Algorithms for Offline Tracking of Connected Components in Large Evolving Networks

Kamer Kaya[1], Erik Saule[1], Onur Küçüktunç[1,2], and Ümit V. Çatalyürek [1,3]

[1]Department of Biomedical Informatics, The Ohio State University
[2]Department of Computer Science and Engineering, The Ohio State University
[3]Department of Electrical and Computer Engineering, The Ohio State University
email:{*kamer,esaule,kucuktunc,umit*}*@bmi.osu.edu*

Abstract

Given a large evolving network with time information on its edges, we are interested in how and when its connected components are formed through time. Such information is useful while analyzing the characteristics of the network's snapshots taken at different time points. This analysis can be used to answer various queries such as what is the time point where two people are first connected in a professional network, how the scientific communities merged over time in a citation graph, or how conversations are formed and attracted new users in a forum discussion. The sensitivity of such an analysis increases with the number of time points and the cost of the analysis increase along with. We propose efficient algorithms and a compact representation of component structures evolving through time for both directed and undirected networks. For an undirected network with m edges, the time complexity of the algorithm is almost linear with m. For the directed case, the time complexity is $\mathcal{O}(m \log \tau)$ where τ is the number of snapshots.

Keywords: Evolving network; connectivity; connected component; time complexity

1 Introduction

A graph is the most commonly used model of today's large networks which arise in nature, life, and science. In this model, the vertices correspond to the nodes in the network and the edges show that there is an interaction between the nodes. Such interactions can be directed or undirected, i.e., one way or two ways. In graph theoretical terms, two nodes are *connected* if it is possible to reach from either one to the other via a set of interactions. A maximal set of connected nodes is called a *component* of the network. For a better understanding of a network, various structural properties, such as size, degree distribution, average distance and connected components of the corresponding graph is analyzed in the literature [3, 10]. For example, Broder et al. showed that the sizes of the connected components of web graph obey the power law [3]. According to their analysis, 90% of the web formed a single connected component when it is modeled as an undirected graph. On the other hand, when the web is modeled as a directed graph, the largest connected components contained only 28% of the pages.

Most of today's networks are dynamic and they evolve over time. The Web is evolving, social networks are evolving, new interactions are set among the network elements perpetually. The properties of such networks also change over time. Evaluating network properties at different time points is a common approach to analyze their dynamics [2, 11, 12]. For a detailed analysis, a sufficient number of snapshots is required since the *sensitivity* increases with that number. To analyze the connectivity properties of each snapshot independently, one may need to construct the component structure for each of them. In addition to independent analysis, investigating the evolution of connected components is a problem that recently gained interest [11, 12].

There are linear time algorithms for finding the connected components of a graph [5, 15, 16]. To track the evolution of the components, one can trivially use these algorithms for each *version* of the graph at different time points. Nevertheless, for a sensitive analysis, the cost of the trivial approach can be a burden since its worst case time complexity linearly increases with the number of versions. In this paper, we are interested in networks with persistent interactions. That is, either the interactions are eternal or when an interaction between two nodes is deleted, it is assumed to be deleted from all versions. We propose efficient algorithms to construct

the connected component structure with its evolution. The proposed algorithms take a network and a set of time points as inputs, and return a very compact representation of the component evolution. To represent this structure, we use an *evolution forest*. For the undirected case, the construction of the evolution forest is an application of the well known disjoint set data structure. We show how the time information can be stored in a compact way and devise an algorithm with $\mathcal{O}(m\alpha(n))$ time complexity where m and n are the number of interactions and elements in the network, respectively, and $\alpha(n)$ is the inverse Ackermann function, which is very small for all practical values of n. For the directed case, we propose a novel $\mathcal{O}(m \log \tau)$ algorithm for the evolution forest construction where τ is the desired number of versions in the analysis.

The proposed connected component structure helps answering time-based connectivity queries efficiently, and has various applications such as finding the time point where two people are connected within a professional network, how the scientific communities merged over time using a citation graph, or how conversations are formed and attracted new users in a forum discussion.

The paper is organized as follows: In Section 2, the notation for the paper is introduced and related work from the literature is discussed. Section 3 gives the algorithms and compact data structures for tracking the connected component evolution, and Section 4 gives some examples for the practical usage of proposed algorithms and data structures. Section 5 shows the results of the experiments, and Section 6 concludes the paper.

2 Notation and Background

Let $G = (V, E)$ be an evolving network modeled as a (directed) graph with n vertices and $m \geq n$ edges. Each edge $e \in E$ is labeled by $time(e)$, the time it is added to G. A *path* is a vertex sequence such that there exists an edge between consecutive vertices. A *cycle* is a path in which the first and the last vertices are the same. If G contains no cycles we say that G is *acyclic*. A vertex $u \in V$ is *connected* to $v \in V$ if there is a path from u to v. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$.

