World Wide Web (2014) 17:351–376
DOI 10.1007/s11280-013-0204-x

Mining most frequently changing component in evolving graphs

Yajun Yang · Jeffrey Xu Yu · Hong Gao · Jian Pei · Jianzhong Li

Received: 20 October 2012 / Revised: 4 January 2013 /
Accepted: 17 January 2013 / Published online: 23 February 2013

Abstract Many applications see huge demands of finding important changing areas in evolving graphs. In this paper, given a series of snapshots of an evolving graph, we model and develop algorithms to capture the most frequently changing component (MFCC). Motivated by the intuition that the MFCC should capture the densest area of changes in an evolving graph, we propose a simple yet effective model. Using only one parameter, users can control tradeoffs between the "density" of the changes and the size of the detected area. We verify the effectiveness and the efficiency of our approach on real data sets systematically.

Keywords Detecting graph changes · Evolving graphs

1 Introduction

Graphs have been widely used to model complex relationships among various entities in real applications, and often evolve over time. For example, in social

Y. Yang · H. Gao · J. Li
Harbin Institute of Technology, Harbin, People's Republic of China

Y. Yang
e-mail: yjyang@hit.edu.cn

H. Gao
e-mail: honggao@hit.edu.cn

J. Li
e-mail: lijzh@hit.edu.cn

J. X. Yu (✉)
Chinese University of Hong Kong, Hong Kong, Hong Kong
e-mail: yu@se.cuhk.edu.hk

J. Pei
Simon Fraser University, Burnaby, Canada
e-mail: jpei@cs.sfu.ca

network, the relationships between any two community or people always evolve over time. Recently, there are many works that study how do the relationships evolve among various entities in networks [6, 21, 22, 33, 34]. An evolving graph can be modeled as a series of snapshots $\mathsf{G} = (G_1, G_2, \cdots, G_{\|\mathsf{G}\|})$, where each snapshot G_i $(1 \leq i \leq \|\mathsf{G}\|)$ is a graph. Discovering frequently changing areas in evolving graphs is critical in many applications.

As a concrete example, consider a road traffic network, where each edge represents a road segment with smooth traffic. As traffic keeps changing over time, the road traffic network is evolving. By analyzing and capturing the most frequently changing areas in a series of snapshots of the traffic network, say a snapshot every 30 minutes, one can obtain meaningful insights into how traffic jams are formed. Specifically, those road segments that are smooth or jammed most of the time, i.e., the infrequently changing edges and areas, either are not for concern or cannot be improved easily. Instead, those frequently changing edges and areas, i.e., those road segments that may have heavy traffic easily and also can be smoothened quickly, are critical for traffic scheduling and route planning. Detecting the most frequently changing areas in such an evolving traffic network is the core of the task.

As another interesting example, consider a financial relationship network among business entities, where an edge between two parties indicates that the total amount of the transactions between the corresponding parties in a period (e.g., a month) is over some threshold. The government finance administration units often monitor such financial relationship network for possible frauds and regulation violation. Those frequently changing edges and areas in such a financial relationship network are particularly interesting since they capture the potentially unusual bursting financial relationship among a set of entities.

Modeling and finding the most frequently changing areas in evolving graphs is far from trivial. On the one hand, it is easy to find the most frequently changing edge, i.e., a pair of vertices that are connected and disconnected the largest number of times in the snapshots. However, such an edge is not informative in analyzing the whole evolving graph. On the other hand, changes may likely occur extensively in many places in a large graph. Returning the whole graph or a large portion of it may be neither interesting nor useful for analysis. In fact, an analyst often has to balance between the change frequency and the size of the changing components detected. Moreover, isolated changes are often less interesting. Instead, we want to find the changes that are connected well in the graph topology.

In this paper, we tackle the problem of discovering the most frequently changing components (MFCC) in evolving graphs, and make important progress. First, we propose an elegant model of frequently changing components. Our model, motivated by the intuition of density of changes in evolving snapshots, encompasses the important factors discussed above. Specifically, our model captures the dense connected components in evolving graphs, and takes only one parameter that allows a user to control the tradeoff between the "density" and the size. Our model is built concretely on the well adopted notions of connectivity and maximum flow analysis on large graphs.

Discovering MFCC is computationally challenging. We develop an efficient algorithm to find MFCC on evolving graphs. Our method has a time complexity of $O(|V|^2 \log |V|))$, where V is the set of vertices in the evolving graph, which allows the mining task to be conducted on many practical middle size graphs. We verify the effectiveness and efficiency of our approach using real data sets systematically.

The rest of the paper is organized as follows. Section 2 develops a new measure of changes called cumulated connectivity change, and defines MFCC. It also presents the properties of MFCC and the problem statement. Section 3 discusses how to compute cumulated connectivity change. Section 4 shows how to find MFCC. We report an empirical evaluation on real-life datasets in Section 5. Section 6 discusses the related work. We conclude the paper in Section 7.

2 Modeling most frequently changing components

In this paper, we consider simple undirected graphs only. An evolving graph can be modeled as a series of undirected graphs, denoted by $\mathsf{G} = (G_1, G_2, \cdots, G_{\|\mathsf{G}\|})$, where $G_t = (V_t, E_t)$ is a snapshot at time t with a set of vertices V_t and a set of edges of E_t. $\|\mathsf{G}\|$ is the number of G_t in G and is called the length of G. Taking $\overline{\mathsf{V}} = \cup_{t=1}^{\|\mathsf{G}\|} V_t$, we can assume each graph G_t is on the vertex set $\overline{\mathsf{V}}$, and thus the vertex set is constant in our analysis. Only the edge set is changing in different snapshots.

2.1 Measuring changes between vertex pairs

We measure the changes between snapshots using the maximum number of independent paths (maximum flow). Given a snapshot $G_t = (V_t, E_t)$, a pair of vertices u and v in G_t is said to be k-edge-connected ($k \geq 1$), if removing any $k-1$ edges in G_t cannot disconnect u and v. Intuitively, if u and v are k-edge-connected, they must be $(k-1)$-edge-connected, but the other way does not hold. Denote by $\mathsf{econ}_t(u, v)$ the maximum value of k such that u and v are k-edge-connected in graph G_t, that is, $\mathsf{econ}_t(u, v) = \max\{k \mid u \text{ and } v \text{ are } k\text{-edge-connected in } G_t\}$. For simplicity, we use k and $\mathsf{econ}_t(u, v)$ alternately to denote maximum value of k such that u and v are k-edge-connected.

We can measure the connectivity change between u and v from G_{t-1} to G_t as follows.

$$\delta_t(u, v) = |\mathsf{econ}_t(u, v) - \mathsf{econ}_{t-1}(u, v)| \tag{1}$$

The k-edge-connectivity of u and v, though being a popularly used and informative measure, does not capture all changes affecting both u and v in an evolving graph. $\delta_t(u, v)$ may be 0, even if some edges are added in G_t and/or deleted from G_{t-1}. For instance, in Figure 1, two edges (a, b) and (e, d) in G_{t-1} (left) are deleted in G_t (right). In addition, two new edges (a, d) and (e, b) are added into G_t. The connectivity change between every pair of vertices is zero. Consequently, using connectivity change we cannot detect the changes between the two snapshots.

In order to capture all possible changes, we have to consider the similarity between two graphs. Graph edit distance can help here.

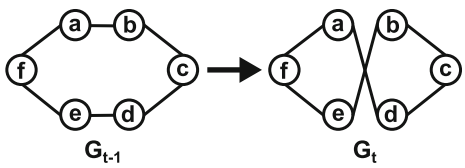


Figure 1 An example: G_{t-1} (*left*) is changed to G_t (*right*).

Given two graphs, $G_t = (V_t, E_t)$ and $G_{t-1} = (V_{t-1}, E_{t-1})$ where $V_t = V_{t-1}$. The edit distance between G_t and G_{t-1}, denoted by $|\Delta E_t|$, is the minimum number of edge edit operations, including edge-insertions and edge-deletions, to transform G_{t-1} into G_t. In our setting, under the assumption that the vertices are unique and $V_{t-1} = V_t$, $\Delta E_t = (E_{t-1} \setminus E_t) \cup (E_t \setminus E_{t-1})$. Let $\Delta E_t = (e_1^{d_t}, \cdots, e_p^{d_t}, e_{p+1}^{a_t}, \cdots, e_{p+q}^{a_t})$ be the set of edge-changes, where $e_i^{d_t}$ is an edge e_i existing in G_{t-1} deleted in G_t $(1 \le i \le p)$, and $e_j^{a_t}$ is an edge e_j inserted in G_t $(p+1 \le j \le p+q)$. Denote by $\overline{\mathsf{g}}_t = (g_1, \cdots, g_p, g_{p+1}, \cdots, g_{p+q})$ a sequence of edge-by-edge changing graphs from G_{t-1} to G_t, where $G_{t-1} = g_0$, g_i is the graph deleting edge $e_i^{d_t}$ from g_{i-1} when $1 \le i \le p$, and g_j is the graph adding a new edge $e_j^{a_t}$ into g_j when $p+1 \le j \le p+q$, and $g_{p+q} = G_t$. Note $|\overline{\mathsf{g}}_t| = p+q$.

We define the **cumulated connectivity change** between two vertices u and v from G_{t-1} to G_t as

$$\Delta_t(u, v) = \sum_{i=1}^{|\overline{\mathsf{g}}_t|} \delta_i(u, v) \tag{2}$$

Here, $\delta_i(u, v)$ measures the connectivity change between u and v when g_{i-1} is changed to g_i in $\overline{\mathsf{g}}_t$.Please note that $\Delta_t(u, v)$ is independent from the specific order of edge deletions and insertions in $\overline{\mathsf{g}}$, respectively. We will prove it in Theorem 2. Consider an intermediate graph G_b between G_{t-1} and G_t, where G_b is constructed by removing all deletion edges $e_i^{d_t}$ $(1 \le i \le p)$ in ΔE_t from G_{t-1}. In fact, $\Delta_t(u, v)$ is only related to the intermediate graph G_b. In the other words, $\Delta_t(u, v)$ is not related to that how G_{t-1} is changed to G_b and how G_b is changed to G_t.

The cumulated connectivity change between u and v in the entire graph sequence G is then

$$\Delta(u, v) = \sum_{t=2}^{\|\mathsf{G}\|} \Delta_t(u, v) \tag{3}$$

For G_{t-1} and G_t in Figure 1, $\overline{\mathsf{g}}_t$ is shown in Figure 2. The cumulated connectivity change between a and b is $\Delta_t(a, b) = \sum_{i=1}^{4} \delta_i(a, b) = 1 + 1 + 1 + 1 = 4$.

2.2 Modeling frequently changing components

For an evolving graph G, we define a universal graph $\overline{\mathsf{G}} = (\overline{\mathsf{V}}, \overline{\mathsf{E}})$, where $\overline{\mathsf{V}} = \bigcup_{1 \le t \le \|\mathsf{G}\|} V_t$, $\overline{\mathsf{E}} = \bigcup_{1 \le t \le \|\mathsf{G}\|} E_t$. A frequently changing component is a subgraph of $\overline{\mathsf{G}}$.

For a subgraph G_s of $\overline{\mathsf{G}}$, how can we measure the quality of G_s as a frequently changing component? We first consider two straightforward objective functions

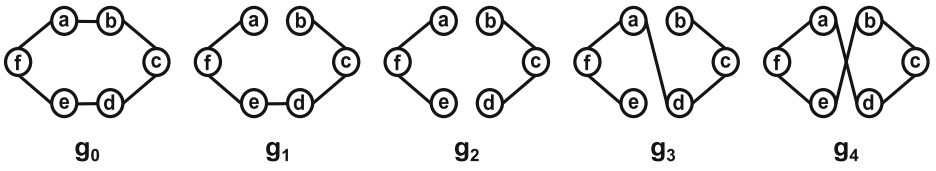


Figure 2 $\overline{\mathsf{g}}_t$ for G_{t-1} and G_t in Figure 1.

based on connectivity change. First, $\Delta(G_s) = \sum_{u,v \in G_s, u \neq v} \Delta(u, v)$. Based on $\Delta(G_s)$, the MFCC of $\bar{G}$ is $\bar{G}$ itself. This does not provide any information about changes. Second, $f(G_s) = \frac{\Delta(G_s)}{N(G_s)}$, where $N(G_s)$ is the number of vertex pairs in G_s, i.e., $N(G_s) = |\{(u, v)|u, v \in G_s\}|$. Intuitively, $f(G_s)$ reflects the density of connectivity change for every vertex pair in G_s. Here, $N(G_s)$ controls the size of G_s such that $f(G_s)$ is larger when $N(G_s)$ is smaller. However, $N(G_s)$ becomes larger very quickly when G_s becomes larger. Hence, based on $f(G_s)$, it is most likely that the MFCC is a single edge that change times is the most in evolving graph. This result is obvious meaningless.

