large networks: Algorithms and observations. In *SDM*, pages 119–130, 2011.

[32] Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *ASONAM*, pages 33–40, 2013.

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

[34] Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. Centrality indices. In *Network Analysis*, pages 16–61, 2004.

[35] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *Social Netw. Analys. Mining*, 3(4):899–914, 2013.

[36] 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, April 2006.

[37] Min-Joong Lee and Chin-Wan Chung. Finding k-highest betweenness centrality vertices in graphs. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, WWW Companion '14, pages 339–340, 2014.

[38] Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. Qube: a quick algorithm for updating betweenness centrality. In *Proceedings of the 21st international conference on World Wide Web*, pages 351–360. ACM, 2012.

[39] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[40] Loet 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.

[41] Seunghyeon Moon, Jae-Gil Lee, and Minseo Kang. Scalable community detection from networks by computing edge betweenness on mapreduce. In *International Conference on Big Data and Smart Computing, BIGCOMP 2014, Bangkok, Thailand, January 15-17, 2014*, pages 145–148, 2014.

[42] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality - incremental and faster. pages 577–588, 2014.

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

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

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

[46] Paul W. Olsen, Alan G. Labouseur, and Jeong-Hyon Hwang. Efficient top-k closeness centrality search. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 196–207, 2014.

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

[48] G Ramalingam and Thomas Reps. *On the computational complexity of incremental algorithms*. University of Wisconsin-Madison. Computer Sciences Department, 1991.

[49] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 112–119, New York, NY, USA, 2005. ACM.

[50] Hai Zhuge. Communities and emerging semantics in semantic link network: Discovery and learning. *Knowledge and Data Engineering, IEEE Transactions on*, 21(6):785–799, June 2009.