If G is undirected and v is connected to u for all $u, v \in V$ we say G is *connected*. We call a subgraph G' *maximally connected* if there exists no connected subgraph G'' of G such that G' is a subgraph of G''. A maximally connected subgraph of G is called a *connected component* of G. The connectivity definition is similar for the directed graphs (digraph): we say a digraph G is *strongly connected* if for all $u, v \in V$, v is connected to u (and vice versa). Similar to the undirected case, a *maximally strongly connected* G' is called a *strong component* (SC) of G. Note that an acyclic digraph has n strong components where each component is *trivial*, i.e., formed by a single vertex.

Given an evolving network $G = (V, E)$, let $\Pi = \{V_1, V_2, \ldots, V_k\}$ be a k-partition of V where each vertex set V_i is nonempty, $V_i \cap V_j = \emptyset$ for $i \neq j$, and $\cup_{i=1}^{k} V_i = V$. Let $G^{\Pi} = (V^{\Pi}, E^{\Pi})$ be the *quotient graph* of G with respect to Π. In the quotient graph, each vertex set $V_i \in \Pi$ of G is clustered into a *supervertex* $\nu_i \in V^{\Pi}$. In addition, for all $uv \in E$ such that $u \in V_i$, $v \in V_j$, and $i \neq j$, there exists $\nu_i\nu_j \in E^{\Pi}$ where $time(\nu_i\nu_j) = time(uv)$. If G is directed, the direction of $\nu_i\nu_j$ should be consistent with the direction of uv. Even though G is a simple graph, G^{Π} can be a multigraph, i.e., there can be multiple edges between two supervertices. The definitions of connectivity and strong connectivity in multigraphs are the same as those in graphs.

Let $P = \{t_1, t_2, \ldots, t_\tau\}$ be a set of τ time points such that $t_i < t_j$ for all $1 \leq i < j \leq \tau$. For each $t \in P$, let $E_t = \{e \in E : time(e) \leq t\}$ be a subset of E. Hence, $E_{t_1} \subseteq E_{t_2} \subseteq \cdots \subseteq E_{t_\tau}$. We call the subgraph $G_t = (V, E_t)$ the *version* of G at time t. We define the connected component tracking problem as follows: given a network G and a set of time points P, we want to know which (strongly) connected components exist in G_t for all $t \in P$. This information then can be used to answer various queries such as the earliest time point $t \in P$ when u and v became connected.

2.1 Related Work:

Tracking the connected components of a dynamic graph has been investigated before for both undirected [4, 9] and directed [6, 7, 13] graphs. Henzinger and King studied the fully dynamic case and provided an $\mathcal{O}(p \log^3 n)$ algorithm to track the connected components of an undirected graph where p is the number of edge insertions/deletions [9]. Ediger et al. investigated the tracking problem for streaming social networks in a fully dynamic setting and proposed efficient parallel algorithms for sparse networks [4]. As far as we know, this is one of the best algorithms for the undirected graphs.

For directed graphs, the problem of incremental maintenance of the strong components in case of edge additions has been investigated by several researchers. In a recent study, Haeupler et al. proposed algorithms with complexity $\mathcal{O}(m^{3/2})$ and $\mathcal{O}(n^{5/2})$ [6, 7]. These algorithms are online: they update their data structures after every edge addition/deletion in an efficient way and wait for the next version. Usually, such algorithms can only answer the connectivity queries related with

the last version. Hence, without any modification, they are not able to answer queries related with previous updates. However, some online algorithms have this property. For example, Roddity and Zwick investigated the dynamic strong component maintentance problem and proposed an algorithm which can also answer queries related with strong connectivity information in all versions of the graph [13]. Similar to our setting, they assume that when an edge is deleted from a version of the graph, it is deleted from all previous versions. Hence, our assumption about persistency is valid in their model where the time of the updates in the online setting correspond to the time points in the offline setting. The time complexity of their algorithm is $\mathcal{O}(m\tau)$ for τ batch updates.

3 Offline Algorithms for Tracking Evolution of Connected Components

When G_{t_τ} is (strongly) connected, our algorithms construct a tree T called *evolution tree* which contains the connectivity information for all snapshots of G in a compact form. The vertices of T correspond to the vertices in V. For each vertex v in T, we store its parent vertex $parent(v)$. We also label each vu edge in T with a time point in P to store when the parent information is set for the first time. Similarly to graph notation, we denote the label by $time(vu)$ for a parent pointer from v to u in T. Hence, $u = parent(v)$ and $t = time(vu)$ imply that u and v are first put into the same component at version G_t. In an evolution tree, each node represents a component. Hence, $u = parent(v)$ and $t = time(vu)$ also implies that all the nodes in the components represented by u and v are in a single component formed at time t. Figure 1 shows the versions of a toy digraph with 4 vertices and 6 edges and its evolution tree for $P = \{1, 2, 3\}$.