We propose the following new objective function.

$$F(G_s) = \frac{\Delta(G_s) - \alpha(G_s)}{\beta(G_s)^\gamma} \tag{4}$$

where $0 \leq \gamma$ is parameter,

$$\Delta(G_s) = \sum_{u,v \in G_s, u \neq v} \Delta(u, v) \tag{5}$$

$$\alpha(G_s) = \sum_{(u,v) \in E_s, u \neq v} \alpha(u, v) \tag{6}$$

$$\beta(G_s) = |E_s| + \beta_c(G_s) \tag{7}$$

$\alpha(u, v) = \sum_{t=2}^{\|\mathsf{G}\|} \alpha_t(u, v)$, that is, $\alpha(u, v)$ is the number of times that the edge (u, v) remains unchanged in G. Here, $\alpha_t(u, v) = 1$ if edge (u, v) appears in both G_{t-1} and G_t; otherwise, $\alpha_t(u, v) = 0$. $\alpha(G_s)$ is the sum of $\alpha(u, v)$ for any two different vertices u and v in G_s. Similar to $N(G_s)$, we use $\beta(G_s)$ to control the size of G_s. $\beta_c(G_s)$ is the number of pairs of vertices in G_s that cannot be connected by a path consisting of change edges. We use $\mathsf{cpath}(u, v)$ to denote a path in $\overline{\mathsf{G}}$ between two vertices u and v such that every edge on this path is changed at least once. Then, $\beta_c(G_s)$ is the number of pairs of u and v in G_s such that there does not exist a $\mathsf{cpath}(u, v)$ between u and v in $\overline{\mathsf{G}}$, or in other words, there are unchanged edges in every path between u and v. The motivation of $\beta_c(G_s)$ is that the less unchanged edges are expected to appear in G_s. Consider a subgraph G_s including two vertices u and v, if there is no $\mathsf{cpath}(u, v)$ between u and v, then there must be an edge that never changes in G_s. It is important to note that $\beta_c(G_s)$ is designed to become smaller when there are more edge-changes in G_s and become larger when there are less edge-changes in G_s. It assists to obtain a reasonable large MFCC when there are many edge-changes.

The ratio in (4) essentially measures the density of connectivity change in subgraph G_s. The nominator measures the changes in G_s. Specifically, $\Delta(G_s)$ captures the changing edges and $\alpha(G_s)$ is the total number of edges unchanged in G_s. The denominator measures the stable elements. Specifically, instead of using the total number of vertex pairs in G_s as the denominator, we only count the number of vertex pairs that cannot be connected by a path consisting of change edges ($\beta_c(G_s)$).

The parameter γ controls the importance of vertex pairs that cannot be connected by a cpath in the frequently changing component found. If $\gamma = 0$, the whole graph $\overline{\mathsf{G}}$ will be returned. If $\gamma = \infty$, then the most frequently changed edge will be returned.

Example 1 Consider the evolving graph $\mathsf{G} = (G_1, G_2, \cdots G_6)$ in Figure 3. $\overline{\mathsf{G}}$ is also shown. The number times an edge is changed is also labeled in $\overline{\mathsf{G}}$. Consider a

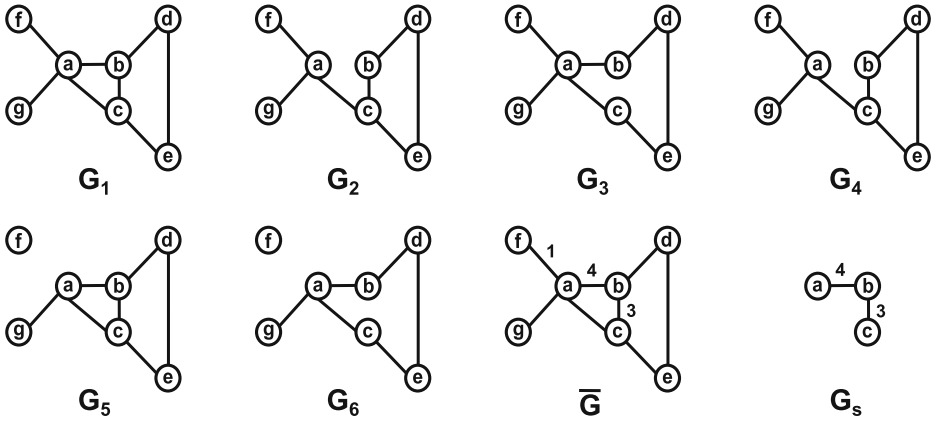


Figure 3 An example, $\mathsf{G} = (G_1, G_2, \cdots, G_6)$, $\overline{\mathsf{G}}$, and the max G_s.

subgraph G_s in the same figure. Assuming $\gamma = 1$, $F(G_s) = (\Delta(a,b) - \alpha(a,b) + \Delta(b,c) - \alpha(b,c) + \Delta(a,c))/(|E_s| + \beta_c(G_s)) = (3+5+4)/(2+0) = 6$. Here, $\Delta(a,c) = 4$, but the edge (a,c) does not change. Thus, the edge (a,c) does not necessarily to be included in the subgraph G_s. In other words, if we insert the edge (a,c) into G_s, the numerator decreases by $\alpha(a,c) = 5$. On the other hand, if we insert another vertex into G_s, the denominator increases. Thus, $F(G_s)$ is the maximum among all possible subgraphs in $\overline{\mathsf{G}}$. This G_s is the graph to be found.

Given an evolving graph $\mathsf{G} = (G_1, G_2, \cdots, G_{\|\mathsf{G}\|})$, and parameter γ, the **problem of discovering the most frequently changing component (**MFCC**)** is to find a connected subgraph G_s ($\subseteq \overline{\mathsf{G}}$) such that $F(G_s)$ is maximized. Note that G_s is a subgraph of $\overline{\mathsf{G}}$, but it does not necessarily need to be an induced subgraph of $\overline{\mathsf{G}}$.

2.3 Some properties of MFCC

We show that the vertices in G_s with the max $F(G_s)$ to be found is a tree.

First, for any pair of vertices u and v in $\overline{\mathsf{G}}$, if there exists a $\mathsf{cpath}(u,v)$, we say $u \stackrel{c}{=} v$. Based on the definition, given vertices, u, u', and v in $\overline{\mathsf{G}}$, if $u \stackrel{c}{=} v$ and $u' \stackrel{c}{=} v$, then $u \stackrel{c}{=} u'$. In other words, if there exists a $\mathsf{cpath}(u,v)$ and a $\mathsf{cpath}(v,u')$, then there must exist a $\mathsf{cpath}(u,u')$. Thus, the relationship $\stackrel{c}{=}$ is reflexive, transitive and symmetry. All vertices of graph $\overline{\mathsf{G}}$ can be partitioned by $\stackrel{c}{=}$ into equivalence classes which we call cpath-components. We denote by PC a subset of vertices in $\overline{\mathsf{G}}$ in which every pair of vertices, u and v, satisfies $u \stackrel{c}{=} v$.

Second, given a subgraph $G_s = (V_s, E_s)$ of $\overline{\mathsf{G}}$, the value of $\beta(G_s)$ consists of two parts, $|E_s|$ and β_c, where $\beta_c(G_s)$ is only related to the cpath-component of vertices set V_s. That is,

$$\beta(G_s) = |E_s| + \sum_{i \neq j} |\mathsf{PC}_i| \times |\mathsf{PC}_j| \tag{8}$$

Here, PC_i and PC_j are two distinct cpath-components in G_s. We explain (8) below. By the definition, $\beta_c(G_s)$ is to compute the number of pairs of vertices u and v in G_s if

there does not exist $\mathsf{cpath}(u, v)$ or $\overset{c}{=}(u, v)$ is not true. Therefore, if u and v in G_s are not in the same cpath-component, $\beta_c(G_s)$ will increase by 1 when computing u and v, and (8) computes all such pairs. As an example, in Figure 3, (a, b, c, f) is a cpath component PC_a. For other 3 vertices, there are four cpath components PC_d, PC_e, and PC_g which consist of one vertex. Therefore, $\beta(\overline{\mathsf{G}}) = |\overline{\mathsf{E}}| + \sum_{i \neq j} |\mathsf{PC}_i| \times |\mathsf{PC}_j| = 23$.

Third, (4) can be computed using (9) below for a graph $G_s(V_s, E_s)$, based on (8).

$$F(G_s) = \frac{\sum_{u,v \in V_s, u \neq v} \Delta(u, v) - \sum_{(u,v) \in E_s} \alpha(u, v)}{\left(|E_s| + \sum_{i \neq j} |\mathsf{PC}_i| \times |\mathsf{PC}_j| \right)^{\gamma}} \tag{9}$$

Theorem 1 *For a given subset of vertices V_s in $\overline{\mathsf{G}}$, G_s over V_s with the max $F(G_s)$ value is a connected tree.*

Proof sketch We prove it by showing that G_s cannot be a non-tree graph. Assume that, for a given subset of vertices $V_s \subseteq \overline{\mathsf{G}}$, G_s that has the max $F(G_s)$ among all connected subgraphs consisting of V_s is not a tree. Then, there must exist a cycle in G_s. Let $\phi = F(G_s)$ using (9). Suppose (u, v) is one of the edges in the cycle. If we delete it, G_s is still a connected subgraph. According to (9) and the definition of $\alpha(u, v)$, the numerator increases by $\alpha(u, v)$ after deleting the edge (u, v), and the denominator $\beta(G_s)$ will decrease by 1 at least, because $|E_s|$ becomes $|E_s| - 1$. Therefore ϕ will increase, when the edge (u, v) is deleted from G_s. This is contradict with the assumption. This leads to the conclusion that the MFCC in $\overline{\mathsf{G}}$ that has the max $F(\cdot)$ value is a tree. □

By Theorem 1, we know G_s is in a tree shape. In fact, $F(G_s)$ is a subset V_s of connected vertices, among which the density of connectivity changes is the largest. In the other words, these vertices are organized in a tree shape.

It is important to note that any valid objective function $F(G_s)$ should satisfy the following property.

Property 1 Given any two graphs G_a and G_c. Let $\mathsf{G}_1 = (G_a, G_c)$ and $\mathsf{G}_2 = (G_c, G_a)$. The $F(G_s)$ computed from both G_1 and G_2 are the same.

We verify this claim by Lemma 2 in Section 3.

3 Compute CCC

A key point of the problem is to compute cumulated connectivity change $\Delta(u, v)$ (3). In order to compute $\Delta(u, v)$, we need to compute $\Delta_t(u, v)$ for $\|\mathsf{G}\| - 1$ times, because $\Delta(u, v)$ is the sum of $\Delta_t(u, v)$ (2), for every pair of vertices (u, v) in G, for $u \neq v$. Suppose that there are p edge-deletions and q edge-insertions from G_{t-1} to G_t in G. This implies that there is a sequence of single-edge-change graphs, $\overline{\mathsf{g}}_t = (g_1, \cdots, g_p, g_{p+1}, \cdots, g_{p+q})$ and $|\overline{\mathsf{g}}_t| = p + q$, from G_{t-1} to G_t. Consequently, in order to compute $\Delta_t(u, v)$, it needs to compute k-edge-connectivity for every two

different vertices over each $g_i \in \overline{\mathsf{g}}_t$, and repeats it $p+q$ times. The cost of doing so is extremely high. We call such a naive approach as a multi-way approach.