Throughout the paper, for simplicity, we assume that the graph in the last version, $G = G_{t_\tau}$, is (strongly) connected. Note that if it has multiple components, one can first find these components using a linear time algorithm. For undirected graphs, a graph traversal algorithm, such as breadth-first search, is sufficient to find the connected components. For directed graphs, one can use an $\mathcal{O}(m)$ algorithm to find the strong components [5, 15, 16]. After all components of G_{t_τ} are found, our algorithms can be executed on each component independently and an *evolution forest* $\mathcal{T} = \{T_1, T_2, \ldots, T_c\}$ is obtained.

3.1 Constructing Evolution Trees for Undirected Networks: For undirected networks, constructing an evolution tree is an application of the well known *Union-Find* algorithm which is an efficient way

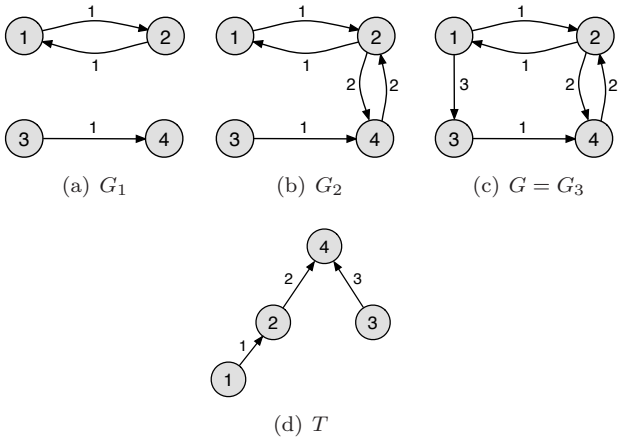


Figure 1: Versions of a simple digraph G with 4 vertices and 6 edges for $P = \{1, 2, 3\}$. The strong components of G_1 are {1,2}, {3}, and {4} where $G = G_3$ is strongly connected. T shows the changes in strong connectivity from G_1 to G_3. To find the (strong) components of G_t, one can remove the edges $\{vu \in T : time(vu) > t\}$ from T, construct a forest, and find the trees inside it. Note that each such tree in this reduced form of T represents a (strongly) connected component in G_t. In the algorithms proposed below, we will use the root of each such tree as a representative of a component.

to manipulate disjoint sets [17]. Our evolution tree data structure supports three main operations which are used by the Union-Find algorithm:

- MakeSet(u) creates a tree with a single node $u \in V$ and sets $parent(u)$ to u.
- $r_u \leftarrow$ Find(u) finds the root r_u of the tree containing u and returns it.
- $r \leftarrow$ Union($R = \{r_1, \ldots, r_\ell\}, t$) combines the trees with roots in R. It selects first the tree with the maximum height. Let r be the root of this tree. For all $r' \in R$ except r, it sets $parent(r')$ to r and $time(r'r)$ to t. When it finishes, it returns the root r of the unified tree.

The Union-Find algorithm has been previously used to find the connected components of an undirected graph without any time information on the edges. Modifying the Union operation for temporality is a non-complex task for the undirected case. We give the description of the evolution tree construction in Algorithm 1 for completeness. Assume that no edge exists in the graph at the beginning. That is, E_{t_0}, the initial edge set, is empty. Hence, there are n components, thus n trees each containing a single node.

The MakeSet operation is used to create these trees. When a new edge uv is added to the network, first the roots r_u and r_v of the trees containing u and v are found by executing two Find operations. If $r_u = r_v$, there is no need for an update since u and v are already connected. If this is not the case, a Union operation is executed to combine the components of u and v.

Algorithm 1 ETreeU$(G(V,E), P = \{t_1, \ldots, t_\tau\})$

```
1:  for each u ∈ V do
2:    MakeSet(u)
3:  E_{t_0} ← {}
4:  for 1 ≤ i ≤ τ do
5:    for each uv ∈ E_{t_i} \ E_{t_{i-1}} do
6:      r_u ← Find(u)
7:      r_v ← Find(v)
8:      if r_u ≠ r_v then
9:        Union({r_u, r_v}, t_i)
10:       if the graph is connected then
11:         return
```

In Algorithm 1, the MakeSet operation is used to construct n disjoint sets (trees), each containing a single vertex. Hence, the complexity of the first loop is $\mathcal{O}(n)$. For the rest of the algorithm, using the techniques proposed by Tarjan, the Union and Find operations can be implemented in such a way that each iteration is handled in $\mathcal{O}(\alpha(n))$ amortized time where $\alpha(n)$ is the inverse of $f(n) = A(n,n)$, and A is the rapidly growing Ackermann function [17]. The value of $\alpha(n)$ is smaller than 5 for any practical number of vertices n. Hence, the overall time complexity of the algorithm is $\mathcal{O}(m\alpha(n))$. For the analysis, we assume that the edges are already partitioned into sets $E_{t_i} \setminus E_{t_{i-1}}$ for $1 \leq i \leq \tau$. If this is not the case they can be sorted in $\mathcal{O}(m + \tau)$ time by using counting sort. Note that since these disjoint edge sets are processed in increasing time order, this algorithm is suitable for the online setting.