In this section, we propose a new approach called 2-WAY-CCC. As the name implies, in order to compute $\Delta_t(u, v)$, we only compute k-edge-connectivity for every two different vertices twice. 2-WAY-CCC does not rely on the number of edge-deletions and edge-insertions when G_{t-1} is changed to G_t, when computing $\Delta_t(u, v)$. Note: $2 \ll p+q$ in general.

3.1 A 2-WAY-CCC approach

We discuss some properties on k-edge-connectivity changes, and then give the algorithm.

Lemma 1 *Let $\overline{\mathsf{g}}_t$ be the sequence of single-edge-change graphs, $\overline{\mathsf{g}}_t = (g_1, \cdots, g_p, g_{p+1}, \cdots, g_{p+q})$ from G_{t-1} to G_t. Let $g_0 = G_{t-1}$, g_i is a graph either (1) by inserting a new edge e that does not appear in g_{i-1} or (2) by deleting an existing edge e from g_{i-1}. For any pair of vertices u and v in g_i, $\delta_i(u, v)$ (refer to (2)) increases by 1 at most for case (1) and decreases by 1 at most for case (2).*

Proof sketch We prove the edge-deletion case by contradiction. If an edge e is deleted from graph g_{i-1}, for any pair of vertices u and v, it is obvious that the connectivity cannot increase. Let $\mathsf{econ}_{i-1}(u, v) = k$ in g_{i-1}. Assume that $\mathsf{econ}_i(u, v) = k-2$ at most in g_i after e deleted from g_{i-1}. There exists an edge-cut set $E_c = \{e_1, \cdots, e_{k-2}\}$. If we delete all $k-2$ edges in E_c from g_i, u and v will be disconnected. Otherwise, u and v are $(k-1)$-edge-connected at least in g_i. This is contradict with the assumption. We add the deleted edge e into E_c and obtain another edge-cut set E'_c of $k-1$ edges. If we delete all edges in E'_c from g_{i-1}, u and v will be disconnected. This is contradict with the fact that u and v are k-edge-connected in g_{i-1}. Similarly, the edge-insertion case can be proved. □

Lemma 1 ensures that the k-edge-connectivity for any pair of vertices u and v is affected by 1 at most when there is an edge change. Next, we investigate whether the order of edge-changes (deletions/insertions) will affect computing $\Delta_t(u, v)$. Reconsider the sequence of single-edge-change graphs, $\overline{\mathsf{g}}_t = (g_1, \cdots, g_p, g_{p+1}, \cdots, g_{p+q})$, from G_{t-1} to G_t. $\overline{\mathsf{g}}_t$ corresponds a sequence of edge changes, namely, $\Delta E_t = \Delta E_t^d \oplus \Delta E_t^a$, where $\Delta E_t^d = (e_1^{d_t}, \cdots, e_p^{d_t})$ represents a sequence of edge-deletions, and $\Delta E_t^a = (e_1^{a_t}, \cdots, e_q^{a_t})$ represents a sequence of edge-insertions. Note $\oplus$ indicates an operator that concatenates two sequences. Based on ΔE_t^d and ΔE_t^a, we consider three graphs in sequence, (G_a, G_b, G_c), for two consecutive graphs, G_{t-1} and G_t, in G. Here, $G_a = G_{t-1}$, $G_c = G_t$, and G_b is constructed by deleting all the edges in ΔE_t^d from G_a. As a sequence, G_c is a graph by inserting all edges in ΔE_t^a into G_b. We show that the order of edge-deletions among those in ΔE_t^d will not affect cumulated connectivity change between G_a and G_b, and the order of edge-insertions among those in ΔE_t^a will not affect cumulated connectivity change computing between G_b and G_c.

Theorem 2 *Consider ΔE_t^d as a set of edge-deletions that transfers G_a to G_b. Assume S_1 and S_2 be two different sequences of ΔE_t^d to transfer G_a to G_b. Let $\Delta^1_{a,b}(u, v)$ be the*

cumulated connectivity change $\Delta_{a,b}(u, v)$ *when* G_b *is obtained from* G_a *using* S_1*, and let* $\Delta^2_{a,b}(u, v)$ *be the cumulated connectivity change* $\Delta_{a,b}(u, v)$ *when* G_b *is obtained from* G_a *using* S_2*.* $\Delta^1_{a,b}(u, v) = \Delta^2_{a,b}(u, v)$*. In a similar fashion, consider* ΔE^a_t *as a set of edge-insertions that transfers* G_b *to* G_c*. Assume* S_3 *and* S_4 *be two different sequences of* ΔE^a_t *to transfer* G_b *to* G_c*. Let* $\Delta^3_{b,c}(u, v)$ *be the cumulated connectivity change* $\Delta_{b,c}(u, v)$ *when* G_c *is obtained from* G_b *using* S_3*, and let* $\Delta^4_{b,c}(u, v)$ *be the cumulated connectivity change* $\Delta_{b,c}(u, v)$ *when* G_c *is obtained from* G_b *using* S_4*.* $\Delta^3_{b,c}(u, v) = \Delta^4_{b,c}(u, v)$*.*

Proof sketch It can be shown based on Lemma 1 that the value of connectivity for any two vertices, u and v, decreases (increases) monotonously for a set of edge-deletions (edge-insertions). □

Given Theorem 2, we know both $\Delta_{a,b}(u, v)$ and $\Delta_{b,c}(u, v)$ are independent of the order of edge-deletions and edge-insertions, and are only related to the intermediate graph G_b. We can compute $\Delta_t(u, v)$ between G_{t-1} and G_t as $\Delta_{a,b}(u, v) + \Delta_{b,c}(u, v)$ between G_a and G_b and between G_b and G_c. It is worth noting that $\Delta_{a,b}(u, v)$ and $\Delta_{b,c}(u, v)$ are order insensitive among all edge-deletions and edge-insertions, respectively.

Next, we prove our objective function (4) (or (9)) satisfies Property 1.

Lemma 2 *Our objective function* (4) *(or* (9)*) satisfies Property* 1.

Proof sketch According to (4), $F(G_s)$ consists of three parts: $\Delta(G_s)$, $\alpha(G_s)$ and $\beta(G_s)$. Given any two graphs G_a and G_c. Let $\mathsf{G}_1 = (G_a, G_c)$ and $\mathsf{G}_2 = (G_c, G_a)$. Furthermore, let $\Delta_1(G_s)$, $\alpha_1(G_s)$, and $\beta_1(G_s)$ be the values computed for G_1, and $\Delta_2(G_s)$, $\alpha_2(G_s)$, and $\beta_2(G_s)$ be the values computed for G_2. Because $\alpha_i(G_s)$ and $\beta_i(G_s)$ are only affected by the edge-changes between G_a and G_c, for $i = 1, 2$, we have $\alpha_1(G_s) = \alpha_2(G_s)$ and $\beta_1(G_s) = \beta_2(G_s)$. We only need to prove $\Delta_1(G_s) = \Delta_2(G_s)$. Following Theorem 2, we can construct G_{b_1} as an intermediate graph for G_1 and construct G_{b_2} as an intermediate graph for G_2. Obviously, $G_{b_1} = G_{b_2}$. Because $\Delta_i(G_s)$ is only related to the intermediate graph G_{b_i}, for $i = 1, 2$, we have $\Delta_1(G_s) = \Delta_2(G_s)$ and Lemma 2 is proved. □

The 2-WAY-CCC algorithm for computing $\Delta(u, v)$ for every pair of vertices in G is shown in Algorithm 1.

The complexity analysis is given below. For 2-WAY-CCC, we need to compute edge-connectivity for every pair of vertices in graph $G(V, E)$ using max-flow algorithm twice at each time step. The Push-Relabel method with time complexity $O(|V|^2 \cdot |E|)$ [5] is the asymptotic fastest in all max-flow algorithms. In fact, by Edmonds–Karp Algorithm, the time complexity is $O(|E| \cdot |f^*|)$ [5], where $|f^*|$ is the max-flow between two vertices. In our problem, we consider the capacity of each edge as 1. The max-flow between u and v must be less than $\min(d(u), d(v))$, where $d(u)$ is the degree of vertex u. Thus, the max-flow is less than $|V|$. Thus, the time complexity of computing edge-connectivity is less than $O(|V| \cdot |E|)$. Gomory and Hu state that edge-connectivity of all pairs of vertices in an undirected graph can be computed using $|V| - 1$ max-flow computations [13]. At each time step, the time complexity of 2-WAY-CCC is $O(|V|^2 \cdot |E|)$. We need two matrices to maintain the edge-connectivity

Algorithm 1 2-WAY-CCC (G)

Input: $\mathsf{G} = (G_1, G_2, \cdots, G_{\|\mathsf{G}\|})$.
Output: compute $\Delta(u, v)$ for every pair of vertices in G.

1: $\Delta(u, v) \leftarrow 0$ for every u and v in G;
2: **for** $t = 2$ to $\|\mathsf{G}\|$ **do**
3: let $G_a \leftarrow G_{t-1}$, $G_c \leftarrow G_t$;
4: let G_b be the graph by deleting all edges in ΔE_t^d from G_a;
5: **for** each pair of vertices $u, v \in \mathsf{G}$ **do**
6: compute $\Delta_{a,b}(u, v)$;
7: compute $\Delta_{b,c}(u, v)$;
8: $\Delta_t(u, v) \leftarrow \Delta_{a,b}(u, v) + \Delta_{b,c}(u, v)$;
9: $\Delta(u, v) \leftarrow \Delta(u, v) + \Delta_t(u, v)$;

and the cumulated connectivity change for every pair. The space complexity is $O(|V|^2)$.

4 Find MFCC

In this section, we give our algorithm to find MFCC. It is important to note that G_s ($\subseteq \overline{\mathsf{G}}$) with the max $F(G_s)$ is a connected tree by Theorem 1.

For a given G_s, to compute $F(G_s)$ (4), there are three main components, $\Delta(G_s)$ (5), $\alpha(G_s)$ (6) and $\beta(G_s)$ (7). Most of the components are already computed when computing CCC. In Section 3, we discussed how to compute $\Delta(u, v)$ (3), with which $\Delta(G_s)$ can be easily computed. $\beta(G_s)$ consists of E_s and $\beta_c(G_s)$, where $\beta_c(G_s)$ is only related to the cpath-components in $\overline{\mathsf{G}}$. We can compute $\beta(G_s)$ using (8). Also, it is straightforward to compute $\alpha(u, v)$ which can be done while computing $\Delta(u, v)$. Note that the value of $\alpha(u, v)$ for any edge in G_s is only related to the times of the edge no change.

A $G_s(V_s, E_s)$ is constructed as a tree for a given subset of vertices V_s (Theorem 1). Given a subset of vertices V_s in $\overline{\mathsf{G}}$, $\Delta(G_s)$, $\beta_c(G_s)$ and $|E_s|$ are determined. Maximizing $F(G_s)$ ((4) or (9)) on V_s, for a subgraph $G_s(V_s, E_s)$, is equivalent to minimizing $\alpha(G_s) = \sum_{(u,v)\in E_s} \alpha(u, v)$. Therefore, we treat $\overline{\mathsf{G}}$ as an edge-weighted graph. Every edge $(u, v) \in \overline{\mathsf{G}}$ is associated with a weight $\alpha(u, v)$. Our problem becomes to find a minimum spanning tree on V_s in terms of $\alpha(u, v)$.

We give two properties regarding minimum spanning tree which will be used later in our algorithm. Below, for a given graph G, we use $G - \{v\}$ to denote an induced subgraph of G after deleting the vertex v from G, and use $G + \{v\}$ to denote a min induced subgraph of $\overline{\mathsf{G}}$ containing G and an additional vertex v.

Lemma 3 *Given an edge-weighted graph G, let its minimum spanning tree be T. If v is any a leaf vertex of T, the minimum spanning tree for $G - \{v\}$ is $T - \{v\}$.*

Proof sketch Note that there may exist more than one minimum spanning tree, say T_1 and T_2 for G. The two trees can be different but both trees must have the same total minimum value, $\sum_{u,v\in T_1} w(u, v) = \sum_{u,v\in T_2} w(u, v)$. Lemma 3 suggests that any

of the minimum spanning trees can be T. We prove it by contradiction. Suppose there exists another minimum spanning tree T' for $G - \{v\}$. We insert the edge (u, v) whose weight is the minimum among all the edges connecting to vertex v into T'. Then, $\sum_{(u',v')\in T'+\{v\}} w(u', v')$ becomes less than $\sum_{(u',v')\in T} w(u', v')$, T is not the minimum spanning tree of G. This leads to the contradiction. □

Lemma 4 *Given a graph G, let its minimum spanning tree be T. Suppose that deleting a vertex v from G divides G into l connected components $G_i = (V_i, E_i)$, for $1 \le i \le l$, where every two G_i and G_j for $i \ne j$ are disconnected. Then, the minimum spanning tree of $G_i + \{v\}$ is $T_i + \{v\}$. Here, $G_i + \{v\}$ represents the induced subgraph of G on the set of vertices $V_i + \{v\}$, and $T_i + \{v\}$ represents the induced subtree of T on the set of vertices $V_i + \{v\}$.*

Lemma 4 can be proved in a similar way as to prove Lemma 3.

The algorithm Our new algorithm, called FIND-MAX, is given in Algorithm 2. There are two main phases.

In the first phase, we collect information to compute $\Delta(u, v)$, $\alpha(u, v)$, $\beta(G_s)$, and construct $\overline{\mathsf{G}}$, for a given evolving graph G (lines 1 and 2). we use L_Δ to denote the set of $\{\Delta(u, v)\}$ for every pair of different u and v in G. It can be computed using 2-WAY-CCC(G) as discussed in the previous section. We use L_α to denote the set of $\{\alpha(u, v)\}$ for every edge (u, v) in G. Note that 2-WAY-CCC(G) needs to scan the entire G once. While doing that, we can compute $\alpha(u, v)$ for every edge (u, v) in G. Also, we use L_{PC} to denote the set of $\{\mathsf{PC}_i(\overline{\mathsf{V}})\}$ for all cpath-components in $\overline{\mathsf{G}}$. We show how to do it below. For a pair of graphs, $G_{t-1}(\mathsf{V}, E_{t-1})$ and $G_t(\mathsf{V}, E_t)$, in $\mathsf{G}(\mathsf{V}, \mathsf{E})$, we construct a delta graph $\Delta G_t(\mathsf{V}, \Delta E_t)$ where $\Delta E_t = (E_{t-1} \setminus E_t) \cup (E_t \setminus E_{t-1})$. Then, we construct $\Delta G = \bigcup_{t=2}^{\|\mathsf{G}\|} \Delta G_t$. Every edge in ΔG changes at least once in G. And every connected component in ΔG must be a cpath-component for the vertices in G. We compute all PCs for $\overline{\mathsf{G}}(\overline{\mathsf{V}}, \overline{\mathsf{E}})$. Every two vertices, u and v, in the same $\mathsf{PC}_i(\overline{\mathsf{V}})$, are $u \stackrel{c}{=} v$. Note that we use $\mathsf{PC}_i(\overline{\mathsf{V}})$ to indicate PC_i is found over $\overline{\mathsf{V}}$. By L_{PC} and (8), we can compute $\beta(G_s)$ easily.

Algorithm 2 FIND-MAX (G)

Input: G.
Output: the subgraph G_s with the max $F(G_s)$.

1: Let L_Δ, L_{PC}, and L_α be the sets of $\Delta(u, v)$, $\beta(u, v)$, $\alpha(u, v)$ for every pairs of u and v in G;
2: compute L_Δ, L_{PC}, L_α using 2-WAY-CCC (G) and obtain $\overline{\mathsf{G}}$ at the same time;
3: let G_s be a spanning tree with the min $\alpha(G_s)$ of $\overline{\mathsf{G}}$;
4: let $\mathcal{T} \leftarrow \emptyset$;
5: $G_s \leftarrow$ FIND-TREE (G_s, L_Δ, L_{PC}, L_α, $\mathcal{T}(\emptyset)$);
6: **if** $G_s = \emptyset$ and $\mathcal{T} \ne \emptyset$ **then**
7: $\quad G_s \leftarrow$ TREE-TRAVERSE ($\mathcal{T}$, L_Δ, L_{PC}, L_α);
8: **return** G_s;

In a summary, we construct the universal graph $\overline{\mathsf{G}}(\overline{\mathsf{V}}, \overline{\mathsf{E}})$ for G, and we treat it as a weighted graph where each edge, (u, v), is associated with a weight $w(u, v) = \alpha(u, v)$. In addition, we have L_Δ and L_{PC}.

In the second phase, we find the connected subtree G_s of $\overline{\mathsf{G}}$ with the max $F(G_s)$ (lines 3–7). We start by finding the minimum spanning tree with the min $\alpha(\overline{\mathsf{G}})$ for all the vertices in $\overline{\mathsf{G}}$ (line 3). Then we find the minimum spanning tree (in terms of α) over the selected subset of vertices, V_s, of $\alpha(G_s)$. Such a subset V_s is formed by removing a vertex from $\overline{\mathsf{G}}$ in an one-by-one fashion. While removing a vertex from a subset of vertices V_s one-by-one, we pay attention to two things: (1) the vertex removal will not miss the subtree G_s with the max $F(G_s)$, and (2) such a subtree G_s must be connected. This is done using a data structure called a partition-tree in two steps: partition-tree construction and partition-tree traversal. In the partition-tree construction, we remove vertices that cannot be included in the final G_s with the max $F(G_s)$, and maintain some G_s that can possibly be the final answer. In the partition-tree traversal, we explore combinations of those maintained in the partition-tree, and compute the final G_s with the max $F(G_s)$.

As shown in Algorithm 2, we call Find-Tree with G_s (a min spanning tree of $\overline{\mathsf{G}}$ in terms of $\alpha(G_s)$) to start the vertex removal process (line 5). Find-Tree will return either a non-empty G_s, which is the answer with the max $F(G_s)$, or an empty G_s. When Find-Tree returns an empty G_s, Find-Tree will return a non-empty partition-tree $\mathcal{T}$ constructed by Find-Tree. With the partition-tree $\mathcal{T}$, we find the subtree G_s with the max $F(G_s)$ by calling Tree-Traverse (line 7).

4.1 Find-Tree

The Find-Tree algorithm is given in Algorithm 3. We compute the value of $\phi_s = F(G_s)$, where G_s, an input to Find-Tree, is a minimum spanning tree (in terms of α). We use $\mathcal{X}$ to maintain the set of vertices such that we cannot obtain a minimum spanning tree (connected) over $G_s - \{x\}$ if $x \in \mathcal{X}$. Such a vertex is called a cut-vertex. Next, we remove a vertex from G_s one-by-one in a loop (lines 3–15). In the loop, for each vertex $u \notin \mathcal{X}$ (which means that removing u may construct a connected subtree), we remove u from G_s. Let the tree after removing u from G_s be G_u. There are two cases when removing u from G_s.

Case 1 (u is a leaf node of G_s): We do not need to compute the minimum spanning tree on $V_s - \{u\}$, because we know the minimum spanning G_u is $G_s - \{u\}$ based on Lemma 3. We compute $\Delta(G_u)$ and $\beta(G_u) = \beta(G_s) - \sum_{u \notin \mathsf{PC}_i(\overline{\mathsf{V}})} \mathsf{PC}_i(\overline{\mathsf{V}}) - 1$, respectively, and then compute $F(G_u)$.

Case 2 (u is not a leaf node of G_s): We try to find the minimum spanning tree G_u over $V_u = V_s - \{u\}$, and compute $\alpha(G_u)$. If G_u is connected, we compute $F(G_u)$ as we do in Case 1. If G_u is not connected, we let ϕ_u be $-\infty$ and record this information by inserting u into $\mathcal{X}$ (line 9).

After attempting to remove every vertex $u \in G_s$ (lines 4–9), we find the subtree with the max $F(\cdot)$ among all possible G_u, and denote it as $G_{\max}$ with $\phi_{\max}$ $(= F(G_{\max}))$ (line 10). We compare $\phi_{\max}$ (after removal of a vertex) with ϕ_s of G_s. Let $\phi_\delta = \phi_{\max} - \phi_s$. If $\phi_\delta \geq 0$, it implies that $F(\cdot)$ increases by removing a vertex from G_s, and we

Algorithm 3 FIND-TREE (G_s, L_Δ, L_{PC}, L_α, $\mathcal{T}(\bullet)$)

Input: $G_s \subseteq \overline{\mathsf{G}}$, L_Δ (a set of $\Delta(u, v)$ for every pair of u and v in $\overline{\mathsf{G}}$),
L_{PC} (a set of $\beta(u, v)$ for every pair of u and v in $\overline{\mathsf{G}}$),
L_α (a set of $\alpha(u, v)$ for every pair of u and v in $\overline{\mathsf{G}}$), and
$\mathcal{T}(\bullet)$ (a partition tree with a node $\bullet$ marked)).
Output: the subgraph G_s with the max $F(G_s)$.

1: $\phi_s \leftarrow F(G_s)$;
2: $\mathcal{X} \leftarrow \emptyset$;
3: **repeat**
4: **for** every vertex $u \in G_s$ and $u \notin \mathcal{X}$ **do**
5: let G_u be a tree by removing u from G_s;
6: **if** G_s is connected **then**
7: $\phi_u \leftarrow F(G_s)$;
8: **else**
9: $\phi_u \leftarrow -\infty$; $\mathcal{X} \leftarrow \mathcal{X} \cup \{u\}$;
10: let $\phi_{\max}$ be the ϕ_u that has the max value and $G_{\max}$ be the corresponding G_u;
11: $\phi_\delta \leftarrow \phi_{\max} - \phi_s$;
12: **if** $\phi_\delta \geq 0$ **then**
13: $G_s \leftarrow G_{\max}$; $\phi_s \leftarrow \phi_{\max}$;
14: $\mathcal{X} \leftarrow \emptyset$;
15: **until** $\phi_\delta \leq 0$
16: **if** $\mathcal{X} = \emptyset$ and $\mathcal{T} = \emptyset$ **then**
17: **return** G_s;
18: **if** $\mathcal{X} \neq \emptyset$ **then**
19: $(x, G_{s_1}, G_{s_2}) \leftarrow$ CUT-NODE (G_s, L_Δ, L_{PC});
20: create a new node $\odot$ labeled with a triple (x, G_s, ϕ_s);
21: **if** $\mathcal{T} = \emptyset$ **then**
22: let $\odot$ be the root of $\mathcal{T}$;
23: **else**
24: add $\odot$ as a child of the marked node $\bullet$ in $\mathcal{T}$;
25: FIND-TREE (G_{s_1}, L_Δ, L_{PC}, L_α, $\mathcal{T}(\odot)$);
26: FIND-TREE (G_{s_2}, L_Δ, L_{PC}, L_α, $\mathcal{T}(\odot)$);
27: **if** $\mathcal{X} = \emptyset$ and $\mathcal{T} \neq \emptyset$ **then**
28: create a new node $\odot$ labeled with a triple $(\emptyset, G_s, \phi_s)$;
29: add $\odot$ as a leaf node of the marked node $\bullet$ in $\mathcal{T}$;
30: **return** $\emptyset$;

replace G_s and ϕ_s with $G_{\max}$ and $\phi_{\max}$, respectively (line 13). In addition, we reset $\mathcal{X}$ to be empty, because in the next iteration in the loop, removing those vertices in $\mathcal{X}$ may possibly result in a connected subtree (line 14).

We repeat the vertex removal process until we cannot remove any vertices (lines 3–15). It is worth noting that after this loop, we cannot remove any vertex u further to make $\phi_u \geq \phi_s$. At this stage, $\mathcal{X} = \emptyset$ means the current G_s have no cut-vertex, and $\mathcal{T} = \emptyset$ means that the vertex removal process will always result in a connected subtree in the removal process until now. Note that $\mathcal{T}$ is passed by an input to FIND-TREE, and FIND-TREE is a recursive procedure. If both $\mathcal{X}$ and $\mathcal{T}$ are empty, then the current G_s is the answer, and we return it (lines 16 and 17).