3.2 Constructing Evolution Trees for Directed Networks:

Contrary to the undirected case, the construction of the evolution tree is complicated for directed graphs. Note that when an edge uv is added to an undirected graph, the only components that can be combined are the ones containing u and v. On the other hand, for a directed graph, the number of strong components which might be combined after an edge addition can be any number between 0 and n. Besides, finding such components is harder since checking only the endpoints of the edges will not suffice. Given G and $P = \{t_1, \ldots, t_\tau\}$, one can compute the strong components of each snapshot G_{t_i} for $1 \leq i \leq \tau$. Since finding strong components of a graph takes linear time, the complexity of this naïve approach is $\mathcal{O}(m\tau)$. One can improve this approach in practice by using quotient graphs. In the online setting, Roddity and Zwick followed this idea and proposed an algorithm [13]. Algorithm 2 shows the offline algorithm based on Roddity and Zwick's idea for the online setting.

Algorithm 2 ETreeD1$(G(V,E), P = \{t_1, \ldots, t_\tau\})$

```
1:  for each u ∈ V do
2:    MakeSet(u)
3:  G' ← G_{t_1}
4:  for 1 ≤ i ≤ τ do
5:    Let s be the number of SCs of G' = (V', E')
6:    for each component C_j = (V_j, E_j) of G' do
7:      if V_j contains a single vertex u then
8:        r_j ← u
9:      else
10:       r_j ← Union(V_j, i)
11:   if s = 1 then
12:     return
13:   if i < τ then
14:     if s ≠ |V'| then
15:       Let Π = {V_1, V_2, ..., V_s}
16:       G^Π = ({r_1, r_2, ..., r_s}, E^Π)
17:       G' ← G^Π
18:     for each uv ∈ E_{t_{i+1}} \ E_{t_i} do
19:       r' ← Find(u)
20:       r'' ← Find(v)
21:       if r' ≠ r'' then
22:         E' ← E' ∪ {r'r''}
```

Algorithm 2 starts with the graph $G' = G_{t_1}$. At ith iteration, after finding the strong components of G' at time t_i, the evolution tree is updated respectively. That is, for each nontrivial component of G', the algorithm combines the subtrees corresponding to the vertices in that component. Then, the graph for the next iteration is constructed in two steps: first, the quotient graph G^Π of G' is computed by using a partition Π corresponding to the strong component decomposition of G', and G' is set to G^Π. Note that if G' is acyclic then the quotient graph will already be equal to G'. So, there is no need to construct the quotient graph in this case. Second, each edge $uv \in E_{t_{i+1}} \setminus E_{t_i}$ is inserted into E^Π between the supervertices r' and r'' if $r' \neq r''$. After computing the new graph, the algorithm sets it to G' and continues with the next iteration.

Although our preliminary experimental results show that using quotient graphs can reduce the tree construction time greatly in practice, in theory, it does not improve the worst case time complexity $\mathcal{O}(m\tau)$ of the naïve approach which computes the strong components of G_{t_i} at each iteration. This can be understood

from the fact that $G_{t_{\tau-1}}$ and hence all G_{t_i} for $i < \tau$ can be acyclic. In this case, there is no difference between the naïve approach and Algorithm 2.

Since Algorithm 2 is originally proposed for the online setting, it always uses the edges in E_{t_i} while computing the strong components of G' at ith iteration. However, in our case, all the edge information is available beforehand. That is, it should be possible to obtain a novel algorithm better than the naïve approach in theory and faster than Algorithm 2 in practice. Such an algorithm is given in Algorithm 3.

Algorithm 3 ETREED2($G(V,E), P = \{t_1, \ldots, t_\tau\}$)

```
1: for each u ∈ V do
2:     MAKESET(u)
3: SETPARENTS(G, P, 0, τ)
```

Algorithm 4 r = SETPARENTS($G = (V,E), P = \{t_1, \ldots, t_\tau\}, i, j$)

Require: $0 \le i < j \le \tau$ such that $t_i, t_j \in P$, $G_{t_i} = (V, E_{t_i})$ is acyclic, and $G = G_{t_j}$ is strongly connected.

Output: The root r of the corresponding evolution tree (an index from 1 to n corresponding to a vertex of the initial call) is returned, and T is constructed.

```
1: if i = j − 1 then
2:     return UNION(V, t_j)
3: else
4:     k = ⌈(i + j)/2⌉
5:     Let s be the number of G_{t_k}'s strong components
6:     Let C_ℓ = (V_ℓ, E_ℓ) denote the ℓth SC of G_{t_k}
7:     for ℓ = 1 to s do
8:         if C_ℓ is nontrivial then
9:             r_ℓ ← SETPARENTS(C_ℓ, P, i, k)
10:        else
11:            r_ℓ ← the vertex in C_ℓ
12:    if s > 1 then
13:        Let Π = {V_1, ..., V_s} be a partition of V
14:        Let G^Π be the quotient graph induced by Π
           where each V_ℓ is represented by a supervertex
           r_ℓ for 1 ≤ ℓ ≤ s
15:        return SETPARENTS(G^Π, P, k, j)
16:    else
17:        return r_1
```

Algorithm 3 first calls MAKESET to create n trees each containing a singleton vertex. Then it calls SETPARENTS given in Algorithm 4 to construct the evolution tree. SETPARENTS takes four arguments: a digraph G, a set of τ time points P, and two integers i and j such that $0 \le i < j \le \tau$, G_{t_i} is acyclic, $G = G_{t_j}$ is strongly connected.[1] Algorithm 4 uses a recursive approach to find the time points k, $i < k \le j$, in which new components are formed and the connectivity information changes. It starts by checking if $i = j - 1$. If this is the case we know that all the vertices of the acyclic G_{t_i} will be combined in the next version G_{t_j}. In this case, the algorithm calls the UNION operation to combine the corresponding trees and to set the new edge labels to t_j. It then returns the root r of the this new tree and terminates gracefully. If $i \neq j - 1$, the algorithm examines the strong components of the version at time t_k for $k = \lceil (i + j)/2 \rceil$. For each nontrivial strong component C_ℓ of G_{t_k}, a recursive call SETPARENTS(C_ℓ, P, i, k) is made. Note that $C_\ell \subset G_{t_k}$ is strongly connected, and its snapshot at time t_i is known to be acyclic. Hence, the last two parameters are set to i and k, respectively. After all the recursive calls at line 9 terminate, we obtain a forest where each tree in the forest corresponds to a strongly connected component of G_{t_k}. Since $G = G_{t_j}$ is strongly connected, the trees in the forest should be connected to each other and form the evolution tree sometime between $(t_k, t_j]$. To set these connections, another recursive call SETPARENTS(G^Π, P, k, j) is made at line 15. Here, G^Π is a quotient of G where the parts in the partition Π correspond to the strong components of G_{t_k}. Since the strong components are maximal by definition, the snapshot of G^Π at time t_k is acyclic. Furthermore, G^Π itself is strongly connected since G is strongly connected. Hence, the last two parameters k and j are correct.

Algorithm 4 does not use any FIND operation because of its structure. While other algorithms construct the tree via the addition of the network edges at each iteration, SETPARENTS uses those edges only for the first call. For the recursive calls, it uses G's edges which is a quotient graph of some strongly connected subgraph of the network. Besides, since the supervertex representing a strong component of G_{t_k} is already obtained from a previous recursive call, we do not need the FIND operation. Note that the FIND operation can be still useful after the evolution tree construction for some operations such as checking if two nodes are in the same component.

THEOREM 3.1. *Algorithm 4 has time complexity* $\mathcal{O}(m \log \tau)$.

Proof. First observe that the UNION operations takes $\mathcal{O}(n)$ time since a UNION operation costs $\mathcal{O}(1)$ amor-

[1] Note that the algorithm is recursive and the letter G in the pseudo-code and the related text does not denote the original network graph except for the initial call.

tized time and there are n nodes in the network. Excluding this part and the recursive calls, the time complexity of the algorithm's body, i.e., finding strong components of G_{t_k} and quotient graph construction, is linear with respect to $|V|$ and $|E|$.

Now consider the call tree for SetParents where each node corresponds to an execution of the algorithm. The root node is the initial call, and a node's children are the recursive calls it makes. We claim that each edge of the graph is traversed exactly once in each level of the call tree. Indeed, when performing recursive calls, the edges of the graph are partitioned into $p+1$ parts: one part for each strong component and one part for the quotient graph. Therefore, each edge exists at most once in each level of the call tree. So each level of the call tree contains at most m edges. Following this, we can conclude that there are also at most m vertices at each level since the input graphs of the recursive calls are strongly connected. That is, the number of the edges they have cannot be less than the number of vertices. These observations lead us to a complexity of $\mathcal{O}(m)$ per level of the call tree.

For each call SetParents(G, P, i, j), there are $j-i$ time points the algorithm needs to check. For further recursive calls, this number is either $k-i$ and $j-k$ where $k = \lceil \frac{i+j}{2} \rceil$. That is, the number of time points that the algorithm examines in the worst case is decreasing geometrically. Since there are τ of them at the beginning, and the algorithm stops when $j = i+1$, the height of the call tree is $\mathcal{O}(\log \tau)$. Combining all these observations concludes the proof.

Corollary 3.1. *Algorithm 3 has time complexity* $\mathcal{O}(m \log \tau)$.

4 Using an evolution forest

As mentioned above, the evolution forest $\mathcal{T}$ is a compact representation for the evolution of connected components. After its construction, it can be used as is or processed for further efficiency while answering several queries on the time-based connectivity information of the network. Here, we consider two examples.