When $\mathcal{X} \neq \emptyset$, let $\mathcal{X} = (x_1, \cdots, x_m)$ be a set of m cut-vertices. We select one vertex x in $\mathcal{X}$ to partition $G_s(V_s, E_s)$ into two subtrees G_{s_1} and G_{s_2} using a procedure Cut-Node, which we explain below.

Note on Cut-Node For any vertex x_i in $\mathcal{X}$, removing x_i cuts the current subgraph G_s into c_i connected components $G'_{i,1}, \cdots, G'_{i,c_i}$, where $G'_{i,j}$ for $1 \leq j \leq c_i$ does not contain x_i. Then we construct c_i connected components $G_{i,1}, \cdots, G_{i,c_i}$, where $G_{i,j} = G'_{i,j} + \{x_i\}$ for $1 \leq j \leq c_i$ is a connected subtree that contains x_i. For each component $G_{i,j}$, we find its minimum spanning tree, and compute $F(\cdot)$. Based on Lemma 4, we know the minimum spanning tree of $G_{i,j}$ can be found from the minimum spanning tree of G_s. We denote $G_{i,\max}$ as $G_{i,j}$ with the max $\phi_{i,\max}$ among $\phi_{i,j} = F(G_{i,j})$ for $1 \leq j \leq c_i$. We do the same for every cut-vertex x_i for $1 \leq i \leq m$ in $\mathcal{X}$. We select the vertex x_i whose $\phi_{i,\max}$ is the max among all $\{\phi_{j,\max}\}$ for every $x_j \in \mathcal{X}$, as the cut-vertex. We denote the cut-vertex as x $(= x_i)$. Then, we create a new node in the partition tree, denoted as $\odot$, labeled with a triple (x_i, G_s, ϕ_s), and insert it as a child of the marked node $\bullet$ in the partition tree $\mathcal{T}$ (passed as an input to Find-Tree). If $\mathcal{T}$ is empty, we treat $\odot$ as the root of $\mathcal{T}$. In addition, we create two subtrees, G_{s_1} and G_{s_2}, where $G_{s_1} = G_{i,\max}$ if x is x_i and $G_{s_2} = \bigcup G_{i,j}$ for $1 \leq j \leq c_i$ and $j \neq i$.

With the result of Cut-Node, we call Find-Tree recursively twice using G_{s_1} and G_{s_2} (lines 25 and 26). The partition-tree $\mathcal{T}$ is a binary tree to be constructed by Find-Tree recursively. The leaf node of $\mathcal{T}$ is added when $\mathcal{X} = \emptyset$ and $\mathcal{T} \neq \emptyset$ (lines 27–29).

4.2 Tree-Traverse

In Find-Max, Tree-Traverse computes G_s with the max $F(G_s)$ using the partition-tree $\mathcal{T}$, where each node is associated with a triple initially. We traverse $\mathcal{T}$ in a bottom-up fashion, and process the non-leaf node with two leaf nodes first at a time. A non-leaf node with two leaf nodes will become a leaf node in the next step. During the traversal, a leaf node may have a set of triples: $\{(\emptyset, G_{q_1}, \phi_{q_1}), (\emptyset, G_{q_2}, \phi_{q_2}), \cdots\}$, if all ϕ_{q_i} are the max and are the same. In general, a non-leaf node, t_s, in $\mathcal{T}$ has a triple associated with (s, G_s, ϕ_s), and has two child nodes, t_q and t_r, where t_q and t_r are associated with two sets of triples $\mathcal{Q} = \{(\emptyset, G_{q_1}, \phi_{q_1}), (\emptyset, G_{q_2}, \phi_{q_2}), \cdots\}$ and $\mathcal{R} = \{(\emptyset, G_{r_1}, \phi_{r_1}), (\emptyset, G_{r_2}, \phi_{r_2}), \cdots\}$. We update (s, G_s, ϕ_s) to be a set $\mathcal{S}$ of triples, $\mathcal{S} = \{(\emptyset, G_{s_1}, \phi_{s_1}), (\emptyset, G_{s_2}, \phi_{s_2}), \cdots\}$. Initially, $\mathcal{S} = \{(s, G_s, \phi_s)\} \cup \mathcal{Q} \cup \mathcal{R}$.

We update $\mathcal{S}$ as follows. First, for any G_{q_i} and G_{r_j}, if one vertex in G_{q_i} is contained in neighbor set of G_{r_j} over $\overline{\mathsf{G}}$ (i.e., $G_{q_i} \cup G_{r_j}$ is connected in $\overline{\mathsf{G}}$), we consider a new triple $(\emptyset, G'_s, \phi'_s)$ in $\mathcal{S}$ among three subtrees: (a) $G_{\text{new}} = G_{q_i} \cup G_{r_j}$, (b) G_{q_i}, and (c) G_{r_j}. For (a), we compute $\phi_{\text{new}} = F(G_{\text{new}})$. By Lemma 4, ϕ_{new} can be computed directly, and there is no need to find minimum spanning tree again. We select G_{new} with ϕ_{new} as G'_s with ϕ'_s and add it into $\mathcal{S}$ if ϕ_{new} is $\max\{\phi_{\text{new}}, \phi_{q_i}, \phi_{r_j}\}$ among (a), (b), and (c). Second, we remove all the triples $(, G_{s_i}, \phi_{s_i})$ from $\mathcal{S}$ if ϕ_{s_i} is less than any other ϕ_{s_j} for a triple $(, G_{s_j}, \phi_{s_j})$ in $\mathcal{S}$.

The final resulting G_s in this process is any G_{s_i} from the set of triples $\{(\emptyset, G_{s_1}, \phi_{s_1}), (\emptyset, G_{s_2}, \phi_{s_2}), \cdots\}$ that are associated with the root of the partition-tree $\mathcal{T}$ at the end of the process.

4.3 An example

We explain it using an example (Figure 4). Here, $\gamma = 1$. Figure 4a shows $\overline{\mathsf{G}}$. The edges marked by "$\|$" mean that they change twice (the edges are deleted and then inserted). Figure 4b shows the minimum spanning tree G_s of $\overline{\mathsf{G}}$. Then, we want to know if there is any subtree of G_s with a larger $F(\cdot)$. Let G'_s be a subtree of G_s by removing one vertex from $\{3, 4, 6, 7, 9, 10\}$. Because $F(G'_s) < F(G_s)$, we cannot remove any of such vertices. Next, we consider removing vertices from $\mathcal{X} = \{1, 2, 5, 8\}$. If we remove any one vertex from $\mathcal{X}$, the resulting G'_s is not connected. A vertex in $\mathcal{X}$ can be a cut-vertex. Figure 4c shows the partition-tree $\mathcal{T}$ to be constructed by FIND-TREE. For $\mathcal{T}$ construction, here the vertex 2 is the cut-vertex to cut G_s into two subtrees G_{s_1} and G_{s_2}. It is because the subtree G_{s_1} over $\{2, 3, 4\}$ is with the max $F(\cdot)$ among all subtrees if we remove any vertex from $\mathcal{X}$ to cut G_s. G_{s_2} is the subtree of G_s over the vertices $\{1, 2, 5, 6, 7, 8, 9, 10\}$. Here, the root of the partition-tree is a triple $(2, G_s, F(G_s))$, where 2 is the cut-vertex to cut G_s. The root has two children. One points to G_{s_1} and the other points to G_{s_2}. Next, consider G_{s_2} which is over the vertices $\{1, 2, 5, 6, 7, 8, 9, 10\}$. We can remove the vertex 2 from G_{s_2}, because the resulting subtree without the vertex 2 is connected and is with a larger $F(\cdot)$. Let G_s be a subtree over $\{1, 5, 6, 7, 8, 9, 10\}$. In a similar fashion, the cut vertex 8 is selected, which results in two subtrees. The completed partition-tree $\mathcal{T}$ is shown in Figure 4c.

In TREE-TRAVERSE, with $\mathcal{T}$ traversal, we find the subtree with max $F(\cdot)$ in a bottom-up manner. Reconsider Figure 4c. The node in $\mathcal{T}$ labeled ⑧ is associated with a triple $(8, G_s, \phi_s)$, where G_s is the subtree over the vertices $\{1, 5, 6, 7, 8, 9, 10\}$ and $\phi_s = F(G_s)$. The node ⑧ has two child nodes. The left is for a subtree G_{q_1} over $\{8, 9, 10\}$ with $\phi_{q_1} = F(G_{q_1})$. The right is for a subtree G_{r_1} over $\{5, 6, 7\}$ with $\phi_{r_1} = F(G_{r_1})$. We also consider if merging G_{q_1} and G_{r_1} can result in a connected subtree with a larger $F(\cdot)$ value. Since G_{q_1} and G_{r_1} do not contain 8 (the cut vertex) together, they cannot be merged as a connected subtree. We update the information associated with ⑧ to include G_{q_1} with ϕ_{q_1} and G_{r_1} with ϕ_{r_1} and exclude G_s with ϕ_s, because $\phi_{q_1} = \phi_{r_1} > \phi_s$. This implies that G_{q_1} or G_{r_1} can be the final answer or be involved in the final answer. We repeat the same process for the root node ②. At the end, any of the three subtrees (indicated by three colors in Figure 4b) can be the final answer.

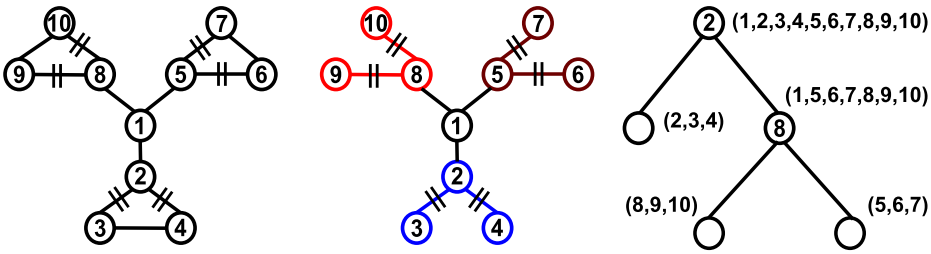


Figure 4 An example.

4.4 Time/space complexity

The dominating factor of FIND-MAX is FIND-TREE. We give the complexity for FIND-TREE. In the vertex removal process, for every vertex u, we compute ϕ_u using the minimum spanning tree G_u. The cost of maintaining minimum spanning tree is $O(\log|V|)$ [9]. Then one removal process take $O(|V|\log|V|)$ times. In the worst case, we need execute $|V|$ times for the vertex removal process. For $\mathcal{T}$ traversal, we only need scan every node in $\mathcal{T}$ once. Thus, the time complexity of FIND-TREE algorithm is $O(|V|^2\log|V|)$. The space required for $\mathcal{T}$ is $O(|V|^2)$. It is because there are $|V|$ nodes in $\mathcal{T}$ at most and we need keep G_s for a node s. Thus, the space complexity is $O(|V|^2)$.

5 Performance studies

In this section, we evaluate the effectiveness and efficiency of our algorithms on three real-life graph datasets including CAIDA Anonymized Internet Traces Datasets, Amazon, and Slashdot. To investigate the effectiveness, we propose five distinct measurements. We also test efficiency and scalability of our method. All experiments were done on a 2.2 GHz Intel Pentium Dual core PC with 2 GB main memory, running Windows XP. All algorithms are implemented by Visual C++ 6.0.

We tested the following three real datasets.

CAIDA anonymized internet traces datasets We use 2 datasets: Chicago and Sanjose. These datasets contain anonymized passive traffic trace from CAIDA's passive monitors on Chicago and Sanjose (2008-7-17 to 2009-10-15). In these datasets, there are more than 400K IP-addresses. We generate an evolving graph as follows. We treat a set of IP-addresses as a subnet if they have the same first p bits. Each subnet is considered as a vertex and there is an edge between two subnets if any two IP-addresses in the two distinct subnets are connected. We generate evolving graphs, $\mathsf{G} = (G_1, G_2, \cdots)$, by generating graph G_t at different time. The size of vertices ranges from 5K to 20K. The length of an evolving graph ranges from 10 to 30. The details are shown in Table 1 for the 8 different evolving graphs. Each dataset is with a name, say Chi-1 (Chicago Number 1) or San-1 (Sanjose, Number 1).