Let $P = \{t_1, \ldots, t_\tau\}$ be the set of time points used for constructing the evolution forest and let the query be about the number of components at each $t_i \in P$. By using the evolution forest $\mathcal{T}$, Algorithm 5 answers this query in $\mathcal{O}(n+\tau)$ time as follows. Let $nc[i]$ be the number of components at time t_i. We know that $nc[0] = n$ since there are n trivial components at the beginning. Since

$$nc[i] - nc[i-1] = |\{uv : time(uv) = t_i\}|,$$

it follows that

$$nc[i] - nc[0] = |\{uv : time(uv) \leq t_i\}|$$

for all $1 \leq i \leq \tau$. Hence, a prefix sum is sufficient to compute $n - nc[i]$ for each t_i. In the last step, subtracting this number from n gives us the number of strong components.

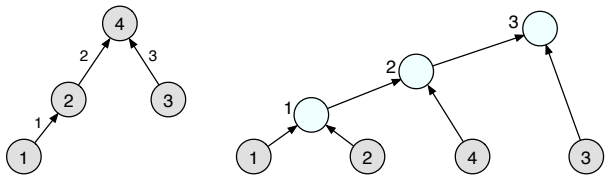
Algorithm 5 NumComponents$(\mathcal{T} = (V, E), P = \{t_1, \ldots, t_\tau\})$

```
1: for 1 ≤ i ≤ τ do
2:     comb[i] ← 0
3: for each edge uv ∈ E do
4:     t ← time(uv)
5:     comb[t] ← comb[t] + 1
6: for 2 ≤ i ≤ τ do
7:     comb[i] ← comb[i] + comb[i − 1]
8: for 1 ≤ i ≤ τ do
9:     nc[i] ← n − comb[i]
```

Another useful information is the earliest time of connectivity for a given pair of nodes u and v in the network. This operation can be answered by searching the lowest common ancestor (LCA) of u and v in $\mathcal{T}$. If no common ancestor of u and v exists they are not connected to each other. Otherwise, they are in the same tree T in the forest. In this case, the LCA of u and v is the ancestor in the lowest level of T. Let r be the LCA. If $r = u$ the connection time is the label of the edge $v'u$ on the path from v to u in T. Similarly, if $r = v$ the label of the edge $u'v$ on the path from u to v is the answer. On the other hand, when r is not equal to u and v, let u' and v' be its two children on the paths from u to r and v to r, respectively. In this case, the connection time can be found by $max(time(u'r), time(v'r))$. These edges can be found by traversing the paths from u and v to the root of T simultaneously. Hence, its complexity is proportional to the height of T which is $\mathcal{O}(\log n)$ due to the implementation of Union operation.

The previous query can be answered in a more efficient way by using a less compact data structure, *components forest* [13], which can be obtained from $\mathcal{T}$ in $\mathcal{O}(n)$ time. In this structure, each vertex in G is represented by a leaf node, and instead of the edges, the non-leaf nodes have time labels. When a set of vertices are combined at time t, a new parent node is added with label t having the connected vertices as its children. Figure 2 shows an evolution tree T and the corresponding tree T' in components forest. Given two nodes, their LCA's label in a components forest is equal to their earliest time of connectivity. Hence, we only need to find the LCA. There exist several

$\mathcal{O}(1)$ algorithms in the literature to find the LCA of two vertices [1, 8, 14]. Hence, one can preprocess the evolution tree T and use one of these algorithms to answer such queries in constant time. For different types of queries, T can be processed in different ways to minimize the response time.



(a) T (b) T'

Figure 2: An evolution tree T and the corresponding components tree T'. The gray colored leafs of T' are the vertices in the graph. The white nodes correspond to the combinations of components through time.

5 Experimental Results

We conduct a set of experiments to evaluate the performance of the algorithms in the directed case. Algorithms 2 and Algorithm 3 are sequentially implemented in the C programming language and their performances are compared on a computer with 2.27GHz dual quad-core Intel Xeon CPUs and 48GB of main memory. The compiler is gcc version 4.5.2, and -O3 optimization flag is used.

We use six directed networks from the Stanford Large Network Dataset Collection.[2] For each network, we are interested in the evolution tree construction for the largest strong component in the corresponding graph. The numbers of vertices and edges in the largest strong component of each network are given in Table 1.

	largest SC	
Name	n	m
amazon0601	395,234	3,301,092
soc-LiveJournal1	3,828,682	65,825,429
soc-sign-epinions	32,223	443,506
web-BerkStan	334,857	4,523,232
web-Google	434,818	3,419,124
WikiTalk	111,881	1,477,893

Table 1: Numbers of vertices and edges in the largest strong component of each network.

Before executing the algorithms on the largest strong component, we set the time labels of its edges randomly as an integer from 1 to τ. To do this, we first choose a number from a random distribution and convert it to an integer by first adding 0.5 and then removing its decimal point. We use four different distributions: uniform, normal, exponential, and *reversed exponential*. To reverse the exponential distribution, we first choose an integer $1 \leq t \leq \tau$ and use $\tau - t + 1$ as the label of an edge.

As Figure 3 shows, the proposed algorithm ETreeD2 is much faster than ETreeD1 especially when τ is large. For $\tau \geq 400$, the proposed algorithm is better for all networks and random distributions. Furthermore, when $\tau < 400$, ETreeD1 is only better in a few instances whose edges are labeled by using the exponential distribution. When $\tau = 2000$, for the largest component we have, which is the one in soc-LiveJournal1, ETreeD2 is approximately 10 times faster than ETreeD1 for the normal and exponential distributions. Similar relative performances can also be observed for other networks.

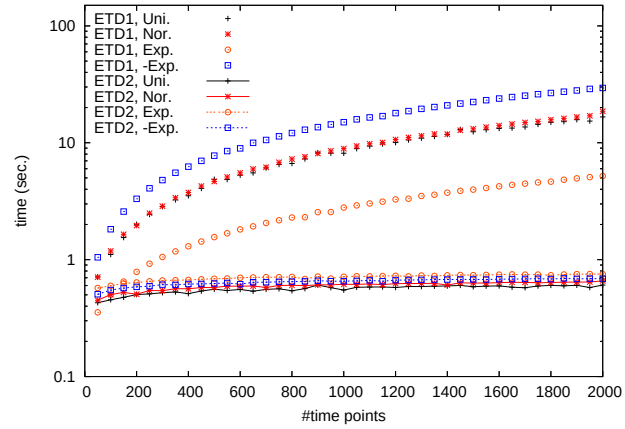
ETreeD1 is very sensitive to the change of random distributions. For example, its performance differs drastically for the exponential and reversed exponential distributions. This can be explained as follows: when the exponential distribution is used, most of the interactions in the network occur in the first few time points. Hence, the intermediate versions have larger strong components, and the quotient graphs have less number of vertices and edges. On the other hand, when the distribution is reversed as described above, the exact opposite happens, and the numbers of vertices and edges processed at each iteration increase. Experimental results show that compared with its performance for the exponential distribution, ETreeD1 can spend 5 times more with the reversed exponential distribution. As Figure 3 shows, although the performance of the proposed algorithm ETreeD2 can decrease when the labels are generated by using the exponential distribution, the algorithm is not as sensitive as ETreeD1 to the distribution. Furthermore, it continues to perform better especially when τ increases. Hence, we can say that ETreeD2 is better than ETreeD1 in general.

6 Conclusion and Future Work

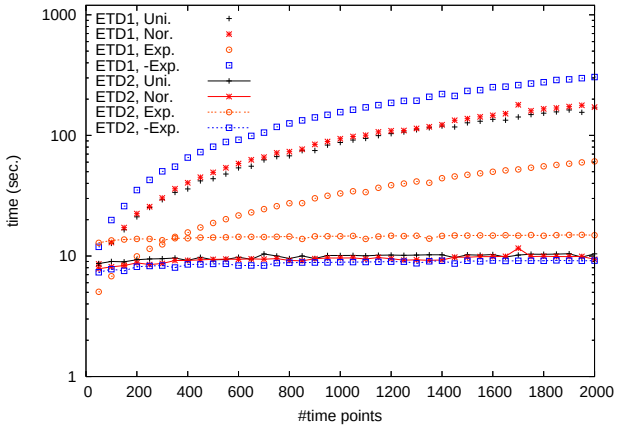
We investigated the problem of tracking the evolution of connected components in a network and propose efficient algorithms. The algorithms are better than the existing solutions both in theory and in practice, especially when one needs to analyze the dynamics of a network in high sensitivity.

In this work, we investigated the networks and problems where the network connections or their effects are persistent. If the network is only growing without