Slahdot dataset (http://slashdot.org/) Slashdot is a technology related news website known for its specific user community. The website features user-submitted and editor-evaluated current primarily technology oriented news. The network was obtained in 2009. The users are nodes, and an edge between u and v represents user u agrees with user v's comment. This dataset contains 82,140 nodes and 349,202 edges. We generate three different snapshots for an evolving graph and the size of vertices ranges from 20K to 80K. Some evolving graphs are shown in Table 1.

Amazon dataset (http://snap.stanford.edu/data/) This network is collected by crawling Amazon website. It is based on *Customers Who Bought This Item Also Bought* feature of the Amazon website. The nodes in this network represent products. If a product is frequently co-purchased with another product, the graph contains an edge between them. This dataset is collected from March to June in 2003. The whole graph includes 262,111 nodes and 1,234,877 edges. We generate five snapshots and the size

Table 1 Dataset characteristics.

Name	$\|V\|$	Max $\|E_t\|$	Min $\|E_t\|$	Avg $\|E_t\|$	Avg $\|\Delta E_t\|$	$\|\|\mathsf{G}\|\|$
Chi-1	5K	21K	20K	20K	421	10
Chi-2	10K	32K	32K	32K	545	10
Chi-3	10K	32K	31K	31K	526	15
Chi-4	10K	32K	31K	32K	577	20
Chi-5	10K	32K	31K	32K	553	25
Chi-6	10K	32K	31K	31K	532	30
Chi-7	15K	39K	37K	38K	663	10
Chi-8	20K	51K	49K	51K	688	10
San-1	5K	20K	19K	19K	398	10
San-2	5K	20K	19K	20K	402	15
San-3	5K	21K	19K	20K	417	20
San-4	5K	21K	19K	20K	385	25
San-5	5K	21K	19K	20K	396	30
San-6	10K	31K	30K	30K	511	10
San-7	15K	38K	36K	38K	597	10
San-8	20K	51K	48K	50K	624	10
Slashdot	20K	87K	86K	87K	627	3
Slashdot	80K	267K	263K	266K	1626	3
Amazon	20K	61K	59K	61K	749	5
Amazon	80K	251K	245K	249K	1892	5

of vertices also ranges from 20K to 80K. The description of some evolving graphs are shown in Table 1.

Let $G_s = (V_s, E_s)$ be the resulting MFCC (a subtree with the max $F(G_s)$ in $\overline{\mathsf{G}}$). We test the effectiveness of our approach using five measures as follows:

- $\mathcal{P}(G_s)$: The percentage of the number of change edges over the total number of edges in G_s, and is defined as $\mathcal{P}(G_s) = \frac{\text{Number of change edges in } G_s}{\text{Total number of edges in } G_s}$. The larger $\mathcal{P}(G_s)$ the better resulting G_s found.
- $A(G_s)$: The average number of edge changes in G_s and is defined as $A(G_s) = \frac{\sum_{e_i \in E_s} a_i}{|E_s|}$ where a_i is the number of the edge e_i change times in G. The larger $A(G_s)$ the better resulting G_s found.
- Percentage τ: Let $\overline{G_s}$ be the induced subgraph of V_s in $\overline{\mathsf{G}}$. τ measures the number of edge changes in $\overline{G_s}$ over that in $\overline{\mathsf{G}}$, and is defined as $\tau = \frac{\sum_{e_i \in \overline{G_s}} a_i}{\sum_{e_i \in \overline{\mathsf{G}}} a_i}$. This measure is essentially the fraction of edge change times among the vertices captured by our algorithm.
- Ratio λ: Consider $f(G_s) = \frac{\Delta(G_s)}{N(G_s)}$ that reflects the density of cumulated connectivity change, where $\Delta(G_s)$ is the cumulative connectivity change of all pairs of vertices in G_s and $N(G_s)$ is the number of pairs of vertices in G_s. λ measures the density of connectivity change of G_s compared with that of the universal graph $\overline{\mathsf{G}}$, and is defined as $\lambda = \frac{f(G_s)}{f(\overline{\mathsf{G}})}$.
- $\mathcal{H}(u)$ order: $\mathcal{H}(u) = \sum_v \Delta(u, v)$ is the affected connectivity count of vertex u. We sort all the vertices in $\overline{\mathsf{G}}$ in the descending order by $\mathcal{H}(u)$. We call it the $\mathcal{H}(u)$ order. The ratio of vertices with $\mathcal{H}(u)$ order measures the percentage of vertices in G_s that are in the top x % of $\mathcal{H}(u)$ order. This measure measures whether the edge-connectivity of vertices in G_s are affected most frequently.

It is worth noting that these measures are used to evaluate the MFCC we found, but cannot be used as an objective function to find such MFCC. It is because we will find only one edge or vertex from $\overline{\mathsf{G}}$ by these measures and this result is obvious meaningless.

Exp-1 $\mathcal{P}(G_s)$ and $A(G_s)$: We test $\mathcal{P}(G_s)$ and $A(G_s)$ using 12 datasets (refer to Table 2). The size of G_s found by our algorithm is also shown in Table 2. When the size of universal graph $\overline{\mathsf{G}}$ is 20K, for Chicago and Sanjose datasets, the size of G_s are 422 and 388 respectively. It is a small fraction of $\overline{\mathsf{G}}$. We see $\mathcal{P}(G_s) \geq 97$ % and $A(G_s)$ are always higher than other regions, $A(\overline{\mathsf{G}} - G_s)$. Take Chi-2 as an example. The subgraph G_s found is with 289 vertices among the total number of 10K vertices in the evolving graph G. The length of the evolving graph for Chi-2 is $\|\mathsf{G}\| = 30$. Therefore, the max number of changes per edge is 30. In the subgraph found for Chi-2, the average number of edge changes per edge is 25.8, and the average number of edge changes per edge for the remaining part of the evolving graph is 0.21. For other datasets, we also find the value of $A(G_s)$ is far more than $A(\overline{\mathsf{G}} - G_s)$. In addition, we study the size $|V_s'|$ of the MFCC when the objective function is $f(G_s) = \frac{\Delta(G_s)}{N(G_s)}$. In Table 2, we find that V_s' are always two vertices. It means G_s is always a single edge that change times is the most in G. This result is obvious meaningless.

Exp-2 $\mathcal{H}(u)$ order: We study $\mathcal{H}(u)$ order of vertices in G_s using 12 datasets. The ratio of vertices with $\mathcal{H}(u)$ order is shown in Table 3. From Table 3, we see that at least 85 % vertices in G_s appears in the top 10 % of $\mathcal{H}(u)$ order, and at least 93 % vertices in G_s appears in the top 20 % of $\mathcal{H}(u)$ order. For almost all the vertices in G_s, they are in the top 30 % of $\mathcal{H}(u)$ order. These results show that the edge-connectivity among vertices in subgraph G_s are affected most in whole graph G.

We also study the relationship between betweenness and cumulated connectivity change. Betweenness is an important centrality measure of a vertex in a way that vertices that occur on many shortest paths between other vertices

Table 2 Effectiveness: $\mathcal{P}(G_s)$ and $A(G_s)$.

Dataset	$\lvert V_s\rvert$	$\mathcal{P}(G_s)$	$A(G_s)$	$A(\overline{\mathsf{G}} - G_s)$	$F(G_s)$	$\lvert V_s'\rvert$
Chi-1	153	0.99	8.4	0.71	237.15	2
Chi-2	228	0.96	8.7	0.27	329.07	2
Chi-4	217	0.97	16.3	0.37	161.81	2
Chi-6	289	0.99	25.8	0.21	165.94	3
Chi-7	397	0.97	8.5	0.18	324.68	2
Chi-8	422	0.98	8.3	0.15	91.28	4
San-1	137	0.98	8.2	0.21	425.38	2
San-3	142	0.96	17.1	0.65	502.55	2
San-5	151	0.99	26.6	0.97	176.41	2
San-6	196	0.98	7.7	0.33	362.67	2
San-7	307	0.97	8.6	0.15	606.33	2
San-8	388	0.99	8.3	0.18	124.13	3
Slashdot	412	0.99	1.8	0.02	273.19	2
Slashdot	886	1	1.6	0.01	176.54	3
Amazon	537	0.99	3.7	0.06	338.98	2
Amazon	934	1	3.3	0.04	185.04	2

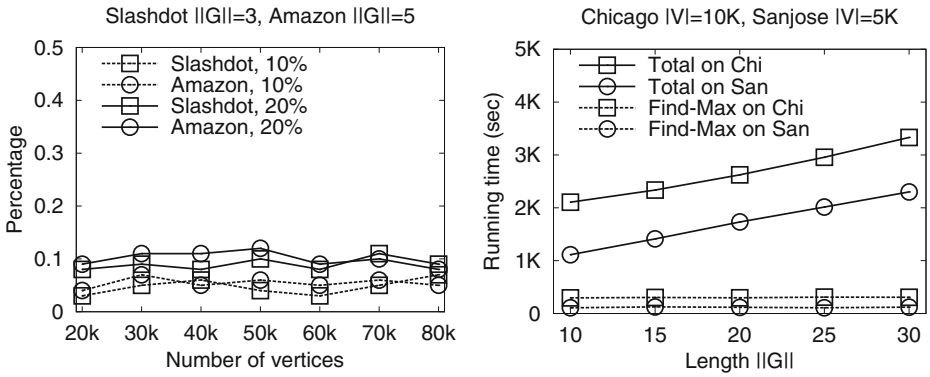
Table 3 Effectiveness: $\mathcal{H}(u)$.

	Top % of $\mathcal{H}(u)$ order				
Dataset	10 %	20 %	30 %	40 %	50 %
Chi-1-5K	0.85	0.93	0.99	1	1
Chi-2-10K	0.89	0.96	1	1	1
Chi-4-10K	0.88	0.96	1	1	1
Chi-6-10K	0.91	0.95	1	1	1
Chi-7-15K	0.95	0.99	1	1	1
Chi-8-20K	0.97	1	1	1	1
San-1-5K	0.82	0.94	0.98	1	1
San-3-5K	0.86	0.97	1	1	1
San-5-5K	0.87	0.95	1	1	1
San-6-10K	0.93	0.99	1	1	1
San-7-15K	0.94	0.97	1	1	1
San-8-20K	0.92	0.96	0.99	1	1
Slashdot-20K	0.96	1	1	1	1
Slashdot-80K	0.98	1	1	1	1
Amazon-20K	0.94	1	1	1	1
Amazon-80K	1	1	1	1	1

have higher betweenness. Given a graph $G = (V, E)$, for any vertex v, the betweenness of v is defined as follows. $C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through vertex v. We define the betweenness change of v for a graph sequence G as $\Delta_B(v) = \sum_2^{\|\mathsf{G}\|} |C_B^t(v) - C_B^{t-1}(v)|$, where $C_B^t(v)$ is the betweenness of v in snapshot G_t. Like $\mathcal{H}(u)$ order, $\mathcal{B}(u)$ order can be defined. We sort all vertices in descending order of $\Delta_B(v)$. We study how many vertices occur in the top x % of $\mathcal{H}(u)$ and $\mathcal{B}(u)$ order at the same time. Formally, we measure $T(x) = \frac{|\mathcal{H}(u) \cap \mathcal{B}(u)|}{|\mathcal{H}(u) \cup \mathcal{B}(u)|}$. For Slashdot and Amazon datasets, in Figure 5a, the value of $T(x)$ are always less than 10 % when x % $= 10$ % and x % $= 20$ %. This fact shows the vertices affected most by cumulated connectivity change are different from that affected most by betweenness change. Therefore, the MFCC found by cumulated connectivity change cannot be obtained by betweenness change.