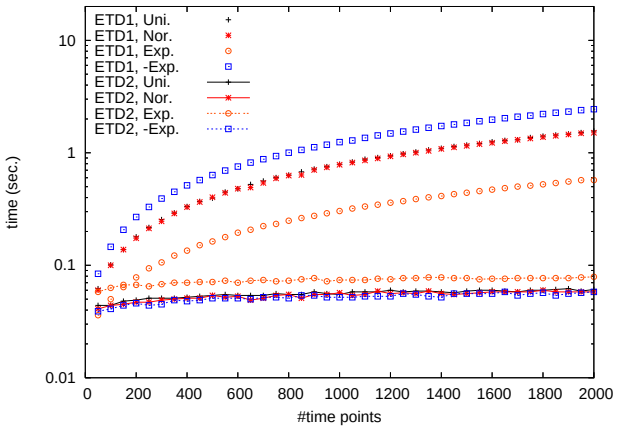
[2] http://snap.stanford.edu/data/



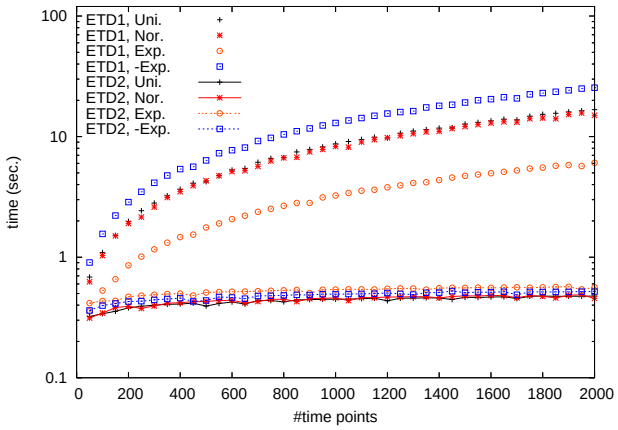
(a) amazon0601



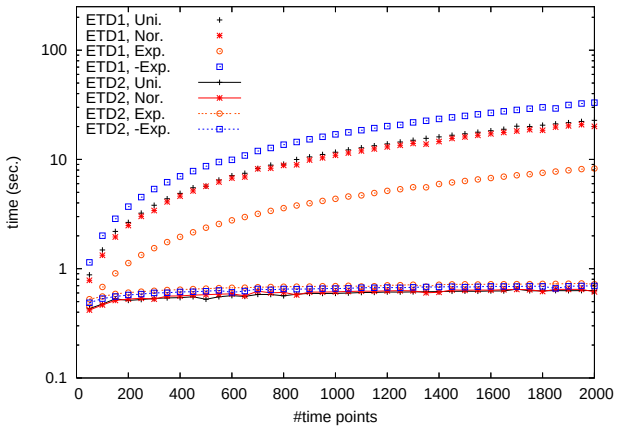
(b) soc-LiveJournal1



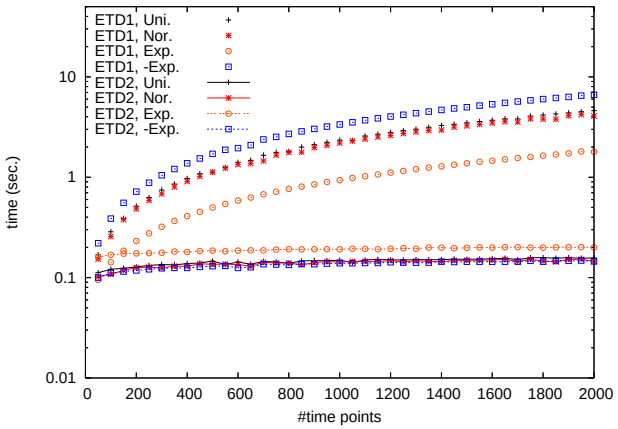
(c) soc-sign-epinions



(d) web-BerkStan



(e) web-Google



(f) WikiTalk

Figure 3: Execution times of the algorithms with respect to number of time points for six different networks. The execution times are given in log scale. We used 4 random time labelings where each one is using a different probability distribution. For each distribution, 20 instances are generated for each network and their evolution trees are created. The average times for these 20 executions are used to evaluate the performances. In the plots, ETD1 and ETD2 denote Algorithms 2 and 3. The suffixes Uni., Nor., Exp., and -Exp. denote uniform, normal, exponential, and reversed exponential probability distributions, respectively.

any edge/node deletions our algorithms can be used as is. The proposed algorithms are also useful for some queries in fully dynamic networks with edge deletions when the effect of a connection is persistent. For example, consider a social network in which u added v as a friend at time t and erased her later. If we read this connection as u knows v since time t this information will be persistent, and related connectivity queries can be answered by using the proposed algorithms. However, if the connection is read as u is a friend of v at time t the algorithms in this paper are not suitable. In the future, we are planning extend our algorithms to handle such cases.

Both Algorithm 2 and Algorithm 3 assume that the whole graph can be stored in memory. Today, this assumption may be a little bit optimistic for some real-world graphs. Hence, another future work is modifying the proposed algorithms for the out-of-core or distributed-memory setting.

Acknowledgments

This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775 and NSF grants CNS-0643969, OCI-0904809 and OCI-0904802.

References

[1] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN '00*, pages 88–94, 2000.

[2] M. Bloznelis, J. Jaworski, and K. Rybarczyk. Component evolution in a secure wireless sensor network. *Networks*, 53:19–26, January 2009.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proc. 9th WWW/Computer Networks*, pages 309–320, 2000.

[4] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. In *Proc. IPDPSW 2011*, pages 1691–1699, May 2011.

[5] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74:107–114, May 2000.

[6] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster algorithms for incremental topological ordering. In *Proc. ICALP'08*, pages 421–433, 2008.

[7] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms (to appear)*, 2011.

[8] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, May 1984.

[9] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46:502–516, July 1999.

[10] P. Holme and B. J. Kim. Vertex overload breakdown in evolving networks. *Phys. Rev. E*, 65:066109, Jun 2002.

[11] U. Kang, M. McGlohon, L. Akoğlu, and C. Faloutsos. Patterns on the connected components of terabyte-scale graphs. In *Proc. ICDM 2010*, pages 875–880, December 2010.

[12] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Proc. KDD'06*, pages 611–617, 2006.

[13] L. Roddity and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. STOC'04*, pages 184–191, 2004.

[14] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, December 1988.

[15] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[16] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[17] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, April 1975.