Exp-3 Ratio λ of G_s to $\overline{\mathsf{G}}$: In Figure 6, we investigate the ratio λ by varying the number of vertices on datasets of Chicago, Sanjose, Slashdot, and Amazon. From Figure 6, we see all curves satisfy a common property: the ratio λ increases while the vertex size increases. In Figure 6a, the size of vertices ranges from 5K to 20K. When the number of vertices is 5K, the density of cumulated connectivity change of G_s is 100 times as much as $\overline{\mathsf{G}}$ for Chicago and Sanjose datasets respectively. When the size increases to 20K, the density of G_s is nearly 800 times as much as $\overline{\mathsf{G}}$ for Chicago and Sanjose datasets respectively. For Slashdot and Amazon datasets, in Figure 6b, λ is 500 when the number of vertices is 20K. We also find λ increases while the size increases. When the vertex size is 80K, λ is nearly 1,000. These results confirm the effectiveness of our approach for larger networks.

Exp-4 Ratio λ' of G_s to random graph G_r: Like λ, we define $\lambda' = \frac{f(G_s)}{f(G_r)}$. We test λ' of G_s using a random subgraph G_r of $\overline{\mathsf{G}}$ with r vertices. If $r < |V(G_s)|$, only vertices in G_s are randomly selected. If $r > |V(G_s)|$, in addition to G_s, G_r

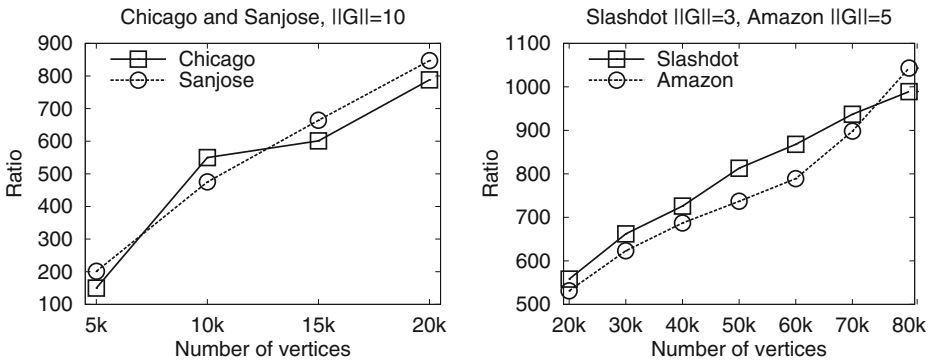


(a) Betweenness $\mathcal{B}(u)$ vs $\mathcal{H}(u)$ (b) Varying the length

Figure 5 Find-max testing.

contains extra vertices randomly selected in $\overline{\mathsf{G}}$ that are reachable from G_s. The random subgraph G_r is generated with distinct size: 50, 100, 200, 500, 1K, and 2K, respectively. We fix the size of the whole graph 20K for Chicago and Sanjose datasets, and fix the size 80K for Slashdot and Amazon. The results of this experiment are shown in Figure 7. For Chicago and Sanjose datasets, in Figure 7a, λ' is high when $r \geq 1{,}000$ and λ' is low when $r = 500$. This is because the size of G_s is nearly 500 and G_r is close to G_s when $200 \leq r \leq 500$. For Slashdot and Amazon, in Figure 7b, λ' is minimum at $r = 1{,}000$. This is also because G_r is close to G_s. For these four datasets, we observe λ' is always larger than 10. It means the density of cumulated connectivity change of G_s is always 10 times larger than random graph G_r. This fact indicates shrinking or expanding G_s will decrease the effectiveness of G_s.

Exp-5 Percentage τ of G_s to $\overline{\mathsf{G}}$: As shown in Figure 8a, for Chicago and Sanjose Datasets, the curves of τ increase marginally while the size of vertices increases from 5K to 20K. When the number of vertices is 20K, τ is larger



(a) Chicago and Sanjose (b) Slashdot and Amazon

Figure 6 Effectiveness: ratio λ of G_s to $\overline{\mathsf{G}}$.

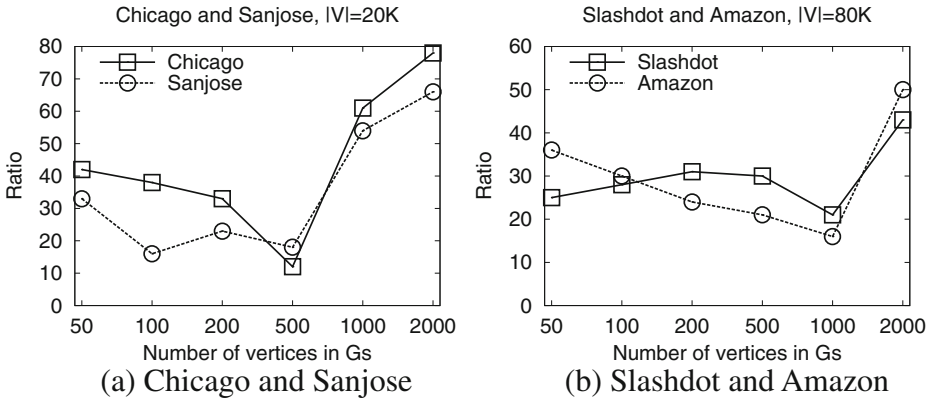


Figure 7 Ratio λ' of G_s to random graph G_r.

than 80 %. It states that 80 % of edge change times in $\overline{\mathsf{G}}$ is captured by a small fraction of vertices found by our approach. Similarly, in Figure 8b, for Slashdot and Amazon, τ also increases while the graph size increases from 20K to 80K. The edge change times among the vertices found by our approach is nearly 90 % of the edge change times among all vertices in $\overline{\mathsf{G}}$. These results confirm that our approach captures the important component where the edges change most frequently.

Exp-6 The impact of parameter γ: In Figure 9, we investigate the impact of parameter γ on Sanjose, Chicago, Amazon and Slashdot datasets. The number of vertices of these four datasets are 5K, 10K, 20K and 80K respectively. We fix the length of evolving graphs 30 for Sanjose and Chicago datasets and fix the length of evolving graphs 3 and 5 for Slashdot and Amazon datasets respectively. We vary the value of γ from 0 to 2.5. As shown in Figure 9a and c, we find that the size of G_s decreases with γ increasing. When $\gamma = 0$, the whole graph $\overline{\mathsf{G}}$ is found. When $\gamma = 2.5$, the most frequently changed edge

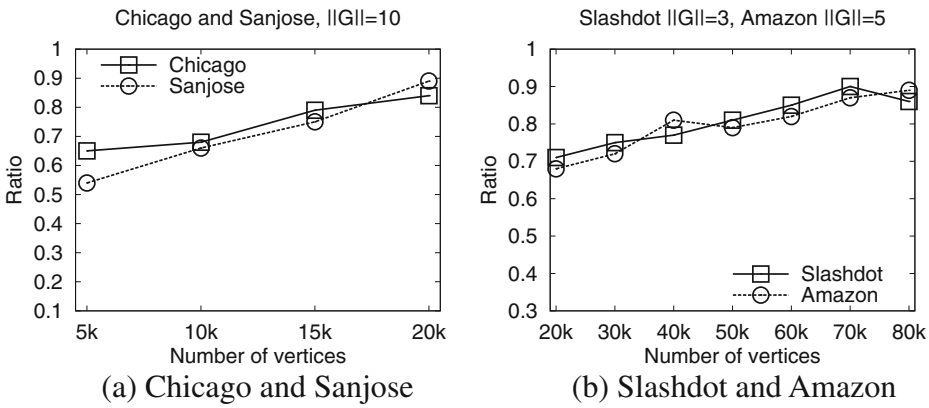


Figure 8 Effectiveness: τ.

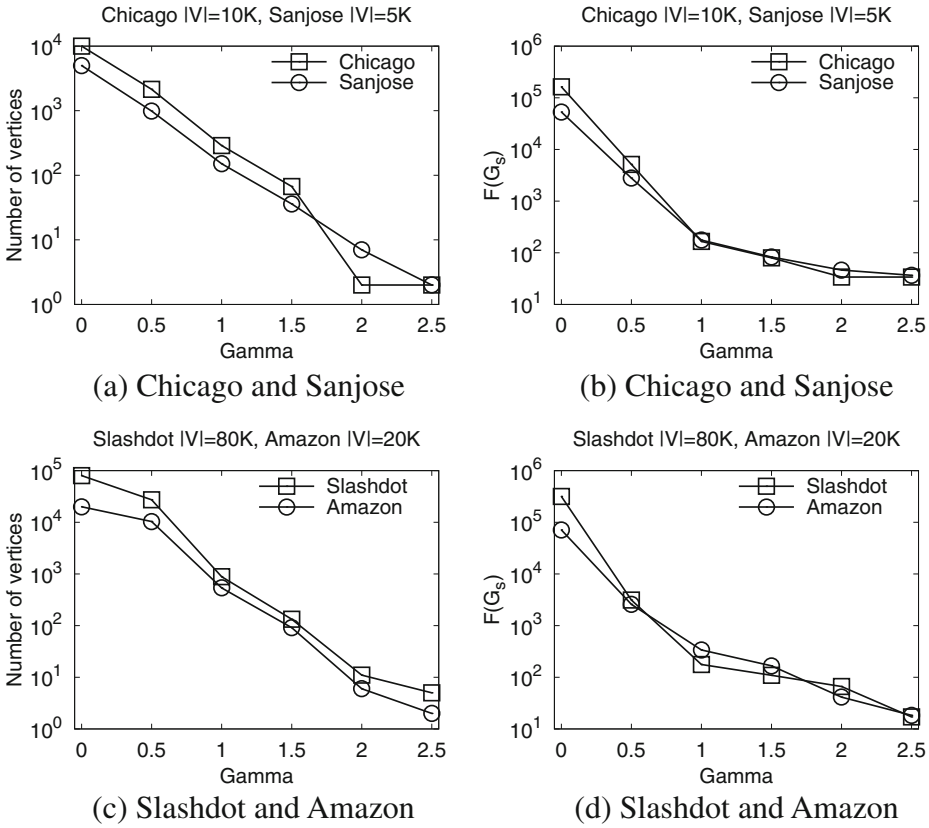


Figure 9 Impact of parameter γ.

is found. The more γ, the less number of vertex pairs in G_s that cannot be connected by a path consisting of change edges. These results state the size of G_s can be controlled by parameter γ. In addition, as shown in Figure 9b and d, we find the value of $F(G_s)$ is also decreases with γ increasing. When $\gamma = 0$, $F(G_s)$ essentially is $\Delta(G_s) - \alpha(G_s)$. Because the whole graph $\overline{\mathsf{G}}$ is found at this time, then $F(G_s)$ is $\Delta(\overline{\mathsf{G}}) - \alpha(\overline{\mathsf{G}})$, which is the largest for the curves in Figure 9c and d. Note that, for a fixed γ, the $F(G_s)$ for G_s found by our algorithm is maximum among all possible subgraphs in $\overline{\mathsf{G}}$, even though $F(G_s)$ decreases with γ increasing.

5.1 Efficiency results

Exp-7 $\Delta(u, v)$ computing: We first study the performance of computing CCC by comparing NAIVE-CCC and 2-WAY-CCC. We use the max-flow algorithm to compute k-edge-connectivity for every two different vertices for a given graph. In Figure 10, we vary the size of the vertices from 5K to 80K with the fixed length of the evolving graph 10.

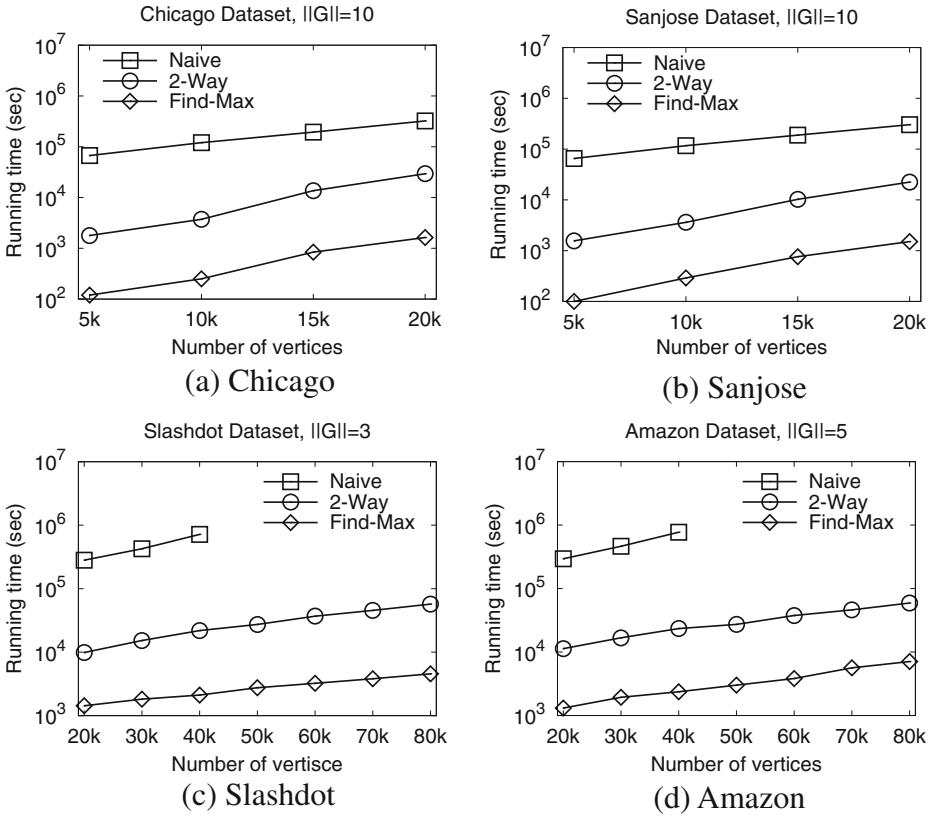


Figure 10 Varying the sizes of the vertices.

Figure 10a and b show the results using Chicago and Sanjose datasets respectively. When the number of vertices is about 5K, the computing time for NAIVE-CCC is in the order of 10^5 already. 2-WAY-CCC is nearly 100 times faster than NAIVE-CCC. Note that 2-WAY-CCC only needs to compute edge-connectivity twice for every G_{t-1} and G_t. To evaluate the scalability, as shown in Figure 10c and d, we test the computing time of our algorithm by varying the size of vertices from 20K to 80K on Slashdot and Amazon datasets. When the number of vertices is larger than 40K, the naive method cannot perform but 2-WAY-CCC perform well.

Exp-8 The Efficiency of FIND-MAX: In Figure 10, FIND-MAX indicates the time to compute lines 3–7 in Algorithm 2, the total time to compute Algorithm 2 is the time to compute 2-WAY-CCC plus FIND-TREE. For Chicago and Sanjose datasets, We test the computing time by varying the number of vertices with the fixed length 10 of the evolving graphs. As shown in Figure 10a and b, the computing time for FIND-MAX is in the order of 10^2 s, and can be done very efficiently. To evaluate scalability, in Figure 10c and d, we test the computing time of FIND-MAX on Slashdot and Amazon datasets. We vary the graph size from 20K to 80K. When the number of vertices is 80K, the computing time is

still in order of 10^3 s and it states excellent scalability of algorithm Find-Max. We also test computing time by varying the length of evolving graphs from 10 to 30 in Figure 5b. We use two datasets: Chi-2 and San-1. The number of vertices are fixed at 10K and 5K respectively. The computing time increases marginally, and is not sensitive to the length of evolving graphs.

6 Related work

In recent years, there are many works on evolving graph problem. Sun et al. in [28] present dynamic tensor analysis, which incrementally summarizes tensor streams (high-order graph streams) as smaller core tensor streams and projection matrices. Sun et al. in [27] propose GraphScope which discovers communities in large and dynamic graphs, as well as detects the changing time of communities. Tong et al. in [31] propose a family of Colibri methods to track low rank approximation efficiently over time. The authors in [15, 25, 26] propose distinct measure functions such as maximum common subgraph to detect when the graph changes. However, because utilizing these measure functions to compute all subgraphs in an evolving graph is infeasible, these works cannot answer where the changes occur frequently in an evolving graph.

There is an increasing interest in mining dynamic graphs. Borgwardt et al. in [3] apply frequent-subgraph mining algorithms to time series of graphs, and extract subgraphs that are frequent within the set of graphs. Bifet and Gavaldà in [2] present three closed tree mining algorithms to mining frequent closed tree in evolving graphs. The extraction of periodic or near periodic subgraphs is considered in [18] where the problem is shown to be polynomial. Inokuchi and Washio in [16] and Robardet in [24] discuss finding frequent evolving patterns in dynamic networks. Chan et al. in [4] introduce a new pattern to be discovered from evolving graphs, namely regions of the graph that are evolving in a correlated manner. All of these works are to finding a subgraph sequence pattern, such that (a) its embedding in a graph sequence is frequent and (b) the behavior of these embedding are identical over time. Liu et al. in [20] proposes a random walk model with restart to discover subgraphs that exhibit significant changes in evolving networks. However, this method is only concerned with changes between two graphs, and cannot quantify changes of subgraph in a time interval. A subgraph which changes significantly in two successive time steps may not change frequently in the whole time interval.

Ren et al. in [23] propose the FVF framework to answer shortest path query for all snapshots in evolving graphs. Aggarwal et al. in [1] use a structural connectivity model to detect outliers which act abnormally in some snapshots of graph streams. Feigenbaum et al. in [11] explore the problem related to compute graph distances in a data-stream model, whose goal is to design algorithms to process the edges of a graph in an arbitrary order given only a limited amount of memory. Tantipathananandh et al. in [29, 30] propose frameworks and algorithms for identifying communities in social networks that change over time. Diehl and Görg in [7] present a generic algorithm for drawing sequences of graphs, and address the problem of computing layouts of graphs which evolve over time. There are many works that focus on some properties such as k-connectivity and minimal spanning tree in dynamic graphs. Even and Shiloach in [10] study the connected component problem ($k = 1$) in the early

1980s. Frederickson in [12] study the minimum spanning tree problem by giving a fully dynamic algorithm. Westbrook and Tarjan in [32] present the first partially dynamic algorithm for the maintenance of both 2-edge connected components and biconnected (2-vertex connected) components. Liang et al. in [19] deal with the fully dynamic maintenance of 2, 3-connected components of a graph in the parallel. Jarry et al. in [17] study the strongly connected components in evolving graphs with geometric properties. Dinitz and Nossenson in [8] propose an abstract model that describes the graph connectivity structure. By this model, they can check whether two vertices belong to the same 5-edge-connectivity class under edge insertion. Henzinger in [14] present an insertion-only algorithm for maintaining the exact and approximate size of minimum edge cut and minimum vertex cut of a graph. The objectives of above works are not to find the subgraph that changes frequently in an evolving graph.

7 Conclusion

In this paper, we studied finding a subgraph which change most in an evolving graph. We proposed an objective function $F(\cdot)$ to find a connected subgraph G_s in G, with the max $F(G_s)$ value. Our function $F(\cdot)$ is based on the cumulated connectivity change, and is effectively used to identify the most changed subgraph with a small number of unchanged edges included. We gave two new algorithms to compute cumulated connectivity change and a novel algorithm to identify the subgraph G_s with the max $F(G_s)$. We confirmed the effectiveness and efficiency of our algorithms using real-life datasets.

Acknowledgements This work is supported by the National Grand Fundamental Research 973 Program of China under grant 2012CB316200, the National Natural Science Foundation of China under grants 61173022, 61173023, and grants of the Research Grants Council of the Hong Kong SAR, China No. 419109, 418512.

References

1. Aggarwal, C.C., Zhao, Y., Yu, P.S.: Outlier detection in graph streams. In: ICDE (2011)
2. Bifet, A., Gavaldà, R.: Mining frequent closed trees in evolving data streams. Intell. Data Anal. **15**(1), 29–48 (2011)
3. Borgwardt, K.M., Kriegel, H.-P., Wackersreuther, P.: Pattern mining in frequent dynamic subgraphs. In: ICDM (2006)
4. Chan, J., Bailey, J., Leckie, C.: Discovering and summarising regions of correlated spatio-temporal change in evolving graphs. In: ICDM Workshops (2006)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
6. Cui, Y., Pei, J., Tang, G., Luk, W.-S., Jiang, D., Hua, M.: Finding email correspondents in online social networks. World Wide Web 1–24. doi:10.1007/s11280-012-0168-2
7. Diehl, S., Görg, C.: Graphs, they are changing. In: Graph Drawing (2002)
8. Dinitz, Y., Nossenson, R.: Incremental maintenance of the 5-edge-connectivity classes of a graph. In: SWAT, pp. 272–285 (2000)
9. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Algorithms and Theory of Computation Handbook (1999)
10. Even, S, Shiloach, Y.: An on-line edge-deletion problem. J. ACM **28**(1), 1–4 (1981)

11. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the data-stream model. SIAM J. Comput. **38**(5), 1709–1727 (2008)
12. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput. **14**(4), 252–257 (1985)
13. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. J. Soc. Ind. Appl. Math. **9**(4), 551–570 (1961)
14. Henzinger, M.R.: A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. J. Algorithms **24**(1), 194–220 (1997)
15. Horst Bunke, M.K., Dickinson, P.J., Wallis, W.D.: A Graph-Theoretic Approach to Enterprise Network Dynamics. Birkhauser (2007)
16. Inokuchi, A., Washio, T.: A fast method to mine frequent subsequences from graph sequence data. In: ICDM (2008)
17. Jarry, A., Lotker, Z.: Connectivity in evolving graph with geometric properties. In: DIALM-POMC (2004)
18. Lahiri, M., Berger-Wolf, T.Y.: Mining periodic behavior in dynamic social networks. In: ICDM (2008)
19. Liang, W., Brent, R.P., Shen, H.: Fully dynamic maintenance of k-connectivity in parallel. IEEE Trans. Parallel Distrib. Syst. **12**(8), 846–864 (2001)
20. Liu, Z., Yu, J.X., Ke, Y., Lin, X., Chen, L.: Spotting significant changing subgraphs in evolving graphs. In: ICDM (2008)
21. Musial, K., Budka, M., Juszczyszyn, K.: Creation and growth of online social network. World Wide Web 1–27. doi:10.1007/s11280-012-0177-1
22. Musial, K., Kazienko, P.: Social networks on the internet. World Wide Web 1–42. doi:10.1007/s11280-011-0155-z
23. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. In: VLDB (2011)
24. Robardet, C.: Constraint-based pattern mining in dynamic graphs. In: ICDM (2009)
25. Schweller, R.T., Gupta, A., Parsons, E., Chen, Y.: Reversible sketches for efficient and accurate change detection over network data streams. In: Internet Measurement Conference (2004)
26. Shoubridge, P., Kraetzl, M., Wallis, W.D., Bunke, H.: Detection of abnormal change in a time series of graphs. J. Interconnect. Netw. **3**(1–2), 85–101 (2002)
27. Sun, J., Faloutsos, C., Papadimitriou, S., Yu, P.S.: Graphscope: parameter-free mining of large time-evolving graphs. In: KDD (2007)
28. Sun, J., Tao, D., Faloutsos, C.: Beyond streams and graphs: dynamic tensor analysis. In: KDD (2006)
29. Tantipathananandh, C., Berger-Wolf, T.Y.: Constant-factor approximation algorithms for identifying dynamic communities. In: KDD (2009)
30. Tantipathananandh, C., Berger-Wolf, T.Y., Kempe, D.: A framework for community identification in dynamic social networks. In: KDD (2007)
31. Tong, H., Papadimitriou, S., Sun, J., Yu, P.S., Faloutsos, C.: Colibri: fast mining of large static and dynamic graphs. In: KDD (2008)
32. Westbrook, J., Tarjan, R.E.: Maintaining bridge-connected and biconnected components on-line. Algorithmica **7**(5&6), 433–464 (1992)
33. Wu, D., Ke, Y., Yu, J., Yu, P., Chen, L.: Leadership discovery when data correlatively evolve. World Wide Web **14**, 1–25 (2011). doi:10.1007/s11280-010-0095-z
34. Yu, Z., Zhou, X., Zhang, D., Schiele, G., Becker, C.: Understanding social relationship evolution by using real-world sensing data. World Wide Web 1–14. doi:10.1007/s11280-012-0189-x